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:

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

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.

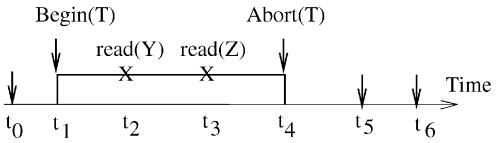

Fig. 1. An illustration of data deadline.

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.

- $L(T)$: the number of object accesses of T
- $L_t^{to}(T)$: the number of remaining temporal object accesses of T at time t
- $L_t^{nto}(T)$: the number of remaining nontemporal object accesses of T at time t
- $L_t(T)$: the number of remaining object accesses of T at time t; $L_t(T) = L_t^{to}(T) + L_t^{nto}(T)$; $L_{a(T)}(T) = L(T)$
- $E_t(T)$: the estimated remaining execution time of T at time t
- $R_t(T)$: the estimated remaining response time of T at time t
- $C_t(T)$: the estimated completion time of T at time t; $C_t(T) = t + E_t(T)$
- $RS_t^{to}(T)$: the readset of T at time t; this set contains the versions of all temporal data objects read by T.
- $P_t(T)$: the priority of T at time t

The data-deadline of transaction T at time t, $dd_t(T)$, is defined as:

$$dd_t(T) = \min_{X \in RS_t^{to}(T)} vi_e(X).$$

4.1 Baseline Scheduling Policies for CPU and Data Access

All the policies presented are preemptive dynamic priority policies that differ in the method used to set priorities. Before explaining the policies that are based on data-deadlines, the baseline policies to which all the policies are compared are outlined.

- *Earliest Deadline First* (EDF): This is the traditional EDF policy where the priority of a transaction is its deadline.

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

  This policy neglects temporal consistency constraints. Note that a transaction aborted due to temporal inconsistency can be restarted.
- *Least Slack First* (LSF): This is the conventional LSF policy where the priority of a transaction corresponds to its estimated slack.

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

  Here again, temporal consistency constraints are ignored and a transaction aborted due to temporal inconsistency can be restarted.

4.2 Data-Deadline Based Scheduling Policies

As mentioned earlier, EDF and LSF are not cognizant of data-deadlines in scheduling. Transactions that read data objects that have become invalid are aborted because of commit criteria. Variations of EDF and LSF that are cognizant of data-deadlines in scheduling could possibly avoid such aborts. They are as follows:

- *Earliest Data-Deadline First* (EDDF): This policy assigns the minimum of the data-deadline and transaction deadline to be the priority of the transaction.

  $$P_t(T) = min(dd_t(T), d(T)).$$

- *Data-Deadline based Least Slack First* (DDLSF): This policy calculates the slack of the transaction based on the minimum of the data-deadline and transaction deadline.

  $$P_t(T) = min(dd_t(T), d(T)) - C_t(T).$$

4.3 Policies with Forced Wait

In many cases, estimates of either the remaining execution or response time are available. Both of these estimates can be used to maintain data temporal consistency. A transaction that reads valid data may not be able to commit because the validity interval of a data object it reads expires before it can commit. Although it is impossible to predict the exact time when a transaction will commit, it is possible to estimate the minimum execution time needed for a transaction to complete. If a transaction reads a data object whose remaining validity interval is less than the remaining execution time of the transaction, there is no chance for the transaction to commit before the validity interval of that data object expires. Such a validity interval is called an *infeasible validity interval* with respect to the transaction. To prevent a transaction from reading invalid data, or data with an infeasible validity interval, it can be forced to wait until the data is updated. That is, there must be a mechanism to check the validity of data when the transaction reads it. In Fig. 1, the data-deadline of T becomes t_4 once it reads data object Z. But, if T cannot commit before t_4 then T will be aborted after t_4 because a transaction commits only if all the data objects it accessed are still valid at commit time. Instead, T can wait until t_4 and read the new version of Z. This approach saves the work a transaction has already done and avoids recovery overhead. However, this method has the disadvantage of incurring the cost to check for infeasible validity intervals which, in turn, may unnecessarily delay the commitment of transactions.

4.3.1 Forced Wait with Estimated Remaining Execution Time

The policies outlined above (EDF, EDDF, LSF, and DDLSF) do not consider the *feasibility* of validity intervals of data objects that a transaction accesses. To remedy this, these policies are combined with the idea of *Forced Wait*. With forced wait, whenever a temporal data object is read by a transaction, the system checks if this transaction will commit before the validity of the data object expires. If the validity could expire before the commit, then the transaction is made to wait until the data object is updated, else the transaction is allowed to continue. If transaction T reads temporal data object X at time t, then the following condition for *Forced Wait policy with Estimated Remaining Execution Time* (FWE) is checked:

$$condition\ 1 : t + E_t(T) \leq vi_e(X).$$

Under FWE, if $condition\ 1$ holds, then the transaction proceeds. Otherwise the transaction waits. This can be coupled with the previously described policies to yield the following combined policies:

- *Earliest Deadline First with FWE* (EDF-FWE),
- *Earliest Data-Deadline First with FWE* (EDDF-FWE),

- *Least Slack First with FWE* (LSF-FWE), and
- *Data-Deadline based Least Slack First with FWE* (DDLSF-FWE).

4.3.2 Forced Wait with Estimated Remaining Response Time

In this case, estimated remaining response time is used in forced wait. This is slightly more sophisticated than FWE in that it estimates the remaining response time based on the statistical data from the system. Our hypothesis is that, when such information is available, it can provide additional performance benefits. In order to estimate the response time needed in forced wait policy, contention due to access to shared resources such as the CPU and data is accounted for. $CPUSF_t$ is the slowdown factor at time t due to CPU resource contention. $CPUSF_t$ is the average time taken to get one unit of CPU work done. $CCSF_t$ is the slowdown factor at time t due to data resource contention. It is the average blocking time to acquire the lock on a nontemporal data object. Thus,

$$R_t(T) = E_t(T) * CPUSF_t \ + \ L_t^{nto}(T) \ * \ CCSF_t.$$

If the estimated remaining response time is used, then the following condition is tested:

$$condition\ 2 : t \ + \ R_t(T) \ \leq \ vi_e(X).$$

It should be noted that since $R_t(T) \geq E_t(T)$, condition 2 is stronger than condition 1. *Forced Wait policy with Estimated Remaining Response Time* (FWR) makes use of condition 2 in addition to condition 1, to perform a more precise check. To implement FWR policy, three queues are maintained in the system, namely, *CPU_Queue*, *Sleep_Queue*, and *Wait_Queue*.

