Priority Assignment in Real-Time Active Databases[1] [2]

Rajendran M. Sivasankaran, John A. Stankovic, Don Towsley
Bhaskar Purimetla, & Krithi Ramamritham
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 three policies, namely PD, DIV and SL, for assigning priorities to parent, immediate and deferred transactions executing on a multiprocessor system and then evaluate the policies through simulation. The policies use different amount of semantic information about transactions to assign the priorities. The simulator has been validated by comparing to three sets of published results. We conducted experiments in three settings: task model, main memory database model and disk resident database model. 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 in all the three settings.

1 Introduction

Traditionally, in soft real-time database 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 deadlines 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 policies that account for deadlines. The performance implications of time cognizant priority assignment policies have been studied in detail in soft real-time database systems in [1, 6]. These studies have concluded that deadline-cognizant priority assignment provide significantly higher performance than priority assignments that ignore deadlines. In this paper, our goal is to study and evaluate time cognizant 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

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

[2]A shorter version of this paper appeared in PDIS 1994.

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 presenting a detailed description of the problem we are addressing for a real-time active database, we give a brief introduction to active databases. The building block of an active database system is the Event-Condition-Action (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[3], 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 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: *immediate*, *deferred* and *independent*. The transactions triggered in those modes are referred to as immediate, deferred and independent transactions, respectively. Immediate and deferred transactions are executed as part of the parent transaction whereas independent transactions are executed independently. Since immediate and deferred transactions are part of the parent transaction, we also refer to them as subtransactions. An immediate transaction is executed as soon as it is triggered and the parent transaction is suspended until it completes. A deferred transaction is executed after the parent completes execution, but before it commits. In this paper, by a parent completing execution we mean that the parent has finished all its work but has not committed its results. The immediate and deferred transactions are committed if and only if the parent commits.

Due to the rule firings, an active transaction dynamically generates additional work. In order for the time-cognizant scheduling policies to perform well, they should take into account the work that is dynamically generated. This aspect of the problem makes this scheduling problem different from the classical hard real-time scheduling problem where execution times are assumed to be known in advance [8, 9, 16]. The priority driven nature of real-time transaction processing gives rise to the question of how to assign priorities to the parent and to all triggered actions [12, 14]. We believe that the strategy to assign priorities has a significant impact on the performance of the system as triggered actions must contend with ongoing transactions for resources. In this paper, we address the problem of assigning priorities to triggering and triggered transactions. We introduce and evaluate three policies for assigning priorities to *immediate* and *deferred* transactions.

Our main contributions are:

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

- the development of two priority assignment policies that account for the work dynamically generated by active transactions;
- an evaluation of the two priority assignment policies against a baseline policy using a real-time active database simulator in three settings, i.e., in a real-time task setting, a main memory database setting and a disk resident database setting;
- a demonstration 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;
- the identification of the trade-offs between the performance of triggering and non-triggering transactions offered by the different policies, thereby enabling an implementor to select from various policies depending on the relative importance of the triggering and non-triggering transactions in the system; and
- the identification of differences and probable reasons for the differences in the relative performance of the policies in the three settings.

We discuss related work in Section 2. In Section 3, we explain our transaction model. Section 4 gives a detailed explanation of the priority assignment policies. We discuss the simulator, experiments and results in Section 5. We summarize our main results and discuss future work in Section 6.

2 Related Work

Over the past few years active databases and real-time databases have become important areas of research. Experimental studies reported in [1, 6, 15] are very comprehensive and cover most aspects of real-time transaction processing, but have not considered active workloads (workloads that generate additional transactions dynamically) and have not addressed the problem of subtransaction priority assignment. There have been both theoretical and experimental studies in active databases [2], [3], [11], [4]. Most of the studies have concentrated on the specification of Event-Condition-Action (ECA) rules [5]. Experimental work on active databases in [2] has been performed in a non real-time setting. Experiments in [2] show the impact of transaction boundaries and data sharing on the performance of active databases. We address a different problem, that of assigning priorities to triggered transactions in real-time active databases.

The relationship between real-time databases and active databases was discussed briefly in [13] where it was noted that a missing ingredient in active databases is the active pursuit of timely processing of actions to do real-time processing. 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.

The problem of assigning deadlines to the parallel and serial subtasks of complex distributed tasks in a real-time system has been studied in [7] through simulation. This work differs from the present work in two ways. First, the structures of all of the complex tasks are assumed to be known in advance. Second, the experimental system considered in [7] is not a database system nor does it have an active component. The system we consider for our experiments is a real-time active database with unpredictable data accesses and rule firings.

3 Transaction Model

In this section we describe the transaction model we have considered and the simplifying assumptions we have made to study priority assignment in a real-time active database. The performance metric we use in our study is Missed Deadline Percentage (**MDP**), i.e., the percentage of transactions that miss their deadlines, which is a traditional metric used to evaluate performance in real-time systems.

Data in our system is modeled as objects that have methods associated with them. A transaction in our system is a series of method executions on objects. We consider two classes of transactions in the system: *Non-Triggering* (class **NT**) and *Triggering* (class **T**). A non-triggering 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. We consider Class **T** transactions that trigger only immediate and deferred transactions. We disable cascading rule firings, that is, triggered transactions do not trigger further transactions. 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. In our current model, only the object events, i.e., an execution of a method on an object, activate rules and cause the triggering of subtransactions. Transaction and temporal events do not activate rules. We make this assumption to simplify the experimentation since the only effect of transaction and temporal 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. Please note that our simulation implementation supports independent transactions, cascading rule firings, and transaction and temporal events.

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. Consequently the transaction is less likely to complete successfully compared to another transaction

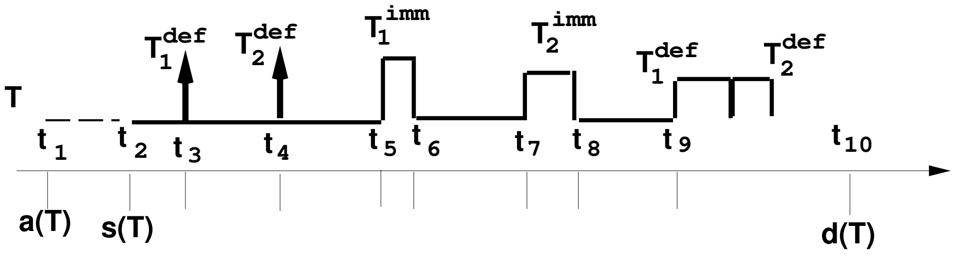

Figure 1: **Life of a Complex Active Transaction**

with the same deadline which doesn't trigger any subtransactions, since it faces more contention for resource during its lifetime.

To better illustrate the problem being addressed, we provide an example of the structure of a complex active transaction executing on a uniprocessor. Figure 1 shows the life of a complex active transaction T that triggers transactions in all three modes. Transaction T arrives at time t_1 $(a(T))$ with t_{10} as its deadline $(d(T))$ and is started at time t_2 $(s(T))$. It triggers a deferred transaction T_1^{def} at time t_3, and another deferred transaction T_2^{def} at time t_4. It triggers an immediate transaction T_1^{imm} at t_5, and an immediate transaction T_2^{imm} at t_7. Figure 1 shows that the transactions T_1^{imm} and T_2^{imm} execute immediately while T is suspended. Finally, the figure shows that once the parent T completes at t_9, the deferred transactions T_1^{def} and T_2^{def} execute. The problems we address are the problem of assigning priorities to subtransactions T_1^{imm}, T_2^{imm}, T_1^{def} and T_2^{def}, 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 in parallel if operating on a multiprocessor or a distributed system.

Later in this section, we describe three priority assignment policies PD, DIV and SL for assigning priorities to the parent and immediate and deferred subtransactions in an active real-time database system. PD is a *static* baseline policy where the priority of the parent does not change with time. DIV and SL are *dynamic* policies that change the priority of the parent depending on the amount of dynamic work it has generated. In all cases, the priorities are assigned to the triggered transactions when they start execution and priorities are not changed subsequently during their execution. It should be noted that the three algorithms use increasingly more knowledge (estimates) about the transactions in their assignment of priorities. PD is a pure EDF policy for both transactions and subtransactions and uses no other information about transactions. DIV makes a simple adjustment to EDF where deadlines are dynamically changed when new subtransactions are triggered. Our hypothesis is that very simple modifications will give significant improvements. Note that DIV uses estimates of execution times of the parent transaction and subtransactions that it has triggered. SL is a Least Slack First (LSF) policy that is slightly more sophisticated than DIV and it uses estimates of execution times of the parent transaction and triggered subtransactions and estimates about the subtransactions that might be triggered in the future. Our hypothesis is that when such information is available, it can provide additional performance benefits.

We will describe how the policies PD, DIV and SL assign priorities to the parent transaction and the triggered subtransactions for two cases: when the triggered subtransaction is of type immediate and when the triggered subtransaction is of type deferred. Before describing the policies, we introduce some attributes of a transaction and other related terms that we require to explain our priority assignment policies.

4.1 Attributes of a Transaction

Let us consider a transaction T for which we define the following attributes. Some attributes of the transaction may change during the execution of the transaction and these attributes are subscripted with t, where t is the time at which the attributes' value is represented.

$a(T)$	:	The arrival time of T
$s(T)$	:	The start time of T
$d(T)$	:	The absolute deadline of T
$n_t^{def}(T)$	:	The number of deferred transactions triggered by T prior to time t
$P_t(T)$	:	The priority of T at time t

The policies we have developed use estimates of certain quantities associated with a transaction such as the length of the transaction and the execution time the transaction. The following are some of the estimates that we use in our policies. We also provide an example of of how these estimates are obtained in section 5 where we describe the parameters for the experiments.

