Programming and Computer Software, Vol. 26, No. 2, 2000, pp. 84–96.

Transaction Control in Real-Time DBMSs

E. Yu. Pavlova

St. Petersburg State University, Bibliotechnaya pl. 2, Petrodvorets, 198904 Russia
e-mail: katya@meta@math.spbu.ru

Received September 29, 1999

1. INTRODUCTION

In recent years, interest in the practical uses of real-time systems (RTS) has been growing rapidly. This has resulted in the development of a number of new systems and has given rise to new requirements imposed upon real-time systems. It is natural that these facts stimulated research in the field of real-time systems.

In the classical sense, a real-time system is one that is somehow related to real (rather than computer) time. Such systems often have to interact with processes that proceed in the external world. A typical example is a system that controls some physical device. As a rule, such systems consist of *controlling* and *controllable* systems. The controllable system is often considered as an environment, and the controlling system collects information on the environment by taking readings from various sensors. This information is used by the controlling system to perform certain actions; thus, it is essential that the state of the environment be perceived as accurate as possible. In the case of a large error, the actions of the controlling system can be inadequate.

At first glance, it would seem that the most important factor in real-time systems is the speed of the code execution. However, this is wrong. Even if the program that takes readings from a certain sensor is very fast, this does not guarantee that the information on the state of this sensor is promptly updated. Indeed, we cannot be sure that the application that is responsible for this sensor gets control over a given time interval. For this reason, the most important feature of real-time systems is *predictability* [16, 27].

Real-time systems often need to manipulate large amounts of shared data, which naturally leads to the use of DBMSs. However, traditional database management systems are of little use in real-time systems, since such systems impose new requirements that are not satisfied in traditional DBMSs [7, 11, 27, 33].

Information control systems and systems that control various process in real time provide typical examples of real-time systems that use DBMSs.

Information control systems illustrate one of the new requirements imposed by real-time systems upon DBMS, namely, the existence of deadlines for tasks to be performed. In contrast to systems for controlling processes in real time, temporal constraints in information control systems are less strict, tasks appear aperiodically, and queries on the database are more complex.

A good example of a real-time control system is an automatic pilot. For example, to calculate the direction of the flight, the system must take into account the current information on the wind force and direction. The more time elapsed from the moment of the wind data acquisition, the less reliable the results of the calculations. This example illustrates another requirement for DBMS used in real-time systems—they have to deal with outdating data.

These examples illustrate the two most important features of real-time DBMSs:

1. Every task must be completed within a certain time frame.

2. The data in the system must be kept up-to-date.

Methods used in classical DBMSs do not take these specific features into account; therefore, different methods must be developed to effectively incorporate DBMSs in real-time systems. Thus, DBMSs intended for use in real-time systems form a specific class of DBMSs called RTDBMSs.

New problems arise in integrity control, logging, and data restore [7, 29]. Another field of research related to real-time databases is data access control including buffer management [8, 38].

The standard SQL does not support temporal aspects, and by now, the development of the Real-Time SQL has been completed, although it has not yet been adopted as a standard.

Investigations are also carried out in such fields as database modeling and process algebras. In the second field, conventional process algebras used for studying parallel systems are extended to support temporal properties.

Performance of real-time DBMSs is another field of investigations. In addition to cost models, various requirements for predictability, such as multiple safety levels and QoS, are studied. The use of distributed databases in real time [19, 20, 31] and multimedia databases [34] are of independent interest.

In this paper, we consider only database consistency support in real-time systems.

0361-7688/00/2602-0084$25.00 © 2000 MAIK "Nauka/Interperiodica"

2. CONSISTENCY SUPPORT IN CLASSICAL DBMSS

An extensive body of literature has been devoted to database consistency support in the last 25 years. This field is beyond the scope of this paper; however, we give an outline of basic concepts and algorithms used in classical DBMSs to demonstrate the salient features of the methods used in real-time systems.

To provide the consistency of the database, the concept of a *transaction* is used in classical DBMSs. All operations with the database are performed by transactions only, which are controlled by a special algorithm that guarantees the integrity of the database.

Transaction management algorithms regulate the concurrent execution of several transactions; for this reason, they are called *transaction management protocols.* Such protocols are based on the assumption that each transaction possesses a certain set of properties. Different protocols use different transaction properties; the most widespread is the so-called ACID set of properties.[1]

A prerequisite of any transaction management protocol is to maintain the integrity of the database when several transactions are performed concurrently. The main property that guarantees the correct work of the protocol is the *serialization* property for the schedules obtained[2] [12].

The efficiency of the algorithms used for transaction management strongly affects the performance of the DBMS; thus, the efficiency of these algorithms is a crucial factor used to compare them.

All transaction management protocols are classified into two classes: *pessimistic* and *optimistic.*

2.1. Pessimistic Approach

The salient feature of the protocols of this class is preventing the very possibility of conflicts. The main method for achieving this goal is the locking mechanism. This method requires that any transaction must lock every resource before using it. The transaction that holds the lock can be sure that no conflict can arise with other transactions that require the same resource. Transactions that can potentially cause a conflict are simply not given the lock. In this case, such a transaction is queued up for the resource and continues executing only after the lock has been obtained.

One of the most widespread pessimistic protocols is the *two-phase locking protocol* (2PL) [1]. According to this protocol, all locking operations must precede all unlocking operations. In the classical version of this protocol, two types of locks are used: read locks and write locks.

A serious problem that can arise when using a pessimistic protocol is the possibility of deadlocks. Deadlock occurs when each transaction from a set of transactions T waits for the right to lock an element that is locked by another transaction from this set [1]. Transactions that are in deadlock cannot leave it in a natural way and can wait forever. For detecting and eliminating deadlocks, special methods are used.

2.2. Optimistic Approach

The main idea of optimistic protocols is maximizing the amount of the work done. These protocols allow all transactions to be executed, but only those of them can be completed successfully that do not violate the consistency of the database.

From the standpoint of the transaction manager, the execution of a transaction is divided into three steps: *reading, validation,* and *writing.* During the reading step, the transaction executes concurrently with other transactions without any restrictions; however, all modified data are written to the private memory of the transaction rather than to the database. When the transaction completes, the transaction manager initiates the validation step. At this step, conflicts with other transactions are detected. If no conflicts have been detected, the changes are made to the database and become available for other transactions. Otherwise, the transaction in question or conflicting transactions[3] are completed abnormally (i.e., all the changes are ignored and not made to the database).

Depending on the type of the validation, the optimistic protocols are divided into forward validation [17] and backward validation [24] protocols. The difference between them is in the choice of the set of transactions that are validated for potential conflicts when the transaction T_0 is completed. Protocols of the first group test all uncompleted transactions and abnormally terminate either those that conflict with T_0 or terminate T_0 itself. Backward validation protocols test all transactions that were completed normally and abnormally terminate T_0 in the case of conflicts.

