PRIORITY ASSIGNMENT IN REAL-TIME ACTIVE DATABASES

B. PURIMETLA, R.M. SIVASANKARAN,
J.A. STANKOVIC, K. RAMAMRITHAM and
D. TOWSLEY

CMPSCI Technical Report 94-29

April 1994

Priority Assignment in Real-Time Active Databases[1]

Bhaskar Purimetla, Rajendran M. Sivasankaran, John A. Stankovic,
Krithi Ramamritham & Don Towsley
Department of Computer Science
University of Massachusetts, Amherst, MA 01003

Abstract

Active databases and real-time databases have been important areas of research in the recent past. It has been recognized that many benefits can be gained by integrating active and real-time database technologies. However, there has not been much work done in the area of transaction processing in active real-time databases. This paper deals with an important aspect of transaction processing in active real-time databases, namely the problem of assigning priorities to transactions. In these systems time-constrained transactions trigger other transactions during their execution. We present various policies for assigning priorities to parent, immediate and deferred transactions executing on a multiprocessor system and then evaluate the policies using simulation. The simulator has been validated by comparing to three sets of published results. Our new results demonstrate that dynamically changing the priorities of transactions depending on their behavior (triggering rules), yields a substantial improvement in the number of triggering transactions that meet their deadline.

1 Introduction

Traditionally, in soft real-time transaction processing systems, a transaction is considered a monolithic unit of work with a given deadline. In these systems, priorities are assigned to these transactions and the transactions are scheduled based on their priorities. The priority assignment usually takes into account the deadline of the transactions, because the underlying assumption is that the deadline reflects the urgency of completing the transaction. Scheduling policies such as Earliest Deadline First (EDF) and Least Slack First (LSF) are examples of priority-cognizant policies. The performance implications of priority assignment policies have been studied in detail in soft real-time transaction processing systems in [1, 6]. In such systems, the priority assignment policies as well as conflict resolution policies must be time cognizant. In this paper, our goal is to study and evaluate priority assignment policies in a real-time active database. A real-time active database is a database system where transactions have timing constraints such as deadlines, where transactions may trigger other transactions, and where data may become invalid with the passage of time. There are many applications such as cooperative distributed navigation systems and intelligent network services where real-time active database technology is extremely useful [12, 14].

Before explaining the problem we are addressing in detail, we will give a brief introduction to active databases. The building block of an active database system is the Event-Condition-Action

[1] This work was supported in part by NSF under grants IRI-9114197 and IRI-9208920.

(ECA) rule. The semantics of the ECA rule is that if the specified event (E) occurs and if the condition (C) is true then the specified action (A) is to be executed. Some examples of events are begin[2], commit/abort of a transaction, accessing a data item, or reaching a specific point in time. A condition is usually a predicate on the database state. An action is the transaction that is executed in *reaction* to the specific *situation* which is the combination of events and the conditions. The transaction that fires the rules is called the triggering transaction and the action that is executed because of the rule firing is called the triggered transaction. We refer to the transactions that trigger other transactions as *active* transactions or *parent* transactions in this paper. An active transaction has a set of triggered transactions that are either executed as part of the active transaction or separately – depending on the type of the *coupling mode* between the parent and the triggered transactions [4]. There are three types of coupling modes. They are *immediate*, *deferred* and *independent* and the transactions triggered in those modes are referred to as immediate, deferred and independent transactions, respectively. The immediate and deferred transactions are executed as part of the parent transaction whereas the independent transactions are executed independently. An immediate transaction is executed as soon as it is triggered and the parent transaction is suspended until the immediate transaction is completed. A deferred transaction is executed after the parent finishes execution, but before it commits. The immediate and deferred transactions are committed only if the parent commits. Since the immediate and deferred transactions are part of the parent transaction, we also refer to them as subtransactions.

Due to the rule firings, an active transaction generates additional work dynamically. In a real-time context, for the time-cognizant scheduling policies to perform well, they should take into account the dynamic work that is being generated. This aspect of the problem makes this scheduling problem different from the classical hard real-time scheduling where execution times are assumed to be known in advance [8, 9, 15]. The priority driven nature of real-time transaction processing brings about the question of assigning priorities to the parent and all triggered actions [12, 14]. We believe that the priority assignment strategy has a significant impact on the performance of the system as triggered actions will contend with ongoing transactions for resources. In this paper, we address the problem of assigning priorities to triggering and the triggered transactions. We discuss three policies for assigning priorities to *immediate* and *deferred* transactions, given the triggering and triggered transactions' characteristics.

To illustrate the problem, we provide an example of the structure of a complex active transaction executing on a uniprocessor. We use figure 1 to illustrate our example. Since we use figure 1 later in our description of priority assignment protocols, it is more complicated than necessary for the problem illustration. Figure 1 shows the life of a complex active transaction T that triggers transactions in all three modes. Transaction T arrives at time $t1$ ($at(T)$) with $t10$ as its deadline ($dl(T)$) and is started at time $t2$ ($st(T)$). It triggers a deferred transaction $dt1$ at time t_3, and another deferred transaction

[2] The begin event may, in turn, have been caused by some external environment event such as an obstacle identified by a sensor [12, 14].

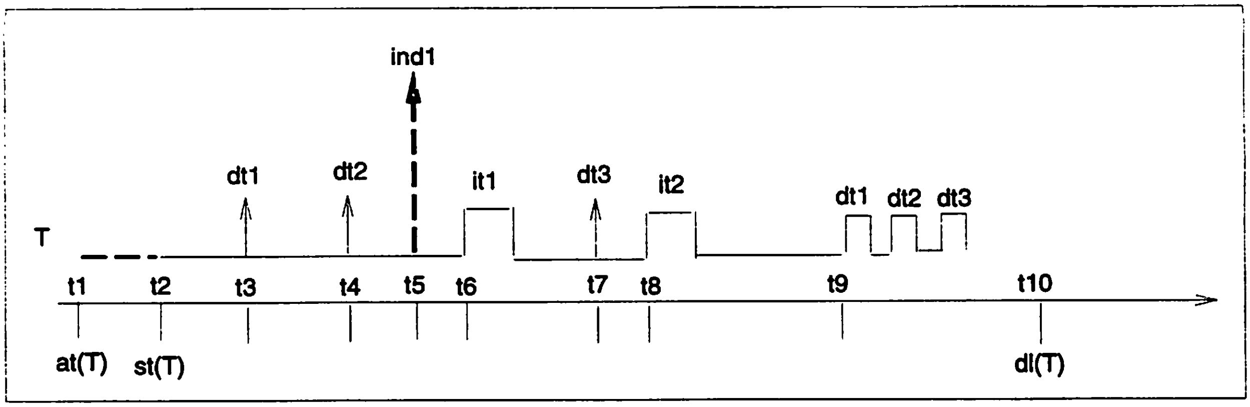

Figure 1: **Life of a Complex Active Transaction**

dt2 at time *t4*. It triggers an independent transaction *ind1* at *t5*, an immediate transaction *it1* at *t6*, another deferred transaction *dt3* at *t7* and an immediate transaction *it2* at *t8*. Figure 1 shows that the transactions *it1* and *it2* execute immediately while *T* is suspended. Finally, the figure shows that once the parent *T* completes, the deferred transactions *dt1*, *dt2* and *dt3* execute. The problem we address in this paper is the problem of assigning priorities to transactions *it1*, *it2*, *dt1*, *dt2* and *dt3*, and the problem of dynamically reassigning the priority of the parent transaction *T* during its lifetime. It should be noted that the deferred transactions can execute simultaneously if operating on a multiprocessor or a distributed system. We assign the priorities to triggered transactions when they start execution and they are not modified later.

Our main contributions are :

- developing priority assignment policies that take into account the dynamic work generated by active transactions;
- evaluating the priority assignment policies against a baseline policy using a real-time active database simulator in two settings, i.e., in a real-time task setting and a main memory database setting;
- demonstrating that for mixed workloads with triggering and non-triggering transactions, priority assignment policies that take into account the dynamic work generated reduce the deadline miss ratio of the triggering transactions significantly at the cost of a very small increase in the deadline miss ratio of non-triggering transactions compared to the baseline policy; and
- conducting experiments to identify the relative bias the policies show for the triggering transactions, thereby enabling an implementor to select from various policies depending on the relative importance of the triggering and non-triggering transactions in the system.