$X_t(T)$	:	The estimated execution time left for T at time t
$C_t(T)$	:	The estimated completion time of T at time t; $C_t(T) = t + X_t(T)$
$S_t(T)$	:	The estimated slack of T at time t; $S_t(T) = d(T)$ - $C_t(T)$
$m_t^{imm}(T)$	:	The estimated number of immediate transactions triggered by T after time t
$m_t^{def}(T)$	:	The estimated number of deferred transactions triggered by T after time t
$\overline{X}^{imm}(T)$	:	The estimated average execution time of an immediate subtransaction triggered by T
$\overline{X}^{def}(T)$	:	The estimated average execution time of a deferred subtransaction triggered by T
$X_t^{worst}(T)$	:	The estimated worst case execution time of T at time t

$$X_t^{worst}(T) = X_t(T) + n_t^{def}(T) * \overline{X}^{def}(T) + m_t^{def}(T) * \overline{X}^{def}(T) + m_t^{imm}(T) * \overline{X}^{imm}(T)$$

$S_t^{worst}(T)$: The estimated worst case slack of T at time t

$$S_t^{worst}(T) = d(T) - t - X_t^{worst}(T)$$

$X_t(T)$, is the estimate (at time t) of the time it will take a transaction or subtransaction to execute. This quantity, when estimated at the start of a transaction or when a subtransaction is triggered is the estimated execution time of that (sub)transaction. The estimated worst case execution time of transaction T, $X_t^{worst}(T)$, includes the execution time left at time t for T, execution times of all of the deferred transactions T has triggered prior to time t and estimates of execution times of the subtransactions which the transaction T may trigger during its remaining lifetime. We believe that by analyzing the characteristics of a real-time database application one might be able to obtain information like the estimated execution time of transactions, the kind of actions they trigger and the probability of triggering.

It should be noted, that a lower value of $P_t(T)$ indicates higher importance. For instance, if there are two transactions T_1 and T_2, and $P_t(T_1) < P_t(T_2)$, then T_1 gets priority over T_2. Also, in all the policies, the absolute deadline (i.e., the deadline by which a transaction is required to complete) of the subtransaction is set to the parents deadline. For instance, if a transaction T triggers a subtransaction T^{sub} transaction at time t, then

$$d(T^{sub}) = d(T)$$

4.2 Priority Assignment When Immediate Subtransactions Are Triggered

1. ***PD***: Assign the deadline of the parent as the priority of the immediate subtransaction. Further, 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. Let us consider a transaction T triggering its ith immediate transaction T_i^{imm} at time t. Then

   $$P_t(T_i^{imm}) = d(T)$$

2. ***DIV***: Divide the parent's estimated effective slack equally among the *current* immediate subtransaction triggered, the deferred subtransactions triggered prior to the triggering point and the parent. This quantity (obtained by dividing the parent's estimated effective slack) is added to the estimated completed time of the immediate subtransaction to give the priority of the subtransaction. The parent's deadline is also adjusted dynamically to reflect the work that has been triggered dynamically.

   Let us consider a transaction T triggering its ith immediate transaction T_i^{imm} at time t. The estimated effective slack is calculated by subtracting the sum of estimated execution time of deferred transactions that have been triggered until time t and estimated execution time of the ith immediate transaction from the parents estimated slack as shown in equation (1).

$$P_t(T_i^{imm}) \quad = \quad C_t(T_i^{imm}) + \frac{S_t(T) - \left(X_t(T_i^{imm}) + \sum_{j=1}^{n_t^{def}(T)} X_t(T_j^{def})\right)}{n_t^{def}(T)+2} \qquad ..... (1)$$

Let us assume T_i^{imm} finishes at time $w > t$. The parents priority is reassigned at time w as follows

$$P_w(T) \quad = \quad P_w(T) - (w - t)$$

For instance, in figure 1, for T_1^{imm} at the triggering point t_5 the absolute deadline and priority are assigned as follows

$$P_{t_5}(T_1^{imm}) \quad = \quad C_{t_5}(T_1^{imm}) + \frac{S_{t_5}(T) - \left(X_{t_5}(T_1^{imm}) + X_{t_5}(T_1^{def}) + X_{t_5}(T_2^{def})\right)}{2+2}$$

The priority of the parent transaction will be modified when T_1^{imm} completes at t_6.

$$P_{t_6}(T) \quad = \quad P_{t_6}(T) - (t_6 - t_5)$$

The main idea behind this policy is that of giving higher priorities to class **T** transactions, which have more work to do before completion. This should 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. The priority assigned to the subtransaction can be thought of as a virtual deadline which is the sum of its estimated completion time and some slack that it gets from the parent. This virtual deadline is then used to schedule the subtransaction using the EDF algorithm.

3. ***SL***: Adjust the worst case slack ($S_t^{worst}(T)$) of the parent at each potential triggering point and use that as the priority for both the triggered and triggering transactions. The initial value of slack is assigned based on the estimates of the remaining execution time for a transaction and its subtransactions. The slack is then adjusted at each object 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. Let us assume a transaction T starts at time t_0 and time t is a potential triggering point where T could trigger its ith immediate transaction T_i^{imm}. Initially the priority of T (worst case slack) at time t_0 is set as follows:

$$P_{t_0}(T) \quad = \quad S_{t_0}^{worst}(T)$$

Now at time t, if subtransaction T_i^{imm} is triggered, the priority is adjusted as follows:[4]

$$P_t(T) \quad = \quad d(T) - t - X_t^{worst}(T) - \sum_{j=1}^{n_t^{def}(T)} X_t(T_j^{def}) - X_t(T_i^{imm})$$
$$P_t(T_i^{imm}) \quad = \quad P_t(T)$$

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

$$P_t(T) \quad = \quad d(T) - t - X_t^{worst}(T) - \sum_{j=1}^{n_t^{def}(T)} X_t(T_j^{def}) \quad .... (2)$$

In figure 1, the priorities for T and T_1^{imm} will be

$$P_{t_5}(T) \quad = \quad d(T) - t_5 - X_{t_5}^{worst}(T) - X_{t_5}(T_1^{imm}) - X_{t_5}(T_1^{def}) - X_{t_5}(T_1^{def})$$

[4]Note that we update priorities only at object events.

4.3 Priority Assignment When Deferred Subtransactions Are Triggered

The priority assignment for deferred transactions is very similar to that of immediate transactions. In the DIV and SL policies the priority of the parent will change when it triggers a deferred transaction. The deferred transaction is executed after the parent transaction finishes execution. Let us consider a transaction T that is triggering its ith deferred transaction at time t. The priority assignment for the parent transaction T under the three policies are as follows:

1. PD Protocol:

$$P_t(T) \quad = \quad d(T)$$

2. DIV Protocol:

$$P_t(T) \quad = \quad P_t(T) - X_t(T_i^{def})$$

3. SL Protocol:

$$P_t(T) \quad = \quad d(T) - t - X_t^{worst}(T) - \sum_{j=1}^{n_t^{def}(T)} X_t(T_j^{def}) - X_t(T_i^{def})$$

For the SL protocol, if the transaction T does not trigger a deferred transaction then the priority assignment will be the same as equation (2) in the last section. The following equations illustrate the deadlines/slack assignments for the deferred transactions. Let us assume that transaction T has triggered m deferred transactions before completing execution. In the following equations, we consider the assignment of priority for T_j^{def}, $j \leq \; m$, at time w, which is the time the deferred transactions start their execution (after the parent transaction completes execution).

1. PD Protocol:

$$P_w(T_j^{def}) \quad = \quad d(T)$$

2. DIV Protocol:

$$P_w(T_j^{def}) \quad = \quad C_w(T_j^{def}) + \frac{S_w(T) - \sum_{k=1}^{m}(X_w(T_k^{def}))}{m}$$

3. SL Protocol:

$$P_w(T_j^{def}) \quad = \quad d(T) - w - \sum_{k=1}^{m}(X_w(T_k^{def}))$$

5 Experimental Results

We begin this section with a brief description of $RADEx$ (*R*eal-time *A*ctive *D*atabase *Ex*perimental simulator) and its validation. We then discuss the experimental setup along with the assumptions 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 $\pm 5\%$ of the point estimate for the Missed Deadline Percentage (MDP).

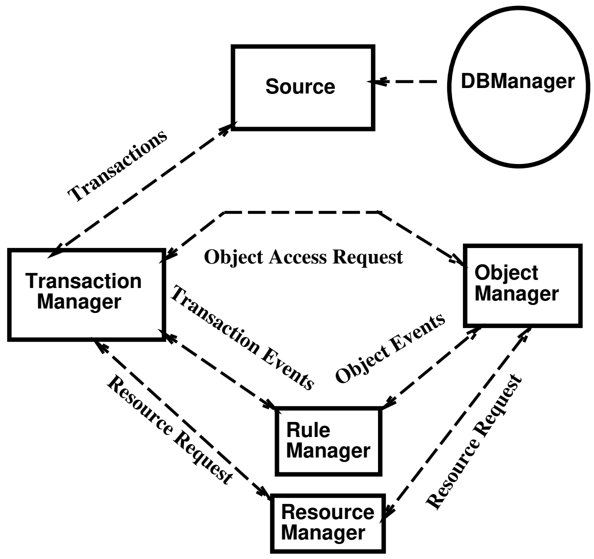

Figure 2: **Simulator Architecture**

5.1 Simulation Model

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

- *Transaction Manager*: The Transaction Manager is responsible for scheduling and execution of the 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 is responsible for concurrency control and sending messages to the rule manager when an object event occurs. Transaction manager requests the Object Manager for access to objects. If the request for an object can be satisfied Object Manager sends the object granted message back to the Transaction Manager. If the request cannot be satisfied it sends an abort message back to the Transaction Manager. Conflict resolution in this module is based on priorities 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 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, disks and the main memory buffer. The Object Manager makes request to the Resource Manager for the necessary pages or for CPU time to execute the methods. The Transaction Manager requests CPU time and buffer space to load transactions from the Resource Manager. 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 randomly selected disk queue depending on the kind of request.