2.3. Comparison of the Approaches

The principal disadvantage of the pessimistic approach is the idle time during deadlocks and the waste of computer resources for detecting and eliminating deadlocks.

Optimistic protocols waste resources executing transactions that will be abnormally terminated later due to conflicts and on consistency validation at the final stage of every transaction.[4]

[1] ACID stands for atomicity, consistency, isolation, and durability.

[2] The schedule of transactions is the order in which the transactions are executed. A schedule is called *serializable* if there exists a schedule according to which the transactions are executed strictly one after the other and the result of the execution coincides with the result of the initial schedule.

[3] In this case, the behavior depends on a concrete transaction management protocol.

[4] When there is a large number of short transactions, the validation step can take a significant part of the time when the transaction stays in the system.

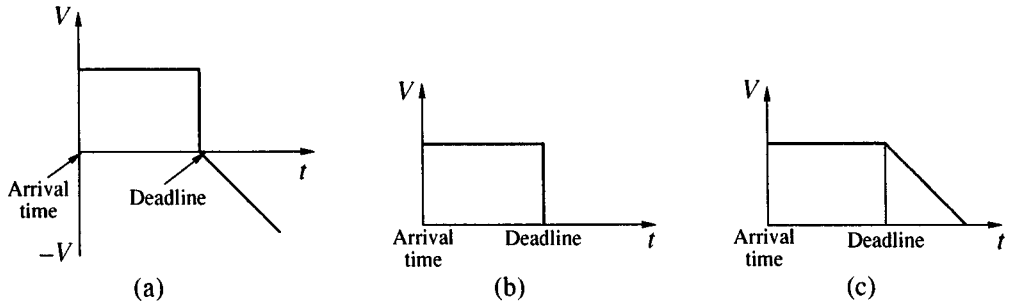

Fig. 1. Utility function in systems with soft, firm, and hard deadlines. (a) Hard deadline, (b) firm deadline, and (c) soft deadline.

Experiments show that in classical DBMSs, pessimistic protocols perform better [5, 9]. For this reason, the greater part of commercial DBMSs use various pessimistic protocols.

3. TRANSACTION MANAGEMENT PROTOCOLS IN REAL-TIME DBMSS

In a real-time system, in contrast to classical DBMSs, every transaction is assigned a deadline, i.e., a time frame by which this transaction must be completed.

Depending on the type of deadlines used, RTDBMSs can be classified into three groups: RTDBMSs with *hard, firm*, and *soft* deadlines [27]. In systems with hard deadlines, any delay is a disaster; in systems with firm and soft deadlines, delays only decrease the performance of the system. The difference between the firm and soft deadlines is as follows: in a system with firm deadlines, the transaction that missed its deadline is aborted from the system, whereas in a system with soft deadlines, only its priority is decreased, but it still can be completed.

The difference between the three types of deadlines can be formally described in terms of the utility function depending on the completion time. This function is depicted in Fig. 1 for all three types of deadlines.

When the deadline in a system with hard deadlines expires, the transaction becomes *harmful*; i.e., it has a negative utility. If the deadline is firm, the transaction becomes useless, rather than harmful, as the deadline expires. In systems with soft deadlines, the utility of overdue transactions decreases gradually as the delay time increases.

Another important difference from the classical DBMSs is the use of different performance indicators. In the classical DBMSs, performance is usually measured as the average number of transactions completed in a unit time; in real-time systems, different indicators are more important, such as the number of transactions that missed their deadlines, the average delay in transaction execution, and the like. Performance is measured by the means of a combination of such parameters.

Methods for calculating these parameters are different for different types of real-time DBMSs. For example, in systems with strict deadlines, the average delay time is irrelevant, whereas in systems with soft deadlines, this characteristic is rather useful [14].

The difference of performance criteria causes a low performance in real-time DBMSs that use transaction management protocols developed for classical DBMSs. Thus, new protocols that take into account specific requirements of real-time systems must be developed.

When resolving conflicts, protocols designed for real-time DBMSs must use priorities of transactions based on their deadlines in order to ensure that low-priority transactions do not block high-priority ones. This requirement is entirely ignored in classical DBMSs, which is one of the main causes of their low performance.

Over the last ten years, a number of specific protocols with improved characteristics have been suggested for real-time systems. There are variations of classical pessimistic and optimistic protocols among them [14, 24, 35–37], as well as relatively new protocols specially designed for real-time systems [6, 15, 18, 30].

3.1. Pessimistic Approach

The most widespread pessimistic protocols for real-time databases are modifications of the two-phase locking protocol (2PL). In this section, we briefly review some of them.

2PL-HP (High Priority). The basic idea of this protocol [2] is to resolve all conflicts in favor of the transactions with higher priority. When a transaction T requests a lock for a data element d, two different situations may occur:

(i) if d is free, then T obtains the lock for it;

(ii) if d is locked, then

• if T requests a lock for d that does not conflict with the locks held by other transactions, it obtains this lock only if the priority of T is higher than the priority of all other transactions that are awaiting the lock for d;

• if the priority of T is higher than the priority of all other transactions that hold locks for d, all of them are terminated, and T obtains the lock;

• otherwise, T is placed in the queue for the lock.

Note that this protocol ensures that no deadlocks can occur.

2PL-WP (Wait Promote). This modification of the 2PL protocol is based on the idea of priority inheritance [3]. When a transaction T with a higher priority requests a lock for a resource d that is blocked by another transaction with a lower priority, then T is placed in the queue (as in 2PL). However, the transaction that is holding the lock for d is assigned the priority equal to that of T. As a result, this transaction can be executed more quickly and the transaction T can hope to obtain the lock for d sooner.

The use of this method can easily lead to a situation (due to recursive increase of priorities) when the greater part of (or even all) transactions have the same priority. In this case, the behavior of this protocol differs little from that of the classical 2PL protocol.

An example of the work of these two protocols is presented in Fig. 2.

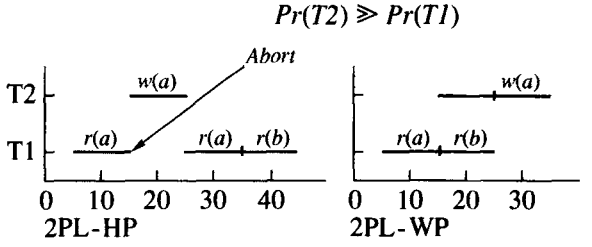

Fig. 2. The 2PL-HP and 2PL-WP Protocols.

3.2. Optimistic Approach

To improve the performance of a real-time DBMS, conflicts between transactions must be resolved in favor of the transactions with higher priority, which is impossible with *backward validation* optimistic protocols. Indeed, in this case, conflicts occur with transactions that have already been completed. For this reason, the greater part of optimistic protocols used in real-time DBMSs belong to the *forward validation* class.[5]

Let us consider two examples of optimistic protocols for real-time systems.

