1

SCHEDULING ACCESS TO TEMPORAL DATA IN REAL-TIME DATABASES

Ming Xiong, Rajendran Sivasankaran, John A. Stankovic, Krithi Ramamritham and Don Towsley

Department of Computer Science, University of Massachusetts, Amherst, MA 01003, USA

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 [14]. While considerable work has been done on real-time databases, most of it assumes that only transactions have deadlines [1, 7]. New solutions that consider data time constraints are required for both concurrency control and cpu scheduling. In this chapter, new solutions are presented based on data-deadlines and time cognizant forced wait policies. 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. When a transaction attempts to access data that is about to expire, the forced wait policy delays it until the data is updated using new sensor inputs. Each of these is integrated with both earliest deadline first (EDF) and least slack first (LSF) cpu scheduling policies. It is shown that, while the data-deadline policy improves performance, the forced wait policy exhibits a more significant performance improvement. Under the conditions tested, it is also shown that LSF policies work better than EDF based policies.

The remainder of this chapter is organized as follows. Related work is discussed in Section 2. Section 3 describes the overall real-time database architecture. Section 4 discusses the main issues and solutions developed to maintain temporal consistency in a real-time database. Section 5 outlines the transaction scheduling policies employed in the solutions. Section 6 discusses the results of the experimental study and section 7 summarizes and concludes the study.

2 RELATED WORK

Over the past few years real-time databases have become an important area of research. Experimental studies reported in [1, 6, 7, 8, 12, 15, 16] cover many 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 were discussed in [17, 9, 10]. Such systems introduce the need to maintain data temporal consistency in addition to logical consistency. In these systems, transactions are aborted (and possibly restarted) when any of the temporal data objects they read become invalid before the transaction commits. Scheduling policies that are cognizant of transactions' deadlines but not the validity of data may not work well because of such data related considerations. Also, it is conceivable that a transaction will not be able to commit before the validity of a temporal data object that it accesses expires. In such a case, observations from our research indicate that waiting for the next valid version of the data object can save unnecessary aborts. The solutions presented here exploit this concept. In addition, sometimes, the version of the data object read by a transaction could be very close to the current version of the same data object (i.e., the two versions are *similar* [9, 10]) obviating the need to abort the transaction.

In the model introduced in [17], 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 [17] are greater than the absolute or relative thresholds allowed by the application. Transactions access temporally inconsistent data can commit if they can meet their deadlines. Two-phase locking and optimistic concurrency control algorithms, as well as rate-monotonic and earliest deadline first scheduling algorithms are studied in [17]. These studies show that the performances of the rate-monotonic and earliest deadline first algorithms are similar 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 [17] that it is difficult to maintain the data and transaction time constraints due to the following reasons:

1. A transient overload may cause transactions to miss their deadlines.
2. Data values may become out of date due to delayed updates.
3. 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.
4. 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 were motivated by these problems.

In [9], 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 [10], weaker consistency requirements based on similarity 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), in this chapter it is coupled with data-deadline and used to improve the performance of transaction scheduling.

3 SYSTEM MODEL AND CORRECTNESS

In real-time database systems, in addition to the conventional transaction correctness and logical data consistency criteria, transactions have to meet their deadlines and the data have to be temporally consistent. In this section, the transaction and data models which characterize the features of such real-time database systems are described. Also, the criteria for transaction correctness and data consistency in such real-time database systems are presented. The model considered is general enough to address one of the most important issues in real-time databases, that of maintaining data temporal consistency by accounting for data constraints while scheduling.

3.1 Data and Transaction Model

An active real-time *database* consists of a set of objects and a set of ECA (Event-Condition-Action)[5] rules. An *object* models a real world entity, for example, the position of an aircraft. The states of those real world entities are typically monitored by sensors. All the data objects in such a database can be characterized according to two orthogonal dimensions:

- *temporal objects* vs. *non-temporal objects*

  The objects in the database are either temporal objects or non-temporal objects. 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 the state of non-temporal objects.

