Parallelizing Serializable Transactions within Distributed Real-time Database Systems

Subhash Bhalla Masaki Hasegawa

The University of Aizu, Aizu-Wakamatsu,
Fukushima PO 965-8580, (Japan) {bhalla,d8041201}@u-aizu.ac.jp,

Abstract. A real-time database system supports a mix of transactions. These include the real-time transactions that require completion by a given deadline. At the support side, existing concurrency control procedures introduce delays due to non-availability of data resources. The present study makes an effort to introduce a higher level of parallelism for execution of real-time transactions. It considers simple extensions within a transaction processing system. These permit a real-time transaction to avoid delays due to ordinary transactions. These also eliminate elements of other unpredictable delays, due to deadlocks or simple waiting for data resources. Thus, the investigated procedures can perform critical functions in parallel to process time-critical transactions. In effect, it is a model of transaction execution that permits execution of real-time transactions without interference from other executing transactions, and by reducing other probabilistic delays.

1 Introduction

The use of real-time database systems is growing in many application areas such as, industrial process control systems, multi-media systems and many critical data access applications. In such a system, a critical transaction (computational task) is characterized by its computation time and a completion deadline. These systems are characterized by stringent deadlines, and high reliability requirements. Many approaches for implementation of Real-Time systems are being studied [9], [1], [4]. In most cases, the designers tend to extend the available approaches for concurrency control for the new environments. However, we propose to eliminate the elements that cause delays and consider introduction of parallelism based on alternative procedures.

Our research is aimed at isolation of real-time transactions (RTTs). Such transactions are proposed to be executed in parallel with no interference from other transactions. Earlier research efforts within concurrency control also try to isolate transaction classes. For example, isolation of read-only transactions [1], [7], multi-class queries [8], and class for restricting admission [1]. Briefly, our goal is -

1. Isolate a RTT and permit it to execute freely in parallel; and
2. Execute two conflicting RTTs with better completion guarantee for both transactions.

For our purpose, we examine the process of data allocation. The characteristics of the 2 Phase Locking based Concurrency Control scheme has been studied. The rest of this paper is organized in the following manner. Section 2. describes the problem with background. Section 3 proposes a validation based selective concurrency control mechanisms that can eliminate delays for the RTTs and reduce interference among other transactions. Section 4 presents a system model of transaction execution. Section 5 examines a criterion for serializability based on the notions of local access graphs (LAGs). Section 6 presents an algorithm for constructing the LAGs. The proof of correctness has been studied in section 7. Section 8 presents a study of performance evaluation. Finally, conclusions and summary are presented in section 9.

2 Nature of Delays

The present study aims at exploring,

- **Transaction Classification**
  - possibility of executing RTTs (time-critical transactions) with no interference from ordinary transactions;
- **Conflicts Among Remaining RTTs**
  - possibility of switching precedence in favor of a more urgent transaction; and
  - eliminate computational losses due to deadlocks, aborts, repeated rollbacks, and excessive overheads associated with access of frequently sought data items.

2.1 Real-Time System Environment

A requirement often imposed on transaction processing systems is that the schedule formed by the transactions, be Serializable [1]. A common method of enforcing serializability is based on two-phase locking. In a real-time environment, preceding ordinary transactions render a substantial portion of database inaccessible to an arriving real-time transaction. The RTDBS environment has the following type of transactions :

1. Real-Time Transactions (RTTs);
   These transactions are characterized by a known computation time estimate (Ct), and a execution time deadline (Dt), as -

$$Ct \ + \ delays \ \ll \ Dt$$

2. Status queries : the read only transactions; and
3. Ordinary Transactions (OTs) : the non-critical transactions that have no execution deadline associated with them.

The prominent delays on account of scheduled transactions are -

1. **Preceding Ordinary Transactions** - The transaction T4 fails to get data items requested (with exclusive access). Also, the transaction T6 fails to get data items requested (with shared access). Thus, an incoming RTT may need to wait until the executing transactions, release the data item [1].
2. **Delays and Conflicts Among RTTs** - The problem is further aggravated in case of RTTs that do not have predeclared read-set and write-set items. These seek locks for data items as these proceed with the computation. Consider a transaction T that reads data item x and writes on data item y. After a delay of d1 units, on receiving a lock for data item x, it computes the update values and seeks an exclusive lock for data item y, which becomes available after a delay of d2 units of time. Hence, the worst time estimate for computation time T is given by :

$$Ct \ + \ (d1 \ + \ d2) \ \ll \ Dt$$

3. **Deadlocks Among RTTs** The situation is further complicated by the occurrence of deadlocks, that can introduce more delays.