We do priority-driven preemptive scheduling. 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.

5.2 Validation of the Simulator

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

- We validated the *active* part of the simulator against the results in [2]. The results are illustrated in the figure 3. We mapped the model in [2] onto ours as closely as possible. We were not able to obtain the exact results because of the following differences in the two models. The original work modeled the buffer explicitly, whereas we model our buffer using a parameter h, which is the probability that a page is resident in the buffer. h is set to 0.9, for all the experiments in this validation. The explanation of the parameters and the experiments can be found in [2]. In figure 3, we observe that our results are within 10 percent of the original results.

- We validated the *real-time* part of the simulator by trying to duplicate the results in [1]. In the NT[5] (*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, we validated our simulator against the results in [7], where the policies for assigning deadlines to parallel and serial subtasks of complex distributed tasks in a real-time system have been evaluated. We do not present the result graph for this experiment to save space. There was an inherent difference in the two models because the one in [7] is for a distributed system whereas ours is for a single site multiprocessor system. The system in [7] 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%.

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

5.3 Baseline Parameters

In this section we describe the workload model and the calculation of estimates required by DIV and SL. Let us assume a transaction T arrives at time t_0 and triggers a subtransaction at time t_1. Let $U(i,j)$ denote a uniformly distributed random variable in the range $[i,j]$. The length of a transaction T is given in number of methods as follows.

$$\begin{aligned} L(T) &= U(5,7), T \in \text{class } \mathbf{T} \\ &= U(4,6), T \in \text{class } \mathbf{NT} \\ L(T^{sub}) &= U(4,6), T \in \text{class } \mathbf{T},\ T^{sub} \text{ subtransaction (immediate or deferred) triggered by } T \end{aligned}$$

Each method takes 1 unit of time to execute. At the start of every method execution a transaction T belonging to class $\mathbf{T}$ triggers a subtransaction with probability ρ. The probability that the triggered subtransaction is of type immediate is q and the probability that it is of type deferred is $1-q$. For any transaction T, we can compute the length of the transaction when it arrives ($L_{a(T)}(T) \equiv L(T)$)

[5] Please note this is not the same as class $\mathbf{NT}$ which denotes the class of non-triggering transactions.

and hence, we know the remaining length of the transaction at any time t. Let $L_t(T)$ denote the remaining length of transaction T at time t. The estimates that we use in the priority assignment policies are calculated as follows:

$$\begin{aligned}
X_t(T) &= L_t(T) \\
m_t^{imm}(T) &= L_t(T) * \rho * q \\
m_t^{def}(T) &= L_t(T) * \rho * (1-q) \\
\overline{X}^{imm}(T) &= L(T^{sub}) \\
\overline{X}^{def}(T) &= L(T^{sub})
\end{aligned}$$

It should be noted that all other estimates discussed in section 4 can be calculated from the above estimates.

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

$$d(T) \;=\; a(T) \;+\; (1+\beta) \;*\; X_{at(T)}(T)$$

where β is a uniformly distributed random variable within a specified range. We consider three types of workloads, one where the class **T** transactions trigger only immediate subtransactions ($q = 1$), one where they trigger only deferred transactions ($q = 0$), and finally where they trigger both immediate and deferred subtransactions with equal probability ($q = 0.5$). 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 deferred subtransactions.

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 times 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.

$$load \;=\; \frac{\frac{\lambda_T}{\mu_T} + \frac{\lambda_{NT}}{\mu_{NT}}}{N_{CPU}}$$

$$frac_T \;=\; \frac{\frac{\lambda_T}{\mu_T}}{\frac{\lambda_T}{\mu_T} + \frac{\lambda_{NT}}{\mu_{NT}}}$$

From the above equations, for a given $load$, $frac_T$, $1/\mu_T$ and $1/\mu_{NT}$ we can compute $1/\lambda_T$ and $1/\lambda_{NT}$. Tables 1 and 2 show the system and transaction parameter settings, respectively, for our baseline experiments.

Parameter	Setting
N_{CPU}	6
Time taken to execute a Method	1 unit
Number of Object Instances	2000
Overload Management Policy	*Not Tardy*
$frac_T$	0.15

Table 1 : System Parameters.

Class	Parameter	Setting
class **NT**	Length in methods ($L(T)$)	$U(4,6)$
	slack parameter (β)	$U(0.5, 1.25)$
class **T**	Length of the parent in methods ($L(T)$)	$U(5,7)$
	Probability of triggering by object event (ρ)	0.8
$q = 1.0$ (all imm.)	Length of the subtransaction ($L(T^{sub})$)	$U(4,6)$
	slack parameter (β)	$U(6.0, 6.5)$
$q = 0.0$ (all def.)	Length of the subtransaction ($L(T^{sub})$)	$U(4,6)$
	slack parameter (β)	$U(2.0, 2.5)$
$q = 0.5$ (both imm. and def.)	Length of the subtransaction ($L(T^{sub})$)	$U(4,6)$
	slack parameter (β)	$U(4.0, 4.5)$

Table 2 : Transaction Parameters

5.4 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, i.e., every data access is a shared access. In the following discussion we use slack to represent the slack parameter of a class which is a uniformly distributed random variable that is used to calculate the deadline (the initial slack) of the transaction belonging to that class, and we use *estSlack*[6] to mean the remaining slack time that is estimated by the policies. The experiments presented here are:

- Load vs. MDP for a fixed slack distribution.
- Average slack vs. MDP for a fixed load.
- Analysis of trade-offs between DIV and SL for a fixed slack distribution and load.

[6] *estSlack* at time t for a transaction T is $S_t(T)$.

5.4.1 Load vs. MDP

The first set of performance results are presented in figures 5, 6, and 7, respectively. Figure 5 deals with the case where all the subtransactions are triggered in immediate mode ($q = 1$). We observe from the graph that when compared to PD, both the DIV and SL decrease the MDP of class **T** transactions at higher loads by as much as 15 to 20 percent, at the cost of a small increase of around 2 percent in the MDP of class **NT** transactions. DIV reduces the MDP of class **T** by a greater amount than SL, accompanied by a smaller increase in the MDP of class **NT** transactions. The first evidence for our hypothesis that extra information like the estimates of execution time and accounting for the dynamic work generated will improve performance of triggering transactions is seen here. This performance improvement, however, comes at the cost of decreased performance of non-triggering transactions.

In figure 6, we present the results for the case where all the subtransactions triggered are executed in deferred mode ($q = 0$). Again, SL and DIV decrease the MDP of class **T** transactions over PD. In this case SL provides substantially better performance to class **T** transactions than **NT** whereas the reverse is true for PD and DIV. SL performs better than DIV at higher loads for class **T**. However, in the case of **NT** class DIV performs better than SL. SL reduces the MDP of class **T** 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 class **T** transactions over class **NT** transactions than DIV. This difference in performance from the previous results in figure 5 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. Hence SL gives a very high preference to class **T** transactions when deferred subtransactions are present. This explains the fact that SL keeps the MDP of class **T** transactions nearly constant while the MDP of class **NT** transactions increases. It also explains the fact that SL gives lower MDP for class **T** than the class **NT**.

The results for the case where the triggered subtransactions execute either in deferred or immediate mode with equal probability ($q = 0.5$), are illustrated in figure 7 . The relative performance trend between the three policies is similar to that of the case where all subtransactions are of the deferred type, but the difference between the MDPs of class **T** transactions for the three policies is lower than the only deferred case because of the presence of immediate subtransactions. The qualitative performance of SL with respect to the triggering and non-triggering transactions is similar to that of the case where all the subtransactions are of the immediate type.

In the above results we observe that while the MDP of class **T** transactions is reduced by the SL and DIV policies, the MDP of the class **NT** transactions increases. This is desirable when class **T** transactions are more valuable to the application than class **NT** transactions. In order to look at the performance of the policies when 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 class **NT** transactions. The results are illustrated in figures 8, 9, and 10. We observe that PD always performs best if we give equal value to both **T** and **NT** class transactions. DIV is the next best and SL is the worst. PD works best here because DIV and SL are biased toward class **T** transactions which require more CPU time than class **NT** transactions. Although the DIV and SL reduce the MDP of class **T** transactions, since the majority of transactions belong to class **NT** (85%), the overall MDP increases in this case. For example, in figure 8, SL exhibits the worst performance and DIV is comparable to PD except at very high loads. So DIV performs comparably with the PD even when transactions of both classes have equal value. In the presence of deferred subtransactions, clearly PD provides the lowest MDP over DIV and SL in that order.