- *base objects* vs. *derived objects*

  Base objects are those that reflect specific entities in the environment. They are updated by transactions that read sensors. Other transactions may derive new data objects from these base objects. Such objects are called derived objects. In an active real-time database system, a transaction may be triggered according to the ECA rules in the database when a base object is updated by a sensor transaction. The triggered transaction may update the states of derived objects whose states are derived from that base object.

An active real-time database system handles both triggering and triggered transactions. Each transaction can be either a read-only, a write-only, or an update transaction. There are three classes of transactions:

- **Sensor transactions**: These are transactions which update base objects.
- **Triggered update transactions**: These are transactions triggered by the updates of base objects. They update the derived objects.
- **User transactions**: These are user-level transactions with deadlines. They can be read-only, write-only or update (read-write) transactions.

The deadlines of sensor transactions and triggered update 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 next.

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. Data read by transactions must be not only fresh, but also temporally correlated. This leads to the notion of temporal consistency which consists of two components [17]: 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 beginning 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 we define the ith version of data object X, X_i (i=1,2,...), as

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

where $value(X_i)$ denotes the current state of data object 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 temporal data 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] This does not consider *data similarity* which is introduced in section 4.4.

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 chapter, when we say 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 valid 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.

4 ISSUES IN ENSURING DATA TEMPORAL CONSISTENCY

Maintaining temporal consistency raises many issues including data-deadline, exploiting the concept of forced wait, developing time cognizant concurrency control, determination of optimal period values for sensor transactions, version selection, handling bursty arrivals of sensor transactions, timely update of derived data, and using temporal data similarity. Among the issues that are discussed here, only data-deadline, forced wait and concurrency control are considered in our experimental study. The rest of the issues will be considered in our future work.

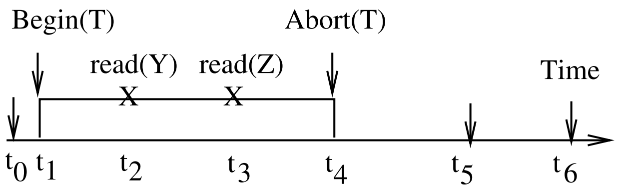

Figure 1 **An Illustration of Data-Deadline**

4.1 Data-Deadline

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 the *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 violates 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 notion of data-deadlines and aborts due to 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 data Z is valid in the interval $[t_0, t_4]$. Transaction T starts at time t_1, and T 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. Note that 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 versions of Y and Z to meet deadline t_6.

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

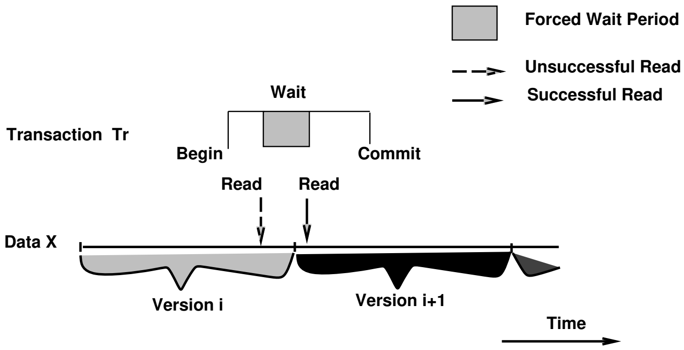

Figure 2 An Illustration of Forced Wait Protocol

4.2 Forced Wait

In many cases, estimates of 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 protocol is illustrated in Figure 2 where a transaction avoids reading the ith version of data X by waiting for the next version of X. 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 Concurrency Control

In our model, base 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 update transactions. When a sensor transaction writes a base object in each period, it creates a new version of the base 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 base objects. There may be multiple valid versions of an object in the database at one instant. As soon as a version becomes invalid, it is discarded. Thus, non-sensor based 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 base objects. To ensure that data is not read while it is being updated, latches (i.e., short term locks or semaphores) must be used. However, concurrency control is necessary for derived objects and non-temporal objects if data integrity is to be maintained.