OPT-SACRIFICE. This protocol [13] is a version of the protocol OCC-FV (Optimistic Concurrency Control with Forward Validation) adapted to the requirements of real-time DBMSs. According to OPT-SACRIFICE, the transaction T_0 that reached the validation stage terminates if at least one of the conflicting transactions has a higher priority. Otherwise, T_0 is completed successfully, and all conflicting transactions terminate abnormally. Thus, the transaction being validated sacrifices itself for the executing transaction T_1 of a higher priority.

This protocol has a number of disadvantages. First, termination of the transaction T_0 means that the work and resources spent on its execution were wasted. Besides, since T_1 can also be terminated abnormally (for its own reasons or as a sacrifice for another transaction with a higher priority), the sacrifice of T_0 can prove useless.

OPT-WAIT. This protocol also belongs to the class of *forward validation* protocols. According to this protocol, a transaction T that reached the validation step and detected a set of conflicting transaction T' behaves as follows:

(i) if the priority of T is higher than the priority of all transactions $T_i \in T'$ that conflict with it, then T completes normally, while all T_i terminate abnormally;

(ii) otherwise, T waits for the completion of all $T_i \in T'$ that have a higher priority; then, it behaves as follows:

• in the case when a higher priority transaction from T' completes normally, T is terminated abnormally;

• if none of the higher priority transaction from T' completes normally, then T_i completes itself normally.

This protocol, in contrast to OPT-SACRIFICE, does not suffer from a large number of useless restarts since all of them are necessary. However, it also has its weak points. The blocking effect that occurs as a result the transaction holdup at the validation stage (instead of immediate completion or termination) results in the conservation of resources and, thus, leads to the same problems that are characteristic of pessimistic protocols in real-time systems. Besides, even in the case when the blocked transaction T is completed successfully, it can cause a large number of lower priority transactions (both executing and blocked) to terminate abnormally. Moreover, the number of abnormally terminated transactions can increase because this number includes low-priority transactions that conflicted with T, but not with higher priority transactions. This number can increase significantly depending on the delay time of T.

An example of the work of two optimistic protocols is presented in Fig. 3.

3.3. Comparison of the Approaches

Conventionally transaction management protocols are distinguished by two factors: the time of detecting conflicts and the method used for this purpose. In terms of these two factors, optimistic and pessimistic protocols constitute extreme cases.

Pessimistic protocols try to detect conflicts even before they emerge and resolve them using the locking mechanism. Optimistic protocols detect conflicts when transactions complete and resolve the conflicts detected by restarting a part of the conflicting transactions.

[5] Although the greater part of optimistic protocols for RTDBMSs belongs to the forward validation class, backward validation protocols are also encountered (see, e.g., [24]).

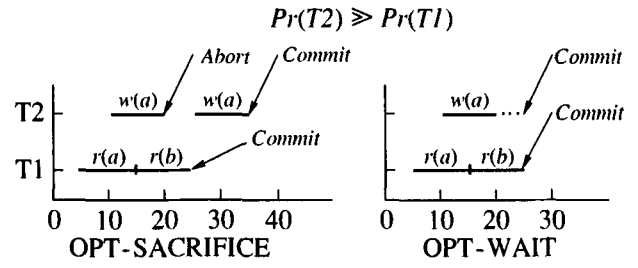

Fig. 3. OPT-SACRIFICE and OPT-WAIT Protocols.

Both methods for resolving conflicts have certain drawbacks. Blocking conserves resources, while restarts waste the resources.

In real-time systems, the transaction management protocol has to deal with a large number of abnormally terminating transactions. While this is quite a normal situation for optimistic protocols, the large number of restarts is a problem for pessimistic protocols.

Another weak point of pessimistic protocols is the idle time of transactions while they wait for the right to lock a required resource. The idle time increases the probability of missing the deadline. In addition, the transaction that holds the lock for the required resource can also miss its deadline, either because it is too near or because it may have to wait for the right to lock other resources. Thus, it is quite possible that no transactions will be executed by their respective deadlines.

Optimistic protocols do not lock resources and do not cause transactions to stay idle,[6] which is certainly a plus for real-time systems. The reverse of the medal is the waste of resources on the parallel execution of conflicting transactions, only part of which can be completed normally. In addition, it should be noted that the validation stage can be labor-intensive, which is especially significant when the transactions are very short.

The comparison analysis of these two approaches for a model of simple transactions in real-time DBMSs with soft and firm deadlines with both a limited and unlimited number of resources shows that, in most cases, the optimistic approach performs better than the pessimistic one [14, 36]. However, because of the large number of various real-time systems with specific requirements, no protocol exists that is ideal for all cases. Sometimes, the pessimistic approach outperforms the optimistic one.

4. SYSTEMS WITH OUTDATING DATA

In the preceding section, we considered systems in which temporal constraints are imposed only on transactions. However, in real-time systems, the data themselves can be of temporal nature [40, 42]. For example, the automatic pilot needs to know information on the wind force and direction, and this information changes continuously and becomes outdated very quickly. The necessity of working with outdating data causes additional problems.

In real-time DBMSs, the data are classified into two types: discrete and continuous. Discrete data are conventional data in the sense of classical DBMSs; these data are not related to real time.

Continuous data are related to external objects whose state is continuously changing. It is clear that the state of a continuous object is represented in the database as a discrete magnitude. This magnitude represents the state of the object at a certain moment in the past and can differ more and more from the real state of the object as time goes on. For example, if hour-old data are used to calculate the course of an aircraft, we can hardly expect the results to be useful.

From the formal point of view, a continuous object can be described by the triple $(d_{value}, d_{timestamp}, d_{avi})$, where d_{value} is the state of the object at the time moment $d_{timestamp}$ and d_{avi} is the time interval during which this state can be considered correct. This time interval is called the absolute validity interval of the data element.

Not only can the elements that are directly related to external objects be continuous; but those with values depending on continuous objects should be considered continuous as well. Such objects are called *basic* and *derived*, respectively.

All changes in the database are performed by transactions. This is also true for continuous data elements. Transactions are classified into three categories, depending on the type of the data changed:

(i) *Sensor* transactions modify basic objects. Such transactions usually execute periodically and do not deal with discrete data elements.

(ii) *Derived* transactions modify derived objects. Such transactions are invoked as a result of changes made to the corresponding basic continuous objects.

(iii) *User* transactions are all other transactions. They cannot modify any continuous objects, but can read them.

As a rule, sensor and derived transactions are assigned firm deadlines. Indeed, this is quite natural: on the one hand, it is useless to complete the task of modifying an object state knowing that this state has already become invalid; on the other hand, if an object state cannot be modified on time, this is not fatal.[7]

4.1. Correctness of Transactions

In the preceding section, a transaction was considered correct if it did not violate the consistency of the database, did not have conflicts with other transactions, and its deadline was not missed. However, in cases

[6] Note that the delay of transactions in the OPT-WAIT protocol is not the idle time (because the transaction has already finished executing); this is only the decision making on whether the transaction completion should be considered normal.