- If condition 2 is true, then the transaction is kept in CPU_Queue. The scheduler removes the highest priority transaction from CPU_Queue to execute.
- If condition 2 is false and condition 1 is true, then the transaction is placed in the Sleep_Queue. Transactions in the Sleep_Queue are executed only if the cpu is idle, i.e., if there are no transactions in the CPU_Queue. This is because transactions satisfying condition 2 have better chance to commit before validity of data expires than transactions do not satisfy condition 2.
- If condition 1 is false, then it is not possible for the transaction to commit before validity of data X expires. The transaction is placed in the Wait_Queue where it remains until X is updated. Once X has been updated the conditions are checked again. After which it is moved to CPU_Queue (if condition 2 is true) or Wait_Queue (if condition 1 is true but condition 2 is false).

The following algorithm summarizes the FWR policy:

Algorithm: Condition checking in FWR policy
if $(t + R_t(T) \ \leq \ vi_e(X))$ then
  T Continues;
else if $(t + E_t(T) < vi_e(X))$ then
  Place T in Sleep_Queue;
else
  Place T in Wait_Queue;
end;

With FWR, the following combined policies are tested:

- *Earliest Deadline First with FWR* (EDF-FWR),
- *Earliest Data-Deadline First with FWR* (EDDF-FWR),
- *Least Slack First with FWR* (LSF-FWR), and
- *Data-Deadline based Least Slack First with FWR* (DDLSF-FWR).

TABLE 1
Latch Compatibility Table

Mode	Read	Write	Notify
Read	Yes	No	Yes
Write	No	No	Yes
Notify	Yes	Yes	Yes

4.3.3 A Mechanism to Implement Forced Wait

As mentioned in Section 3.3, there is no database concurrency control for temporal objects. However, to implement forced wait, there are three latch modes for temporal objects in the system: *read (R)*, *write (W)* and *notify (N)*. The compatibility of these latches is shown in Table 1. When a temporal object is read or written, a read or write latch is set. When a user transaction is forced to wait, a notify latch [6] is set. A user transaction that is forced to wait needs to be notified when a write latch is released, i.e., the corresponding temporal object has been updated successfully. Notify mode is compatible with all other modes, and can be treated like a free mode. When a write latch is released, all the user transactions with notify latches are awakened.

4.4 Policies with Similarity

It is often the case that the values of data objects do not change much from one update to another. In this case, versions of a data object that are only slightly different are indistinguishable to transactions reading the data. This is the basic concept of *data similarity* that has been explored in detail in [13], [14]. Similarity is based on the idea that a past version of data that is not current but close to the current version can be used in processing if it does not adversely affect the outcome. This can be used to improve performance. Two versions of a data object are *similar* if a transaction which reads the data object considers them as similar. As pointed out in [13], [14], similarity is a binary relation on the domain of a data object. It is reflexive and symmetric, but not necessarily transitive. In our paper, the assumption is that it is not transitive. In our model, if $value(X_i)$ read by a transaction is similar to $value(X_{i+1})$, then its vi is extended as follows:[3]

$$vi(X_i) \ = \ [vi_b(X_i), vi_e(X_{i+1})),$$

In Fig. 1, the data-deadline of T becomes t_4 once it reads data object Z. But, if T cannot commit before t_4 then T is aborted after t_4. However, if the version of Z before and after t_4 are similar then T can commit after t_4. Again, EDF,

3. Based on symmetry of data similarity, the $vi(X_{i+1})$ can also be extended to become $[vi_b(X_i), vi_e(X_{i+1}))$. However, this does not help extend data-deadlines.

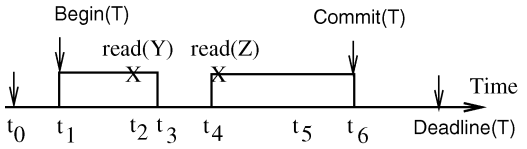

Fig. 2. An illustration of policies with both forced wait and similarity.

EDDF, LSF, and DDLSF policies do not take advantage of the similarity semantics of temporal data objects that a transaction accesses. It is reasonable to combine the notion of data similarity with data-deadline. For example, a transaction may miss its data-deadline but not its transaction deadline because a data object that it read becomes invalid (because it is updated). If this happens and similarity is used, the system checks if the current version of the data is similar to the version that it read. If the current version is similar to the version that it read, then the data deadline of the transaction is extended and the transaction continues instead of being aborted; otherwise the transaction is aborted. Again, our hypothesis is that, by taking advantage of similarity, data-deadline misses can be reduced and performance can be improved.

4.5 Policies with Both Forced Wait and Similarity

Not surprisingly, forced wait and similarity can be combined. In this case, forced wait is employed when a data object is read, and similarity is employed if a version read by a transaction becomes invalid, but is similar to the current version. Consider the situation illustrated by Fig. 2. As in Fig. 1, transaction T needs to read both data Y and Z. T begins at time t_1. At this time, the current version of Y is valid in the interval $[t_0, t_5)$, and the current version of Z is valid in the interval $[t_0, t_4)$. At time t_2, T reads a version of Y that is valid in the interval $[t_0, t_5)$. However, at time t_3, T is not allowed to read the current version of Z that is valid in the interval $[t_0, t_4]$ by the forced wait policy. Here T is forced to wait for the next version of Z, which is valid from time t_4. After T reads this new version of Z, it proceeds and completes its work at time t_6. But at t_6, the version of Y read by T is already invalid. However, the current version of Y is similar to the version of Y read by T. Because similarity is employed, transaction T can commit at time t_6, which is before its deadline. It should be noted that T may be aborted at time t_5 if the version of Y read by T is *not similar* to the current version of Y.