We discuss related work in Section 2. In Section 3, we explain our transaction model, system model and describe the various attributes of a transaction and other related terms. Section 4 gives a detailed explanation of the priority assignment policies. We discuss the experiments and results *in Section* 5 in detail. We conclude our paper in Section 6 with a summary of our main results and discuss future work.

2 Related Work

Over the past few years active databases and real-time databases have become important areas of research. There have been both theoretical and experimental studies in active databases [2], [3], [11], [4]. Experimental work on active databases in [2] is done in a non real-time setting. Most of the research done on active components in object oriented databases and real-time databases has concentrated on the specification of Event-Condition-Action (ECA) rules [5]. Experimental work done on real-time transaction processing systems [1, 6] has not considered active workloads. Experimental studies reported in [1] are very comprehensive and cover most aspects of real-time transaction processing, but have not considered active workloads and have not addressed the problem of subtransaction priority assignment. Experiments in [2] show the impact of transaction boundaries and data sharing on performance of active databases. We address a different problem, that of assigning priorities to triggered transactions in real-time active databases. In [13] the relationship between real-time databases and active databases is discussed briefly. Lack of active pursuit of timely processing of actions to do real-time processing was identified as a missing ingredient in active databases. In short, past studies on real-time transaction processing have not dealt with active workloads and studies on active databases have not dealt with real-time transactions.

Simulation studies have been conducted to study the problem of assigning deadlines to subtasks in real-time systems [7]. The problem of assigning deadlines to the parallel and serial subtasks of complex distributed tasks is addressed in [7]. The structure of the complex tasks is assumed to be known in advance. Also the experimental system considered in [7] is not a transaction processing system and does not have an active component. The system we consider for our experiments is an real-time active database with unpredictable data accesses and rule firings. In contrast with [7], in our study the structure of complex transactions is not known in advance.

3 The Model

In this section we describe the transaction model and the system model used to study the priority assignment policies in a real-time active database. We also define some key attributes of a transaction and other related terms.

3.1 Transaction Model

A transaction in our system is a series of method executions on objects. We consider two types of transactions in the system: *Triggering* (class **T**) and *NonTriggering* (class **NT**). A nontriggering transaction is a simple transaction which does not cause any rules to fire. A triggering transaction, on the other hand, can trigger subtransactions upon the occurrence of an event. All subtransactions are simple transactions. An event in our model is one of the following types of events: *Transaction Event*, *Object Event*, or *Temporal Event.* An object event occurs, whenever the transaction executes a method on an object instance, a transaction event occurs during ***begin***, ***commit*** or ***abort*** of a transaction and a temporal event occurs when a particular point in time is reached. The subtransaction that is triggered can be executed in one of the three modes: immediate, deferred or independent.

3.2 Attributes of a Transaction

In order to understand the details of priority assignment policies, we define the following attributes of a transaction T, and other terms that we require to explain our priority assignment policies.

$at(T)$	:	The arrival time of T
$st(T)$	:	The start time of T
$dl(T)$	:	The absolute deadline of T
$vdl_t(T)$	:	The virtual deadline of T at time t (this is used for scheduling)
$avgLen(T)$	:	Average length of T (average number of method invocations)
$lenLeft(T)$	:	Length Left (remaining number of method invocations based on *avgLen(T)* assumption)
$slack_t(T)$	:	The slack of T at time t
$wslack_t(T)$	:	The predicted worst case slack of T at time t
$eet(T)$	:	The estimated execution time of T
$ect_t(T)$	:	The estimated completion time of T at time t
$eetl_t(T)$	:	The estimated execution time left for T at time t
$wcet_t(T)$	:	The predicted worst case execution time of T at time t
$nDef_t(T)$	:	The number of deferred transactions triggered by T until time t
$nImm_t(T)$	:	The number of immediate transactions triggered by T until time t
$eetDef_t(T)$	:	The sum of estimated execution time of all deferred transactions triggered until time t
$probTrig_{obj}$	:	Probability of an object event triggering a subtransaction
$probTrig_{trans}$	:	Probability of a transaction event triggering a subtransaction

The $vdl_t(T)$, the virtual deadline of transaction t at time T is used to assign the transaction's priority by the EDF scheduler. To increase the priority of a transaction we assign a virtual deadline which is earlier than the absolute deadline $dl(T)$. Thus, it is not necessary to abort a transaction when its virtual deadline expires. $eet(T)$, estimated execution time information, is used by some priority assignment policies. $eetl_t(T)$, is the difference between the estimated execution time and the time taken to execute the transaction so far. The $wcet_t(T)$, includes the execution time left at time t as well as the worst case time to execute the subtransactions which the transaction T can trigger during its remaining lifetime. The predicted worst case slack $wslack_t(T)$ is obtained by

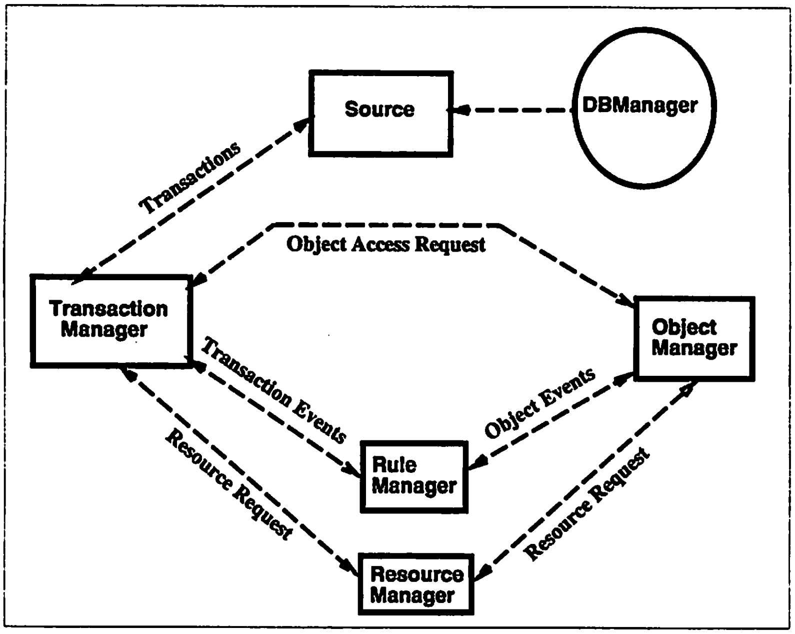

Figure 2: **Simulator Architecture**

subtracting the worst case execution time from the absolute deadline. The calculation of worst case execution time uses the value of $probTrig_{obj}$ and $probTrig_{trans}$, which are the probabilities that a subtransaction is triggered by an object event or a transaction event of the transaction, respectively. This information is used by some of the priority assignment policies. We believe that by analyzing the characteristics of an application one might be able to obtain information like execution time of transactions, the kind of actions they trigger and the probability of triggering. In our experiments, we assume that this information is known accurately. In future experimentation, we plan to study the implication of errors in this information, on the performance of the priority assignment policies.

The following equations show the relationship between the some of the terms defined above.

$$\begin{aligned} ect_t(T) &= clock + eetl_t(T) \\ slack_t(T) &= dl(T) - ect_t(T) \end{aligned}$$

where *clock* is the current clock value.

3.3 Simulation Model

Our performance model of an active real-time active database was implemented using the DeNet Simulation Language [10]. The simulator is made up of five active components : *Source*, *Transaction Manager*, *Object Manager*, *Resource Manager*, *Rule Manager*, and a passive component *DB Manager*. Figure 2 illustrates the architecture of the simulator. Following is a detailed description of the modules:

- *DB Manager*

  This is the passive module that models the data. The data is modeled as having a certain number of object classes and each object class having a certain number of instances. Each object class has a certain number of methods defined which are used to access the object. Each object instance in the database is mapped to a page or number of pages in the secondary storage.