5.4.2 Average Slack vs. MDP

It is understandable that accounting for the dynamic work generated could result in better performance for class **T** transactions. However the explanation of difference in the performance between DIV and SL is not clear from the above experiments. One of the main differences between DIV and SL policies is that, while DIV always favors class **T** transactions, SL depending on the slack parameters, could favor either class. DIV changes (increases) the priority of class **T** transactions as it triggers other transactions, but keeps the priority of class **NT** unchanged. SL, on the other hand, modifies the priority of both the classes as they are executing. Hence, it is possible that depending on the slack parameters, the performance of the DIV and SL could change. Experiments were conducted where we varied the slack of class **T** transactions, holding the slack of class **NT** transactions constant for a load of 0.75. Figure 11 shows the result of this experiment where the class **T** trigger only immediate subtransactions. The MDP of each transaction class is given as a function of average slack of class **T** transactions for all the three policies. The average slack of class **T** is plotted on the x-axis and MDP is plotted on the y-axis. We observe that when the slacks of class **T** are low, SL provides a lower MDP for class **T** transactions than DIV, but performs worse for class **NT**. At higher slack values of class **T**, DIV starts performing better than SL in both classes. Figure 5 is an instance of this latter behavior at different loads. Also, increasing the slack of class **T** means increases the *estSlack* (remaining slack time) of class **T** transactions which results in relatively lower priorities for class **T** transactions compared to class **NT** transactions. This effect can be observed from the fact that the difference between MDPs of DIV and SL for class **NT** is decreasing as we increase the slack of class **T** transactions. We also conducted experiments where we held the slack of class **T** at a constant value and varied the slack of class **NT**. We don't present the graph of this experiment, but it suffices to say that the same kind of phenomenon was observed.

Figure 12 shows the result of an experiment where the class **T** transactions trigger only deferred subtransactions. The MDP of the different transaction classes is given as a function of average slack time for class **T** transactions. The average slack of class **T** transactions is plotted along the x-axis and the MDP is plotted along the y-axis. SL performs better than DIV for class **T** with low slack values and the DIV starts performing better than SL when the slacks are really high. As mentioned earlier, very large slacks for class **T** implies large *estSlack* for class **T** transactions and hence lower priorities for class**T** relative to class **NT** transactions. Hence, SL starts performing better for class

NT at high slack values of class **T** as seen in the figure. We observe that under SL MDP for class **T** reaches a minimum and then slowly starts to increase. This is the slack value of class **T** which performs the best for SL for a given slack value of class **NT**. Other slack values to the left are too tight and values to the right are too loose. The difference between the case where class **T** transactions trigger only immediate transactions and where they trigger only deferred transactions is that the MDP of class **T** where DIV starts performing better than SL is much higher in the first case. In other words, DIV starts performing better than SL for class **T** transactions that trigger only immediate subtransactions at *relatively* low slack values when compared to case where class **T** transactions trigger only deferred subtransactions. In all the cases seen so far, we observe that PD does worse than DIV and SL for class **T** and does better than DIV and SL for class **NT** transactions.

5.4.3 Analysis of Trade-offs between DIV and SL Policies

We observed in the previous experiments that DIV and SL give preference to **T** class transactions over the class **NT** transactions. In this section we will describe and evaluate two algorithms that provide the capability to tradeoff the MDPs for classes **T** and **NT**. The algorithms are constructed from DIV and SL in the following way. We introduce 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. It should be noted that experiment is conducted for a particular load and slack parameter values.

$$\text{ALPHA-DIV: } P_t(T) \ = \ \alpha * P_t^{DIV}(T) \ + \ (1-\alpha) * P_t^{PD}(T)$$

where P_t^{DIV} is the priority assigned to the transaction T by the DIV protocol and P_t^{PD} is the deadline of the parent transaction.

$$\text{ALPHA-SL: } P_t(T) \ = \ \alpha * P_t^{SL}(T) \ + \ (1-\alpha) * P_t^{PD}(T)$$

where P_t^{SL} is the priority assigned to the transaction by the SL protocol.

When α is zero both the policies reduce to PD. When α is one they reduce to DIV and SL, 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 as α is varied from zero to one. The results are found in figures 13, 14 and 15 where the MDPs of classes **NT** and **T** transactions are plotted. 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 class **T** transactions by 20 percent with a slight increase of only 1 percent in the MDP of class **NT** transactions, compared to the ALPHA-SL protocol which achieves lesser reduction at the cost of a 2 percent increase. Hence ALPHA-DIV protocol performs better than ALPHA-SL protocol in this case.

However, when the workload consists only of deferred transactions the ALPHA-SL protocol reduces the MDP of class **T** transactions to a very negligible value with a rise in the MDP of class **NT** transactions. ALPHA-DIV is not able to reduce the MDP of class **T** transactions that significantly. So if class **T** transactions have a higher value than class **NT** transactions, then ALPHA-SL is the protocol of choice in the deferred only case. ALPHA-DIV marginally outperforms ALPHA-SL at lower α values. Similar behavior is observed in the case where 50% of the subtransactions are deferred and 50% immediate. The results are illustrated in figure 15. Hence in te case where most subtransactions are deferred, ALPHA-SL provides more flexibility to achieve a higher reduction in the MDP of class **T** transactions with an increase in the MDP of the class **NT** transactions than the ALPHA-DIV policy.

5.4.4 Summary of Performance for the Real-Time Task Model

In summary, for a real-time task model, the dynamic priority assignment policies DIV and SL reduce the MDP of class **T** transactions, while increasing the MDP of class **NT** transactions. Between DIV and SL, the policy of choice depends upon the slacks of the transactions and the load in the system. SL is a pure slack based policy that performs well for a certain class of transactions depending on the slack of that class, its absolute value and its value relative to the other class. DIV always favors class **T** transactions. We also looked at two policies, ALPHA-DIV and ALPHA-SL, that will enable us to trade-off performance between different classes. Our hypothesis that accounting for the work generated dynamically and using knowledge about transactions will benefit the triggering transactions is substantiated by the results of this section.

5.5 Main Memory Database

We now consider the effects that data contention can have on the behavior of the policies. Since we are changing just one parameter any difference in the behavior is due to data contention. We introduce data contention by requiring every data access to be an exclusive access. It should be noted that the way we calculate the estimates in this case is the same as in real-time task case. The concurrency control algorithm we use is a modified wound-wait algorithm. According to the concurrency control mechanism, all of the subtransactions triggered by a transaction (whether in deferred or immediate mode) are considered part of the transaction and share the locks. Similarly, two subtransactions of the same parent transaction share the locks. All of the subtransactions and the parent transaction release the locks at their commit time which occurs after the parent transaction and the deferred subtransactions have finished. Since different subtransactions may have different priorities, deadlocks could occur in spite of using the wound-wait protocol. Deadlocks are prevented by checking for deadlocks each time a transaction waits for another transaction and abort 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].

The results are illustrated in figures 16 and 17 for workloads consisting of only immediate, and only deferred subtransactions, respectively. In the case that all subtransactions are immediate, DIV and SL provide better performance to class **T** while providing worse performance to class **NT** as was observed in the case with no data contention. Figure 16 shows the performance for varying loads at a fixed slack distribution. The relative performance of DIV and SL change with the varying load for class **T** transactions, but for class **NT** transactions DIV performs better than SL.

The level of data contention could affect the performance of the priority assignment policies. In order to study the effect of data contention we changed the number of objects in the system. For a given load, reducing the number of objects will amount to a higher data contention. Figures 18 and 19 show the immediate and deferred cases of workload where the number of objects is 500. For the case where all the subtransactions are of type immediate, SL starts performing better than DIV for class **T** at high load values. This is not the case when the number of objects is 2000.

In the case where all the subtransactions are triggered in immediate mode, for the main memory database (figure 16) the MDP of class **NT** keeps increasing for SL when compared to DIV. This is not the case for the immediate only case of real-time task model (figure 5). The difference between the two cases is that there is data contention in the main memory case. Moreover, when the level of data contention is higher (figure 18) the difference between the MDPs of class **NT** between DIV and SL has increased. One could attribute this to the slacks being too tight. But that should affect DIV and PD too. In our model, SL estimates the number of steps but does not take into account the blocking delays due to data contention. This could lead SL to underestimate the time taken to execute the remaining steps (i.e., overestimate the *estSlack*) and miss the deadlines. But the *estSlack* (remaining slack time) of all the transactions is being uniformly overestimated. If we assume that on an average all transactions undergo equal amount of blocking delays, then SL can overestimate *estSlack* of transactions that have not been blocked until a certain point in time and the transaction can miss its deadline.

We observed that the number of concurrency control aborts in the case of SL was two to three times the number of aborts in PD or DIV. The number of aborts were highest both for class **T** and class **NT** transactions for SL. DIV had higher aborts for class **NT** than PD and PD had higher aborts for class **T**. This is explainable because DIV favors class **T** transactions over class **NT** and PD favors class **NT** over class **T**. For SL the high number of aborts for class **NT** was definitely affecting its performance for this class. In a policy like SL the relative priority ordering between two transactions can keep changing throughout the life of these transactions depending on how many steps are executed by these two transactions whereas in a deadline based priority scheme the relative ordering is fixed. When there is data contention then the number of steps executed by transactions that start around the same time can differ which would affect the *estSlack* of these transactions. Transactions that don't experience contention could end up with more *estSlack* than transactions that have experienced some data contention.

As an example, let us consider two transactions T_1 and T_2 that arrive at time 1 and 3 respectively.

Let their lengths be 5 units each and deadlines be 12 ($d(T_1)$) and 14 ($d(T_2)$) respectively. At time 5 let us say T_1 has completed 4 steps and T_2 has completed 1 step because it faced data contention. Therefore, $X_5(T_1)$ is 1 and $X_5(T_2)$ is 4. Now *estSlack* of T_1 at time 5 is $S_5(T_1)$, i.e., 12 - 5 - 1 = 6 and the *estSlack* of T_2 at time 5 is $S_5(T_2)$, i.e., 14 - 5 - 4 = 5. *estSlack* of T_2 at time 5 is less than *estSlack* of T_1 at time 5. In such a scenario, if T_2 needs a data item locked by T_1 then T_2 will abort T_1, and T_1 will be restarted with a very low slack (a high priority) and abort other transactions that are nearing their end. It is reasonable to assume that the chances are higher for a transaction that came earlier to hold the lock on a data item that is wanted by a transaction that came later than the other way. We conjecture that this kind of a scenario is happening in SL resulting in large number of concurrency control aborts.

