Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics*

Ming Xiong, Rajendran Sivasankaran, John A. Stankovic, Krithi Ramamritham & Don Towsley
Department of Computer Science
University of Massachusetts
Amherst MA 01003
{xiong, sivasank, stankovic, krithi, towsley}@cs.umass.edu

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 non-waiting transaction scheduling policies are proposed. Informally, data-deadline can be viewed as the deadline that a transaction implicitly gets 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.*

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, and program stock trading [18]. While considerable work has been done on real-time databases, most of it assumes that only transactions have deadlines [1, 5, 6, 7, 8, 9, 10, 13, 16, 19].

The main contributions in this 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 that a transaction implicitly gets 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.
- The development of priority assignment policies that account for transaction deadlines and data time constraints, by using the idea of data-deadline. This includes the development of 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.
- 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.

*This work was supported, in part, by the National Science Foundation grant IRI-9114197.

The remainder of this 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.

2. Related Work

Over the past few years real-time databases have become important areas of research. Experimental studies reported [1, 5, 6, 7, 8, 9, 10, 13, 16, 19] 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 [20, 11, 12]. 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 [20]. In the model introduced in [20], 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 [20] 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 [20]. 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.

In [11], 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 [12], 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 to improve the performance of transaction scheduling.

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

A firm real-time database system is considered where transactions that have missed their deadlines add no value to the system and are, hence, aborted. 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 non-temporal. 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 is no temporal validity interval associated with non-temporal objects. Two approaches to modeling temporal data have been proposed in the literature [15] : 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, object versioning which maintains multiple versions of each objectis used. 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 non-temporal 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 absolute consistency which is introduced in the next subsection.

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, not only must the data read by transactions be fresh, but also be temporally correlated. This leads to the notion of temporal consistency which consists of two components [20]: absolute consistency and relative consistency. To define temporal consistency formally, the attributes of a temporal data object X are defined first:

X_i	:	the ith version of data object X
$avi_b(X_i)$	:	the begining of the absolute validity interval of X_i;
$avi_e(X_i)$	:	the end of the absolute validity interval of X_i;
$avi(X_i)$	:	the absolute validity interval of X_i; $avi(X_i) = [avi_b(X_i), avi_e(X_i)]$ where $avi_b(X_i) \leq avi_e(X_i)$.

Now the ith version of data object X, X_i (i=1,2,...), is defined as

$$(value(X_i), avi(X_i))$$

where $value(X_i)$ denotes the ith state of X, and $avi(X_i)$ denotes $value(X_i)$'s *absolute validity interval*, i.e., the time interval during which $value(X_i)$ is considered to be absolutely consistent. After $avi_e(X_i)$, $value(X_i)$ is no longer valid[1].

Consider a relative consistency set R in which each element is a version of a different temporal object. Associated with this set R is a *relative validity interval* denoted by $rvi(R)$. R has a correct state at time $t > 0$, if and only if

1. $\forall X_i \in R, value(X_i)$ is logically consistent - satisfies all integrity constraints.
2. R is temporally consistent:
   (a) $\forall X_i \in R$, $value(X_i)$ is absolutely consistent, i.e.,
   $avi_b(X_i) \leq t \leq avi_e(X_i)$.
   (b) R is relatively consistent, i.e.,
   $\forall X_i, Y_j \in R, \mid avi_b(X_i) - avi_b(Y_j) \mid \leq rvi(R)$.

In the rest of the paper, when it is mentioned that a transaction T reads a data object X at time t, it is understood that T reads a version of X that is absolutely consistent at time t.

Therefore, a transaction in our real-time database can commit if and only if

1. it is logically consistent,
2. it meets its deadline, and
3. it reads temporally consistent sets of 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.

[1] This does not consider *data similarity* which is introduced in section 4.4.

3.3. Concurrency Control for Temporal Objects

In the current model, 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.