4.4 Other Issues

Now consider the freshness, or absolute consistency of data in the database. Absolute consistency of data is provided by the periodic sampling of the environment, that is, by executing the sensor transaction that obtains data from a sensor and periodically updates the database. The issue here, as pointed out in [14], is how the period of the sensor transaction should be set. The reader is referred to [14] for a detailed discussion.

Another approach for the execution of sensor transactions is demand driven. When the value of a data item is needed, the corresponding sensor transaction is executed and the data value is installed. The data value obtained in this way is expected to be more accurate than the periodic approach if the delay to get a valid value of a data item can be ignored.

Since there may be more than one version of a temporal object in the database, the transaction should select a version to maintain the relative consistency of data it reads. If a transaction reads the most recent version of an object, it has more time to commit since this version has the longest validity. On the other hand, if the transaction reads the most recent version of an object, it may violate the relative consistency requirement. Consider the following opportunistic version selection approach: if X is the first data item in a relative consistency set read by T, then transaction T reads the most recent version of X; otherwise, transaction T reads the most recent version of X which satisfies the relative consistency requirement. An alternative approach is to predeclare the read set of a transaction and choose the versions of data that satisfy temporal consistency. However, neither of the approaches guarantees that a transaction can maintain temporal consistency. But the system is expected to exhibit better performance by such opportunistic approaches.

Overload can easily lead to temporal inconsistency. Consider n periodic sensor transactions $T_1, T_2, ..., T_n$ with periods $P_1, P_2, ..., P_n$, respectively. If they start *in phase*, i.e., the first periods of all the transactions begin at time $t = 0$, then the instances of those sensor transactions simultaneously start at $i * LCM(P_1, P_2, ..., P_n)$ $(i = 1, 2, ...)$ where $LCM(P_1, P_2, ..., P_n)$ is the least common multiple of $P_1, P_2, ..., P_n$. Sensor transactions along with user transactions may cause the system to be overloaded. Under overloads, one solution is to *update on demand*: Give higher priority to user transactions and lower priority to sensor transactions. When a user transaction needs to read an object and finds it invalid, the corresponding sensor transaction which updates the object inherits the priority of the user transaction and updates the object. The user transaction can then read valid data and proceed.

Since a base object and its corresponding derived objects are not updated by the same transaction, there may be a time interval during which the base object has already been updated but the derived objects, whose values depend on the base object have not yet been updated. If a user transaction attempts to read derived objects within that time interval, it may read invalid values. Since an update to a base object may trigger the updates of more than one derived object, and the value of one derived object may depend on more than one base object, an eager policy to update derived objects may cause considerable repetitive computation, and a lazy policy to update derived objects may cause user transactions to read invalid values. An alternative is to use an update on demand policy as discussed before. However, the drawback of this approach is its latency. Adelberg et. al. [3] study the intricate balance between recomputing derived data and transaction execution. They propose a policy which does not permit the recomputation of derived data until a fixed amount of time passes. Thus the recomputations triggered within that time can be batched together resulting in a reduction of the recomputation load.

Because an update to a base object may trigger updates to more than one derived object, two alternatives exist. One is to treat each update as a transaction. This increases transaction overhead although it permits the earlier release of data locks. The other is to combine all of the updates into one transaction, which can reduce transaction overhead. In this case, if the updated objects are not related to each other, an object lock can also be released as soon as the update is done.

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 [9, 10]. Similarity is based on the idea that, past versions of data that are not current but are 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 [9, 10], similarity is a binary relation on the domain of a data object. It is reflexive and symmetric, but not necessarily 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[3]:

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

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 will be 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. 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 aborts 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

[3] 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.

object which is about to become invalid, the 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.

5 SCHEDULING TRANSACTIONS IN REAL-TIME DATABASES

In the rest of the chapter, protocols dealing with temporal data accesses based on the ideas of data-deadline and forced wait are presented. For simplicity, the model used in our exerimental study has no triggered update transactions which update derived data since there are no derived data in the system. 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 our system, sensor transactions always receive higher priority than user transactions. Policies to assign priorities to user transactions based on their deadlines and the constraints on temporal data objects that they access are studied.