In these experiments, as expected, the absolute MDP values are higher 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. One more phenomenon that occurs is, for immediate only case at low loads with tight slacks for class **T** transactions, PD does better than SL. This is due to the high number of concurrency control aborts in the case of SL.

5.5.1 Summary of Performance in Main Memory Databases

The same results that DIV and SL perform better than PD for class **T** and PD performs better than DIV and SL for class **NT**, are also seen in the case of main memory databases. But there is a difference in the performance between these two cases mainly due to the high number of concurrency control aborts of SL. At low loads, high concurrency control aborts lead SL to behave worse than PD for class **T** transactions that trigger only immediate subtransactions. The very high number of concurrency control aborts of SL for class **NT** transactions makes SL behave worse than DIV for class **NT**. This problem can be solved by considering the amount of work performed by a transaction before aborting it, and checking for *feasibility* (i.e., if the transaction has a chance to complete before the deadline). Also, accounting for concurrency control delays in estimating the slack might help.

5.6 Disk Resident Database

The last scenario considered is one where the objects are not always in main memory in which case it is necessary to retrieve it from the disk. In our simulator, we do not explicitly manage the buffer. The database buffer is modeled using a parameter h, which is the probability that a page is resident in the buffer. Hence, with a probability $1 - h$ the disk subsystem is accessed. In our calculation of estimates we take the i/o time in to account. The estimates are calculated as follows.

$$
\begin{array}{lcl}
t^{io} & : & \text{average time taken to do an i/o} \\
X_t(T) & = & L_t(T) * (1 + ((1-h) * t^{io})) \\
m_t^{imm}(T) & = & L_t(T) * \rho * q \\
m_t^{def}(T) & = & L_t(T) * \rho * (1-q)
\end{array}
$$

$$\overline{X}^{imm}(T) = L(T^{sub}) * (1 + ((1-h) * t^{io}))$$
$$\overline{X}^{def}(T) = L(T^{sub}) * (1 + ((1-h) * t^{io}))$$

It should be noted that all other estimates can be derived from the above estimates. Table 3 shows important parameter values that we used in our experiments.

Parameter	**Setting**
N_{Disk}	2
t^{io}	30 units
h	0.9

Table 3 : Setting for Disk Resident Databases

We experimented by varying the slack of class **T** while maintaining the same slack distribution for **NT** for a particular load value (0.35). Figures 20 and 21 show the results of this experiment for the cases where all the subtransactions are triggered in immediate mode and deferred mode, respectively. The MDP of the different transaction classes is given as a function of average slack time for class **T** transactions. The MDP of the different transaction classes is given as a function of average slack time for class **T** transactions. The results are similar to the real-time task case for class **T**, i.e., the MDP of class **T** is reduced by DIV and SL policies. When the slack values are lower, SL performs better than DIV and at higher slack values DIV performs better than SL. Also, as in the real-time task case there is a particular slack value of class **T** (relative to class **NT** slack) where SL performs the best. But for class **NT** SL performs at least as well as DIV. The reason is that SL accounts for i/o times in its calculation of slack for class **NT** transactions. The lack of estimate for blocking delays due to concurrency control could get subsumed by the i/o estimate since it is so much larger. On the other hand DIV does not account for the i/o times in the case of class **NT** transactions and hence, does not perform as well as SL. One can think of i/o as the dynamic work that is generated by the class **NT** transactions. Since SL takes into account this dynamic work that is being generated by class **NT**, it performs better than DIV and PD for this class.

One anomaly that can be observed in the case where all the subtransactions are triggered in deferred mode (figure 21) is that in spite of increasing the slack, the MDP of class **T** transactions for PD does not decrease at high slack values (9.75, 11.25). This is because very high slacks result in long deadlines (low priorities) that were causing a lot of concurrency control aborts and restarts resulting in a lot of wasted resources. This problem can be solved by considering the amount of work performed by a transaction before aborting it, checking for *feasibility* (i.e., if the transaction has a chance to complete before the deadline) when restarted and giving higher priorities to transactions that are restarted.

We also experimented by varying the load of the system while maintaining fixed slack distributions. Here SL performs the best for both class **T** and class **NT** for both the cases where only immediate subtransactions are triggered and where only deferred subtransactions are triggered. In the case where all the subtransactions are triggered in deferred mode, SL reduces the MDP of both the classes almost by half in over loaded situations.

5.6.1 Summary of Performance in Disk Resident Database

In the disk resident database case DIV and SL perform better than PD for class **T**. DIV performs the worst for class **NT**. SL performs as good or better than PD for class **NT**. At high loads SL performs the best for both classes **T** and **NT**. Since SL accounts for the i/o in its estimates for class **NT** it tends to perform better than DIV consistently for this class. Our claim that accounting for dynamic work generated, and using the knowledge about the i/o requirements and triggering characteristics of transactions will improve the performance is substantiated by the above results.

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 introduced a simple baseline policy PD and two other policies, namely DIV and SL, which assign priorities taking into consideration the active work generated by a transaction. The policies use different amounts of information about the transactions. PD uses the deadline to schedule the transactions, DIV uses estimates of execution times of the parent transaction and subtransactions that it has triggered and SL uses estimates of execution times of the parent transaction, triggered subtransactions and estimates about the subtransactions that might be triggered in the future.

The main conclusions are:

- There are three *key* parameters: load, slack parameter of class **T** and slack parameter of class**NT** of the workload which determine the relative performances of the three algorithms.
  - For a real-time task case in all of the load/slack space that we have experimented with, DIV and SL perform better than PD for class **T**. PD performs best for class **NT** transactions. It might be possible for SL to perform better than PD for class **NT** for extremely low slack values. But it would be almost impossible for DIV to perform better than PD for class **NT** because both schedule class **NT** depending on the deadline and DIV favors class **T**. So DIV can do only as good as PD for class **NT**. Similarly, it is difficult to conceive of cases where PD could do better than DIV for class **T**.
  - In a main memory database setting with data contention, the concurrency control aborts could change the performance of the algorithms. For instance. at very low load values for the immediate only case, SL could do worse than PD for class **T** because of the high number of concurrency control aborts in the case of SL. Similarly, for the deferred only case, DIV could do worse than PD for class **T**. SL performs consistently worse than DIV

for class **NT** at low slack values because of the high number of concurrency control aborts and restarts of transactions with very high priorities.

  – In a disk resident database setting, DIV and SL perform better than PD for class **T**. SL's performance is comparable or better than that of PD for class **NT**. This improvement in the performance of SL for class **NT** is because of the fact that it accounts for the i/o time in its calculation of estimates.

- For class **T** the sources of unpredictability are triggering of other subtransactions, concurrency control delays and i/o and for class **NT** they are concurrency control delays and i/o. In the real-time task model, SL and DIV account for active work and hence perform better than PD for class **T**. For class **NT** there is no unpredictability. But since SL and PD perform better for class **T** they will penalize class **NT**. DIV behaves qualiatively the same for a main memory database setting as it does for real-time task setting. But SL due to the high number of concurrency control aborts performs relatively poorly for class **NT** when compared to the real-time task model. SL, in the case of a disk resident database accounts for all sources of unpredictability in its estimate of $estSlack$ (remaining slack time) for both class **T** (triggering and i/o) and class **NT** (i/o) except blocking delay which might not be significant compared to i/o. DIV accounts for the sources of unpredictability as SL does for class **T** but does not account for the i/o for class **NT**. Hence SL performs better than DIV for class **NT**.

- Using the extra information like estimates of execution times enhances the performance of class **T** transactions as can be seen by DIV and SL performing better than PD in all three settings. This benefit comes at the cost of increased MDP for class **NT**. Using the extra information about i/o makes SL perform better than DIV and PD for class **NT** in the disk resident database setting.

Some of the extensions we want to address are

- Experiment with variations of DIV and SLACK policies that use 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 $L_t(T)$, $X_t(T)$ and ρ on the performance of the DIV and SL policies.

- Evaluate algorithms that change the priority of triggered subtransactions during their execution. This will be particularly useful to study systems where the triggered subtransactions can trigger further subtransactions.

- Consider the concurrency control blocking delays in the estimate of slacks.
- Consider the amount of work a transaction has completed before aborting due to a concurrency control conflict.

References

[1] R.K. Abbott, and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *ACM Trans. on Database Systems*, Vol. 17, No. 3, 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.*, Vol. 3, No. 3, 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 SIGMOD, 1990.

[5] Special Issue on Active Databases, Bulletin of The Technical Committee on *Data Engineering*, Vol. 15, No. 1-4, Dec. 1992.

[6] J. Huang, J. Stankovic, D. Towsley, and K. Ramamritham, "Experimental Evaluation of Real-time Transaction Processing," *Proceedings of the 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 SIGMOD, June 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*, Vol. 1, No.2, 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] S. H. Son and S. Park, "Scheduling Transactions for Distributed Time-Critical Applications", *Readings in Distributed Computing Systems*, edited by T. L. Casavant and M. Singhal, IEEE Computer Society Press, 1994.

[16] 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.