[7] The first case corresponds to *soft* deadlines and the second to *hard* deadlines.

when temporal characteristics of the data are taken into account, this is not sufficient. For example, the transaction that calculates the heading of an aircraft cannot be considered correct if it is based on hour-old information about the wind. Thus, temporal characteristics of the data must be taken into account when determining whether a transaction is correct or not.

As was noted above, every value of a continuous data element is assigned an interval of absolute validity. It may seem that the additional requirement that the data used in the transaction be absolutely correct for a certain moment in time is sufficient to obtain an adequate definition of the transaction correctness; however, this is not the case.

The interval of absolute validity of the data element only guarantees that the deviation of the value stored in the database from the true value does not exceed a certain margin of error during this time interval. Since the error can accumulate in the course of calculations, we must impose stricter requirements upon the errors of the data in order to ensure a reasonably small resulting error. It is very important to take into account the interrelation of the time moments at which the modification of the corresponding data elements was made. For example, the information on the left engine can differ from the information on the right engine taken a minute later, although both engines operate in the same mode. Thus, the results obtained can be incorrect.

To avoid such problems, an additional condition is imposed upon the data; this condition is called the *relative consistency condition.* It is defined as follows.

For every set of continuous data objects R, the length of the relative validity interval R_{rvi} is specified.[8] The set R is called *relatively consistent* if the following condition is satisfied:

$$\forall d1, d2 \in R,$$
$$|d1_{timestamp} - d2_{timestamp}| \le R_{rvi}. \qquad (*)$$

Now we can define the notion of the correctness of the set of continuous data. The state of the set of continuous data objects R is correct at the time moment t ($t > 0$) if the following conditions are satisfied:

1. $\forall d \in R$, d_{value} is logically consistent, i.e., satisfies all consistency constraints;

2. R is temporally consistent; that is

(a) all elements d from R are absolutely consistent at the time moment t; i.e.,

$$\forall d \in R, \quad (t - d_{timestamp}) \le d_{avi};$$

(b) R is relatively consistent; i.e., relation (*) holds.

Now we are ready to describe a transaction correctness criterion with regard for the constraints imposed on the data used.[9] The transaction can be completed successfully if

(1) it is logically consistent;

(2) its deadline has not been missed;

(3) it used a temporally consistent set of continuous data (i.e., both absolutely and relatively consistent) and these data remain up-to-date at the moment of the completion of the transaction.

Systems with outdating data have much in common with real-time systems that use only discrete data. In particular, they can use the same transaction management protocols, provided that a special criterion for the correctness of transactions is used.

However, the practical implementation of the temporal consistency maintenance raises some new questions that will be considered in the following sections

4.2. Choosing the Period for Sensor Transactions

Sensor transactions are executed periodically and maintain the absolute consistency of the basic continuous objects. If such transactions are executed too rarely, a time interval may exist when the basic data elements that are modified by this transaction are not absolutely valid; thus, they cannot be used by any transaction.[10] On the other hand, if sensor transactions are executed too often, the system can be overloaded.

Thus, we have the problem of choosing a period P of the sensor transaction modifying the value of the element d such as to ensure that d be in the absolutely valid state.[11] Let a transaction require time ϵ ($0 < \epsilon < P$) to be executed. The period of the sensor transaction guarantees that this transaction will be executed during this time interval; however, it does not specify the moment when the execution starts. Consider the worst case scenario when a sensor transaction starts at the moment t (and completes at $t + \epsilon$), and the next transaction starts at $t + 2P - \epsilon$ (and is completed at $t + 2P$). In this case, the value of d remains the same over the time interval $[t + \epsilon, t + 2P)$. Therefore, in order for this value to be absolutely valid over this interval, it is necessary that $P \le d_{avi}/2$.

4.3. Choosing the Version of a Continuous Object

By the time the next sensor transaction is completed, the previous value of the data element can be still absolutely valid. In this case, the previous value is not discarded, but forms a separate version of this data element. Formally, this is a version of the data element that the triple (d_{value}, $d_{timestamp}$, d_{avi}) describes; several

[8] Informally, R_{rvi} can be interpreted as the length of the time interval over which the resultant error remains within the tolerance range.

[9] This criterion is chiefly applied to *user* transactions. For *sensor* and *derived* transactions, coarser criteria are often sufficient.

[10] Transactions that use data that are not absolutely valid have no chance of being completed successfully.

[11] Here, we assume that any sensor transaction completes successfully. In fact, this is not the case, and such transactions are often sacrificed for the successful execution of user transactions.

such versions may exist for one and the same data element.

The existence of several versions of one and the same data element may seem to be redundant. At first glance, it seems that it is always best to use the most recent version of the data element; however, this is not the case.

Consider the following example. Let the set of continuous data consist of two objects a (versions $(a_0, 0, 20)$ and $(a_1, 10, 20)$) and b (version $(b_0, 0, 20)$). Let the relative validity interval for this set be $R_{rvi} = 9$. Let the transaction T start at the moment $t = 10$, be completed by $t = 20$, and use both a and b. For the period of the execution of T, there exist two correct versions of a: a_0 and a_1. However, if the transaction uses a_1, it would not be able to be completed, since the data used would not be relatively valid. On the other hand, if the earlier value a_0 is used, the relative validity criterion holds.

Thus, we face the problem of choosing the version of the data element x to be used in every particular case. Unfortunately, this problem has no universal solution. One approach consists in using the most recent version that does not violate the relative consistency of the data set used.[12] Another variant is to declare elements used by the transaction and choose versions that do not violate their relative consistency. However, neither of these methods guarantees the temporal consistency.

4.4. How to Control the System Overload due to Large Number of Sensor Transactions?

The large number of sensor transactions can create a significant extra load in the system, which can affect the quality of processing user transactions. This problem becomes especially pressing when a large number of sensor transactions must be executed concurrently. This situation is very probable even if the transactions have different periods.

In this case, the strategy of "updating by necessity" may be useful. According to this strategy, user transactions basically have higher priorities as compared to sensor transactions. If a user transaction needs to use a continuous data element that has no correct version at this moment, the corresponding sensor transaction is temporarily assigned the priority of the user transaction that requires this element. As a result, the data element would be updated and the user transactions would be able to continue executing.

Note that this approach also has its weaknesses. In particular, the execution of a transaction that uses a large number of continuous data elements can be delayed for a long time due to frequent waits for the correct version of objects. As a result, it may be unable to be completed correctly, either because it misses its deadline or because the data set used becomes relatively invalid.

4.5. When to Update Derived Objects?

Since the derived data elements are updated by derived rather than by sensor transactions, choosing the strategy for the execution of these transactions presents a separate problem.