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 [4] 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:

$$dd_t(T) = \min_{X \in RS_t^{t_0}(T)} avi_e(X)$$

5.1 Baseline Scheduling Policies for CPU and Data Access

All the policies described in this section are preemptive dynamic priority policies that differ in the methods that they use 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. They are

- *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.

5.2 Data-Deadline Based Scheduling Policies

As mentioned earlier, EDF and LSF are not cognizant of data-deadlines. This can result in the aborting of transactions that have read data objects that become invalid. Variations of EDF and LSF that are cognizant of data-deadlines are now described:

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

5.3 Policies with Forced Wait

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:

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

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, 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:

```
if (t + R_t(T) ≤ avi_e(X)) then  /* condition 2 is true */
   T Continues;
else if (t + E_t(T) < avi_e(X)) then  /* condition 1 is true */
   Place T in Sleep_Queue;
else
   Place T in Wait_Queue;
end;
```

Algorithm : Condition checking in FWR policy

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 by the 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)

6 EXPERIMENTAL EVALUATION OF THE PROTOCOLS

This section presents the experimental setup and the assumptions made in our experiments. A simulator, called *RADEx* [15], was developed to perform our experiments. It was implemented using the DeNet Simulation Language [11]. A table of important parameters and their values is presented. Each set of experiments and an analysis of the results are presented. All the 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 reasonable metric to evaluate performance in firm real-time database systems. The goal of the scheduling policies that are presented is to minimize the number of transactions that miss their deadlines. This does not guarantee that all transactions will meet their deadlines as in hard real-time systems. Hard real-time systems are not considered because database systems for efficiently supporting hard deadline applications appear infeasible due to large variance between the average case and worst case execution times of a typical database transaction [16]. We also expect that understanding scheduling issues in firm deadlines will provide a foundation for addressing more complex real-time systems with hard and soft deadlines.

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.

Useful CPU utilization is also measured to evaluate different policies. Let N_{CPU} denote the number of CPUs, CPU_{total} denote the total CPU time used, and CPU_{wasted} denote the wasted CPU time by aborted transactions in a simulation. Useful CPU utilization is calculated as follows:

$$Useful\ CPU\ Utilization \quad = \quad \frac{CPU_{total} \; - \; CPU_{wasted}}{N_{CPU} \times \ Simulation\ \ Time} \times 100$$

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

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

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).

6.1 Baseline Parameters

The workload model is introduced as follows. The estimated remaining execution time of transaction T at time t, $E_t(T)$ is:

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

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**	**Setting**
N_{CPU}	Number of CPUs	2
N_{to}	Number of temporal object instances	50
N_{nto}	Number of non-temporal object instances	500
avi	Absolute validity interval	U(40,200)
rvi	Relative validity interval	U(100,200)

Table 1 : System Parameters and Default Settings.

Parameter	**Meaning**	**Setting**	
		ST	UT
MethodTime	CPU time per method	1.0	1.0
Slack	User transaction slack		U(8.0,12.0)
Period	Sensor transaction period	U(40,200)	
P_{comp}	Probability of method compatibility	0.0	0.0
Length	Number of method invocations	1	U(6,12)
$Prob_{to}$	Probability to access temporal objects	1.0	0.4

Table 2 : Transaction Parameters and Default Settings.

The parameter *load* that is used in our experiments is very similar to the one in [15]. Let **UT** and **ST** stand for the classes of user transactions and sensor transactions. In order to define *load* the arrival rates and service rates of **UT** and **ST** transactions are specified. The arrivals of **UT** transactions are generated according to a Poisson process with a mean interarrival time of $1/\lambda_{UT}$ time units, and the arrivals 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, *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 sensor 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 ($load = 1.0$), $frac_{ST}$ is

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

System settings are controlled by the parameters listed in Table 1. Transaction characteristics are controlled by the parameters listed in Table 2. In Table 1 and 2, $U(i, j)$ denotes a uniformly distributed random variable in the range $[i, j]$. The *Period* of a sensor transaction is equal to the length of the *avi* of the data object that it updates.