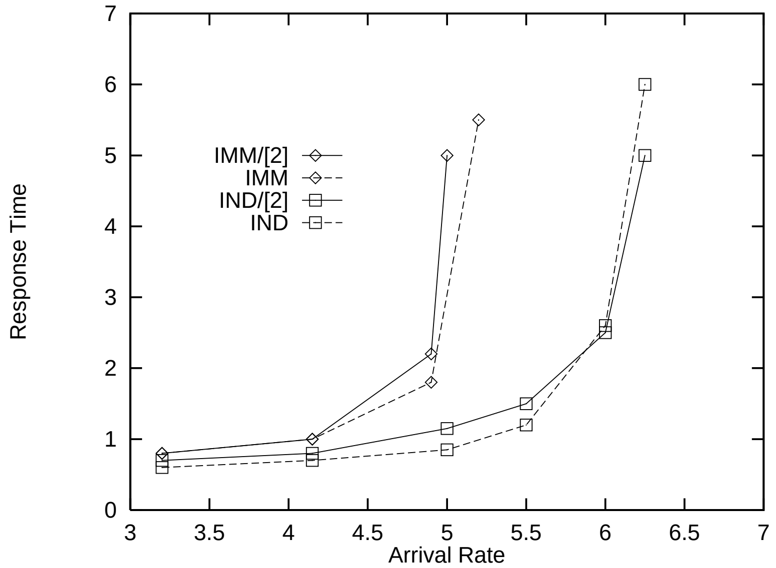

Figure 3: **Validation of Active Part of Simulator**

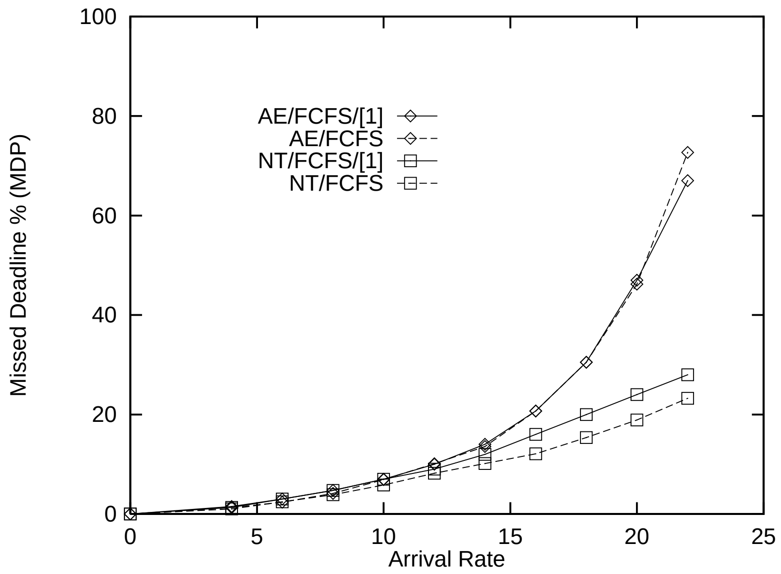

Figure 4: **Validation of Real-time Part of Simulator**

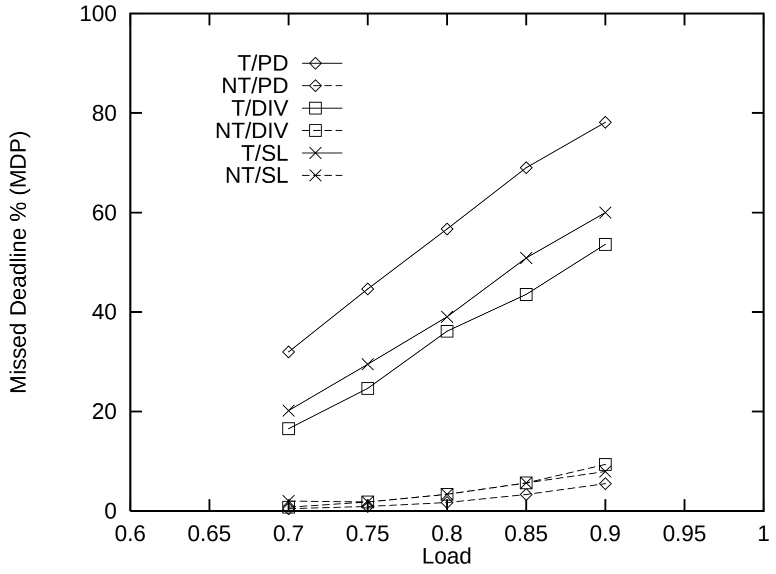

Figure 5: **Only Immediate/ Task Model**

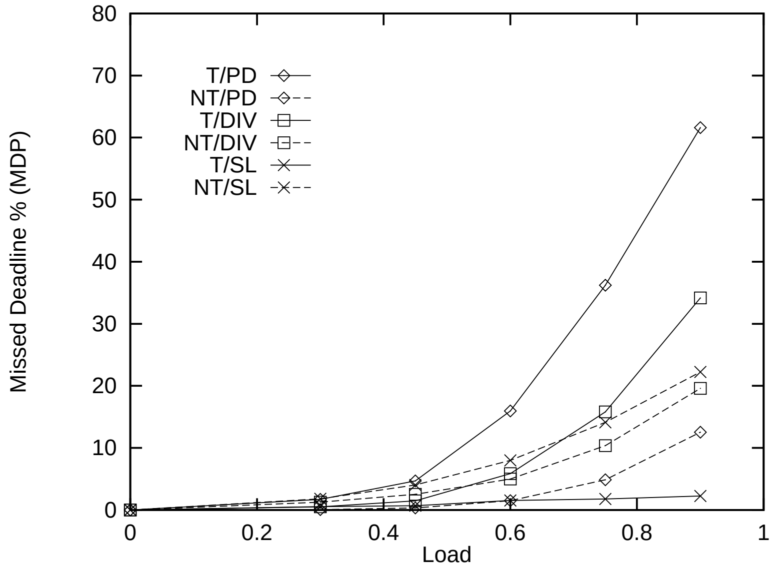

Figure 6: **Only Deferred / Task Model**

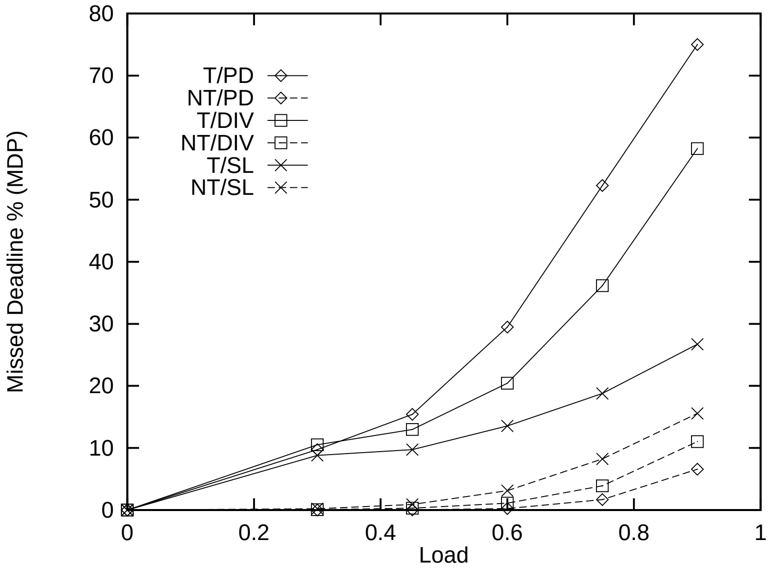

Figure 7: **Immediate-Deferred / Task Model**

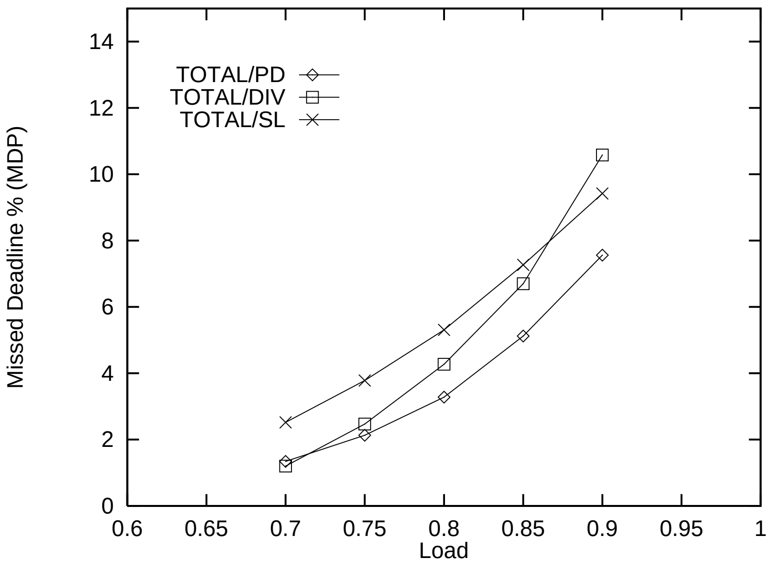

Figure 8: **Only Immediate / Task Model / Total**

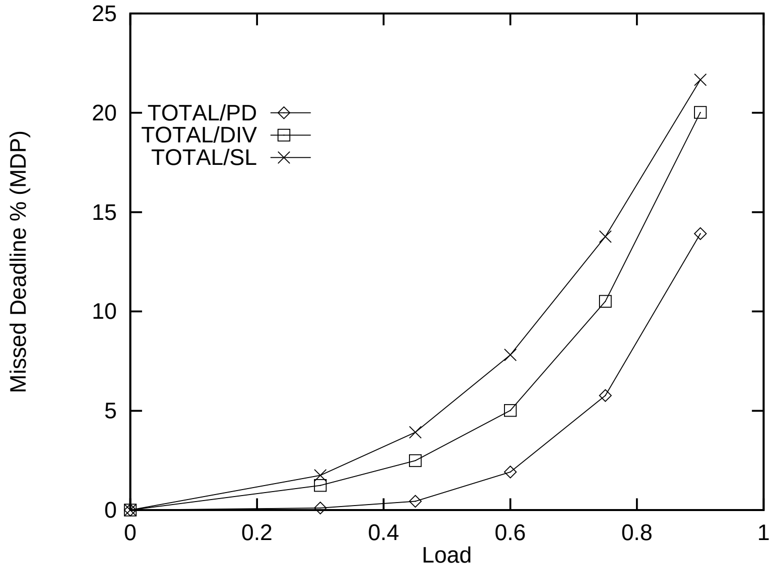

Figure 9: **Only Deferred / Task Model / Total**