Thus, RTTs can have worst case estimates depending on the number of transactions executing in the system and the number of items sought by the transactions. This possibility can lead to a failure of an RTT.

3 The Proposed Model

3.1 Interference from Ordinary Transactions

In many of cases, a large component of the data resources are held by the scheduled (executing) ordinary transactions. The existing approaches are primitive in nature. These make a transaction manager (TM) dependent on multiple data managers (DMs) for priority based executions and aborts [1].

A conceptual model of transaction processing is shown in Figure 2. As per the idea, only the RTTs are permitted to lock data items. The ordinary transactions can execute by performing an additional validation based check [1], [3] [2]. It ensures that, there is no overlap among items in the RTT lock table, and the items read by ordinary transactions. In case of an overlap, the ordinary transactions roll-back. The resulting transaction processing system can perform all RTTs by ignoring existence of ordinary transactions.

For sake of implementation, in case of conflict between a RTT and an OT, the locks granted to ordinary transactions are ignored. The ordinary transaction is informed about possibility of failure during validation. The transaction commitment by the data manager is denied success, in case of a failure (of validation). Such an implementation reduces the data conflicts. These remain to exist among the few executing RTTs.

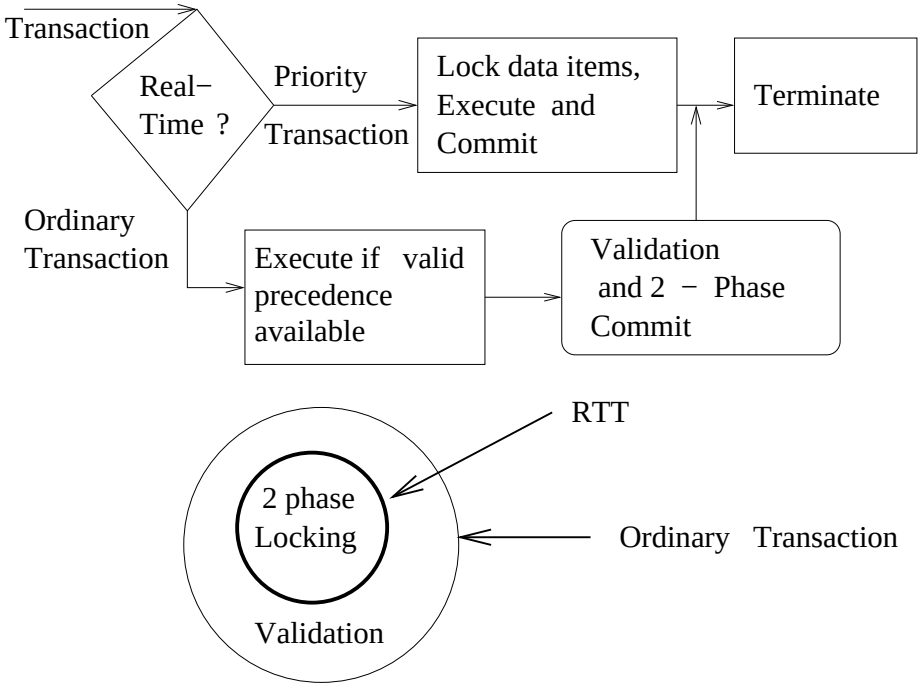

Fig. 1. Execution of real-time transactions in isolation (Brute-force Method)

3.2 Delays and Conflicts Among RTTs

In case of blocking, transactions begin to prepare Transaction-Wait-For graphs (TWFGs). Alternatively, probe messages are sent in order to detect the cause of blocking [1]. Consider an example, a frequently sought after item 'X'. Transactions T4, T3, T5, and T1 arrive in the same order. Other than T4, the three transactions detect that they need to wait for T4. Depending on there relative priority of execution, T3 may abort T4. Subsequently, T3 is aborted by T1, due to a higher priority. Similar aborts in the case of multiple frequently sought data items (**hot-spots** [1]), can cause delays due to deadlocks and frequent aborts. The proposed technique transforms the lock table at the DM level into a local access graph. It aligns the incoming transactions as T4 - T1, T3, T5 (if T4 is completing 2nd phase of '2-phase commit'), or as, T1 - T3, T4, T5.