5 Experimental Evaluation of the Protocols

This section presents the experimental setup and the assumptions made in the experiments. Each set of experiments is discussed and an analysis of the results is presented. All experiments are conducted in a main memory database setting. In the following discussion, EDF, EDDF, and all the variations of those policies are referred to as *deadline* based, and LSF, DDLSF, and their variations are referred to as *slack* based policies. The primary performance metric is Missed Deadline Percentage (**MDP**) of user transactions, i.e., the percentage of user transactions that miss their deadlines, which is a traditional metric used to evaluate performance in real-time database systems.

Let N_{miss} denote the number of user transactions that miss deadlines, and $N_{succeed}$ denote the number of user transactions that succeed. The MDP is given by the following expression:

$$MDP = \frac{N_{miss}}{N_{miss} + N_{succeed}} \times 100.$$

In our study, a transaction is aborted as soon as its deadline expires. This corresponds to a firm real-time transaction. This policy assumes that finishing a transaction after its deadline expires does not impart any value to the system. As mentioned in Section 4, sensor transactions always get higher priority than user transactions, and they are scheduled by the earliest deadline first policy. Our study focusses on policies to assign priorities to user transactions based on their deadlines and the constraints on temporal data objects that they access.

Another metric, Data-Deadline Abort Ratio (DDAR), is defined as:

$$DDAR = \frac{Total\ No.\ of\ data - deadline\ aborts}{Total\ No.\ of\ User\ Transactions}.$$

A simulator, called $RADEx$ [23], was developed to perform our experiments. It was implemented using the DeNet Simulation Language [17]. In the experiments, 90 percent confidence intervals have been obtained whose widths are less than ± 10 percent of the point estimate for the Missed Deadline Percentage (MDP).

5.1 Baseline Parameters

In this section, the workload model is described. Let $U(i, j)$ denote a uniformly distributed integer valued random variable in the range $[i, j]$. The length, $L(T)$, of a transaction T is given as a number of object accesses:

$$L(T) = \begin{cases} U(6,12), & \text{T}\epsilon\ \mathbf{UT} \\ U(1,1), & \text{T}\epsilon\ \mathbf{ST}, \end{cases}$$

where **UT** and **ST** stand for user transactions and sensor transactions, respectively. Let $L_t(T)$ denote the remaining length of transaction T at time t. The estimate that is used for $E_t(T)$ is:

$$E_t(T) = L_t(T) * CPU\ time\ per\ object\ access.$$

The deadline of a user transaction T is set using the following formula:

$$d(T) = a(T) + (1 + Slack) * E_{a(T)}(T),$$

where $Slack$ is a uniformly distributed random variable within a specified range. The deadline of a sensor transaction is equal to the end of its period.

The parameter *load* used in our experiments is very similar to that used in [23]. In order to define *load* the arrival rates and service rates of **UT** and **ST** transactions are specified. The arrivals of **UT** are generated according to a Poisson process with mean interarrival time of $1/\lambda_{UT}$ time units, and those of **ST** transactions are generated according to the periods of sensor transactions which are uniformly distributed between a specified range with mean of $1/\lambda_{ST}$ time units. The arrival rates are calculated using the following two equations, where all other quantities except the arrival rates are assumed to be known. In the first equation, the *load* is defined to be the ratio of work generated to the total processing capacity of the system.

TABLE 2
System Parameters

Parameter	Meaning
N_{CPU}	Number of CPUs
N_{to}	Number of temporal object instances
N_{nto}	Number of non-temporal object instances
P_{sim}	Probability that a temporal object version is similar to the previous one
vi	Validity interval

Let $1/\mu_{UT}$ and $1/\mu_{ST}$ denote the average total execution time of **UT** and ST transactions, respectively, and N_{cpu} the number of CPUs in the system. In the second equation, $frac_{ST}$ is the fraction of *load* that is contributed by the class ST transactions.

$$load = \frac{\frac{\lambda_{UT}}{\mu_{UT}} + \frac{\lambda_{ST}}{\mu_{ST}} * N_{to}}{N_{cpu}}$$

$$frac_{ST} = \frac{\frac{\lambda_{ST}}{\mu_{ST}} * N_{to}}{\frac{\lambda_{UT}}{\mu_{UT}} + \frac{\lambda_{ST}}{\mu_{ST}} * N_{to}}.$$

If the values of $load$, $frac_{ST}$, $1/\mu_{UT}$, and $1/\mu_{ST}$ are given, then $1/\lambda_{UT}$ and $1/\lambda_{ST}$ can be computed from the above equations. When the system is fully loaded, i.e., $load = 1.0$, $frac_{ST}$ is

$$frac_{ST} = \frac{\lambda_{ST} * N_{to}}{\mu_{ST} * N_{cpu}}.$$

System settings are controlled by the parameters listed in Table 2. In our experiments, the similarity relationship is not transitive. Transaction characteristics are controlled by the parameters listed in Table 3. Table 4 and Table 5 show the system and transaction parameter settings, respectively, for our baseline experiments. The $Period$ of a sensor transaction is equal to the length of the vi of the data object that it updates.

With the parameter settings in Table 4 and Table 5, the load from sensor transactions is about 20 percent when the system is fully loaded, i.e., $load = 1.0$. No sensor transactions miss their deadlines in our experiments because they are assigned higher priority than user transactions. In our simulations, since the load is fixed for a single run, running averages of $CPUSF_t$ and $CCSF_t$ are kept from the begining of the simulation. The performance numbers that are referred to in the following discussion are absolute values of MDP.