- *Source*
  The source (transaction generator) generates the incoming transactions into the system. One can view the source as the application or the environment in which the Real-Time Active Database is used. It generates transactions with timing constraints with a specified arrival distribution. It can generate both periodic and aperiodic transaction streams and three types of real-time transactions namely hard, firm and soft. One of our future goals is to study the transaction characteristics of specific applications [12, 14] and design the generator to model it as closely as possible.

- *Transaction Manager*
  Transaction Manager is responsible for scheduling and execution of transactions it receives from the source. It executes the submitted transactions by requesting the Object Manager to execute the specific methods on specific objects. It handles the various transaction events: begin, commit and abort. It informs the Rule Manager of the transaction events.

- *Object Manager*
  Object Manager executes the methods of the objects in the system as requested by the Transaction Manager. It is also responsible for concurrency control. If the request can be executed it executes and sends the success message back to the Transaction Manager. If the request cannot be satisfied it either sends a blocked or abort message back to the Transaction Manager. It also informs the Rule Manager of the various object events that occur. Conflict resolution in this module is priority based where the lower priority conflicting transaction waits or gets aborted depending on whether it is the requester or holder of locks.

- *Rule Manager*
  The Rule Manager models the active workload in the system. The rule firings are modeled probabilistically, i.e., a rule is fired with a certain probability. The Rule Manager checks to see if any rules are triggered whenever it gets an event notice from the Transaction Manager or the Object Manager. It models the condition evaluation, and, finally it generates the transactions corresponding to the actions of the rules triggered if their conditions are satisfied and submits them to the Transaction Manager.

- *Resource Manager*
  The Resource Manager simulates the CPUs and disks and the main memory buffer. The Object Manager requests the Resource Manager for the necessary pages or for CPU time to execute the methods. The Transaction Manager requests the Resource Manager the CPU time and buffer

space to load transactions. The CPU, disk, and memory resource scheduling is priority driven. Our resource model is a multiprocessor, multi disk, shared memory system. The incoming resource requests are queued in a common CPU queue or a common disk queue depending on the kind of request.

The scheduling and conflict resolution decisions made in different modules of the simulator are independent of each other. No global scheduling decisions are made. For instance, the scheduling decision made in the Resource Manager is independent of the one made in the Transaction Manager. There are three types of overload management policies available in the simulator. In the *all eligible* policy all the transactions are run to completion. In the *not tardy* policy, a transaction is aborted as soon as its deadline expires. This corresponds to firm real-time transactions. This policy assumes that finishing a transaction after its deadline expires doesn't impart any value to the system. In *feasible deadline* policy, we check the feasibility of a transaction, i.e., if it can finish by its deadline based on the estimated execution time and abort it if it cannot finish by its deadline.

4 Priority Assignment for Triggered Transaction

In real-time active databases, a transaction with time constraints can trigger subtransactions. Traditionally, priority driven scheduling has been used in real-time systems. So the problem arises as to how to assign a priority to a subtransaction given the priority of the parent transaction. Another problem is that as a transaction triggers subtransactions either in immediate or deferred modes, the amount of work to be done on behalf of the transaction before it commits also increases. So the transaction is less likely to complete successfully compared to another transaction with the same deadline which doesn't trigger any subtransactions, since it faces more resource contention during its lifetime. In this section, we describe some policies for dynamically assigning priorities to the parent and *Immediate* and *Deferred* subtransactions in an active real-time database system. In all cases, the priorities are assigned to the triggered transactions when they start execution and priorities are not changed subsequently during their execution. The priority of the triggering transactions may change dynamically, depending on the policy.

4.1 Priority Assignment Policies for Immediate Subtransactions

Three priority assignment policies are studied for immediate subtransactions. The first two assignment policies assume that the scheduling discipline is Earliest Deadline First (EDF) and the third policy assumes that the scheduling discipline is Least Slack First(LSF). We use the active transaction T in figure 1 to explain our policies and discuss the priority assignment policies for the immediate transaction $it1$. We request the the reader to refer to section 3.2 to understand the semantics of the terms used in the following description.

1. ***PD***: Assign the same deadline as the parent deadline. The priority of the parent transaction which is based on its deadline does not change with the triggering of subtransactions. This is a very simple baseline algorithm. All the actions done on behalf of a transaction get the same priority as the transaction itself at any point during its lifetime.
   For example, at the triggering point, i.e., at time $t6$

$$
\begin{aligned}
dl(it1) &= dl(T) \qquad [1] \\
vdl_{t6}(it1) &= dl(T) \qquad [2]
\end{aligned}
$$

2. ***DIV***: Equally divide the parent's effective slack among all the immediate and deferred subtransactions triggered until that point. The parent's deadline is also adjusted dynamically to reflect the work that has been triggered dynamically. The sum of estimated execution time of deferred transactions that have been triggered until that point is subtracted from the parents slack to determine the effective slack.

$$
\begin{aligned}
vdl_{t6}(it1) &= ect_{t6}(it1) + \frac{slack_{t6}(T) - (eet_{t6}(it1) + eet_{t6}(dt1) + eet_{t6}(dt2))}{nDef_{t6}(T) + nImm_{t6}(T)} \qquad [3] \\
dl(it1) &= dl(T) \qquad [4] \\
vdl_{t6}(T) &= vdl_{t6}(T) - eet(it1) \qquad [5]
\end{aligned}
$$

   The main idea behind this policy, is that of giving higher priority to class **T** transactions, which have more work to do before completion. This will increase the probability of the class **T** transactions meeting their deadlines. This policy only uses the estimates of execution times of subtransactions that have already been triggered. It does not use any knowledge about future triggering of transactions.

3. ***SL***: Adjust the worst case slack ($wslack_t(T)$) of the parent at each potential triggering point. In this policy, the transaction with the smallest slack has the highest priority. The initial value of slack is assigned based on the *predictions* about the total execution time for a transaction and its subtransactions as indicated by $probTrig_{obj}$. The slack is then adjusted at each object or transaction event based on whether the parent transaction triggers a subtransaction or not. The triggered transactions are assigned the same slack as the parent, i.e., they are executed at the same priority. We assume that the transaction events do not trigger any subtransactions. This assumption holds for the rest of the paper and is explained later in more detail.
   Initially slack is set as follows:

$$
\begin{aligned}
wslack_{t6}(T) &= dl(T) - clock \\
&\quad - eet(T) - (avgLen(T) * probTrig_{obj} * eet(st)) \qquad [6]
\end{aligned}
$$

   where st is a triggered subtransaction.
   If a subtransaction, say $it1$ is triggered the slack is adjusted as follows:[3]

[3] Note that we update slack only at object events.

$$\begin{aligned} wslack_{t6}(T) &= dl(T) - clock - \\ &\quad eetl_{t6}(T) - eet(it1) - (lenLeft(T) * probTrig_{obj} * eet(st)) \qquad [7] \\ wslack_{t6}(it1) &= wslack_{t6}(T) \qquad [8] \end{aligned}$$

If a subtransaction is not triggered then the slack adjustment is as follows :

$$\begin{aligned} wslack_{t6}(T) &= dl(T) - clock - \\ &\quad eetl_{t6}(T) - (lenLeft(T) * probTrig_{obj} * eet(st)) \qquad [9] \end{aligned}$$

For a given transaction T, $wslack(T)$ is used to calculate its priority during scheduling. The deadlines are assigned as follows:

$$\begin{aligned} vdl_{t6}(it1) &= dl(T) \qquad [10] \\ dl(it1) &= dl(T) \qquad [11] \end{aligned}$$

4.2 Deadline Assignment Policies for Deferred Subtransactions

The priority assignment for deferred transactions is very similar to that of immediate transactions. There are three policies, PD, DIV and SL for deferred transactions but in the case of DIV the denominator of equation [3] considers only the deferred transactions that are yet to be executed, because all the immediate transactions would have already finished execution. The following equations illustrate the deadlines/slack assignments in the case of deferred transactions. Once again, we use the active transaction T in figure 1 in our explanation and discuss priority assignment policies for deferred transaction $dt1$.