3.3 Delays due Dead-locks

A distributed deadlock occurs when a transaction waits for locks held by another transaction, which in turn, is waiting (directly or indirectly) for locks held by the first transaction. The presence of any deadlock introduces delays and degrades the database performance [5].

4 The System Model

Based on the models of 2 phase locking and real-time computational environment [6], a set of assumptions are organized. It is assumed that a 2 phase locking discipline is followed. In the following section, a scheme to execute transactions in accordance with a precedence order is described.

4.1 Definitions for Real-time Database System

The distributed database system (DDBS) consists of a set of data items (say set 'D'). The DDBS is assumed to be based on a collection of (fixed) servers that are occasionally accessed by real-time hosts. A data item is the smallest accessible unit of data. It may be a file, a record, an array, or an object. Each data item is stored at a site (one only). However, this assumption does not restrict the algorithm in any way and can be relaxed in a generalized case. Transactions are identified as $T_i, T_j, \ldots$; and sites are represented by $S_k, S_l \ \ldots$; where, $i, j, k, l \ \ldots$ are integer values. The data items are stored at database sites connected by a computer network.

Each site supports a transaction manager (TM) and a data manager (DM). The TM supervises the execution of the transactions. The DMs manage individual databases. The network is assumed to detect failures, as and when these occur. When a site fails, it simply stops running and other sites detect this fact. The communication medium is assumed to provide the facility of message transfer between sites. A site always hands over a message to the communication medium, which delivers it to the destination site in finite time. For any pair of sites S_i and S_j , the communication medium always delivers the messages to S_j in the same order in which they were handed to the medium by S_i.

4.2 The Transaction Model

We define a transaction as a set of atomic operations on data items. The system contains a mixture of real-time transactions and ordinary transactions. We assume that the ordinary transactions can be aborted, in case of a data conflict with the real-time transactions. An operation is either a read (returns the value of the data item), or a write (updates the item with a specified new value). For sake of simplicity, we assume that every transaction can read and write on any data item at most once. For any T_i and data item X, $r_i[X]$ denotes a read executed by T_i on X. Similarly, $w_i[X]$ denotes a write executed by T_i on X. The notation o_i denotes an operation of transaction T_i (i.e., either r_i or w_i). We let OS_i denote the set of all operations in T_i (i.e., $OS_i = \cup_j O_{ij}$). Where the notation, $O_{ij}[X]$ or O_{ij} denotes j-th operation o_i of transaction T_i on a data item X.

We denote by N_i the termination condition for T_i, where $N_i \in \{abort, commit\}$. In general, a transaction does not have to be a totally ordered sequence. When two operations are not ordered relative to each other, these can be executed in any order. However a read and a write on the same element, must be ordered.

Definition 1 : Two operations $o_i[X]$ and $o_j[X]$ conflict with each other, if they operate on data item (X) and at least one of them is a Write.
Definition 2 : A transaction T_i is a partial order $T_i = \{\sum_i, \ll_i\}$, where

1. $\sum_i = OS_i \cup \{N_i\}$.
2. For any two operations $O_{ij}, O_{ik} \in OS_i$, if $O_{ij} = r(X)$ and $O_{ik} = w(X)$ for any data item X, then either $O_{ij} \ll_i O_{ik}$ or $O_{ik} \ll_i O_{ij}$.
3. $\forall\ O_{ij} \in OS_i$, $O_{ij} \ll_i N_i$.

The items to be locked by the transaction for the purpose of read and write steps are termed as read-set (RS) and write-set (WS), respectively. The union of the read-set and the write-set of a T_i constitutes locking variables (LV_i). Our definition of a transaction conflict is the same as the commonly accepted notion [1]. Hence, $LV_i \cap LV_j = \phi$, implies that no conflict exists between $T_i,\ T_j$.

A transaction asks for an exclusive lock, if it reads and writes on a data item. Non-compatible locks are granted after completion of the preceding transaction. However, the compatible locks are granted in parallel. Local computation starts after the lock grants have been received. After the computation, the write phase is initiated. Updated data items are sent to the data sites and are stored in a temporary memory. It is assumed that, the 'two-phase commit' protocol [1] is employed to guarantee the atomicity of transactions that involve multiple sites. The real-time transactions are accorded priority and are always committed by local DMs, in preference over local computations.

Let $T = T_1, \ldots, T_n$ be a set of active transactions in a DDBS. The notion of correctness of transaction execution is that of serializability [1]. Hence, a transaction number (TN) assigned to a transaction for its identity has a 5 element value, as (site-id,local-clock,type,priority,global-identity).