Variation of the basic object can cause modifications in several derived objects. Besides, the value of the derived object may depend on several basic objects. Therefore, direct modification of the derived object d every time when the basic objects are updated can result in a waste of resources. On the other hand, if a derived element is updated too rarely, there arises the possibility that there would be invalid value of this element at a certain moment. The strategy of "updating by necessity" provides a reasonable solution. One drawback of this strategy is the delayed updates. In paper [4], another solution was proposed: the state of any derived object cannot be updated more often than in specified time intervals, which decreases the load.

Updating the state of one basic object may cause the modification of several derived objects. Therefore, we have two alternatives: either each modification is done by a separate transaction or all modifications are done within one transaction. A drawback of the first method is the large number of transactions and, consequently, the increased load on the transaction manager. On the other hand, locked objects are freed more quickly than under the second method. To decrease the locking time of data elements under the second method, the locks may be removed immediately after the corresponding methods are updated, provided that the modified objects are independent of each other. In other words, the semantics of such transactions can be used to decrease their *atomicity* and *isolation* while retaining these properties for every individual operation of updating the state of the derived object.

4.6. How to Decrease the Number of Abnormal Terminations?

Problems with the validity of the set of the data used can be the cause of abnormal termination of a large number of transactions. Even if the required continuous data were correct at the moment when they were read by the transaction, they can become invalid by the end of this transaction. This difficulty cannot be overcome altogether; however, we can try to improve the situation by making additional checks at the stage of choosing a version of the continuous object.

Unfortunately, we cannot know in advance the exact amount of time that the transaction execution will take; however, we can often evaluate the minimum execution time. In this case, the version of the object is chosen with regard to the additional requirement that the entire set of data used in the transaction must remain correct

[12] Thus, the most recent version is chosen for the first continuous object that is used in the transaction.

at least over this minimal time interval. In the case when an update is required to obtain such a version of this element, the user transaction is delayed for the corresponding time period. If such a version cannot be obtained, the transaction has no chance of being completed successfully, and it is terminated abnormally.

Note that the approach described avoids doing work that is known to be useless; however, it can result in useless delays in certain transactions.

4.7. Dispatching Transactions

In systems with outdating data, transactions can be classified into two classes, depending on their importance: user transactions, which are the most important, and others (i.e., sensor and derived transactions). It is clear that the most important task is the execution of user transactions, and the main task of the transaction management algorithm, is the maximization of the number of successfully completed user transactions.

The priority mechanism is used to take into account the importance of transactions. Transactions with a higher priority have advantage over low-priority transactions; i.e., they are executed first. To increase the chances of the most important transactions being successfully completed, they can be assigned a higher priority.

Various strategies of assigning priorities have been studied, both for different transaction classes and within the same class, [32, 41, 40]. By way of example, we consider several strategies of assigning priorities to different transaction classes.

1. Both classes are assigned the same priority.
2. User transactions are assigned a higher priority.
3. User transactions are assigned a lower priority.
4. User transactions are assigned a higher priority; however, if such a transaction has to wait for updating of a piece of continuous data, the corresponding sensor (or derived) transaction is temporarily assigned an increased priority in order to complete it as soon as possible.

For the transaction to be completed correctly, it is necessary that all data used by it be temporally consistent. In this connection, the concept of the *data deadline* may be useful. The *data deadline* of a transaction is the moment when the temporal consistency of the data used begins to be violated. It is clear that the further execution of such a transaction becomes useless, since it would not be able to be completed successfully. Thus, the information on data deadlines of transactions can help improve the performance of the system.

The data deadline of the transaction T can be calculated by the following algorithm.

- **Start of the transaction** T

$$data\_deadline_T = deadline_T.$$

- **Reading the object** $((x, x_{timestamp}, x_{avi})$

$$data\_deadline_T = \min\begin{cases} data\_deadline_T \\ x_{timestamp} + x_{avi}). \end{cases}$$

To put data deadlines to good use, this information may be used to assign priorities to transactions. In the general form, a simple formula for evaluating the transaction priority is as follows ($0 < \alpha < 1$):

$$priority_T = \alpha * data\_deadline_T + (1 - \alpha) * deadline_T.$$

Depending on α, the following basic strategies for transaction distribution can be distinguished.

- **By the nearest deadline:**

$\alpha = 0$.

In this case, priorities are assigned on the basis of the deadlines only, while information on the constraints on temporal consistency is neglected. As a result, a transaction can be terminated because of the violation of temporal consistency.

- **By the nearest data deadline:**

$\alpha = 1$.

This algorithm takes into account only data deadlines of transactions. The transaction T_1 with a distant deadline and near *data deadline* gets a higher priority than the transaction T_2 with a nearer deadline. Therefore, it is possible that T_2 will not have enough time before its deadline to be completed. However, if we assign a higher priority to T_2, then both T_1 and T_2 have chances to be completed successfully.

- **Hybrid approach:**

$$\alpha = \begin{cases} 0, \text{ the percentage of the work done is } \le 0.5 \\ 1, \text{ the percentage of the work done is } > 0.5. \end{cases}$$

This algorithm is a hybrid version of the first two approaches. During the first half of the time of the transaction execution, its priority is determined by its deadline; in the remaining time, the priority is determined by the data deadline.

- **Proportionally to the work done:** α is the percentage of the work done by the transaction. This algorithm implements an intermediate approach. The dependence of the transaction priority on the data deadline increases with the increase in the volume of the work done.

Note that the two last approaches can be used only if we know the amount of work required to execute every transaction or, at least, know how to approximate this amount.

5. OPTIMIZATION FOR A PARTICULAR SYSTEM

Certain real-time systems often have very stringent requirements for the performance of transaction man-

agement algorithms, and generalized approaches are not well suited for them. On the other hand, such systems often have specific features that make it possible to develop a specific transaction management protocol for this particular system that provides enhanced performance. This is usually achieved by optimizing the performance of a class of the most important transactions at the expense of all others.

One example of this approach is the classification of transactions into sensor, derived, and user transactions, and assigning a higher priority to user transactions, which are considered more important.

5.1. Classification of User Transactions

Generalizing what was said in the preceding sections, we reveal the following attributes of user transactions in real-time databases:

(1) the time of arrival in the system;

(2) period;

(3) temporal constraints;

(4) priority;

(5) approximate execution time;

(6) necessary data;

(7) utility function.

Depending on the existence and values of these attributes, transactions can be classified by the following criteria:

- type of the deadline (soft, firm, or hard);
- type of arrival in the system (periodic or aperiodic);
- type of data access, which may be predefined (read only, write only, or updating) or arbitrary (i.e., not known in advance);
- are the necessary data known or not?
- is the approximate execution time known or not?
- type of the data used (continuous, discrete, or both).

