Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics

Ming Xiong, *Member, IEEE*, Krithi Ramamritham, *Fellow, IEEE*,
John A. Stankovic, *Fellow, IEEE*, Don Towsley, *Fellow, IEEE*, and Rajendran Sivasankaran

Abstract—In this paper, issues involved in the design of a real-time database which maintains data temporal consistency are discussed. The concept of *data-deadline* is introduced and time cognizant transaction scheduling policies are proposed. Informally, data-deadline is a deadline assigned to a transaction due to the temporal constraints of the data accessed by the transaction. Further, two time cognizant *forced wait* policies which improve performance significantly by forcing a transaction to delay further execution until a new version of sensor data becomes available are proposed. A way to exploit temporal *data similarity* to improve performance is also proposed. Finally, these policies are evaluated through detailed simulation experiments. The simulation results show that taking advantage of temporal data semantics in transaction scheduling can significantly improve the performance of user transactions in real-time database systems. In particular, it is demonstrated that under the forced wait policy, the performance can be improved significantly. Further improvements result by exploiting data similarity.

Index Terms—Real-time database systems, temporal consistency, earliest deadline first, least slack first, data-deadline, transaction processing.

1 Introduction

A real-time database system is a transaction processing system designed to handle workloads in which transactions have deadlines. However, many real-world applications involve not only transactions with time constraints, but also data with time constraints. Such data, typically obtained from sensors, become inaccurate with the passage of time. Examples of such applications include autopilot systems, robot navigation, avionics systems, and process control systems [22], [18]. While considerable attention has focused on real-time databases, most of it assumes that only transactions have deadlines [1], [7], [8], [9], [10], [11], [12], [16], [20], [23]. New solutions that consider data time constraints are required for both concurrency control and cpu scheduling. Ample evidence now exists that such time-cognizant protocols are considerably better at supporting real-time transaction and data correctness than standard database protocols [24].

In this paper, novel solutions that explicitly deal with data time constraints in firm real-time database systems are proposed and evaluated. A firm real-time database system is one in which transactions that have missed their deadlines add no value to the system and, hence, can be aborted. The main contributions of the paper are:

- The development of notions of *data-deadline* and *forced wait* for scheduling transactions that access temporal data. Informally, data-deadline can be viewed as the deadline assigned to a transaction due to the temporal constraints of the data accessed by the transaction. Forced wait entails forcing a transaction to delay further execution until a new version of sensor data becomes available.
- A class of priority assignment policies that account for transaction deadlines and data time constraints. These include policies that force transactions to wait for new versions of data objects and policies that take advantage of *data similarity*.
- The comparison of the different policies with baseline Earliest Deadline First (EDF) and Least Slack First (LSF) policies. It is found that deadline based policies outperform slack based policies at medium loads, and this trend is reversed at high loads.
- A demonstration that while there is some improvement in performance when only data-deadline (without wait) is taken into account, there is significant improvement when it is combined with the notion of *forced wait*.
- A demonstration that, taking data similarity into consideration, improves performance significantly when the forced wait policy is not applied. But, when combined with forced wait, data similarity does not perceptibly impact performance. When estimates of execution (response) time are not available, using data similarity enhances performance.

The remainder of the paper is organized as follows: Section 2 discusses the related work. Section 3 describes the system and transaction model that is considered in the study. Section 4 outlines the transaction scheduling policies that have been considered in the study. Section 5 discusses the results of the experimental study and Section 6 summarizes and concludes the study.

- *M. Xiong is with Bell Labs, 600 Mountain Ave., Murray Hill, NJ 07974. E-mail: mxiong@lucent.com.*
- *K. Ramamritham is with the Indian Institute of Technology, Bombay. E-mail: krithi@cs.umass.edu.*
- *J.A. Stankovic is with the Department of Computer Science, University of Virginia, Charlottesville, VA 22904. E-mail: stankovic@cs.virginia.edu.*
- *D. Towsley is with the Department of Computer Science, University of Massachusetts, Amherst, MA 01003. E-mail: towsley@cs.umass.edu.*
- *R. Sivasankaran is with Knumi Inc., Cambridge, MA 02139. E-mail: raju@knumi.com.*

Manuscript received 30 May 2000; revised 13 Apr. 2001; accepted 19 Apr. 2001.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 112198.

2 Related Work

Over the past few years, real-time databases have become important areas of research. Experimental studies reported in [1], [7], [8], [9], [10], [11], [12], [16], [20], [23], are very comprehensive and cover most aspects of real-time transaction processing, but have not considered time constraints associated with data.