5 Ordering of Transactions in a Distributed System

In the proposed approach, the transactions are ordered by constructing local access graphs (LAGs) for access requests. In this section, we define a LAG.

Definition 3 : A directed graph G consists of a set of vertices $V = V_1, V_2, \ldots$, a set of edges $E = E_1, E_2, \ldots$, and a mapping function Ψ that maps every edge on to some ordered pair of vertices $< V_i, V_j >$. A pair is ordered, if $< V_i, V_j >$ is different from $< V_j, V_i >$. A vertex is represented by a point, and an edge is represented by a line segment between V_i and V_j with an arrow directed from V_i to V_j.

Insertion of an edge $< V_i, V_j >$ into the graph G =(V,E) results in graph G' =(V',E'), where $V' = V \cup \{V_i, V_j\}$ and $E' = E \cup \{< V_i, V_j >\}$. The union of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is another graph G_3 (written as $G_3 = G_1 \cup G_2$), whose vertex set is $V_3 = V_1 \cup V_2$ and the edge set is $E_3 = E_1 \cup E_3$.

Let, there be a partial ordering relation $\ll_T$ defined over T (the collection of executing transactions $T_1, \ldots, T_n$), to indicate a precedence order among transactions, based on criteria of serializability.

Definition 4 : An access graph of T_i (AG_i) is a graph $AG_i(V, E)$, where $V \subseteq T$, and $E = \{< T_j, T_i > | LV_j \cap LV_i \neq \phi$ and $T_j \ll_T T_i\}$.

Example 2 : Consider the transactions RT_a, RT_b, RT_c, RT_d and RT_e as shown below. Let X,Y,Z be data items. Also,

- $RT_a = r_a(X)r_a(Y)w_a(X)w_a(Y)$.
- $RT_b = r_b(X)r_b(Y)r_b(Z)w_b(X)w_b(Y)w_b(Z)$.
- $RT_c = r_c(Z)w_c(Z)$.
- $RT_d = r_d(X)r_d(Z)w_d(X)w_d(Z)$.
- $RT_e = r_e(Y)r_e(Z)w_e(Y)w_e(Z)$.

Consider a situation, where X,Y and Z are located at one site. The execution of above transactions' operations can follow any one of the sequences in accordance with the criteria of serializability [1]. For each execution (equivalent to a serial execution), the AGs of transactions are different. If we consider the arrival pattern of transactions in the order RT_b, RT_c, RT_d and RT_e, RT_a, then, the corresponding AGs of above transactions are shown in Figure 4. Noting that the real-time transaction priority is highest for RT_a, and lowest for RT_e. In this, $RT_i \rightarrow^x RT_j$ indicates, RT_j is waiting for data item X which will be released after completion of RT_i.

Definition 5 : A local access graph (LAG) of T_i at S_k, is a graph $LAG_{ik}(V, E)$, where, $V \subseteq T$, and $E = \{< T_j, T_i > | LV_{jk} \cap LV_{ik} \neq \phi$ and $T_j \ll_T T_i\}$. In this expression, T_j has previously visited site S_k, and LV_{ik} denotes the part of LV_i, resident at S_k.

When a locking request LR_i (for RT_i) is sent to S_j, a LAG_{ij} is constructed at S_j.

$.RT_a$ z: free	$RT_a \longrightarrow^{x,y} RT_b$	$RT_b \longrightarrow^z RT_c$	$RT_a \longrightarrow^x RT_d$ $RT_b \longrightarrow^{x,z} RT_d$ $RT_c \longrightarrow^z RT_d$	$RT_a \longrightarrow^y RT_e$ $RT_b \longrightarrow^{y,z} RT_e$ $RT_c \longrightarrow^z RT_e$ $RT_d \longrightarrow^z RT_e$
1. AG_a	2. AG_b	3. AG_c	4. AG_d	5. AG_e

Fig. 2. Access Graphs (AGs) of transactions.

Observation 1 : Let LV_i be stored at sites $S_1, \ldots, S_m$. And, LAG_{ij} be the LAG of T_i at S_j. Then,

$$AG_i = \cup_{j=1}^{m} LAG_{ij}$$

6 An Algorithm to Construct LAG

In this section, we describe some of the terms used in the algorithm. The algorithm is presented in the following section.