In the general case, a variety of transaction classes exist. However, for every particular case, only a few of them are often sufficient. There is no general rule for choosing the best set of classes; however, an answer to the question "What are the most important transactions and what do they have in common?" is a good starting point.

5.2. Real-Time Main Memory DBMSs

In this section, we apply the idea of optimization of certain classes of transactions to a specialized real-time DBMS.

The system in question is the database residing in the main memory and used in telephone networks. Main memory databases offer very rapid data access, since they do not need to access slow permanent storage devices. For this reason, this approach is often used in systems with high performance requirements.

Since the access to data residing in the main memory is much faster than the access to data stored in the secondary memory, the cost of controlling integrity becomes very high.

In our case, the system has very tight transaction deadlines, which makes the use of generalized algorithms impossible. Fortunately, this system has another salient feature: the overwhelming majority of transactions are very short, these transactions only read data, they are the most important, and have the tightest deadlines.

The main idea of the optimization is to decrease the cost of processing short and read-only transactions. At the expense of the performance of update transactions, an algorithm can be developed that reduces the expenses of executing read-only transactions to almost zero[13] [21].

6. USING COMPLEX TRANSACTION MODELS

In the preceding sections, we used the simple transaction model in which transactions are independent of each other. However, more complex transaction models are also used in real-time systems, e.g., the nested model [10, 11, 22].

By way of example, we consider the use of active databases in real-time systems [10, 11, 39].

6.1. Active Databases

In contrast to conventional DBMSs, active database management systems can perform certain operations when certain events occur. Formally, such systems contain sets of rules of the form *event—condition—action.* In the case when the event specified occurs and the corresponding condition is satisfied, the action specified is performed.

Examples of such events are COMMIT or ABORT of a transaction, data access operations, or external events. The condition is usually a predicate depending on the current state of the database. The action is a transaction that is executed in response to a situation, where the situation is a combination of the event and the condition described above.

The transaction that initiates the execution of a rule is called *triggering*; the transaction that is executed according to this rule is called *triggered.* The concepts of the triggering and triggered transactions are not mutually exclusive; i.e., a triggered transaction can, in turn, triggering another transaction. The transaction that has triggering at least one transaction is called

[13] When this protocol is used, the probability of update transactions to complete is significantly reduced if several of them are executing concurrently. However, if update transactions are relatively rare, the overall performance of the system becomes higher.

active or *parent*. The active transaction is associated with a set of the corresponding triggering transactions.

Various types of relationships between the triggering and triggering transactions can be distinguished. By the type of the relationship, triggered transactions are most often classified into *immediate, deferred*, and *independent. Immediate* and *deferred* transactions are executed as a part of the active transaction; i.e., for this transaction to be completed successfully, all immediate and *deferred* transactions triggered by it must be completed successfully. Independent transactions are executed independently; even if such a transaction is unsuccessfully completed, this does not affect the triggering transaction. The *immediate* transaction begins executing right after it is generated, while the deferred transaction starts executing after the triggering transaction has done its work, but before the corresponding changes has been fixed in the database.

6.2. Real-Time Active Databases

In real-time systems, transactions have deadlines. The possibility of triggering transactions causes additional problems. Indeed, in order to complete a transaction, all immediate and deferred transactions triggered by it must be completed before the deadline expires.

One problem consists in assigning priorities to subtransactions [28]. Since a transaction can trigger other transactions (immediate or deferred) the amount of work to be done increases. Therefore, such a transaction has fewer chances of being completed successfully than one that does not trigger any other transactions. Keeping track of this information is another problem.

To analyze these issues, we consider a model of the active real-time database in which only immediate or deferred transactions can be triggered and, in addition, we assume that triggered transactions cannot trigger other transactions. We also assume that transactions have firm deadlines; i.e., a transaction that missed its deadline is terminated, together with all transactions that it triggered.

6.2.1. Transaction attributes. We define the following attributes for the transaction T:

$a(T)$ is the arrival time;

$s(T)$ is the starting time;

$d(T)$ is the deadline;

$n_t^{def}(T)$ is the number of transactions triggered deferred at the moment t;

$P_t(T)$ is the priority at the moment t.

Some attributes may vary in the course of the execution of the transaction. These attributes are labeled by the subscript t indicating the value of this attribute at the moment t.

Methods for assigning priority, which we describe further, use the following estimates.

$X_t(T)$ is the remaining execution time;

$C_t(T)$ is the expected completion time

$$(C_t(T) = t + X_t(T))$$

$S_t(T)$ is the time reserve

$$(S_t(T) = d(T) - C_t(T))$$

$m_t^{imm}(T)$ is the number of immediate transactions triggered by T after the time moment t;

$m_t^{def}(T)$ is the number of deferred transactions triggered by T after the time moment t;

$\bar{X}^{imm}(T)$ is the average execution time of the immediate transaction triggered by T;

$\bar{X}^{def}(T)$ is the average execution time of the deferred transaction triggered by T;

$X_t^{avg}(T)$ is the average remaining execution time of the transaction T with all its subtransactions triggered by the moment t taken into account: $X_t^{avg}(T) = X_t(T) + n_t^{def}(T) * \bar{X}^{def}(T) + m_t^{def}(T) * \bar{X}^{def}(T) + m_t^{imm}(T) * \bar{X}^{imm}(T)$;

$S_t^{avg}(T)$ is the average time reserve at the moment t with all subtransactions of T taken into account: $S_t^{avg}(T) = d(T) - t - X_t^{avg}(T)$.

We note that the minimum value of $P_t(T)$ is equivalent to the largest possible priority. In all methods described below, the deadline of any subtransaction is set equal to the deadline of the triggering transaction; i.e., $d(T^{sub}) = d(T)$.

6.2.2. Priority of the immediate transaction. In this subsection, we consider some strategies for assigning priorities to immediate subtransactions.

PD method. According to this method, the immediate transaction is assigned the priority equal to the deadline of the parent transaction. Moreover, the priority of the parent transaction does not change after a subtransaction is triggered. That is, if T_i^{imm} is the ith immediate transaction triggered by T at the time moment t, then $P_t(T_i^{imm}) = d(T)$.

DIV method. Consider a transaction T that triggers an immediate transaction T_i^{imm} at the time moment t. Then

$$P_t(T_i^{imm}) = C_t(T_i^{imm}) + \frac{S_t(T) - \left(X_t(T_i^{imm}) + \sum_{j=1}^{n_t^{def}(T)} X_t(T_j^{def})\right)}{n_t^{def}(T) + 2}.$$

Let the transaction T_i^{imm} complete at the moment $w > t$. Then, the priority of the parent transaction is set to

$$P_w(T) = P_t(T) - (w - t).$$

According to this strategy, the estimated free time of the parent transaction is evenly divided between the triggered and parent transactions. This number, plus the estimated completion time of the subtransaction, determines the priority of this subtransaction. The priority of the parent transaction is dynamically adjusted to take into account the work already done.