1. PD Protocol:

$$\begin{aligned} dl(dt1) &= dl(T) \qquad [12] \\ vdl_{t9}(dt1) &= dl(T) \qquad [13] \end{aligned}$$

2. DIV Protocol:

$$\begin{aligned} vdl_{t9}(dt1) &= ect_{t9}(dt1) + \frac{slack_{t9}(T) - (eet_{t9}(dt1) + eet_{t9}(dt2) + eet_{t9}(dt3))}{nDef_{t9}(T)} \qquad [14] \\ dl(dt1) &= dl(T) \qquad [15] \end{aligned}$$

3. SL Protocol:

$$\begin{aligned} wslack_{t9}(T) &= dl(T) - clock - eetDef_{t9}(T) \qquad [16] \\ wslack_{t9}(dt1) &= wslack_{t9}(T) \qquad [17] \\ vdl_{t9}(dt1) &= dl(T) \qquad [18] \\ dl(dt1) &= dl(T) \qquad [19] \end{aligned}$$

5 Experimentation and Results

We begin this section with a brief description of the validation of our simulator. We then discuss the experimental setup along with the assumptions and decisions made in our experiments. We also present a table of important parameters and their values. Finally, we describe each set of experiments and an analysis of the results. In the experiments, 95% confidence intervals have been obtained whose widths are less than ±2.5% of the point estimate for the Missed Deadline Percentage (MDP).

5.1 Validation of the Simulator

Experiments were conducted to validate the simulator. This was accomplished in three steps.

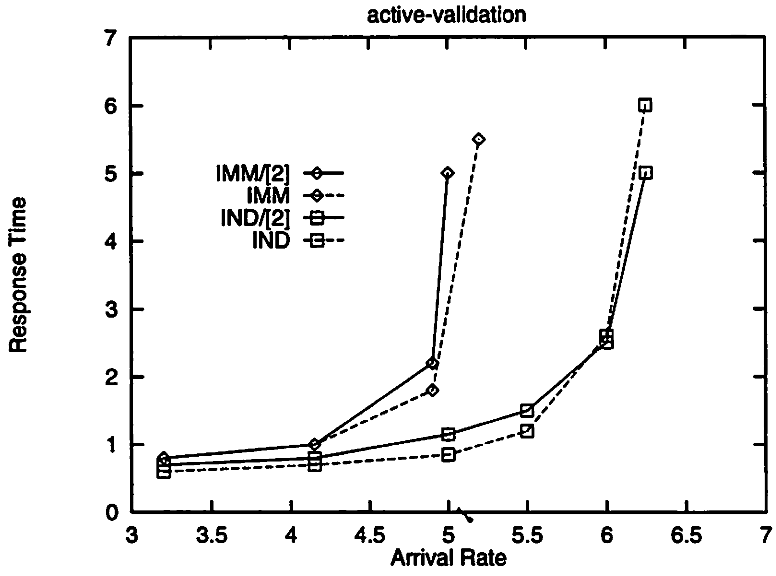

Figure 3:

- We validated the *active* part of the simulator against the results in [2]. The results are illustrated in the figure 3. We mapped their model onto ours as closely as possible. We were not able to obtain the exact results because of differences in the two models. The original work modeled the buffer explicitly, but we model our buffer using a parameter *hitratio*, which is the probability that a page is resident in the buffer. Hitratio is set to 0.9, for all the experiments in this validation. The explanation of parameters and the experiments can be found in [2]. In figure 3 solid lines (marked as /[2]) represent the original results and the dotted lines represent our simulation results. From figure 3, we can see that for Independent coupling mode (no coupling) and the Immediate coupling mode (strict coupling), our results are within 10 percent of the original results.

- We validated the *real-time* part by trying to duplicate the results in [1]. In the NT (*not tardy*) overload management policy, a transaction is aborted as soon as it becomes tardy. In the AE (*all eligible*) policy, a transaction is run until it finishes. The results are illustrated in figure

4. Again our results were very similar to previously published results. The slight performance improvement obtained by our policies in the *not tardy* case, can be explained by the fact that checking for tardiness is done more often in our model.

- Finally, to validate the combined model, we conducted experiments as in [7] (No graphs are plotted for these experiments). There was an inherent difference in the two models because the one in [7] is a distributed system and ours is a single site multiprocessor system. So their system has multiple servers with a queue for each server whereas ours has multiple servers with a single queue. We experimented with *UD* and *DIV-1* policies mentioned in [7] and our missed deadline percentages were lower by no more than 5-10% which can be attributed to the above difference and other subtle differences in parameters.

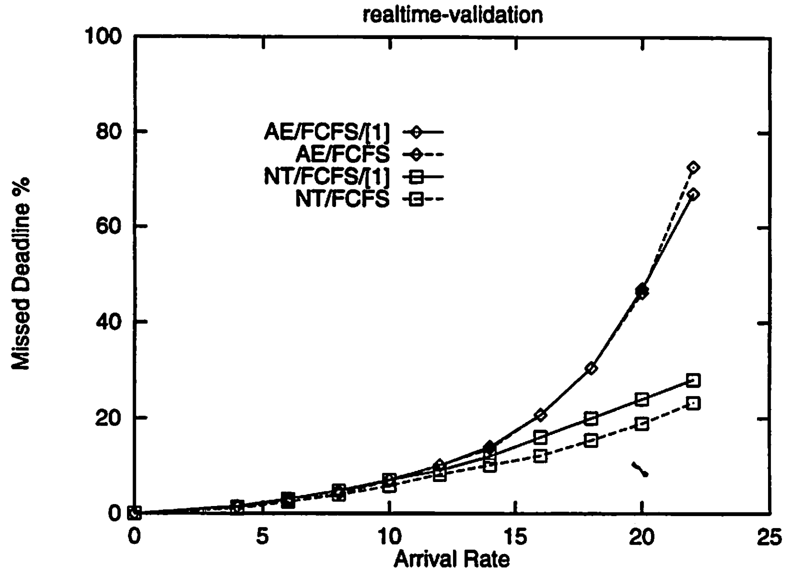

Figure 4:

In our validations we did not perform the complete set of experiments that are in [2], [1] or [7], but just certain baseline experiments.

5.2 Experimental Setup

In our experiments

- We address only *firm real-time* systems where the *value* of the transaction drops to zero once the deadline expires. We abort the transaction that misses its deadline along with all its related subtransactions. This corresponds to the not tardy case.

- We do priority-driven preemptive scheduling.

- We consider active real-time databases that trigger only immediate and deferred transactions.

- We disable cascading rule firings, that is triggered transactions do not trigger further transactions.

- Transaction events do not trigger rules. We make this assumption to simplify the experimentation as the only effect of transaction events triggering rules in our study, would be to increase the number of subtransactions triggered. This effect can be achieved by increasing the probability that an object event can trigger a rule.

- There are three types of class **T** transactions: IMM which trigger only immediate subtransactions, DEF that trigger only deferred transactions, and IMMDEF that trigger both immediate and deferred subtransactions.

- We normalize the slacks for the three types of class **T** transactions taking into account the fact that the deferred transactions can be executed in parallel and the immediate transactions are executed in sequence. Therefore, transactions that trigger only immediate subtransactions will get more slack than those that trigger only deferred subtransactions.

- The Missed Deadline Percentage (**MDP**) is the performance metric we use. This is the traditional metric used to compare performance in firm real-time systems.

- We experiment with two types of data access characteristics of transactions. The first is a main memory based system with no data conflicts. This is essentially a task model as opposed to transaction model. Second is a main memory database, i.e., main memory based system with data conflicts. We have no explicit buffer modeling, but a parameter *hitratio* that probabilistically says if a page is in memory, which is set to 1.0 to simulate main memory database system. Similarly, there is no explicit mapping of objects to specific pages, but every object access is converted in to a corresponding number of page accesses.