TABLE 3
Transaction Parameters

Parameter	Meaning
$AccessTime$	CPU time per object access
$Slack$	User transaction slack
$Period$	Sensor transaction period
P_{comp}	Probability that two object accesses are compatible
$Length$	No. of object accesses by a transaction
$Prob_{to}$	Probability that a transaction accesses a temporal object

TABLE 4
Default System Parameter Settings

Parameter	Setting
N_{cpu}	2
N_{to}	50
N_{nto}	500
P_{sim}	0.0
vi	$U(40, 200)$

TABLE 5
Default Transaction Parameter Settings

Transaction Class	Parameter	Settings
class **ST**	$AccessTime$	1.0
	$Length$	1
class **UT**	$AccessTime$	1.0
	$Slack$	$U(8.0, 12.0)$
	P_{comp}	0.0
	$Length$	$U(6, 12)$
	$Prob_{to}$	0.4

5.2 Data-Deadline Based Policies

In the first set of experiments the performances of policies that account only for the data-deadline, i.e., EDDF and DDLSF are compared to the performances of the baseline policies EDF and LSF. The results are presented in Fig. 3. The results show that at medium loads, the policies exhibit similar performance. At high loads, the *slack*-based policies outperform the *deadline*-based policies. For instance, when the load is 0.9, DDLSF reduces the MDP by 8 percent with respect to EDDF whose performance is very close to the performance of EDF. As load increases, the difference between deadline-based policies and slack-based policies increases. Also, between DDLSF and LSF, DDLSF reduces the MDP for all loads by around 3 percent. Among the *deadline*-based policies, taking the data-deadline into account does not seem to make much difference.

The total and useful CPU utilizations of all policies were analyzed and the following was observed: The useful CPU utilizations for all policies are about the same at medium load. But as load increases, the useful CPU utilizations of *slack*-based policies are higher than those of *deadline*-based policies. This is because, as observed by Haritsa et al. [8], *deadline*-based policies give the highest priority to transactions that are close to missing their deadlines at high loads,

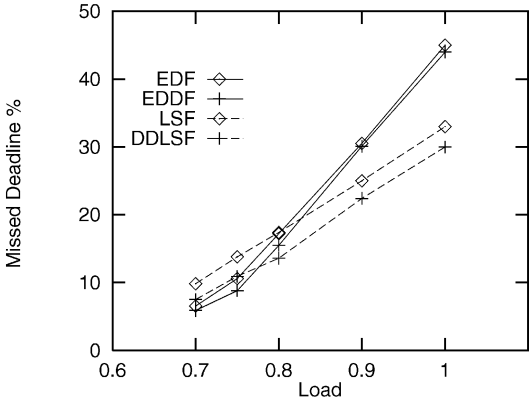

Fig. 3. Baseline and data-deadline policies.

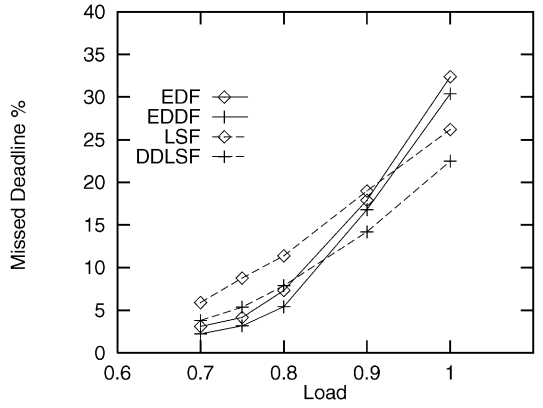

Fig. 4. FWE.

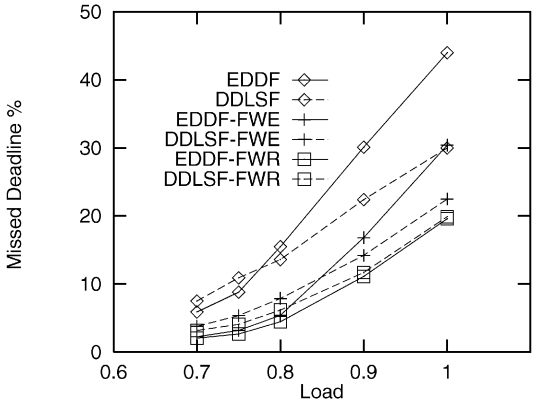

Fig. 6. No forced wait versus FWE/FWR.

thus aggravating deadline misses and perform worse than *slack*-based policies. This explains why the *slack*-based policies outperform *deadline*-based policies at high load.

At medium loads, deadline-based policies outperform the slack-based policies and at high loads this trend is reversed. Overall, taking only data-deadline into account only marginally improves performance.

5.3 Policies with Forced Wait

The second set of experiments was conducted to evaluate the benefits of using execution/response time estimates in choosing between the present version of a data object and a future version. The results are presented in Fig. 4, Fig. 5, and Fig. 6, respectively. Fig. 4 compares the performance of the four forced wait algorithms EDF-FWE, EDDF-FWE, LSF-FWE, and DDLSF-FWE. The results show that at medium loads, deadline-based policies perform better than slack-based policies. At high loads DDLSF-FWE outperforms the other policies. For instance, DDLSF-FWE outperforms LSF-FWE by 5 percent, EDF-FWE and EDDF-FWE by about 3 percent when the load is 0.9. As load increases, *deadline*-based policies deteriorate more sharply than *slack*-based policies.

With the forced wait policy FWR, as shown in Fig. 5, the LSF-FWR performs the worst for all workloads. From the graph it can be observed that, when compared to the priority assignment policies without forced wait and with forced wait, the difference between *slack*-based policies and *deadline*-based policies decreases significantly at high loads. In this case, there is almost no difference between deadline-based policies and slack-based policies. In general, it was observed that using forced wait brought down the MDP of *deadline*-based policies more than it brought down the MDP of *slack*-based policies at high loads. This, however, is due to the nature of EDF at high loads. As observed by Haritsa et al. [8], EDF aggravates deadline misses at high loads and performs worse than most other policies. This is because EDF gives the highest priority to transactions that are close to missing their deadlines. When EDF is integrated with forced wait, high priority transactions that are close to missing their deadlines may be forced to wait. In this manner, lower priority transactions can be executed and the number of deadline misses is reduced.