This strategy uses estimates of completion time for subtransactions that have already been triggered and does not use any knowledge of transactions that can be triggered later. The priority assigned to the subtransaction can be interpreted as a virtual deadline equal to the sum of completion time of this subtransaction and the time reserve obtained from the triggering transaction.

SL method. According to this method, the remaining average time reserve of the parent transaction, $S_t^{avg}(T)$, evaluated at the potential moment of a transaction triggering is used as the priority of both the triggering and triggered transactions; that is, both the parent and triggered transactions are assigned equal priorities.

For example, let the transaction T start at the moment t_0, and the ith immediate transaction T_i^{imm} can be triggered at the moment t. The initial, i.e., for the moment t_0, priority of T is determined as

$$P_{t_0}(T) = S_0^{avg}(T).$$

If the transaction T_i^{imm} was triggered at the moment t, then

$$P_t(T) = d(T) - t - X_t^{avg}(T) - \sum_{j=1}^{n_t^{def}(T)} X_t(T_j^{def}) - X_t(T_i^{imm}),$$

$$P_t(T_i^{imm}) = P_t(T).$$

If no transaction was triggered at the moment t, then

$$P_t(T) = d(T) - t - X_t^{avg}(T) - \sum_{j=1}^{n_t^{def}(T)} X_t(T_j^{def}).$$

6.2.3. Priority of the deferred transaction. We consider the same strategies for assigning priorities as were suggested for immediate transactions.

If DIV or SL methods are used, the priority of the parent transaction changes when a delayed transaction is triggered; for the PD method, the priority of the parent transaction remains the same.

Consider the transaction T that triggers the ith deferred transaction at the time moment t. Then, the priority of the parent transaction is as follows.

1. PD method:

$$P_t(T) = d(T).$$

2. DIV method:

$$P_t(T) = P_t(T) - X_t(T_i^{def}).$$

3. SL method:

$$P_t(T) = d(T) - t - X_t^{avg}(T) - \sum_{j=1}^{n_t^{def}(T)} X_t(T_j^{def}) - X_t(T_i^{def}).$$

Suppose that T triggered m deferred transaction before its completion. Let T_j^{def} (the jth deferred transaction, $j \le m$) start executing at the time moment w (after the parent transaction has been already completed). Then, the priority of T_j^{def} is as follows.

1. PD method:

$$P_w(T_j^{def}) = d(T).$$

2. DIV method:

$$P_w(T_j^{def}) = C_w(T_j^{def}) + \frac{S_w(T) - \sum_{k=1}^{m} (X_w(T_k^{def}))}{m}.$$

That is, the reserve of the free time of the parent transaction is evenly divided between all deferred transactions. This quantity, plus the estimated completion time of the subtransaction, determines its priority.

3. SL method:

$$P_w(T_j^{def}) = d(T) - w - \sum_{k=1}^{m} (X_w(T_k^{def})).$$

That is, the priority is calculated as the time remaining to the deadline minus the estimated time required to execute all deferred transactions.

7. CONCLUSION

The requirements imposed upon real-time DBMSs imply additional study and revision of many divisions of database theory, including transaction protocols.

In real-time systems, transactions are assigned deadlines, and performance indexes differ from those used in classical DBMSs. For this reason, protocols used in traditional DBMSs do not perform very well in real-time systems.

In recent years, a number of specialized protocols for transaction management were suggested that take

the specific features of real-time systems into account. It is notable that in real-time systems, in contrast to traditional DBMSs, optimistic protocols perform better than pessimistic ones.

In real-time systems, not only transactions, but also data themselves can be of a temporal nature. In order for a transaction to be executed correctly, it must be ensured that the correct data set is used. It is not sufficient to consider the correctness of every individual data element since data can be interrelated. This specific feature requires substantial modifications in transaction management protocols.

In real-time systems, not only simple transaction models are used. In particular, active databases are widely used in such systems. In addition to specific features inherent in real-time DBMSs, issues related to assigning priorities to subtransactions must be taken into account.

Concrete real-time systems often present very high requirements for the performance of transaction management algorithms. For this reason, universal approaches to solving this problem are often unacceptable. On the other hand, such systems often have specific features that make it possible to develop a specific transaction management protocol for this particular system that provides enhanced performance. This is usually achieved by optimizing the performance for a class of most important transactions at the expense of all other transactions.

In this paper, we considered only a part of all problems related to consistency control in DBMSs; many other issues were not discussed. In particular, attempts to use alternative correctness criteria different from serialization were made [25, 26]; other sets of transaction properties (instead of ACID) were used. Much attention is also paid to formal methods used for modeling, specification of constraints, and correctness verification [23].