6.1 Terms used to describe the Algorithm

- Home site ($\mathbf{SH_i}$) :
  The site of origin of T_i is referred to as the home site.
- Transaction number ($\mathbf{TN_i}$) :
  A unique number (TN_i), is assigned to the transaction T_i on its arrival at the home site. In this paper, both notations TN_i and T_i are used interchangeably and represent individual transactions. A real-time transaction is allotted a distinct identity indicating its type and priority.
- Locking variables ($\mathbf{LV_i}/\mathbf{LV_{ik}}$) :
  The items read or to be written by a T_i, constitute the LV_i. The locking variables at S_k constitute LV_{ik}.
- Lock request ($\mathbf{LR_{ik}}$) :
  It consists of TN_i and LV_{ik}. It is prepared by SH_i on arrival of T_i, and is sent to each concerned site S_k.
- **Odd edge, Even edge** :
  As per access ordering based on dataflow graphs, an edge $< T_j, T_i >$, such that $TN_j > TN_i$, is called as odd edge. It is treated as a negative priority edge as it needs to be interchanged (verified), upon its occurance.

  The even edges are also called priority edges. A real-time transaction always forms a priority edge (or even edge) with other transactions, due to its real-time priority.
- Access grant status ($\mathbf{AGS_{ik}}$) :
  It has values 0 or 1. After granting of all the requested locks of data items at S_k to LR_{ik}, the AGS_{ik} is changed to 1 at site S_k. Otherwise, AGS_{ik} is 0 for waiting transactions.
- Active list :
  The Active list is maintained by each S_k. The Active list of S_k is divided into two tables: active list of lock requests at S_k ($\mathbf{ALT_k}$), and active list of LAGs at S_k ($\mathbf{ALG_k}$). These tables are:
  - $ALT_k = \{(LV_i, AGS_{ik}) \mid T_i$ requested data items at $S_k \}$.
  - ALGk = { LAG_{ik} — T_i requested data items at S_k }.

  A transaction T_i is inserted into the ALT_{ik}, after initializing AGS_{ik}. On getting the access grants for LV_{ik}, AGS_{ik} is changed to 1. As a next step, these access grants are sent to SH_i.
- Data table ($\mathbf{DT_i}$) :
  This table is maintained at the SH_i for each T_i. The DT_i contains the lock grants (with values) of T_i. Whenever S_k receives any lock grant from another site, it stores it in a corresponding DT_i.
- Status of a transaction ($\mathbf{ST_i}$) :
  For a T_i, ST_i is maintained at the SH_i. It has values 0 or 1. Initially, ST_i is 0. After receiving all commitment messages in phase 1 of '2-phase commit', ST_i is changed to 1. After this, the final phase of commitment of T_i begins.
- Conflict-set of LAG ($\mathbf{LAG_{ik}.conflict-set}$) :
  Set of transactions in LAG_{ik} which are in conflict with T_i. That is,
  $LAG_{ik}.conflict - set = \{vertex - set[LAG_{ik}] - [T_i]\}$.

6.2 Informal description of the Algorithm

If a transaction needs to access data items, its LR_{ik} are sent to each (concerned) site S_k. The LAG_{ik} is updated at these sites. At any site S_k, if LAG_{ik} contains odd edge $< T_j, T_i >$, then it is an indication of possible blocking or delay. It is proposed that the real-time transactions exchange the precedence with the lock holding transaction, to generate a normal precedence (even edge). In all cases, the odd edge is nullified by exchange of precedence to revoke the grant. This is called the confirmation of the edge.

An **edge is confirmed** by checking the existing AGS_{jk} **locally**, or by sending an "abort the edge" message to the SH_j. That is, if the AGS_{jk} is 0, then an even edge $< T_i, T_j >$ is inserted into the LAG_{jk} and odd edge $< T_j, T_i >$ is deleted from LAG_{ik}. Otherwise, at the SH_j, if T_j is under execution, then a **reverse** grant message is sent to SH_j, and the odd edge $< T_j, T_i >$ is substituted at the LAG_{ik}. The executing transaction T_j performs a partial roll-back.

If T_i is a **real-time transaction**, but the local site is participating in a '2-phase commit' for T_j, the SH_j is sent a message $'T_j.STATE'$. In response, its home site terminates T_j, or communicates the updated values, within a time-out period. Else, the site is treated as a failed site.

If T_i is an **ordinary transaction**, but the local site is executing the 2nd phase of 2 phase locking for T_j, the SH_j is sent a message $'T_j.STATE'$. In response, its home site completes T_j, and communicates the updated values (within a larger time frame). Thus, a time-critical transaction is permitted to execute a revoke grant (if necessary) for the conflicting item, in order to cancel an odd edge.