With the parameter settings in Tables 1 and 2, 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 beginning of the simulation.

Two sets of experimental results are now presented: those evaluating data-deadline based policies and those evaluating forced wait policies. See [19] for additional experiments including the evaluation of similarity based protocols.

6.2 Data-Deadline Based Policies

The performance numbers referred to in the following discussion are absolute values of MDP. For example, whenever it is said that policy A is 10better than B (or reduces the MDP by 10improvement is 10

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 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% 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.

In Figure 4, the total and useful CPU utilizations of all policies are plotted. In the graph, T and U stand for total and useful utilization, respectively. The useful CPU utilizations for all policies are about the same at medium loads. 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 loads.

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.

6.3 Policies with Forced Wait

A 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 5, 6, and 7, respectively. Figure 5 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 6, 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 7, the performances of data-deadline based policies (EDDF and DDLSF) with different forced wait policies are compared. The comparisons of variations of EDF and LSF policies are similar and not repeated. As can be seen from Figure 7, 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 8, 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. This, in turn, increases the useful CPU utilization as seen in Figure 9 and leads to less MDP. The evidence for our hypothesis, that using extra information like the estimates of remaining execution time to avoid reading *infeasible validity interval* leads to less data-deadline aborts, is seen here. The estimate of remaining response time, which leads to a more accurate estimate for the completion time of a transaction, helps reduce CPU wastage, and improves the performance of user transactions significantly for *deadline* based policies, but only marginally for *slack* based policies as can be seen from Figure 7. But the addition of forced wait improves performance significantly for both *slack* and *deadline* based policies.

In Figure 10, the data contention is varied by changing the number of non-temporal objects in the database. Data contention is increased by shrinking the database, while maintaining a fixed workload. The results show that as database size decreases, the performance improvements with DDLSF-FWE and DDLSF-FWR decreases more sharply than EDDF-FWE and EDDF-FWR. With a priority abort conflict resolution policy, the number of transaction aborts due to concurrency control increases with the increase in data contention level. But this increase in transaction aborts is significantly more in the case of slack based policy than deadline based policy [15].

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

7 CONCLUSIONS

In this chapter, issues involved in designing a real-time active database which ensures data temporal consistency have been discussed. 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. As observed from the experiments, taking data-deadline into consideration for scheduling improves performance, but not as much as when it is combined with a forced wait policy. Using extra information such as estimates of execution times and response times enhances the performance of user transactions significantly. In the future, we intend to extend this work by considering the temporal consistency problem in distributed real-time database systems.

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] B. Adelberg, H. Garcia-Molina and B. Kao, "Database Support for Efficiently Maintaining Derived Data," Technical Report, Stanford University, 1995.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

[16] J. Stankovic and W. Zhao, "On Real-Time Transactions," *ACM SIGMOD Record*, Vol. 17, No. 1, 1988.

[17] 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.

[18] M. Xiong, J.A. 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.

[19] M. Xiong, R. M. Sivasankaran, J.A. Stankovic, K. Ramamritham and D. Towsley, "Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics," *17th IEEE Real-Time Systems Symposium*, December, 1996.