In Fig. 6, the performances of data-deadline based policies (EDDF and DDLSF) are compared with different forced wait policies. The comparisons of variations of EDF and LSF policies are similar and not repeated. As can be seen from Fig. 6, the performance gain due to the addition of forced wait to *deadline*-based policies increases sharply as load increases. For instance, when the load is 0.9, EDDF-FWR outperforms EDDF-FWE by 7 percent, while EDDF-FWE outperforms EDDF by more than 15 percent. As can be seen from Fig. 7, when the load is 0.9, forced wait policies reduce the data-deadline abort ratio by more than 10 percent with respect to the policies without forced wait. For *slack*-based policies, the addition of forced wait also improves

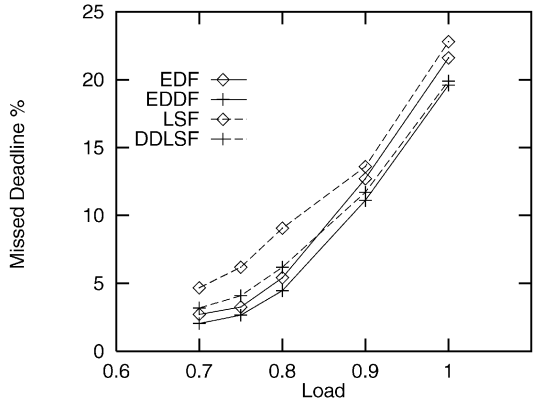

Fig. 5. FWR.

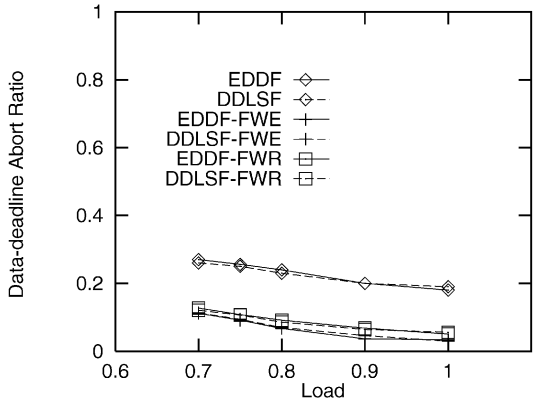

Fig. 7. No forced wait versus FWE/FWR.

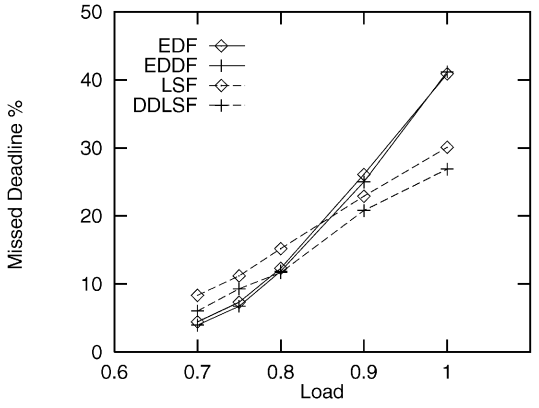

Fig. 8. Baseline and data-deadline policies ($P_{sim} = 0.5$.)

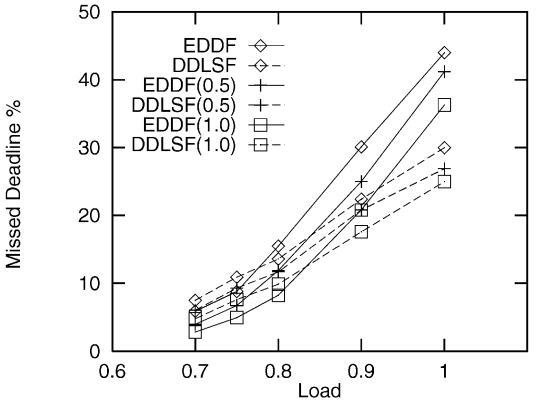

Fig. 10. Using versus not using similarity.

performance significantly. But, as can be seen from Fig. 6, DDLSF-FWR outperforms DDLSF-FWE only marginally.

In summary, forced wait policies improve the performance significantly in comparison to policies without forced wait.

5.4 Policies with Similarity

The third set of experiments was conducted to evaluate the performance of policies that are based on the idea of similarity. For these experiments, P_{sim} = 0.5, 1.0, respectively. When P_{sim} = 1.0, every version of a data object is similar to its preceding version. The results are presented in Fig. 8 and Fig. 9. From those graphs, it is observed that similarity does not affect the conclusion drawn from Fig. 3. That is, with similarity, *deadline*-based policies outperform *slack*-based policies marginally at medium loads, and *slack*-based policies outperform *deadline*-based policies significantly at high loads. This should be obvious since similarity does not affect the relative performance of different scheduling policies. In Fig. 10, the results of experiments with baseline policies (EDDF and DDLSF) which do not take similarity into account and policies which consider similarity are plotted. Two values of P_{sim} are considered, 0.5 and 1.0. It is observed from the graph that policies taking similarity into account do reduce the MDP. The higher the probability of similarity, the better the performance. In particular, when estimates of execution (response) time are not available, using data similarity enhances performance. This can be observed from the performance of deadline based policies in Fig. 10.