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 the future work. All the sensor transactions are scheduled by 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 Figure 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

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

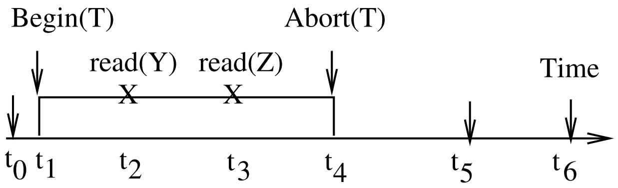

Figure 1. **An Illustration of Data-Deadline**

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 occurs later than time t_5. Next, transaction T proceeds and reads data Z with a value which is 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
$L(T)$	:	the number of method invocations [3] 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 non-temporal 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:

$$dd_t(T) \;=\; \min_{X \in RS_t^{to}(T)} avi_e(X)$$

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.

Conflict resolution for concurrency control is *priority abort*, where the conflicting transaction with lower priority waits or gets aborted depending on whether it is the requester or holder of locks, respectively.

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 commit. 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 Figure 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 for a transaction to commit all the data objects it accessed have to be 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.

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:

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

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)
- *Data-Deadline based Least Slack First with FWE* (DDLSF-FWE)

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

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

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 proceeds. 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. If condition 1 is false then 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 scheduler removes the highest priority transaction from CPU_Queue to execute. The following algorithm summarizes the FWR policy:

Algorithm : Condition checking in FWR policy

```
if (t + R_t(T) <= avi_e(X)) then
    T Continues;
else if (t + E_t(T) < avi_e(X)) then
    Place T in Sleep_Queue;
else
    Place T in Wait_Queue;
end;
```

FWR 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 FWR, 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 non-temporal data object. Thus,

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

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)
- *Data-Deadline based Least Slack First with FWR* (DDLSF-FWR)

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 by transactions reading the data. This is the basic concept of *data similarity* that has been explored in detail in [11, 12]. Similarity is based on the idea that, past versions 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 [11, 12], 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 avi is extended as follows[4]:

$$avi(X_i) \;=\; [avi_b(X_i), avi_e(X_{i+1})]$$

In Figure 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 the version of Z before and after t_4 are similar then T can commit after t_4. Again, EDF, EDDF, LSF, and DDLSF policies do not use the similarity semantics of temporal data objects that a transaction accesses. It is reasonable to combine the notion of data similarity with data-deadline. With similarity, whenever a transaction misses its data-deadline but not its transaction deadline because a data object that it read becomes invalid (because it is updated), 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.

However, similarity and forced wait are not orthogonal issues. Actually, they compete with each other. For example, when a transaction accesses a temporal object which is going to be invalid, forced wait policy might decide to wait for the next version. However, if the next version of the data is similar, the wait might have been unnecessary, i.e., if the transaction proceeds without waiting, the similarity of the next version to this version might allow the transaction to meet its data-deadline. For instance, in Figure 1, the data-deadline of T becomes t_4 once it reads data object Z. If T realizes that it cannot commit before t_4, T will wait until t_4 and get the new value of Z with forced wait policy. But, if the next version of Z is similar to the current version, then T's wait might have been unnecessary. This delay might result in a deadline miss.

[4] Based on symmetry of data similarity, the $avi(X_{i+1})$ can also be extended to become $[avi_b(X_i), avi_e(X_{i+1})]$. However, this does not help extend data-deadlines.

5. Experimental Evaluation of the Protocols

This section presents the experimental setup and the assumptions made in the experiments. A table of important parameters and their values is presented. Finally, each set of experiments and an analysis of the results are discussed. All our experiments are conducted for 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**), i.e., the percentage of 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 transactions that miss deadlines, and $N_{succeed}$ denote the number of 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.

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

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

A simulator, called $RADEx$ [19], was developed to perform our experiments. It was implemented using the DeNet Simulation Language [14]. In the experiments, 90% confidence intervals have been obtained whose widths are less than $\pm 10\%$ 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 in number of method invocations:

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

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:

$$E_t(T) \quad = \quad L_t(T) * CPU\ time\ per\ method$$

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 the period.

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
avi	Absolute validity interval
rvi	Relative validity interval

Table 1 : System Parameters.

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

Table 2 : Transaction Parameters.

The parameter *load* used in our experiments is very similar to the one in [19]. 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 Poisson processes with mean interarrival times 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, respectively. 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. 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.

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

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

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

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

System settings are controlled by the parameters listed in Table 1. In our experiments, the similarity relationship is not transitive. Transaction characteristics are controlled by the parameters listed in Table 2. Tables 3 and 4 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 avi of the data object that it updates.