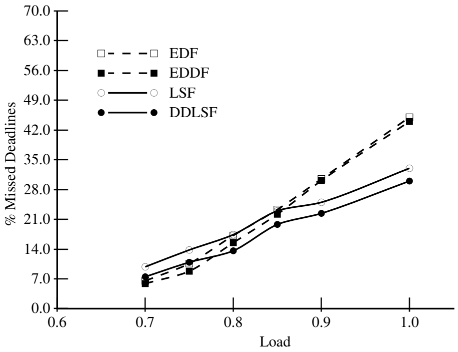

Figure 3 Baseline and data-deadline policies

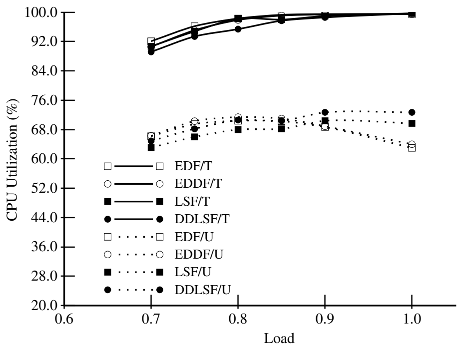

Figure 4 Total/Useful CPU utilization

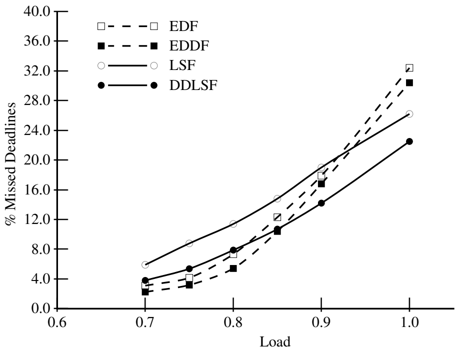

Figure 5 FWE

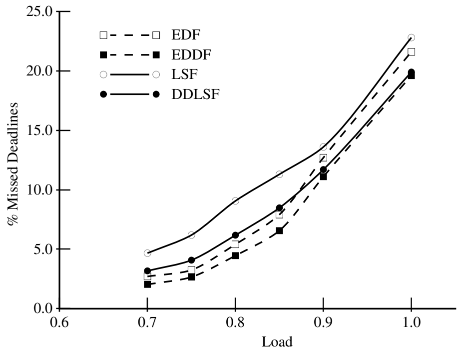

Figure 6 FWR

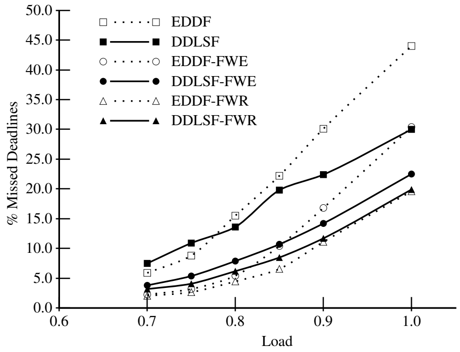

Figure 7 No forced wait vs. FWE/FWR

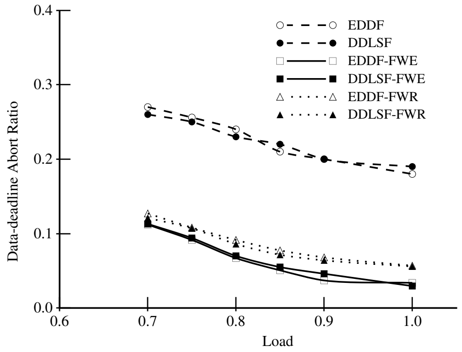

Figure 8 No forced wait vs. FWE/FWR

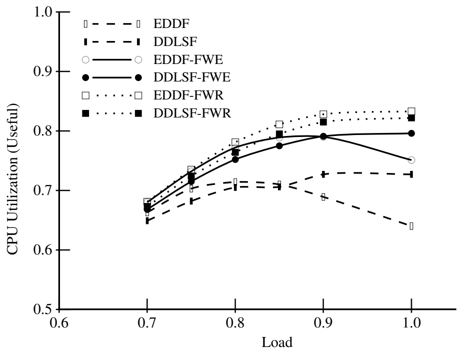

Figure 9 Useful CPU utilization

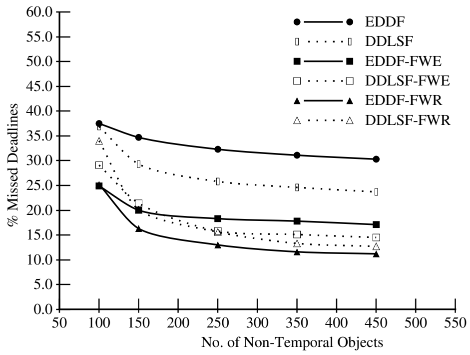

Figure 10 Varying database size