5.5 Policies with Both Forced Wait and Similarity

The last set of experiments was conducted to evaluate the performance of policies with both forced wait and similarity. As discussed in Section 4.4, if similarity is exploited, it may reduce the additional performance gain from using forced wait. The results are presented in Fig. 11. In the experiments, P_{sim} is set to 1.0. The performances of the EDDF and DDLSF policies without forced wait, and with FWE and FWR are compared. From the graph, it is observed that EDDF (DDLSF) policies with FWR outperform EDDF (DDLSF) policies with FWE, which again outperform EDDF (DDLSF) policies without forced wait. For instance, EDDF-FWE outperforms EDDF by about 5 percent when the load is 0.9. But, compared to Fig. 6, the performance gain of EDDF-FWE over EDDF is reduced from 14 percent to 5 percent. The additional performance improvement due to forced wait is less when similarity is used compared to when similarity is not used.

Further experiments with forced wait policies were conducted to explore the sensitivity of the result to the similarity probability. The results are presented in Fig. 12 and Fig. 13. In each figure, 0.0, 0.5, and 1.0 represent the value of similarity probability (P_{sim}). It is observed from both graphs that the value of P_{sim} has little effect on performance. This shows that forced wait policies play a dominant role in the performance when both forced wait and similarity policies are used. Fig. 14 compares the performance of variations of DDLSF policy. P_{sim} is set to 1.0 for policies with similarity. It is

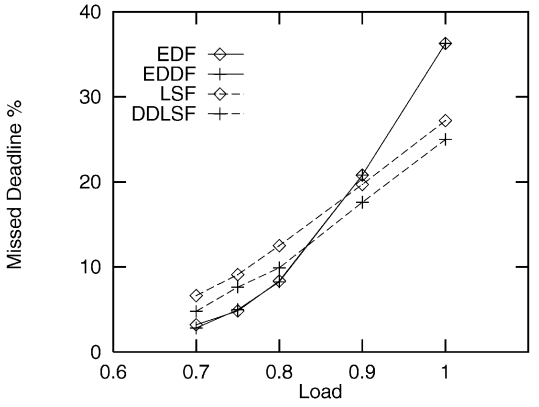

Fig. 9. Baseline and data-deadline policies ($P_{sim} = 1.0$).

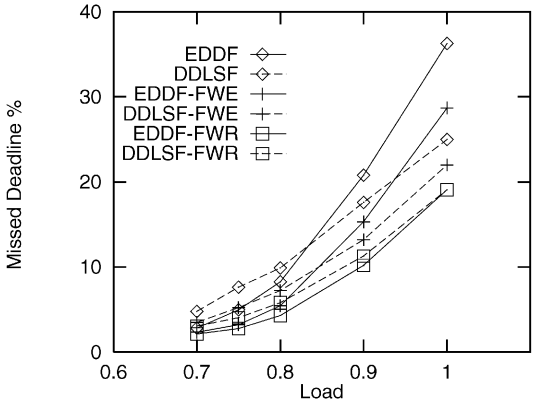

Fig. 11. No forced wait versus FWE/FWR ($P_{sim} = 1.0$).

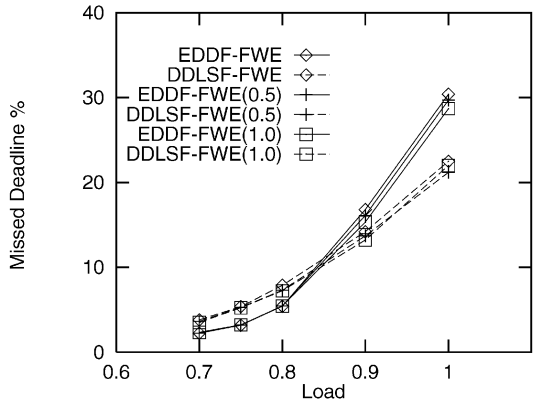

Fig. 12. FWE with different P_{sim} (0.0, 0.5, and 1.0).

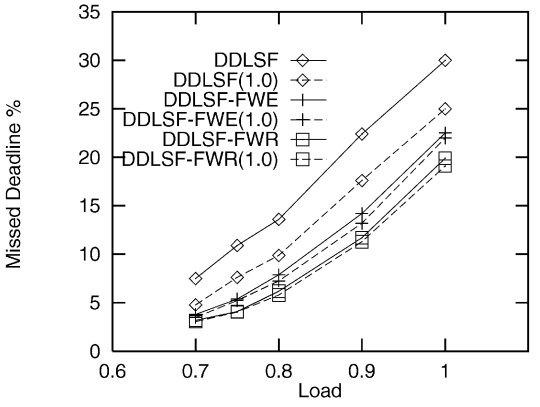

Fig. 14. Variations of DDLSF.

observed that, for the given workload, DDLSF benefits more from forced wait than from similarity.

From these experiments, it is observed that taking data similarity into consideration improves performance significantly when forced wait policy is not applied. But when combined with forced wait, the influence of data similarity decreases.

6 Conclusions

Although transaction scheduling and concurrency control aspects of real-time databases have been studied in detail, not much attention has focused on maintaining temporal consistency. The question of how to improve system performance while transactions maintain data temporal consistency and meet their deadlines poses a new and challenging problem. In this paper, issues involved in designing a real-time database which ensures data temporal consistency are adressed. The ideas of *data-deadline* and *forced wait* are developed. Based on these ideas, transaction scheduling and concurrency control policies are developed and evaluated with detailed simulation studies. Scheduling policies that take advantage of the similarity between different versions of temporal data objects are also evaluated. The main conclusions are:

- Taking data-deadline into consideration for scheduling improves performance. Considerable additional performance improvement is achieved when it is combined with forced wait policy.
- Using extra information such as estimates of execution times to avoid infeasible validity intervals enhances the performance of user transactions significantly, as can be seen from the experiments conducted in Section 5.3. Further, using more accurate information such as estimates of remaining response times to avoid infeasible validity interval makes forced wait policy perform even better than just using estimates of execution times.
- Exploiting similarity in transaction scheduling can reduce data-deadline aborts, thereby resulting in better performance. But data similarity does not produce additional performance improvement when combined with forced wait. However, when estimates of execution (response) time are not available, using data similarity enhances performance.
- When forced wait and similarity policies are combined, forced wait plays a dominant role in enhancing the performance of user transactions.