Parameter	**Setting**
N_{cpu}	2
N_{to}	50
N_{nto}	500
P_{sim}	0.0
avi	$U(40, 200)$
rvi	$U(100, 200)$

Table 3 : Default System Parameter Settings.

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

Table 4 : Default Transaction Parameter Settings.

With the parameter settings in Tables 3 and 4, the load from sensor transactions is about 20% when the system is fully loaded, i.e., $load = 1.0$. 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.

5.2. Data-Deadline Based Policies

In the first set of experiments the performance of policies that account only for the data-deadline, i.e., EDDF and DDLSF is compared to the performance of the baseline policies EDF and LSF. The results are presented in Figure 2. 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% 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% . 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 plotted 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 is higher than that of *deadline* 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 improves performance only marginally.

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 Figures 3, 4, and 5, respectively. Figure 3 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%, EDF-FWE and EDDF-FWE by about 3% 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 Figure 4, 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. However, further investigation is necessary to find out exactly how the forced wait policy interacts with basic *deadline* and *slack* based policies.

In Figure 5, 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 Figure 5, for *deadline* based policies (dotted line in the figure), the performance gain with forced wait policies increases sharply as load increases. For instance, when the load is 0.9, EDDF-FWR outperforms EDDF-FWE by 7% while EDDF-FWE outperforms EDDF by more than 15%. As can be seen from Figure 6, when the load is 0.9, forced wait policies reduce the data-deadline abort ratio by more than 10% with respect to the policies without forced wait. For *slack* based policies, the addition of forced wait also improves performance significantly. But, as can be seen from Figure 5, 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 only similar to its previous version. The results are presented in Figures 7 and 8. From those graphs, it is observed that similarity does not affect the conclusion drawn from Figure 2. 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 Figure 9, 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 Figure 9.

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 Figure 10. 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% when the load is 0.9. But, compared to Figure 5, the performance gain of EDDF-FWE over EDDF is reduced from 14% to 5%. 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 by varying similarity probability. The results are presented in Figures 11 and 12. 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 both forced wait policies are relatively stable with different similarity probability. That is, not much difference in their performance can be observed from the graph as the similarity probability is varied. This shows that forced wait policies play a dominant role in the performance when both forced wait and similarity policies are used. Figure 13 compares the performance of variations of DDLSF policy. P_{sim} is set to 1.0 for policies with similarity. It is 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, but not as much as 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 5.4. 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 in forced wait policy.
- Exploiting similarity in transaction scheduling can reduce data-deadline aborts. Therefore, it leads to better performance when forced wait policy is not applied. But when combined with forced wait, data similarity does not significantly impact performance. However, when estimates of execution (response) time is 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.

Some of the extensions to be addressed in the future include:

- Using transaction savepoints to roll back transactions when a transaction misses its data-deadline but not its transaction deadline. This will save part of the work the transaction has done and avoid recovery overhead.
- Performing sensor updates on demand. That is, they need not always be given higher priority than user transactions. Using priority inheritance between user transactions and sensor transactions can achieve on demand updates of temporal objects. This can result in better performance for user transactions.
- Extending the protocols to include consideration related to relative consistency.
- In distributed real-time database systems, data temporal consistency is even harder to maintain than centralized systems. 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.

References

[1] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *ACM Transactions on Database Systems*, Vol. 17, No. 3, pp. 513-560, September 1992.

[2] B. Adelberg, H. Garcia-Molina and B. Kao, "Applying Update Streams in a Soft Real-Time Database System," *Proceedings of the 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 Transactions on Knowledge and Data Engineering*, Vol. 3, No. 3, pp. 320-336, September 1991.

[4] U. Dayal et. al., "The HIPAC Project: Combining Active Databases and Timing Constraints," *SIGMOD Record*, Vol. 17, No. 1, pp. 51-70, March 1988.

[5] J.R. Haritsa, M.J. Carey and M. Livny, " On Being Optimistic about Real-Time Constraints," *Proc. of 9th SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, April, 1990.

[6] J.R. Haritsa, M.J. Carey and M. Livny, "Earliest Deadline Scheduling for Real-Time Database Systems," *Proceedings of the Real-Time Systems Symposium*, pp. 232-242, December 1991.