7 Proof of Correctness

Theorem 1 : Let H be the history over T. And there are 'm' sites in the system. Then the execution produced by the algorithm is Serializable.
proof : In the algorithm, the conflicts are ordered through the LAGs. So, if $G_L = \cup LAG_{ik}$, where i=1 to n, and k= 1 to m, then, we prove the following.

(i) $G_L = SG(H)$.

(ii) G_L is acyclic.

(i) For any two transactions T_i and T_j, if these have a conflict between them, then - either $< T_i, T_j > \in LAG_{jk}$ or $< T_j, T_i > \in LAG_{ik}$ at some S_k. So, the edge set of G_L contains all conflicting pairs. So, $G_L = SG(H)$.
(ii) The algorithm follows two phase locking principle. That is, the data manager does not releases any lock until it acquires all required locks. Then, G_L is acyclic [1].
Theorem 2 : Let $T = \{T_1, T_2, \ldots, T_n\}$ be the set of conflicting transactions in the system. The above algorithm results in a deadlock free environment.
Proof : Every deadlock cycle results in the formation of at least one odd edge in some LAG_{ik} at some S_k. Suppose, T_i forms an odd edge $< T_k, T_i > \in LAG_{ip}$ at some S_p. In this case, in accordance with the proposed algorithm, if T_k has

not started completion of transaction commitment, the odd edge is removed, and corresponding even edge is inserted in the respective LAG, which breaks the deadlock cycle. Otherwise, if T_k has started completion and broadcast of committed values, then also the cycle will be broken.

Theorem 3 : Let $T = \{T_1, T_2, \ldots, T_n\}$ be the set of conflicting transactions in the system. The above algorithm results in a priority inversion free environment.

If all transactions access data items in accordance with the precedence order assigned to them, no priority inversion occurs. Priority inversion results from formation of an odd edge in some LAG_{ik} at some S_k. As shown above, odd edges are eliminated, as these occur.

8 Summary and Conclusions

In the distributed locking based approaches, if transactions from different sites, are in serializability conflict, then some of the submitted transactions are rejected. Transaction rejects and delays are two of the main problems that concern real-time transaction processing activity. In this study, a procedure has been identified that shows a possibility of execution of critical transactions under serializability conditions. A higher level of concurrency is achieved, as a result of removal of excessive blocking, and roll-backs for critical transactions.

References

1. P.A.Bernstein, V.hadzilacos and N.Goodman, "Two Phase Locking", Chapter 3, *Concurrency control and recovery in database systems*, Addison-Wesley,1987.
2. Bhalla, S. and S.E. Madnick, "Asynchronous Backup and Initialization of a Database Server for Replicated Database Systems", The Journal of Supercomputing, Vol. 27, No. 1, pp. 69-89, Kluwer Academic Publishers, January 2004.
3. Reddy, P.K., and S. Bhalla, "Asynchronous Operations in Distributed Concurrency Control", IEEE Transactions on Knowledge and Data Engineering, Vol. 15, No. 3 May-June 2003, pp. 721-733.
4. Bhalla, S., "The Performance of an Efficient Distributed Synchronization and Recovery Algorithm", The Journal of Supercomputing, Kluwer Academic Publishers, vol. 19, no. 2, June 2001, pp. 199-220.
5. A.N.Choudhary, " Cost of distributed deadlock detection: A performance study", *Proceedings of the Sixth International Conference on Data Engineering*, pp.174-181, February 1990.
6. Korth H.F., E. Levy, and A. Silberschatz, "Compensating Transactions: a New Recovery Paradigm," in *Proc. 16th Intl. Conf. Very Large Databases*, Brisbane, Australia, 1990, pp. 95-106.
7. Lam Kwok-wa, S.H. Son, V.C.S. Lee, and Sheung-Lun Hung, "Using Separate Algorithms to Process Read-Only Transactions in Real-Time Systems", *Proceedings of IEEE Real Time Systems Symposium*, Dec. 1998, pp. 50-59.
8. Pang, H., M.J. Carey, and M. Linvy, "Multiclass Query Scheduling in Real-time Database Systems", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 4, August 1995.
9. Ramamritham K., "Real-Time Databases", *Distributed and Parallel Databases*, Kluwer Academic Publishers, Boston, USA, Vol. 1, No. 1, 1993.