REFERENCES

1. Ullman, J., *Principles of Database Systems*, Computer Science, 1980. Translated under the title *Osnovy sistem baz dannykh*, Moscow: Finansy i Statistika, 1983.
2. Abbott, R. and Garcia-Molina, H., Scheduling Real-Time Transactions with Disk Resident Data, *Proc. 14th VLDB Conf.*, 1988.
3. Abbott, R. and Garcia-Molina, H., Scheduling Real-Time Transactions: a Performance Evaluation, *Proc. 15th VLDB Conf.*, 1989.
4. Adelberg, B., Garcia-Molina, H., and Kao, B. Database Support for Efficiency Maintaining Derived Data, *Technical Report, Stanford University*, 1995.
5. Agrawal, R., Carey, M.J., and Livny, M., Concurrency Control Performance Modeling: Alternatives and Implications, *ACM Trans. Database Syst.*, 1987, vol. 4, no. 12, pp. 609–654.
6. Bestavros, A. and Braoudakis, S., Timelines via Speculation for Real-Time Databases, *Proc. of RTSS'94: The 14th Real-Time Systems Symposium*, San Juan (Puerto Rico), 1994.
7. Bestavros, A., Kwei-Jay Lin, and Sang Hyuk Son, *Real-Time Database Systems: Issues and Applications*, Kluwer, 1997.
8. Brinkschulte, U., Tree-Based Buffer Management in Real-Time Database Systems, *Proc. 8th Int. Workshop on Database and Expert Systems Applications (DEXA'97)*, Toulouse, 1997, pp. 260–264.
9. Carey, M.J. and Stonebraker, M.R., The Performance of Concurrency Control Algorithms for Database Management Systems, *Proc. 10th VLDB Conf.*, Singapore, 1984.
10. Datta, A. and Son, S.H., *A Study of Concurrency Control in Real-Time Active Database Systems*, 1996.
11. Eriksson, J., *Real-Time and Active Databases: A Survey*, 1996.
12. Eswaran, K., Gray, J., Lorie, R., and Traiger, I., The Notions of Consistency and Predicate Locks in Database Systems, *ACM*, 1976, no. 11.
13. Haritsa, J.R., Carey, M.J., and Livny, M., Dynamic Real-Time Optimistic Concurrency Control, *Proc. IEEE Real-Time Systems Symposium*, Orlando, 1990.
14. Juhnyoung, L. and Son, S.H., *Concurrency Control Algorithms for Real-Time Database Systems*, Prentice Hall, 1996, pp. 429–460.
15. Kim, W. and Srivastava, J., Enhacing Real-Time DBMS Performance with Multiversion Data and Priority-Based Disk Scheduling, *Proc. 12th Real-Time Systems Symposium*, San Francisco, 1991.
16. Kim, Y. and Son, S.H., Supporting Predictability in Real-Time Database Systems, *Proc. IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, Boston, 1996.
17. Kung, H.T. and Robinson, J.T., On Optimistic Methods for Concurrency Control, *ACM Trans. Database Syst.*, 1981, vol. 2, no. 6.
18. Lam, K., Son, S.H., Lee, V. and Hung, S., Using Separate Algorithms to Process Read-Only Transactions in Real-Time Systems, *Proc. IEEE Real-Time Systems Symposium (RTSS'98)*, Madrid, 1998, pp. 50–59.
19. Kam-yiu Lam, Law, G.C.K., and Lee, V.C.S., Scheduling of Triggered Transactions in Distributed Real-Time Active Databases, *Lect. Notes Comput. Sci.*, 1998, vol. 1553, pp. 119–140.
20. Madria, S.K., Timestamp-Based Approach for the Detection and Resolution of Mutual Conflicts in Real-Time Distributed Systems, *Proc. Eighth Int. Workshop on Database and Expert Systems Applications (DEXA'97)*, Toulouse, 1997, pp. 692–699.
21. Nekrestyanov, I., Novikov, B., Pavlova, E., and Pikalev, S., Concurrency Control Protocols for Persistent Shared Virtual Memory Systems, *Proc. Second Int. Workshop on Advances in Databases and Information Systems (ADBIS'97)*, St. Petersburg, 1997, vol. 1, pp. 35–40.
22. Nekrestyanov, I. and Pavlova, E., Concurrency Control Protocol for Nested Transactions in Real-Time Databases, *Proc. Second Int. Workshop on Advances in Databases and Information Systems (ADBIS'97)*, St. Petersburg, 1997, vol. 1.
23. Pavlova, E. and Hung, D.V., A Formal Specification of the Concurrency Control in Real-Time Databases, *Proc.*

of Asia-Pacific Software Engineering Conf. (APSEC'99), Takamatsu, Japan, 1999.

24. Krzyzagórski, P. and Morzy, T., Optimistic Concurrency Control Algorithm with Dynamic Serialization Adjustment for Firm Deadline Real-Time Database Systems, *Proc. Second Int. Workshop on Advances in Databases and Information Systems (ADBIS'95)*, 1995, vol. 1, pp. 21–28.
25. Calton, P., Generalized Transaction Processing with Epsilon-Serializability, *Proc. Fourth Int. Workshop on High Performance Transaction Systems*, Asilomar, Calif., 1991.
26. Calton, P. and Leff, A., Autonomous Transaction Execution with Epsilon-Serializability, *Proc. 1992 RIDE Workshop on Transaction and Query Processing*, IEEE Computer Society, Phoenix, 1991.
27. Ramamritham, K., Real-Time Databases, *Int. J. Distributed Parallel Databases*, 1992, vol. 1, no. 1.
28. Sivasankaran, R.M., Stankovic, J.A., Towsley, D.F., Purimetla, B., and Ramamritham, K., Priority Assignment in Real-Time Active Databases, *VLDB J.*, 1996, vol. 5, no. 1, pp. 19–34.
29. Sivasankaran, R.M., Ramamritham, K., Stankovic, J.A., and Towsley, D.F., Data Placement, Logging, and Recovery in Real-Time Active Databases, *Proc. First Int. Workshop on Active and Real-Time Database Systems*, Sweden, 1995, pp. 226–241.
30. Son, S.H., Lee, J., and Lin, Y., Hybrid Protocols Using Dynamic Adjustment of Serialization Order for Real-Time Concurrency Control, *J. Real-Time Syst.*, 1992, no. 4, pp. 269–276.
31. Son, S.H. and Zhang, F., Real-Time Replication Control for Distributed Database Systems: Algorithms and Their Performance, *Proc. 4th Int. Conf. on Database Systems for Advanced Applications (DASFAA)*, Singapore, 1995, pp. 260–264.
32. Stankovic, J.A., Lu, C., Son, S.H., and Tao, G., The Case for Feedback Control Real-Time Scheduling, *Proc. 11th Euromicro Conf. on Real-Time Systems*, York, 1999.
33. Stankovic, J.A., Son, S.H., and Hansson, J., Misconceptions about Real-Time Databases, *IEEE Comput.*, 1999, vol. 32, no. 6, pp. 29–36.
34. Stankovic, J.A., Son, S.H., and Liebeherr, J., Bee-Hive: Global Multimedia Database Support for Dependable Real-Time Applications, *Lect. Notes Comput. Sci.*, 1998, vol. 1553, pp. 51–69.
35. Ulusoy, O., Analysis of Concurrency Control Protocols for Real-Time Database Systems, *Technical Report of Bilkent Univ.*, 1995, no. BU-CEIS-9514.
36. Ulusoy, O., Research Issues in Real-Time Database Systems, *Inf. Sci.*, 1995, vol. 87, nos. 1–3.
37. Ulusoy, O. and Belford, G.G., A Performance Evaluation Model for Distributed Real-Time Database Systems, *Int. J. Modeling Simul.*, 1995, vol. 15, no. 2.
38. Viguier, I.R. and Datta, A., Buffer Management in Active Real-Time Database Systems—Concepts and an Algorithm, *Lect. Notes Comput. Sci.*, 1998, vol. 1553, pp. 141–158.
39. Xiong, M. and Ramamritham, K., Specification and Analysis of Transactions in Real-Time Active Databases, in *Real-Time Database and Information Systems: Research Advances*, Kluwer, 1997.
40. Xiong, M., Ramamritham, K., Stankovic, J.A., Towsley, D.F., and Sivasankaran, R.M., Maintaining Temporal Consistency: Issues and Algorithms, *Proc. First Int. Workshop on Real-Time Databases*, Newport Beach, Calif., 1996.
41. Xiong, M., Sivasankaran, R.M., Ramamritham, K., and Towsley, D.F., Scheduling Access to Temporal Data in Real-Time Databases, in *Real-Time Database Systems: Issues and Aplications*, Kluwer, 1997.
42. Xiong, M., Sivasankaran, R.M., Stankovic, J.A., Ramamritham, K., and Towsley, D.F., Scheduling with Temporal Constraints: Exploiting Data Semantics, *Proc. 17th IEEE Real-Time Systems Symposium*, Washington, 1996.