[7] J.R. Haritsa, M.J. Carey and M. Livny, "Data Access Scheduling in Firm Real-Time Database Systems," *The Journal of Real-Time Systems*, Vol. 4, No. 3, pp. 203-241, 1992.

[8] J. Huang, J.A. Stankovic, D. Towsley and K. Ramamritham, "Experimental Evaluation of Real-Time Transaction Processing," *Real-Time Systems Symposium*, pp. 144-153, December 1989.

[9] J. Huang, J.A. Stankovic, K. Ramamritham and D. Towsley, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes," *Proceedings of the 17th Conference on Very Large Databases*, pp. 35-46, September 1991.

[10] 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 Journal*, Vol. 4. No. 3, September 1992.

[11] T. Kuo and A. K. Mok, "SSP: a Semantics-Based Protocol for Real-Time Data Access," *IEEE 14th Real-Time Systems Symposium* , December 1993.

[12] T. Kuo and A. K. Mok, "Real-Time Data Semantics and Similarity-Based Concurrency Control," submitted to *IEEE Transactions on Knowledge and Data Engineering*, 1995.

[13] Y. Lin and S.H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *Proceedings of the Real-Time Systems Symposium*, pp. 104-112, December 1990.

[14] M. Livny, "*DeNet Users Guide*," version 1.5, Dept. Comp. Science, Univ. of Wisconsin, Madison, WI 1990.

[15] 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, December 1991.

[16] H. Pang, M.J. Carey and M. Livny, "Multiclass Query Scheduling in Real-Time Database Systems," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 4, August 1995.

[17] B. Purimetla, R. M. Sivasankaran, J.Stankovic and K. Ramamritham, "Network Services Databases - A Distributed Active Real-Time Database (DARTDB) Applications," *IEEE Workshop on Parallel and Distributed Real-time Systems*, April 1993.

[18] K. Ramamritham, "Real-Time Databases," *Distributed and Parallel Databases* 1(1993), pp. 199-226, 1993.

[19] R.M. Sivasankaran, J.A. Stankovic, D. Towsley, B. Purimetla and K. Ramamritham, "Priority Assignment in Real-Time Active Databases," *The International Journal on Very Large Data Bases*, Vol. 5, No. 1, January 1996.

[20] X. Song and J. W. S. Liu, "Maintaining Temporal Consistency: Pessimistic vs. Optimistic Concurrency Control," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 5, pp. 786-796, October 1995.

[21] M. Xiong, J. Stankovic, K. Ramamritham, D. Towsley and R. M. Sivasankaran, "Maintaining Temporal Consistency: Issues and Algorithms," *The First International Workshop on Real-Time Databases*, March, 1996.