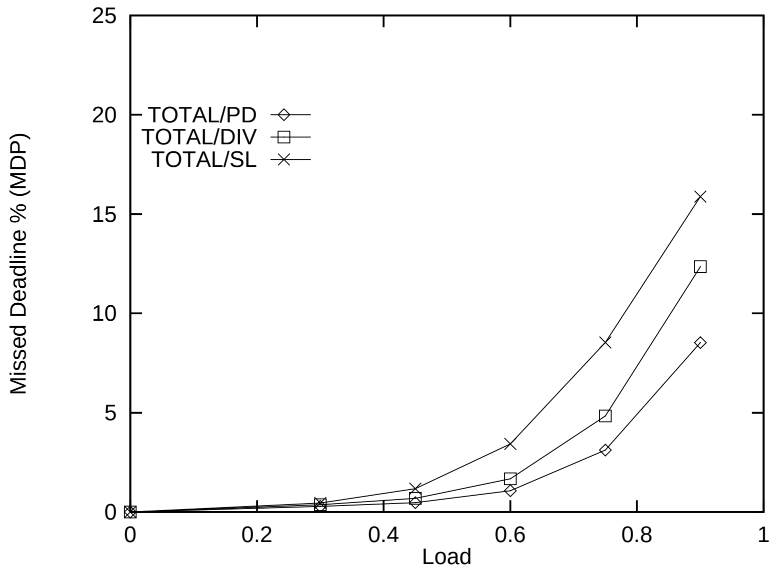

Figure 10: **Immediate-Deferred / Task Model / Total**

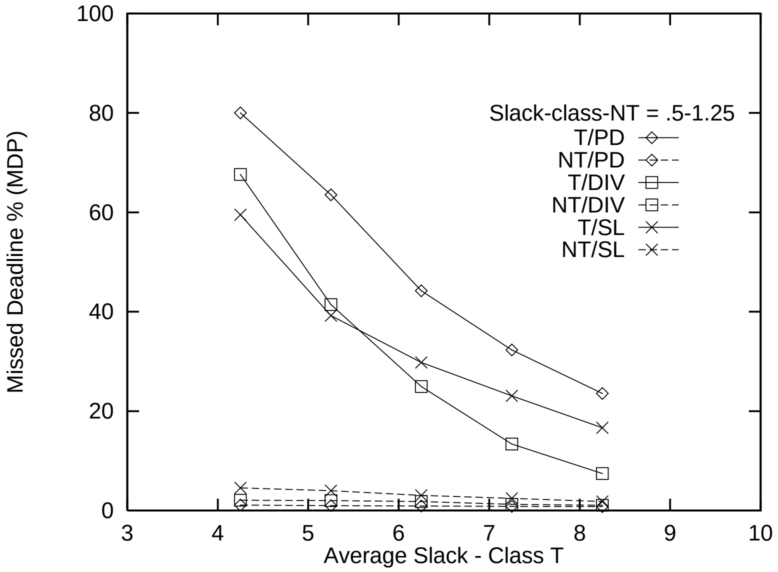

Figure 11: **Only Immediate / Task Model / Vary Class T Slack**

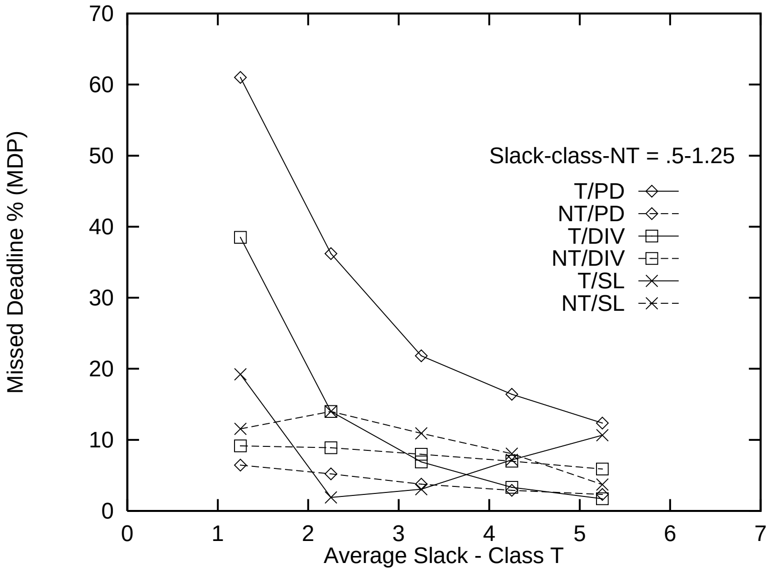

Figure 12: **Only Deferred / Task Model / Vary Class T Slack**

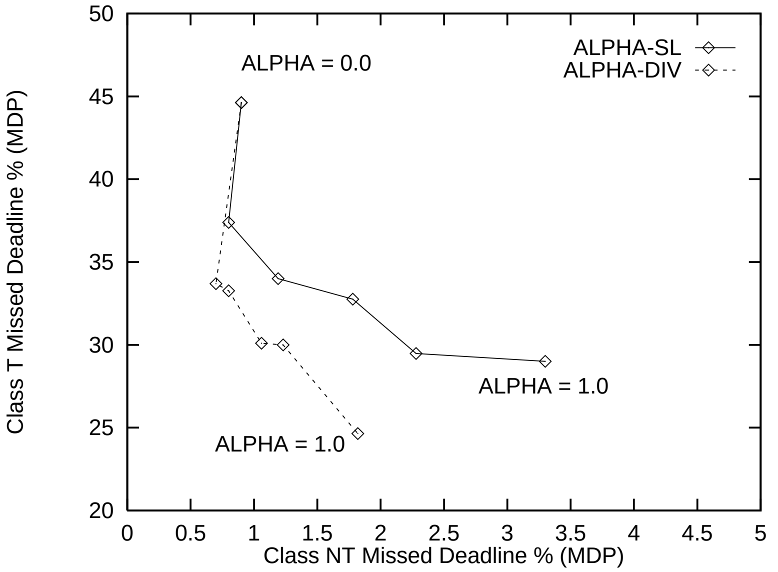

Figure 13: **Only Immediate / Task Model / Trade-offs**

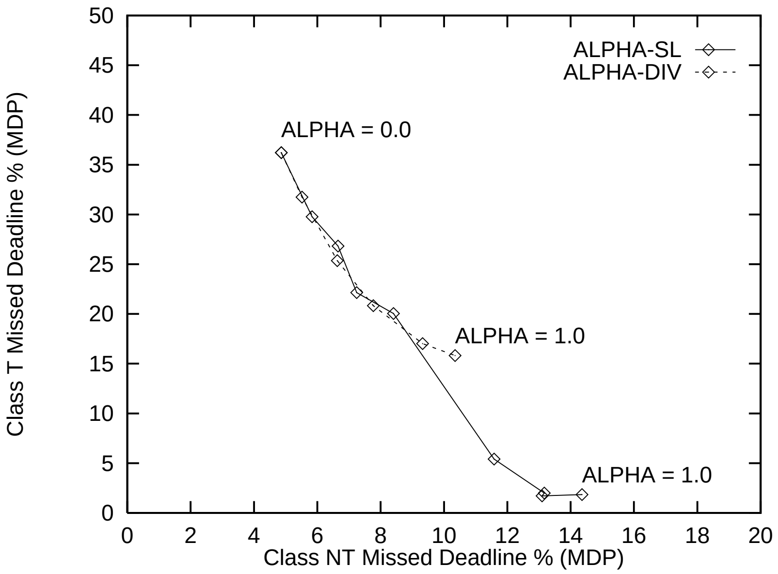

Figure 14: **Only Deferred / Task Model / Trade-offs**

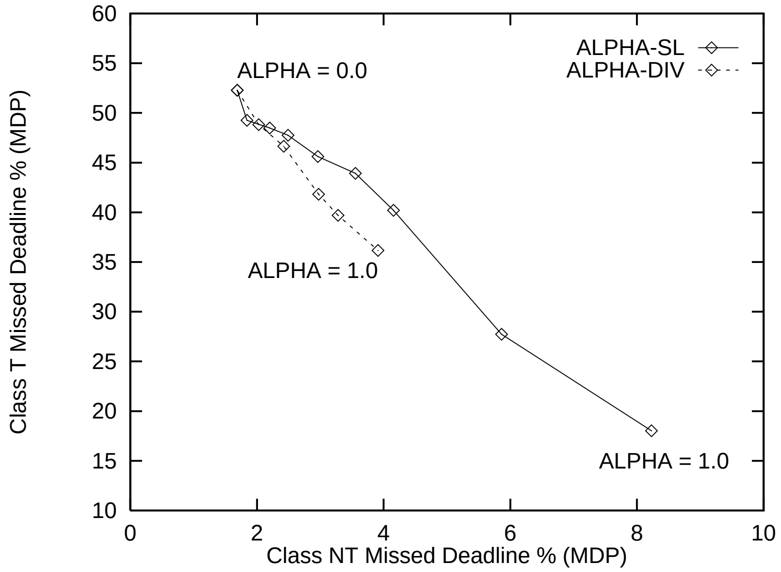

Figure 15: **Immediate-Deferred / Task Model / Trade-offs**