We use a parameter *load* in our experiments which is very similar to the one in [7]. In order to define *load* we specify the arrival rates and service rates of class **T** and **NT** transactions. The arrivals of class **T** and class **NT** transactions are generated according to Poisson processes with mean interarrival time of $1/\lambda_T$ and $1/\lambda_{NT}$ time units, respectively. The arrival rates are calculated using the following two equations, where all other quantities except the arrival rates are known to us. In the first equation, we define the *load* to be the ratio of work generated to the total processing capacity of the system. $1/\mu_T$ and $1/\mu_{NT}$ are the average total execution time of class **T** and **NT** transactions respectively, and N_{CPU} is the number of CPUs in the system. In the second equation, $frac_T$ is the fraction of *load* that is contributed by the class **T** transactions.

$$load = \frac{\frac{\lambda_T}{\mu_T} + \frac{\lambda_{NT}}{\mu_{NT}}}{N_{CPU}} \quad [20]$$

$$frac_T = \frac{\frac{\lambda_T}{\mu_T}}{\frac{\lambda_T}{\mu_T} + \frac{\lambda_{NT}}{\mu_{NT}}} \quad [21]$$

Table 1 shows parameter settings for our baseline experiments. The deadline of a transaction T is set using the following formula:

$$dl(T) \quad = \quad at(T) \ + \ (1 + slack) \ * \ eet(T)$$

where slack is randomly selected from a specified range.

Mode	Parameter	Setting
ALL	N_{CPU}	6
	Number of Object Instances	2000
	Overload Management Policy	*Not Tardy*
	Avg. Length of class NT Transactions	5
	slack of class NT transactions	0.5-1.25
	Avg. Length of class T Transactions	6
	Avg. Length of Subtransactions	5
	$frac_T$	0.15
	Probability of triggering by object event	0.8
	Probability of triggering by transaction event	0.0
IMM	*slack*	5.0-5.5
	% of Immediate Subtransactions	100
	% of Deferred Subtransactions	0
DEF	*slack*	2.0-2.5
	% of Immediate Subtransactions	0
	% of Deferred Subtransactions	100
IMMDEF	*slack*	4.0-4.5
	% of Immediate Subtransactions	50
	% of Deferred Subtransactions	50

Table 1 : Baseline setting

5.3 Real-time tasks

In the first set of experiments we deal with real-time active *tasks* executing in a multiprocessor environment. The purpose of these experiments is to isolate and study the effect of scheduling on performance. We simulate a main memory database system where there are no data conflicts. We have a parameter *dataContention* which, if set to 1.0, makes all the methods of an object class compatible with each other and when set to 0.0 makes all the methods incompatible with each other. So in these experiments, we limited the number of object classes to 1 and the number of methods to 1. Depending on the *dataContention* the method is either compatible with itself or not. All the object instances in the database belong to this single class. We experiment with all three types of class **T** transactions, i.e., IMM, DEF and IMMDEF. The performance of the transactions belonging to both classes is presented in figures 5, 6 and 7, respectively. Figure 5 deals with the case where all the subtransactions are triggered in immediate mode (IMM). We can see from the graph that compared to PD both the DIV and SL policies decrease the MDP of **T** class transactions significantly at higher loads by as much as 10 to 12 percent, at the cost of a small increase of around 3 percent in the MDP of **NT** class transactions. The protocol DIV reduces the MDP of **T** class by a greater amount than SL, accompanied by a smaller increase in the MDP of **NT** class. So clearly DIV performs better than SL in the case of immediate transactions.

In figure 6, we present the results for the case where all the subtransactions triggered are executed in deferred mode (DEF). Again, SL and DIV policies perform better than the PD protocol for class **T** transactions. In this case SL and DIV reverse their roles, SL outperforming DIV substantially at higher loads for **T** class. However, in the case of **NT** class DIV performs better than SL. For the **T** class SL reduces the MDP by 30 percent at high loads compared to DIV with a slight increase in the MDP of **NT** class. Essentially SL gives higher preference to **T** class transactions over **NT** class transactions than DIV. This change of order from the previous result can be explained by the fact that deferred subtransactions can be executed in parallel on a multiprocessor system whereas the immediate subtransactions are executed sequentially. So SL gives a very high preference to **T** class transactions when deferred subtransactions are present. This explains the fact that SL keeps the MDP of **T** class transactions nearly constant while the MDP of **NT** class increases. It also explains the fact that SL gives lower MDP for **T** class than the **NT** class.

The results for the case where the subtransactions triggered execute either in deferred or immediate mode with equal probability (IMMDEF), are illustrated in figure 7. The performance trend is similar to that of the deferred only case, but the difference is lower than the DEF case because of the presence of immediate subtransactions.

In the above results we observe that while the MDP of **T** class transactions is reduced by the SL and DIV policies, the MDP of the **NT** class transactions increases. This is desirable, where **T** class transactions are more valuable than **NT** class transactions. To see the performance of the policies if transactions of both classes have the same value, we evaluated the combined MDP of all the transactions. Again 85 percent of the workload comes from **NT** class transactions. The results are illustrated in figures 8,9 and 10. As we can see PD performs best if we give equal value to both **T** and **NT** class transactions. DIV is the next best and the SL is the worst. This is because DIV and SL are biased toward **T** class transactions. Also the CPU time required by one **T** class transaction is much higher. Though the MDP of **T** class is reduced, since **NT** class transactions constitute the majority, the overall MDP increases in the case of DIV and SL. For example, in figure 8, SL shows the worst performance and DIV is comparable to PD except at very high loads. So DIV protocol performs comparably with the PD protocol even when transactions of both classes have equal value. In DEF and IMMDEF case clearly PD provides the lowest MDP over DIV and SL policies in that order.

5.4 Analysis of DIV and SL Policies

We observed in the previous experiments that DIV and SL give preference to **T** class transactions over the **NT** class. In this section we will study what happens when we design algorithms which range between PD and DIV and similarly between PD and SL. So we introduced a parameter α which controls the priority assignment in DIV and SL policies as follows. We call these parameterized policies ALPHA-DIV and ALPHA-SL.

ALPHA-DIV: priority = $\alpha \star \mathit{deadline}_{DIV} + (1-\alpha) \star \mathit{deadline}_{parent}$

$\mathit{deadline}_{DIV}$ is the deadline assigned to the transaction by the DIV protocol. $\mathit{deadline}_{parent}$ is the deadline of the parent transaction.

ALPHA-SL : priority = $\alpha \star \mathit{estSlack}_{SL} + (1-\alpha) \star \mathit{deadline}_{parent}$

$\mathit{estSlack}_{SL}$ is the slack assigned to the transaction by the SL protocol.

A lower priority value indicates higher importance. So when α is zero both the policies reduce to PD. When α is one they are DIV and SL policies respectively. For this set of experiments the load was kept constant at 0.75. We studied the performance of the ALPHA-DIV and ALPHA-SL policies as α is changed from zero to one. The results are plotted in figures 11,12 and 13. On X and Y axes we plot the MDP of **NT** and **T** class transactions, respectively. The points on the curves, correspond to α values varied from 0 to 1 with an increment of 0.2. We also plotted some points with intermediate values of α to clearly distinguish between the different policies. We observe that ALPHA-DIV works better than ALPHA-SL throughout the range for the immediate only case. It reduces the MDP of **T** class transactions by 22 percent with a little increase of only 1 percent in the MDP of **NT** class, compared to the ALPHA-SL protocol which achieves the same effect at the cost of a 3.5 percent increase. So the ALPHA-DIV protocol performs better than the ALPHA-SL protocol in this case.

But in the deferred only transactions case the ALPHA-SL protocol reduces the MDP of **T** class to a very negligible value with a rise in the MDP of **NT** class. ALPHA-DIV is not able to reduce the MDP of **T** class transactions that significantly. So if **T** class transactions have a higher value than **NT** class transactions, then ALPHA-SL is the protocol of choice in the deferred only case. ALPHA-DIV marginally outperforms ALPHA-SL at lower α values. In the case where subtransactions can be deferred or immediate with equal probability, the trend is the same as the deferred only case. The results are illustrated in figure 13. So ALPHA-SL gives more flexibility to achieve a higher reduction in the MDP of **T** class with an increase in the MDP of the **NT** class than the ALPHA-DIV policy.