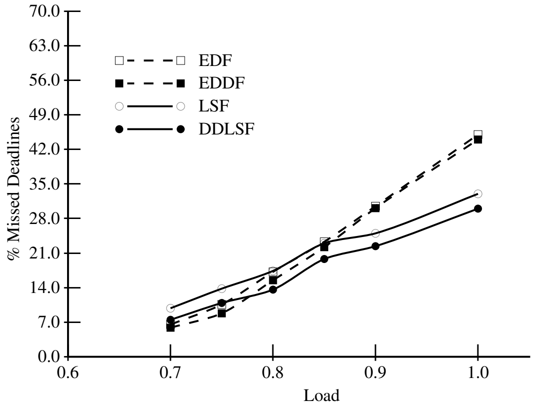

Figure 2. **Baseline and data-deadline policies**

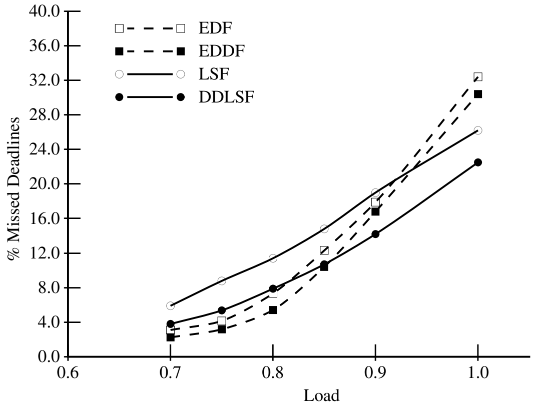

Figure 3. **FWE**

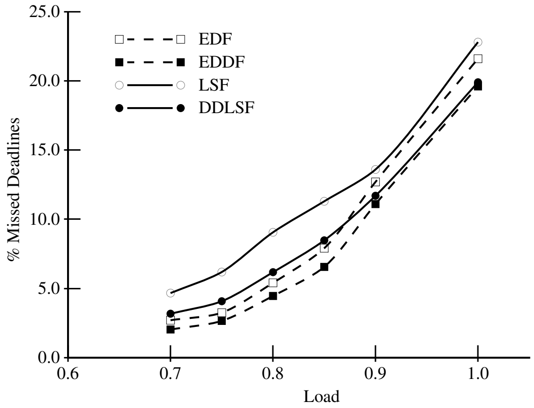

Figure 4. **FWR**

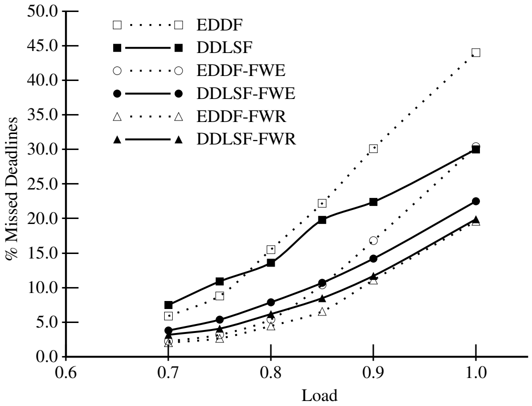

Figure 5. **No forced wait vs. FWE/FWR**

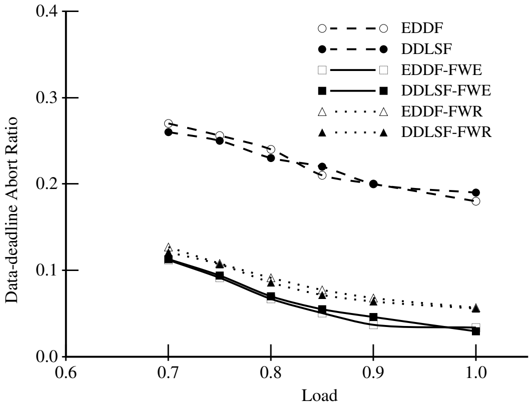

Figure 6. **No forced wait vs. FWE/FWR**

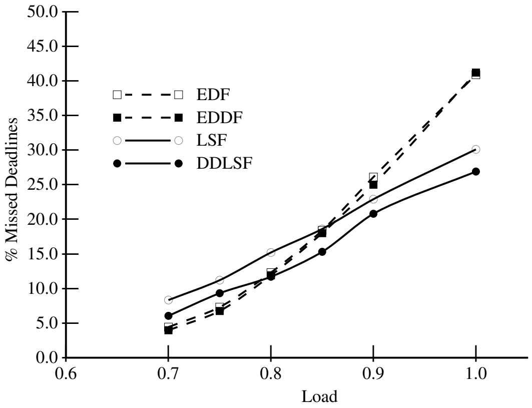

Figure 7. **Baseline and data-deadline policies** ($P_{sim} = 0.5$)

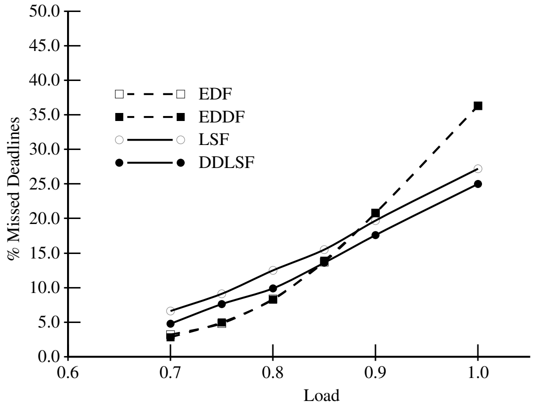

Figure 8. Baseline and data-deadline policies ($P_{sim} = 1.0$)

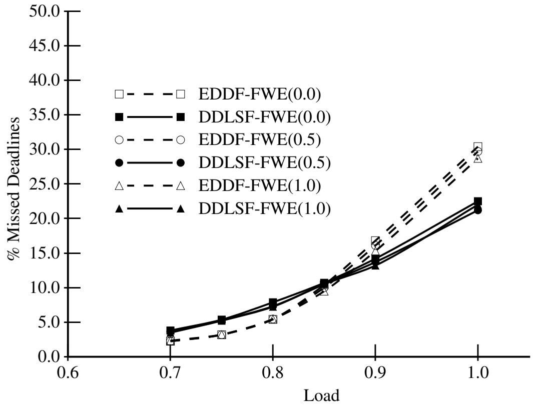

Figure 11. FWE with different P_{sim} (0.0, 0.5, 1.0)

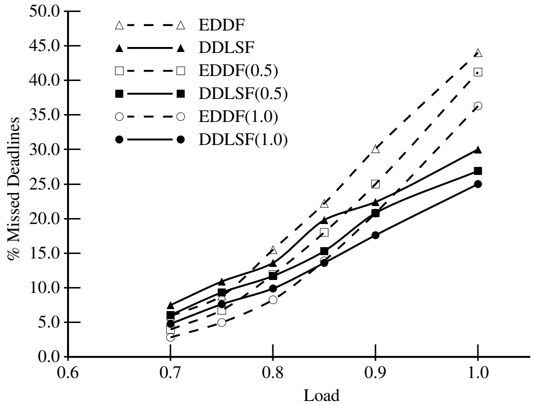

Figure 9. Using vs. not using similarity

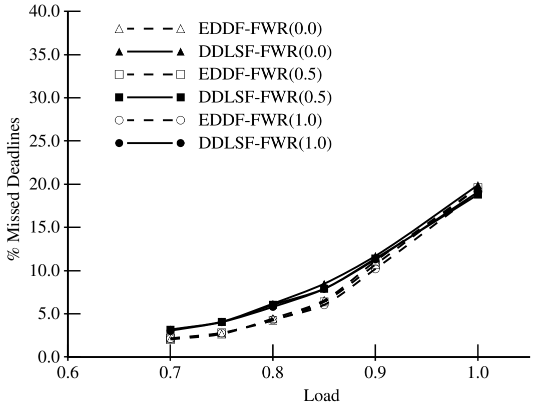

Figure 12. FWR with different P_{sim} (0.0, 0.5, 1.0)

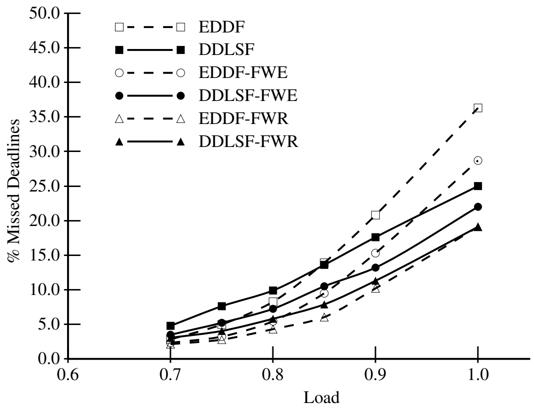

Figure 10. No forced wait vs. FWE/FWR ($P_{sim} = 1.0$)

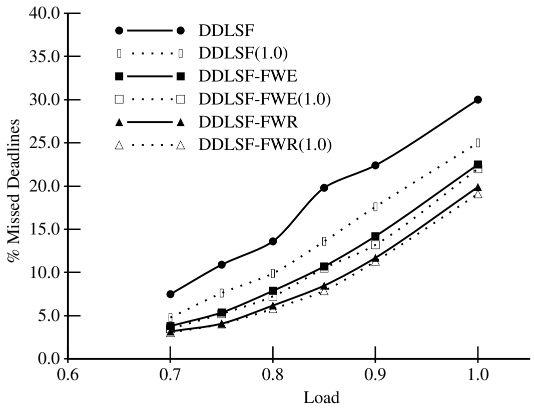

Figure 13. Variations of DDLSF