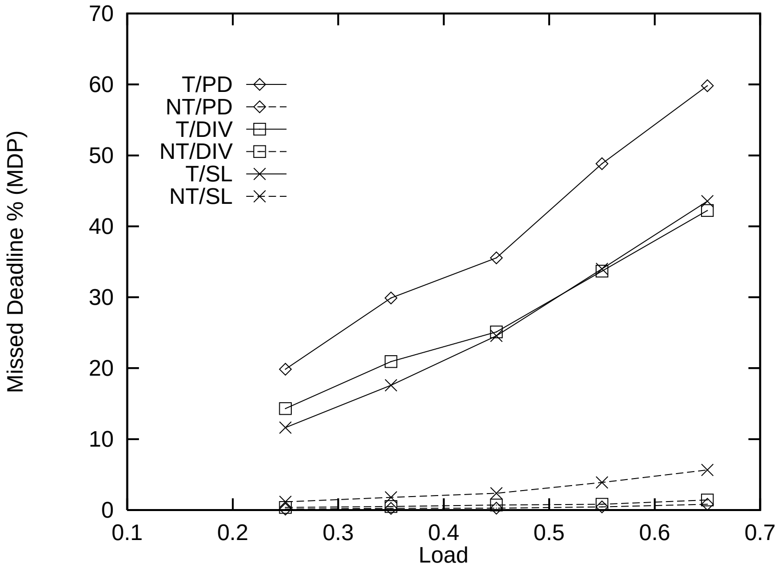

Figure 16: **Only Immediate / Main Memory DB / 2000 Objects**

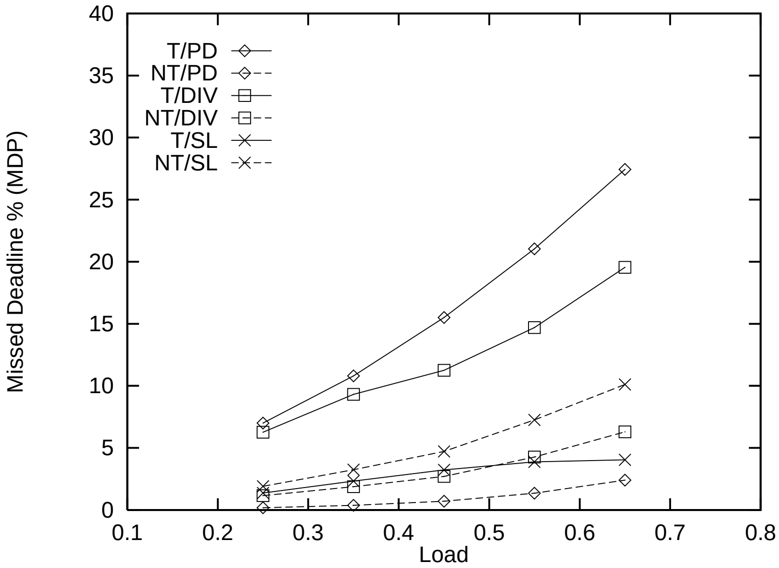

Figure 17: **Only Deferred / Main Memory DB / 2000 Objects**

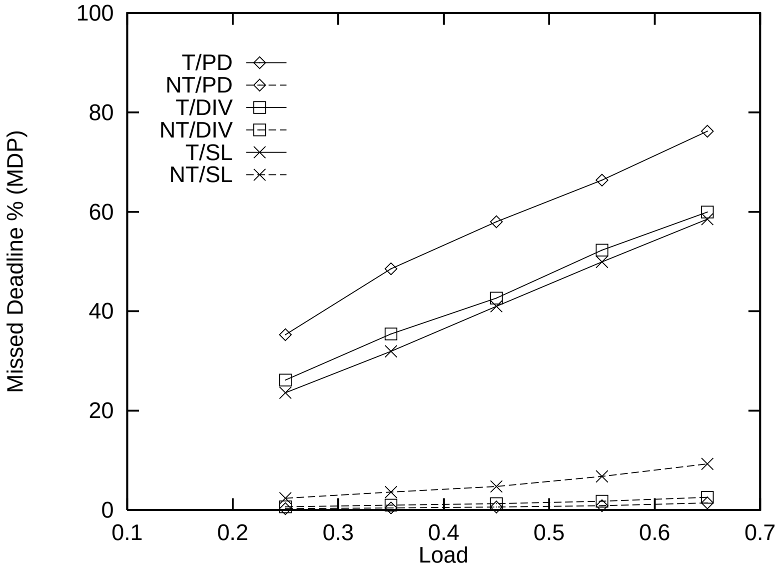

Figure 18: **Only Immediate / Main Memory DB / 500 Objects**

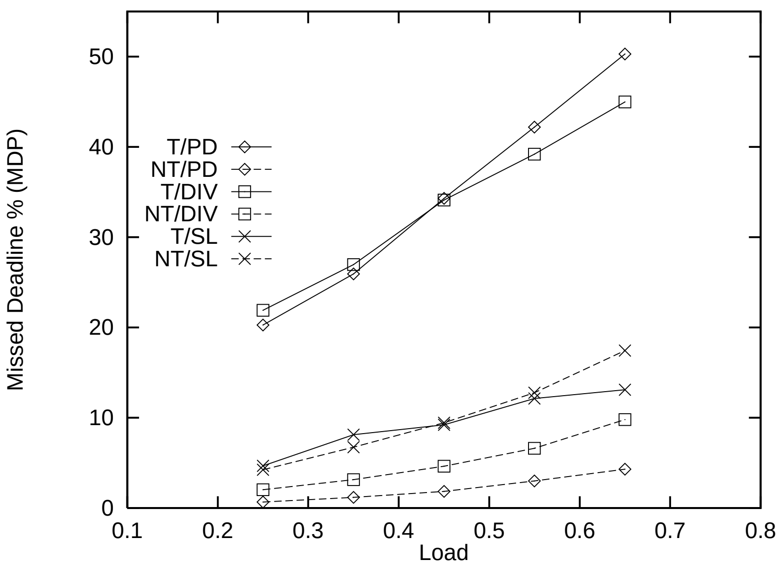

Figure 19: **Only Deferred / Main Memory DB / 500 Objects**

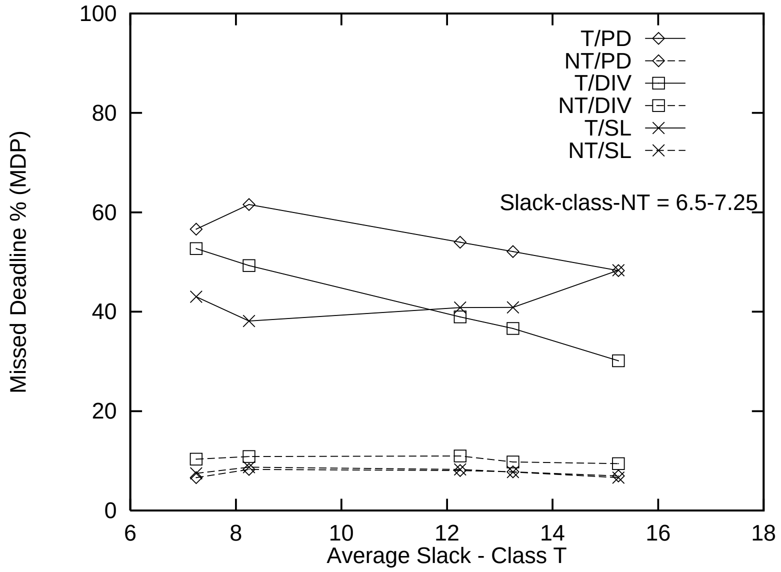

Figure 20: **Only Immediate / Disk Resident DB / 2000 Objects**

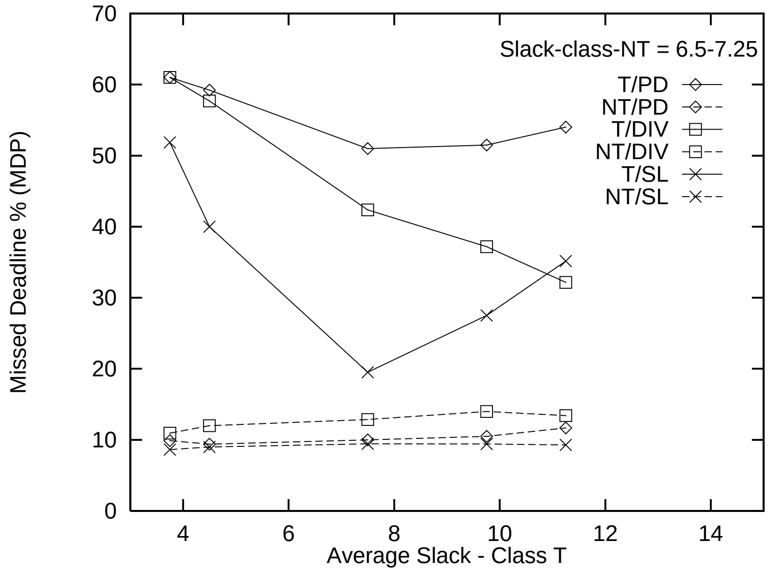

Figure 21: **Only Deferred / Disk Resident DB / 2000 Objects**