Database systems in which time validity intervals are associated with the data are discussed in [13], [14], [25]. Such systems introduce the need to maintain data temporal consistency in addition to logical consistency. The performance of several concurrency control algorithms for maintaining temporal consistency are studied in [25]. In the model introduced in [25], a real-time system consists of periodic tasks which are either read-only, write-only, or update (read-write) transactions. Data objects are temporally inconsistent when their ages or dispersions [25] are greater than the absolute or relative thresholds allowed by the application. Two-phase locking and optimistic concurrency control algorithms, as well as rate-monotonic and earliest deadline first scheduling algorithms are studied in [25]. These studies show that the performances of the rate-monotonic and earliest deadline first algorithms are close when the load is low. At higher loads, earliest deadline first outperforms rate-monotonic when maintaining temporal consistency. They also observed that optimistic concurrency control is generally worse at maintaining temporal consistency of data than lock based concurrency control, even though the former allows more transactions to meet their deadlines. It is pointed out in [25] that it is difficult to maintain the data and transaction time constraints due to the following reasons:

- A transient overload may cause transactions to miss their deadlines.
- Data values may become out of date due to delayed updates.
- Priority based scheduling can cause preemptions which may cause the data read by the transactions to become temporally inconsistent by the time they are used.
- Traditional concurrency control ensures logical data consistency, but may cause temporal data inconsistency.

Our development of the notion of *data-deadline* and the associated algorithms that make use of it are motivated by these problems.

In [13], a class of real-time data access protocols called SSP (Similarity Stack Protocols) is proposed. The correctness of SSP is based on the concept of similarity which allows different but sufficiently timely data to be used in a computation without adversely affecting the outcome. SSP schedules are deadlock free, subject to limited blocking and do not use locks. In [14], weaker consistency requirements based on the similarity notion are proposed to provide more flexibility in concurrency control for data-intensive real-time applications. While the notion of data similarity is exploited in their study to relax serializability (hence increase concurrency), here it is coupled with data-deadline and used to improve the performance of transaction scheduling. The notion of similarity is used to adjust transaction workload by Ho et al. [15] and incorporated into embedded applications (e.g., process control) in [4].

Temporal consistency guarantees are also studied in distributed real-time systems. In [28], *Distance constrained scheduling* is used to provide temporal consistency guarantees for real-time primary-backup replication service.

3 System Model and Correctness

In this section, the transaction and data models which characterize the features of real-time database systems are described. Also, the criteria for transaction correctness and data consistency in such real-time database systems are presented.

3.1 Transaction and Data Model

An *object* in the database models a real world entity, for example, the position of an aircraft. The objects in the database can be either temporal or nontemporal. A temporal object is one whose state may become invalid with the passage of time. Associated with the state is a temporal validity interval. An object whose state does not become invalid with the passage of time is a non-temporal object. Thus, there are no temporal validity intervals associated with non-temporal objects. Two approaches to modeling temporal data have been proposed in the literature [19]: attribute versioning and object versioning. In attribute versioning, a validity interval is associated with each attribute of an object, whereas in object versioning, a validity interval is associated with the aggregate object. Here, the focus is on object versioning, which maintains multiple versions of each object. Each state of a temporal object has a validity interval during which the state is valid. Temporal objects reflect specific objects in the environment and are updated periodically by transactions that read sensors. Two kinds of transactions are considered:

- **Sensor transactions**: These are the periodic transactions which write to temporal objects.
- **User transactions**: These are user-level transactions with deadlines. They read temporal objects and read/write nontemporal objects.

The deadlines of sensor transactions are derived from the requirement that transactions should update an object before the end of the validity interval associated with the data in order to keep the data fresh. Here, fresh refers to temporal consistency, which is introduced in the next section.

3.2 Transaction Correctness and Data Temporal Consistency

In real-time applications, the values of objects in a database must correctly reflect the state of the environment. Otherwise, decisions based on the data in the database may be wrong, and potentially disastrous. For example, data read by transactions must be fresh. This leads to the notion of temporal consistency. To define temporal consistency formally, the attributes of a temporal data object X are introduced first.

As mentioned earlier, a temporal data object has multiple versions. The ith version of data object X, X_i ($(i = 1, 2, \ldots)$), is defined as:

$$(value(X_i), vi(X_i)),$$

where $value(X_i)$ denotes the ith state of X, and $vi(X_i)$ denotes $value(X_i)$'s *validity interval*, i.e., the time interval during which $value(X_i)$ is considered to be temporally consistent. After $vi_e(X_i)$, $value(X_i)$ is no longer valid.[1] So, the attributes of a temporal data object X are defined as follows:

- X_i: the ith version of data object X
- $vi_b(X_i)$: the beginning of the validity interval of X_i;
- $vi_e(X_i)$: the end of the validity interval of X_i;
- $vi(X_i)$: the validity interval of X_i;
  $vi(X_i) = [vi_b(X_i), vi_e(X_i))$, where
  $vi_b(X_i) < vi_e(X_i)$.

The ith version of data object X, X_i $((i = 1, 2, \ldots))$ is temporally consistent at time t if and only if:

$$vi_b(X_i) \le t < vi_e(X_i).$$

In the rest of the paper, when we say that a transaction T reads a data object X at time t, it should be understood that T reads a version of X that is temporally consistent at time t.

A transaction in our real-time database can commit if and only if

1. it is logically consistent, i.e., it is serializable and satisfies all the data integrity constraints,
2. it meets its deadline, and
3. it reads temporally consistent data and the data it read are still fresh when it commits.

In certain circumstances, it may be possible to relax one or more of these constraints but these possibilities are left for future work.

3.3 Concurrency Control for Data Objects

In the current model, user transactions need to obtain database locks in order to read or write a non-temporal objects. Temporal objects are only written by sensor transactions, which are write-only transactions. Furthermore, sensor transactions do not read any data in the database; their write sets are disjoint from each other and from the write sets of user transactions. When a sensor transaction writes a temporal object in each period, it creates a new version of the temporal object. In this case, there is no need for a sensor transaction to obtain database locks in order to write. On the other hand, no transactions other than sensor transactions can write temporal objects. Thus, user transactions do not need to obtain database locks in order to read a valid version of an object. Therefore, there is no database concurrency control for these temporal objects. However, to ensure that data is not read while it is being updated, latches (i.e., short term locks or semaphores) must be used.

For nontemporal data, conflict resolution in case of concurrent access is based on the *priority abort* protocol, where the conflicting transaction with lower priority waits or gets aborted depending on whether it is the requester or holder of locks, respectively.

1. This ignores *data similarity* which is introduced in Section 4.4.

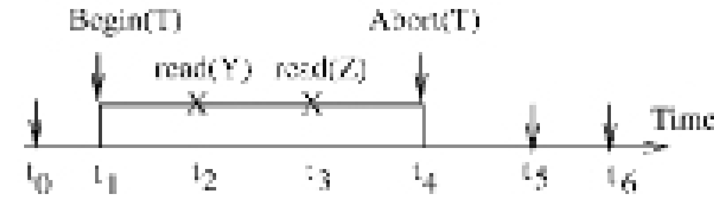

Fig. 1. An illustration of data deadline.

4 Scheduling Transactions in Real-Time Databases

The transactions in the system can be classified into two classes: user transactions and sensor transactions. The scheduling algorithm should maximize the number of user transactions which meet their deadlines while maintaining temporal consistency. In the system studied, sensor transactions always get higher priority than user transactions. Other policies to assign priorities to user and sensor transactions will be studied in future work. All the sensor transactions are scheduled by the earliest deadline first policy. Policies to assign priorities to user transactions based on their deadlines and the constraints on temporal data objects that they access are studied.

Data read by a transaction must be valid when the transaction completes, this leads to another constraint on completion time, in addition to a transaction's deadline. This constraint is referred to as *data-deadline*. Within the same transaction class, the scheduling algorithm should be aware of the *data-deadline* of a transaction, that is, the time after which the transaction will violate temporal consistency.[2] The scheduling algorithm should account for data-deadlines when it schedules transactions whenever a data-deadline is less than the corresponding transaction deadline. Consider the following example which illustrates the concept of data-deadlines.

In Fig. 1, transaction T needs to read two temporal data objects, Y and Z, to produce results. T reads these data objects at time t_2 and t_3, respectively. The deadline of transaction T is t_6. Data Y is valid in the interval $[t_0, t_5]$, and data Z is valid in the interval $[t_0, t_4]$. Transaction T starts at time t_1, and it has no data-deadline at this time. At time t_2, transaction T reads data Y. The data-deadline of transaction T becomes t_5 since it will violate temporal consistency after time t_5. In order to satisfy temporal consistency, T has to be scheduled to commit before time t_5, i.e., before the value of Y it read becomes invalid. Notice the deadline of transaction T is later than time t_5. Next, transaction T proceeds and reads data Z with a value which becomes invalid at time t_4. Now, the data-deadline of T is adjusted to t_4. At time t_4, transaction T has not completed. Thus, it aborts. Note that it might be possible to restart T and use subsequent version of Y and Z to meet deadline t_6.

The following are some of the attributes of a transaction T that are useful in explaining the policies:

- $a(T)$: the arrival time of T
- $s(T)$: the start time of T
- $d(T)$: the deadline of T
- $dd_t(T)$: the data-deadline of T at time t

2. A transaction can violate temporal consistency without missing its deadline.