Therefore, forced wait and similarity are recommended for temporal consistency maintenance in a centralized real-time database system.

In distributed real-time database systems, data temporal consistency is even harder to maintain than centralized systems. Many open questions still remain. For example, how to maintain data temporal consistency in distributed systems? How can data similarity be used to help maintain temporal consistency? Such issues need to be studied in the future.

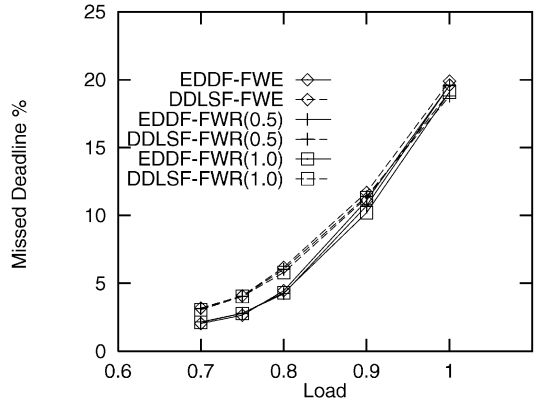

Fig. 13. FWR with different P_{sim} (0.0, 0.5, and 1.0).

Acknowledgments

A preliminary version of this paper appeared in IEEE 17th Real-Time Systems Symposium, December 1996. This work was supported, in part, by the US National Science Foundation grants IRI-9114197 and EIA 9900895.

References

[1] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *Proc. ACM Trans. Database Systems*, vol. 17, no. 3, pp. 513-560, Sept. 1992.

[2] B. Adelberg, H. Garcia-Molina, and B. Kao, "Applying Update Streams in a Soft Real-Time Database System," *Proc. 1995 ACM SIGMOD*, pp. 245-256, 1995.

[3] M.J. Carey, R. Jauhari, and M. Livny, "On Transaction Boundaries in Active Databases: A Performance Perspective," *IEEE Trans. Knowledge and Data Eng.*, vol. 3, no. 3, pp. 320-336, Sept. 1991.
[4] D. Chen and A.K. Mok, "SRDE-Application of Data Similarity to Process Control," *Proc. 20th IEEE Real-Time Systems Symp.*, Dec. 1999.
[5] U. Dayal, "The HIPAC Project: Combining Active Databases and Timing Constraints," *SIGMOD Record*, vol. 17, no. 1, pp. 51-70, Mar. 1988.
[6] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1992.
[7] J.R. Haritsa and M.J. Carey, and M. Livny, "On Being Optimistic About Real-Time Constraints," *Proc. Ninth SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, Apr. 1990.
[8] J.R. Haritsa, M.J. Carey, and M. Livny, "Earliest Deadline Scheduling for Real-Time Database Systems," *Proc. Real-Time Systems Symp.*, pp. 232-242, Dec. 1991.
[9] J.R. Haritsa, M.J. Carey, and M. Livny, "Data Access Scheduling in Firm Real-Time Database Systems," *J. Real-Time Systems*, vol. 4, no. 3, pp. 203-241, 1992.
[10] J. Huang, J.A. Stankovic, K. Ramamritham, and D. Towsley, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes," *Proc. 17th Conf. Very Large Databases*, pp. 35-46, Sept. 1991.
[11] J. Huang, J.A. Stankovic, D. Towsley, and K. Ramamritham, "Experimental Evaluation of Real-Time Transaction Processing," *Proc. Real-Time Systems Symp.*, pp. 144-153, Dec. 1989.
[12] J. Huang, J.A. Stankovic, K. Ramamritham, D. Towsley, and B. Purimetla, "On Using Priority Inheritance in Real-Time Databases," *Special Issue of Real-Time Systems J.*, vol. 4. no. 3, Sept. 1992.
[13] T. Kuo and A.K. Mok, "SSP: A Semantics-Based Protocol for Real-Time Data Access," *Proc. IEEE 14th Real-Time Systems Symp.*, Dec. 1993.
[14] T. Kuo and A.K. Mok, "Real-Time Data Semantics and Similarity-Based Concurrency Control," *Proc. IEEE Trans. Knowledge and Data Eng.*, 1995.
[15] S. Ho, T. Kuo, and K. Mok, "Similarity- Based Load Adjustment for Static Real-Time Transaction Systems," *Proc. 18th Real-Time Symp.*, 1997.
[16] Y. Lin and S.H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *Proc. Real-Time Systems Symp.*, pp. 104-112, Dec. 1990.
[17] M. Livny, *DeNet Users Guide.* version 1.5, Dept. Computer Science, Univ. of Wisconsin, Madison, WI, 1990.
[18] D. Locke, "Real-Time Databases: Real-World Requirements," *Real-Time Database Systems: Issues and Applications*, eds., A. Bestavros, K.-J. Lin, and S.H. Son, Kluwer Academic, pp. 83-91, 1997.
[19] E. McKenzie and R. Snodgrass, "Evaluation of Relational Algebras Incorporating the Time Dimension in Databases," *ACM Computing Surveys*, vol. 23, no. 4, pp. 501-543, Dec. 1991.
[20] H. Pang, M.J. Carey, and M. Livny, "Multiclass Query Scheduling in Real-Time Database Systems," *IEEE Trans. Knowledge and Data Eng.*, vol. 7, no. 4, Aug. 1995.
[21] B. Purimetla, R.M. Sivasankaran, J. Stankovic, and K. Ramamritham, "Network Services Databases-A Distributed Active Real-Time Database (DARTDB) Applications," *Proc. IEEE Workshop Parallel and Distributed Real-time Systems*, Apr. 1993.
[22] K. Ramamritham, "Real-Time Databases," *Distributed and Parallel Databases*, vol. 1, pp. 199-226, 1993.
[23] R.M. Sivasankaran, J.A. Stankovic, D. Towsley, B. Purimetla, and K. Ramamritham, "Priority Assignment in Real-Time Active Databases," *Int'l J. Very Large Data Bases*, vol. 5, no. 1, Jan. 1996.
[24] J.A. Stankovic, S. Son, and J. Hansson "Misconceptions about Real-Time Databases," *IEEE Computer*, vol. 32, no. 6, pp. 29-36, June 1999.
[25] X. Song and J.W.S. Liu, "Maintaining Temporal Consistency: Pessimistic versus Optimistic Concurrency Control," *Proc. IEEE Trans. Knowledge and Data Eng.*, vol. 7, no. 5, pp. 786-796, Oct. 1995.
[26] M. Xiong, J. Stankovic, K. Ramamritham, D. Towsley, and R.M. Sivasankaran, "Maintaining Temporal Consistency: Issues and Algorithms," *Proc. First Int'l Workshop Real-Time Databases*, Mar. 1996.
[27] M. Xiong, R.M. Sivasankaran, J. Stankovic, K. Ramamritham, and D. Towsley, "Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics," *Proc. IEEE 17th Real-Time Systems Symp.*, pp. 240-251, Dec. 1996.
[28] H. Zhou and F. Jahanian, "Real-Time Primary-Backup (RTPB) Replication with Temporal Consistency Guarantees," *Proc. Int'l Conf. Distributed Computing Systems*, May 1998.

Ming Xiong received the BS degree in computer science and engineering in 1990 from Xi'an Jiaotong University, the MS degree in computer science in 1993 from Sichuan University, and the PhD degree in computer science in 2000 from the University of Massachusetts, Amherst. He is currently a member of the technical staff at Bell Laboratories, Lucent Technologies. His research interests include database systems and real-time systems. He is a member of IEEE and ACM.

Krithi Ramamritham received the PhD degree in computer science from the University of Utah and then joined the University of Massachusetts. He has a visiting position at the Indian Institute of Technology Bombay as the Verifone chair professor in the Department of Computer Science and Engineering. He was a science and engineering research council (U.K.), a visiting fellow at the University of Newcastle upon Tyne, U.K, and has held visiting positions at the Technical University of Vienna, Austria and at the Indian Institute of Technology Madras. His interests span the areas of real-time systems, transaction processing in database systems, and real-time databases systems. He is applying concepts from these areas to solve problems in mobile computing, e-commerce, intelligent Internet, and the Web. He is a fellow of the IEEE and a fellow of the ACM. He has served on the Real-Time Systems Symposium as program chair in 1994 and as general chair in 1995, the conference on data engineering as vice-chair in 1995 and 2001, and the conference on management of data as program chair in 2000. He has also served on numerous program committees of conferences and workshops devoted to databases as well as real-time systems. His editorial board responsibilities include *IEEE Transactions on Parallel and Distributed Systems*, the *Real-Time Systems* Journal and ACM SIGMOD's Digital Review. He has coauthored two IEEE tutorial texts on real-time systems, a text on advances in database transaction processing, and a text on scheduling in real-time systems.

John A. Stankovic received the PhD degree from Brown University. He is the BP America professor and chair of the Computer Science Department at the University of Virginia. He is a fellow of both the IEEE and the ACM for research contributions in distributed and real-time computing. He also serves as an elected member on the board of directors of the Computer Research Association and has recently been elected its treasurer. Before joining the University of Virginia, he taught at the University of Massachusetts. At the University of Massachusetts, he won an outstanding scholar award. His PhD thesis was chosen as one of the best CS PhD thesis and published as a book. He has also received the IEEE Technical Committee on Real-Time Systems award for outstanding technical contributions and leadership. He has also held visiting positions in the Computer Science Department at Carnegie-Mellon University, at INRIA in France, and Scuola Superiore S. Anna in Pisa, Italy. He has served as the chair of the IEEE technical committee on Real-Time Systems. He has also served as an IEEE Computer Society distinguished visitor, has given distinguished lectures at various Universities, and has been a keynote speaker at various conferences. He is the editor-in-chief of the *Real-Time Systems* Journal. His research interests are in distributed computing, real-time systems, using feedback control in real-time systems, developing a toolset for composing embedded and real-time systems, operating systems, and distributed multimedia database systems. He has extensively published archival journal articles and conference papers.

Don Towsley received the BA degree in physics in 1971 and the PhD degree in computer science in 1975 from the University of Texas. From 1976 to 1985, he was a member of the faculty of the Department of Electrical and Computer Engineering at the University of Massachusetts, Amherst. He is currently a distinguished professor at the University of Massachusetts in the Department of Computer Science. He has held visiting positions at the IBM T.J. Watson Research Center, Yorktown Heights, New York (1982-1983); Laboratoire MASI, Paris, France (1989-1990); INRIA, Sophia-Antipolis, France (1996); and AT&T Labs—Research, Florham Park, New Jersey (1997). His research interests include networks, multimedia systems, and performance evaluation. He currently serves on the editorial board of *Performance Evaluation* and has previously served on several editorial boards including those of the *IEEE Transactions on Communications* and *IEEE/ACM Transactions on Networking*. He was a program cochair of the joint ACM Sigmetrics and Perfromance '92 conference. He is a member of ACM, ORSA, and the IFIP Working Groups 6.3 and 7.3. He has received the 1998 IEEE Communications Society William Bennett Paper award and three best conference paper awards from ACM Sigmetrics. Finally, he has been elected fellow of both the ACM and IEEE.

Rajendran Sivasankaran received the MS degree in computer science from the University of Texas. He is a doctoral candidate at the University of Massachusetts at Amherst. He worked as a senior member of technical staff at Verizon labs (previously GTE) working on SuperPages and Shopping Engine, a major private label yellow pages service provider to properties including AOL and Lycos. His work in SuperPages has led to two patent filings and recognition in the form of the Leslie Warner award, GTE's highest technical award. He has more than a dozen publications in journals, conferences and workshops, and has also written two book chapters. He is currently the Vice President of Engineering at Knumi Inc., a start-up that he cofounded in 2000.