5.5 Main Memory Database

We now extend the performance study to consider priority assignment policies in the presence of data contention. The concurrency control algorithm we use is a modified wound-wait algorithm. All the subtransactions of a transaction (deferred and immediate modes) are considered part of the transaction by the concurrency control mechanism. They share the locks. Similarly, two subtransactions of the same parent transaction share the locks. All the subtransactions and the parent transaction release the locks at their commit time which is after the parent transaction and the deferred subtransactions have finished. Since different subtransactions may have different priorities, deadlocks might occur in spite of using the wound-wait protocol. So we use a deadlock prevention mechanism, which checks for deadlocks whenever a transaction waits for another transaction and aborts the transaction that causes the deadlock. It has been shown in previous studies that the choice of the transaction to be aborted to resolve a deadlock does not have a significant impact on the performance [6]. For these experiments we set the parameter ***dataContention*** to 0.0, making any two method accesses incompatible. This is equivalent to locking the whole object exclusively for each operation. We keep the rest of the parameter values the same as in previous experiments.

The results are illustrated in figures 14,15 and 16 for immediate only, deferred only and both immediate and deferred cases, respectively. The relative ordering of the performance of the policies remains the same as in the case with no data contention. DIV outperforms SL in the immediate only case and SL reduces the MDP of class **T** transactions more than the DIV policy with an increase in the MDP of class **NT** transactions, in the deferred only and immediate and deferred cases. In figure 14, the SL policy gives a higher MDP than PD at low loads for the **T** class transactions. This effect is due to the fact that SL policy gives a very high priority to a **T** class transaction at the beginning. When there is a data conflict, it leads to a transaction which has just started to abort a transaction which is in its later stages. The absolute MDP values for these experiments is greater than the MDP values of the previous set of experiments with no data contention, that is, more transactions are aborted due to deadline misses. Due to data conflicts more transactions experience waits or get aborted. This shows the effect of data contention on the performance. We are exploring the parameters further to determine if the trend observed also holds for other combination of parameter values, including different degrees of data contention, different amounts of CPU resources and with disk resident data.

6 Conclusions

We have studied the problem of assigning priorities to triggered transactions and reassigning priorities of triggering transactions in a firm real-time active database. We discussed a simple baseline policy PD and two other policies namely DIV and SLACK which assign priorities taking into consideration the active work generated by a transaction. We showed that DIV and SL reduce the deadline miss rate of transactions that trigger other subtransactions (class **T**) with a small increase in the miss deadline percentage of transactions that don't trigger other transactions (class **NT**). We showed that DIV outperforms SL when the transactions are triggered only in immediate mode, and that SL favors class **T** transactions more than DIV when transactions are triggered only in deferred mode. We also showed that SL gives more range flexibility to reduce the MDP of **T** class transactions with an increase in the MDP of **NT** class transactions, when the transactions are triggered only in deferred mode or when they are triggered in immediate or deferred mode with equal probability. We studied the performance of policies in a task only model (no data conflicts) and in a main memory database model. The performance of the policies did not differ drastically in these two settings.

Some of the extensions we want to address are

- Experiment with DIV and SL policies on a disk resident database to find out if disk I/O has an effect on their performance.
- Experiment with variations of DIV and SLACK policies that have more information about a transaction, for instance the exact number of subtransactions that it is going to trigger, and the type of subtransactions it is going to trigger, i.e., immediate or deferred, to see what advantage is obtained by exploiting this information.
- Study variations of DIV and SLACK that will assign priorities at every scheduling instance instead of at every object event.
- Study the effect of errors in the knowledge about transactions like *eet* and *probTrig* on the performance of the DIV and SL policies.
- Explore the relative performance of the DIV and SLACK policies in different resource contention regions.

References

[1] R.K. Abbott, and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *ACM Trans. on Database Systems*, Sept. 1992.

[2] M.J. Carey, R. Jauhari, and M. Livny, "On Transaction Boundaries in Active Databases: A Performance Perspective," *IEEE Trans. on Knowledge and Data Engg.*, Sept. 1991.

[3] U. Dayal, et. al., "The HIPAC Project: Combining Active Databases and Timing Constraints," *SIGMOD Record*, 17, 1, March 1988.

[4] U. Dayal, M. Hsu, and R. Ladin, "Organizing Long-Running Activities with Triggers and Transactions," ACM 1990.

[5] Special Issue on Active Databases, *Data Engineering*, Dec. 1992.

[6] J. Huang, J. Stankovic, D. Towsley, and K. Ramamritham, "Experimental Evaluation of Real-time Transaction Processing," *IEEE Real-Time Systems Symposium*, Dec. 1989, 144-153.

[7] B. Kao, and H. Garcia Molina, "Subtask Deadline Assignment for Complex Distributed Soft Real-Time Tasks," *Technical Report STAN-CS-93-1491*, Stanford University, Oct. 1993.

[8] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour, "*A Practitioners Handbook for Real-Time Systems - Guide to Rate Monotonic Analysis for Rea-Time Systems*," Kluwer Academic Publishers, 1993.

[9] E. L. Lawler, "Recent Results in Theory of Machine Scheduling". *Mathematical Programming : The State of the Art*, A.Bachem et. al., Springer-Verlag, New York, 1983.

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

[11] D. McCarthy, and U. Dayal, "The Architecture of an Active Data Base Management System," ACM 1989.

[12] B. Purimetla, R. M. Sivasankaran and J.Stankovic, "A Study of Distributed Real-Time Active Database Applications," *IEEE Workshop on Parallel and Distributed Real-time Systems*, April 1993.

[13] K. Ramamritham, "Real-Time Databases," *International Journal of Distributed and Parallel Databases*, 1993.

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

[15] J. Xu and D. L. Parnas, "Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations," *IEEE Transactions on Software Engineering*, Vol. 16, No. 3, March 1990.

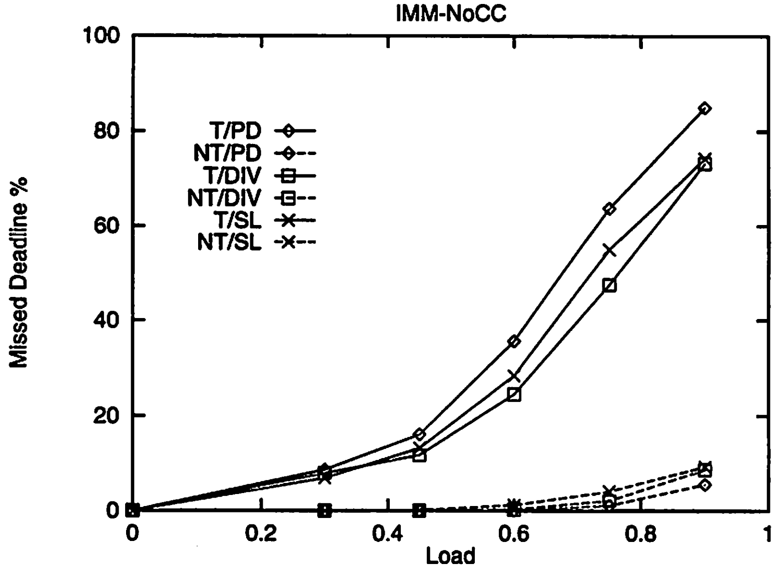

Figure 5:

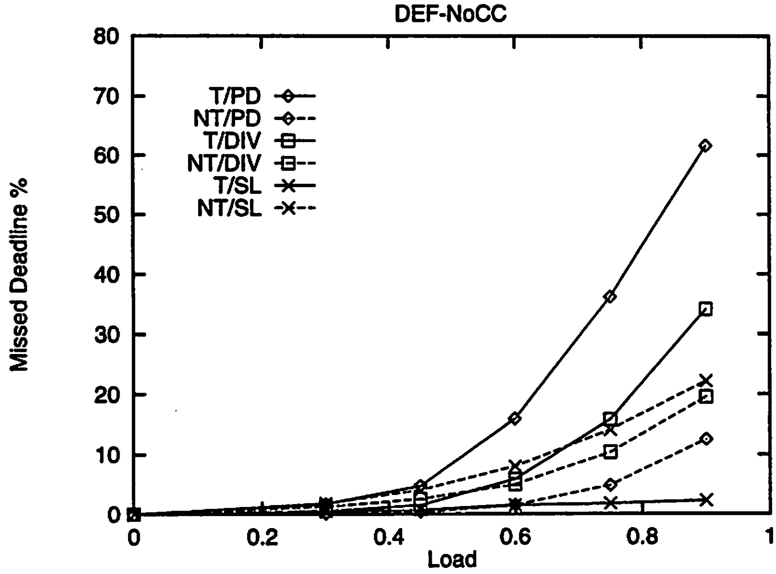

Figure 6:

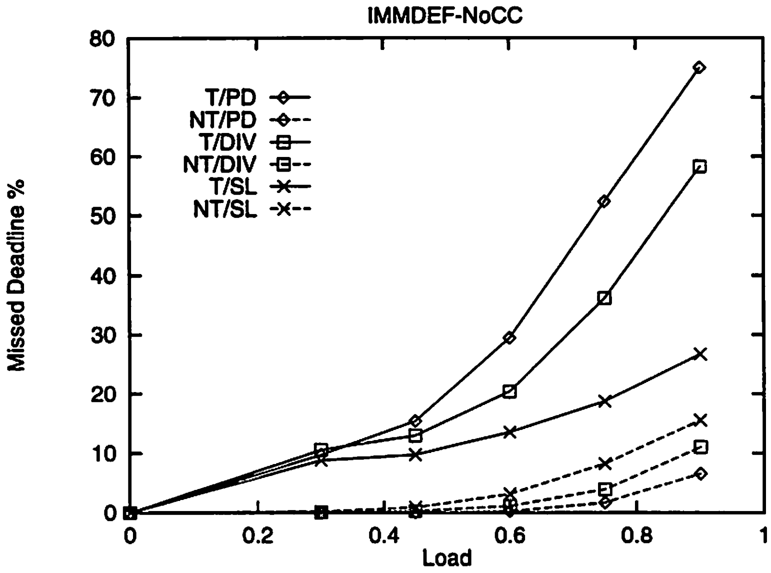

Figure 7:

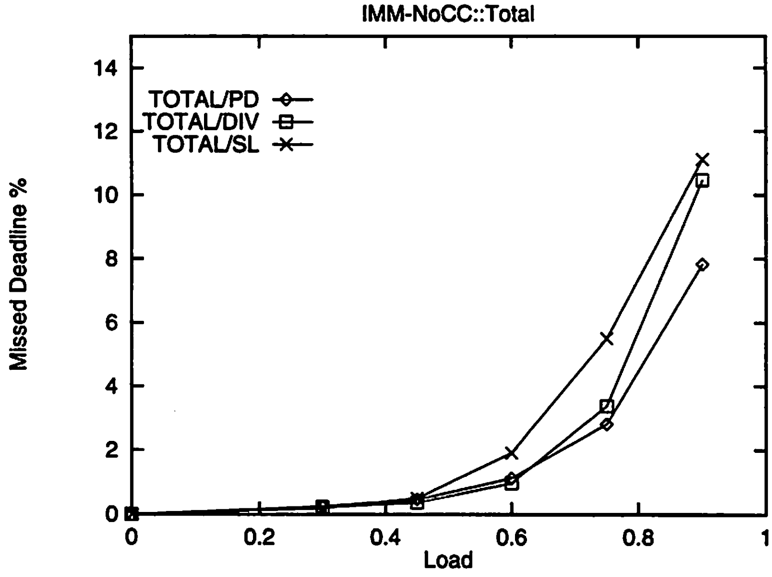

Figure 8:

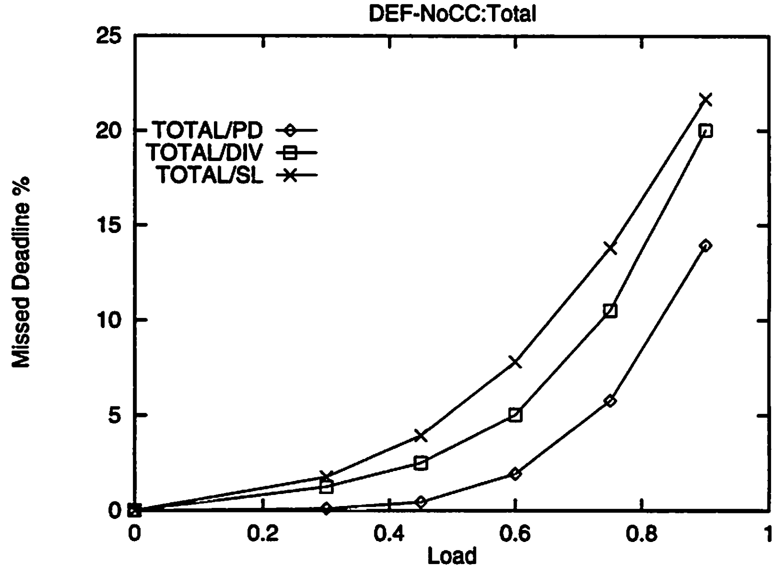

Figure 9:

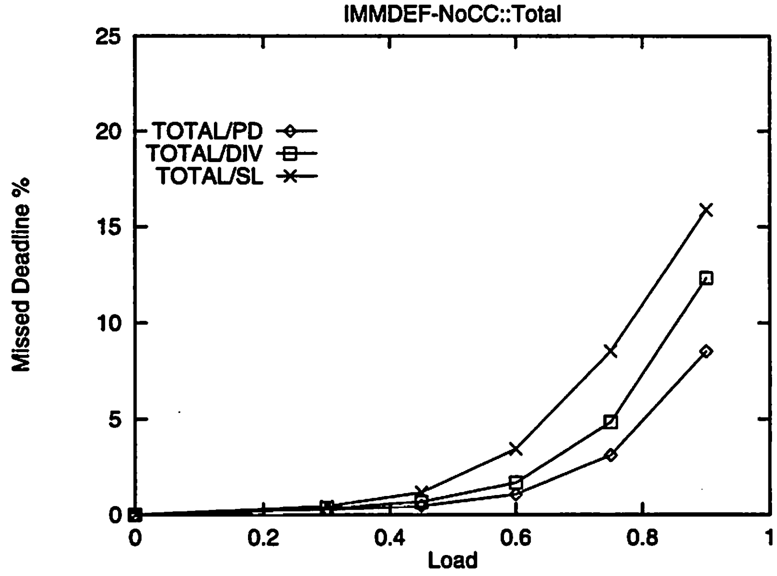

Figure 10:

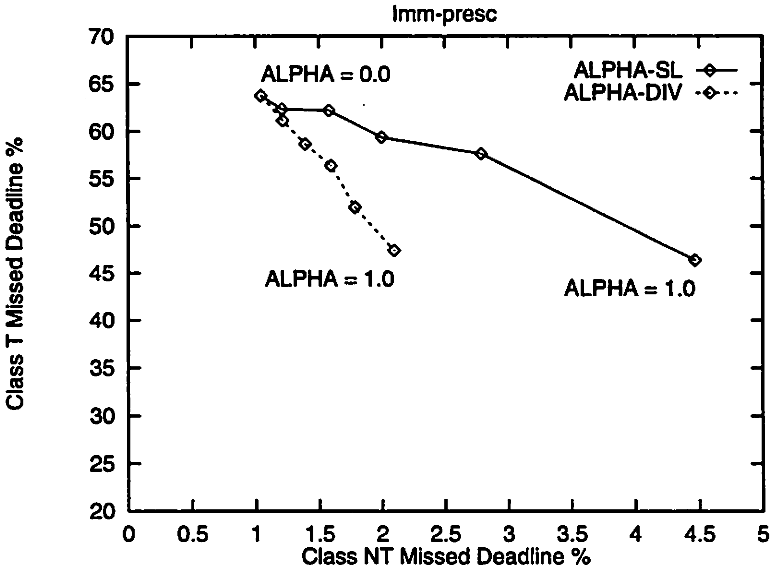

Figure 11:

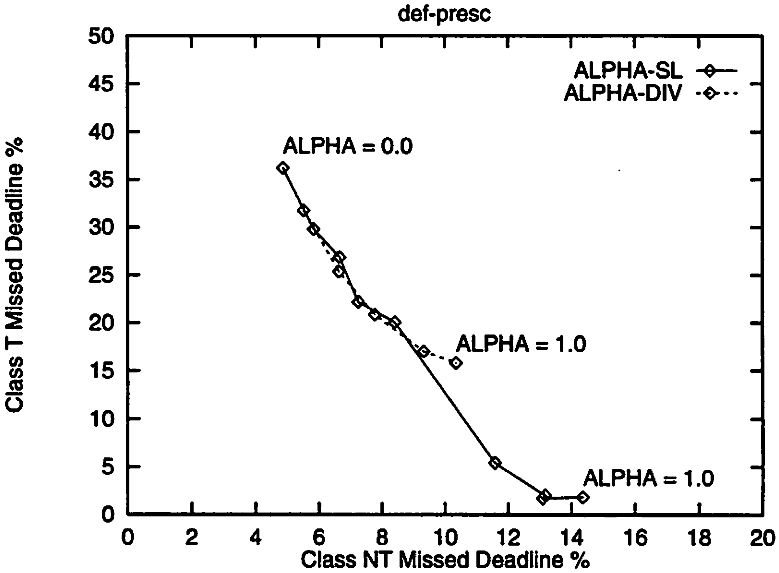

Figure 12:

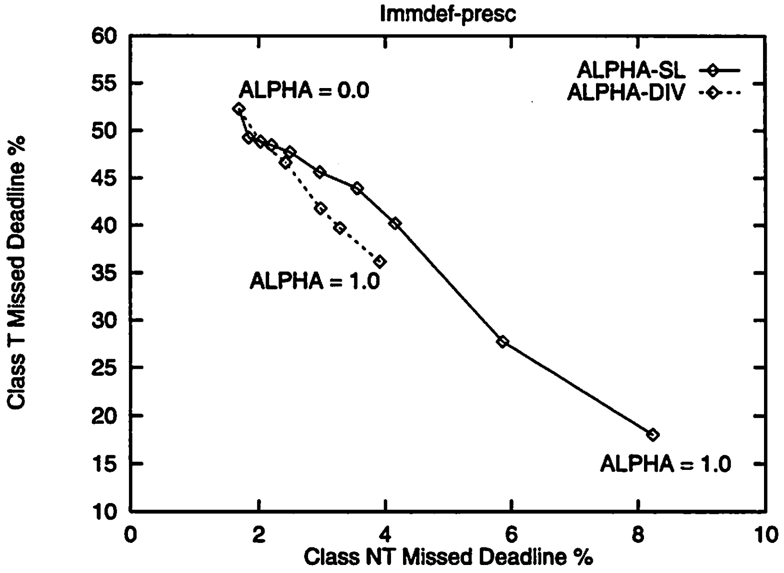

Figure 13:

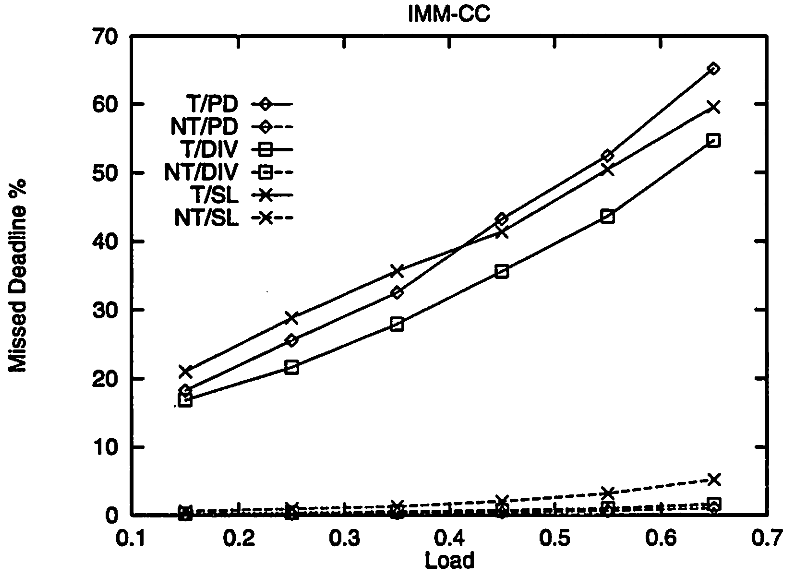

Figure 14:

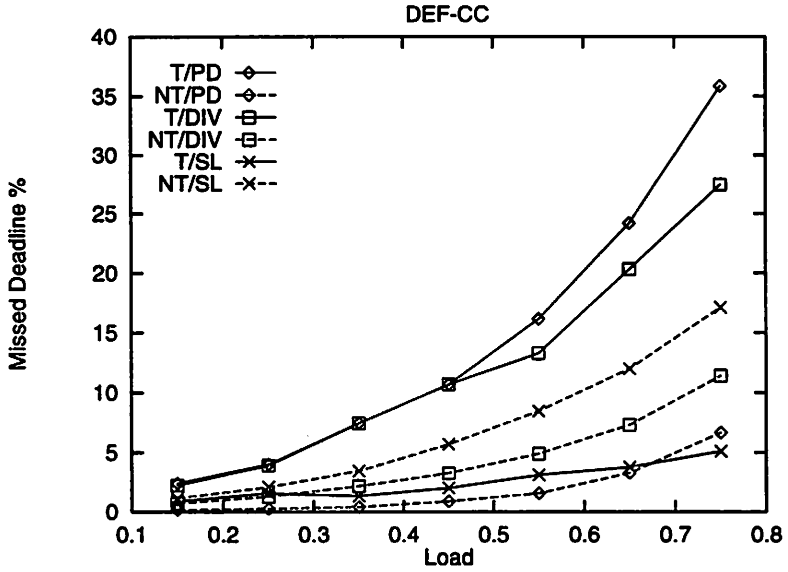

Figure 15:

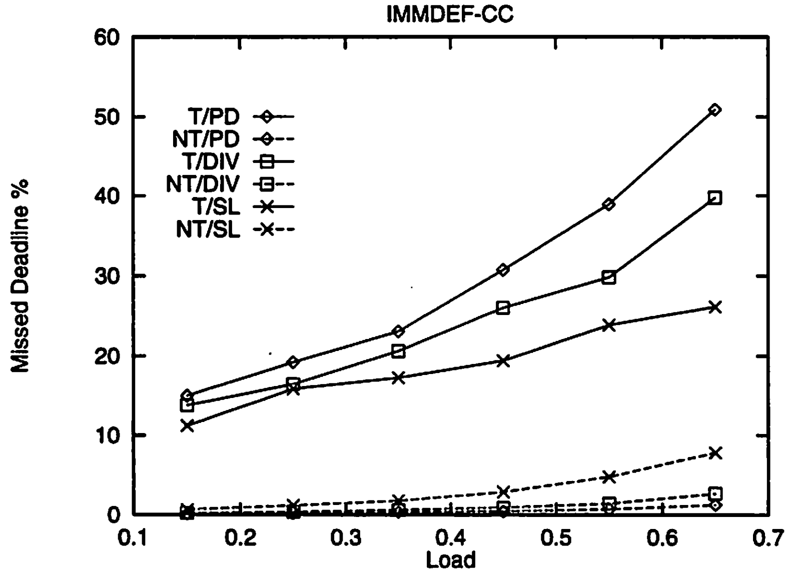

Figure 16: