EXPERIMENTAL EVALUATION OF REAL-TIME TRANSACTION PROCESSING

J.Huang, J.A. Stankovic, D. Towsley
and K. Ramamritham

COINS Technical Report 89-48

April 1989

Experimental Evaluation of Real-Time Transaction Processing*

Jiandong Huang
Department of Electrical and Computer Engineering
University of Massachusetts

John A. Stankovic
Don Towsley
Krithi Ramamritham
Department of Computer and Information Science
University of Massachusetts

April 1989

Abstract

This paper presents results of empirical evaluations carried out on the RT-CARAT testbed. This testbed was used for evaluating a set of integrated protocols that support real-time transactions. Using a basic locking scheme for concurrency control, several algorithms for handling CPU scheduling, data conflict resolution, deadlock resolution, transaction wakeup, and transaction restart are developed and evaluated. The performance data indicates (1) that the CPU scheduling algorithm has the most significant impact on the performance of real-time transactions, (2) that various conflict resolution protocols which directly address deadlines and criticalness produce better performance than protocols that ignore such information, (3) that deadlock resolution and transaction restart policies tailored to real-time constraints seem to have negligible impact on overall performance, (4) that criticalness of transactions is a very important factor in real-time transaction processing, and (5) that deadline distributions strongly affect transaction performance. We believe that these results represent the first experimental results for real-time transactions from an actual system.

*This work was supported, in part, by the U.S. Office of Naval Research under Grant N00014-85-K0398, by the National Science Foundation under Grant DCR-8500332, and by Digital Equipment Corporation.

1 Introduction

A real-time database is a database system where (at least some) transactions have explicit timing constraints such as deadlines. In such a system, transaction processing must satisfy not only the database consistency constraints but also the timing constraints. Real-time database systems can be found in program trading in the stock market, radar tracking systems, battle management systems, and computer integrated manufacturing systems.

Most work on databases focuses on query processing and database consistency, but not on meeting any time-constraints associated with transactions. On the other hand, real-time systems research deals with task scheduling to guarantee responses within deadlines, but it has largely ignored the problem of guaranteeing the consistency of shared data, especially for data that resides on a disk. Therefore, the traditional mechanisms for consistency enforcement and for timing constraint enforcement are not suited for real-time transaction processing. Rather, algorithms from the two areas have to be extended and combined. To study real-time transaction systems, some new criteria are required which unify timing and consistency constraints, specify the properties of transactions, define correctness, and specify proper metrics.

Several papers have recently been published in the area of time-critical database systems [11,1,2,9,4,3]. The topics covered in these papers include the identification of the characteristics of real-time transactions as well as the model of the underlying real-time operating system primitives [11,1,3], access control and conflict resolution [11,2], transaction scheduling [2,3], and deadlock prevention [9]. These previous studies provide insight into many of the issues encountered in the design of real-time transaction systems, and present some basic ideas for solving some of the problems.

Because research on real-time transactions is still in its infancy, current work has many shortcomings. First, they are incomplete. For example, timing constraints and criticality are two important but independent factors in describing real-time transactions. The relation between the two factors and their combined effect with respect to system performance has not been addressed. Second, some of the previous work concentrates on only one or two specific issues, thus lacking an integrated systemwide approach. Third, none of the ideas or proposed algorithms in the previous studies, though well thought out, are evaluated in real systems. Indeed, only [2] presents any experimental results. However this work is based on simulation and the effect of many important overheads, such as locking and deadlock detection, are ignored.

In the research described here we have implemented and evaluated 4 real-time CPU scheduling algorithms, 5 policies for data conflict resolution, 3 policies for transaction wakeup, 4 deadlock resolution policies, and 3 transaction restart policies. We compare various combinations of these algorithms to each other and to a baseline system where timing constraints are ignored. We also studied:

- the relationship between transaction timing constraints and criticality, and their combined effects on system performance,
- the behavior of a CPU bound system vs. an I/O bound system, and

- the impact of deadline distributions on the conducted experiments.

The performance data indicates (1) that the CPU scheduling algorithm is the most significant of all the algorithms in improving the performance of real-time transactions, (2) that conflict resolution protocols which directly address deadlines and criticalness can have an important impact on performance over protocols that ignore such information, (3) that deadlock resolution and transaction restart policies tailored to real-time constraints seem to have negligible impact on overall performance, (4) that criticalness is a very important factor in real-time transaction processing, and (5) that deadline distributions strongly affect transaction performance. We believe that these empirical results represent the first experimental results for real-time transactions on an actual system.

The rest of this paper is organized as follows. In section 2, we describe a real-time transaction model and summarize our basic assumptions. The protocols and policies for real-time transaction processing are described in section 3, where we focus on CPU scheduling, data conflict resolution, transaction wakeup policies, deadlock resolution, and transaction restart. In section 4, we present our real-time database testbed, RT-CARAT, where the proposed protocols and policies are implemented and evaluated. The performance results are demonstrated and discussed in section 5. Finally, we make concluding remarks in Section 6.

2 Transaction Model

In this study we investigate a real-time database that is secondary storage resident and centralized. As is usually required in traditional database systems, we also require that all the real-time transaction operations should maintain data consistency as defined by serializability[1]. In our system, serializability is enforced by using the two-phase locking protocol.

A real-time transaction is characterized by four factors:

- a deadline,
- a criticalness,
- a value function, and
- a size (not always known) that is dependent on
  - the number of data objects to be accessed, and
  - the amount of computation to be performed.

[1]The property of data consistency may be relaxed in some real-time database systems, depending on the application environment and data properties. The relaxation of consistency is not considered in this paper.

Deadlines constitute the time constraints of real-time transactions. Depending on the timing requirements, transactions can be defined as hard real-time and soft real-time [11]. The transactions considered here are soft real-time, i.e. transactions should meet their deadlines, but there may still be some (but diminishing) value for completing the transactions after their deadlines. Here the goal is to maximize the number of transactions that meet their deadlines.

The criticalness represents the importance of a transaction. In real-time database systems, not all transactions are equally important. For instance, in a stock market database, a transaction that updates the database with new information about the financial market may be more critical than the one that searches the database for arbitrage opportunities. Thus, the criticalness of a transaction will have an impact on transaction execution.

Criticalness and deadline are two independent characteristics of real-time transactions. A transaction with a short deadline does not mean that it has high criticalness. Transactions with the same criticalness may have different deadlines and transactions with the same deadline may have different criticalness values. The use of a value function is one way to combine the effects of transaction's deadline and criticalness. In a real-time database each transaction imparts a value to the system which is related to its criticalness and to when it completes execution (relative to its deadline). We define the value of a transaction as a function of its criticalness, deadline and current time of the system. Basically, the higher the criticalness of a transaction, the larger its value to the system. On the other hand, the value of a transaction is time-variant. A transaction which has missed its deadline will not be as valuable to the system as if it completed before its deadline. The following formula expresses the value of a transaction T: [2]

$$\mathrm{V(T)} = f(c,d,t) = \begin{cases} c, & t \le d \\ c + (1 - t/d) \times c^2, & d < t < d + d/c \\ 0, & t \ge d + d/c \end{cases}$$

where t - current time;
d - deadline of transaction T;
c - criticalness of transaction T.

As an example, Figure 2 shows the value functions of two transactions. In this model, a transaction has a constant value before its deadline. The value decays if the transaction passes its deadline and decreases to zero at time z. In this paper we model the decay rate as a function of deadline and criticalness where z is inversely proportional to the criticalness, which means that the higher the criticalness of a transaction, the faster it loses its value after its deadline. At the extreme, if a transaction is extremely critical ($c \rightarrow \infty$), its value drops to zero immediately after its deadline.

Clearly, given value functions, real-time transactions should be processed in such a way that the value of completed transactions is maximized. If a transaction does not

[2] There are other ways to present and interpret transaction value functions depending on the application. [1]

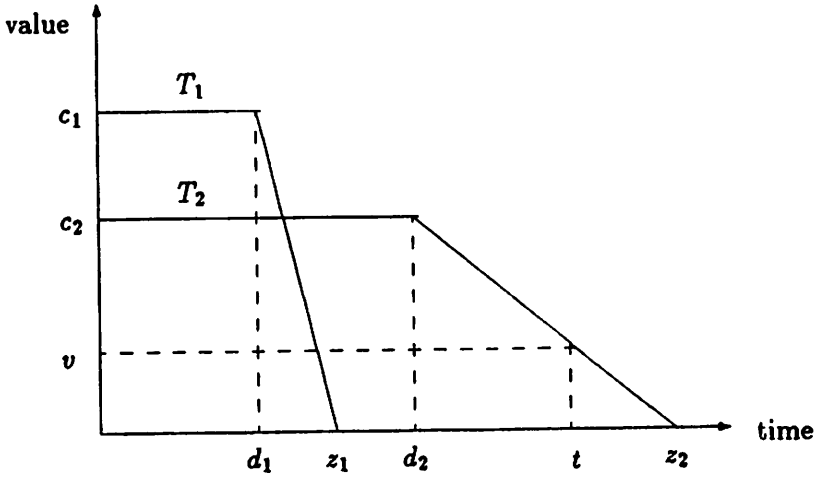

Figure 1: Value functions for transaction T_1 and T_2

complete before time z (see Figure 2), it should abort as soon after z as is feasible, since its execution after z does not contribute any value to the system at all. (Note that aborting the transaction at exactly point z would require preempting the currently executing transaction and subsequently may cause the preempted transaction to miss its deadline.) On the other hand, a transaction aborted because of deadlock or data conflict may be restarted if it has not missed its deadline or it may still provide some value to the system.

For some of our protocols we assume that the estimated execution time of a transaction is known. This information may be helpful in making more informed decisions regarding which transactions are to wait, abort, or rollback. This hypothesis is tested in our experiments.

3 Real-Time Transaction Processing

In this section we present the various real-time protocols and policies that we developed, implemented, and tested.

3.1 CPU Scheduling

In traditional database systems, transactions compete for the CPU based on any number of algorithms. Such scheduling usually emphasizes fairness and an attempt to balance CPU and I/O bound transactions. These scheduling schemes are not adequate for real-time transactions. In real-time environments, transactions should get access to the CPU based on criticalness and deadline, not fairness. If the complete semantics of transactions, e.g. the data access requirements and timing constraints, are known in advance, then scheduling can be done through transaction preanalysis [3]. On the other hand, in many cases there

will exist no complete knowledge of timing and a priority based scheduling scheme may be used, where the priority is set based on deadline, criticalness, length of the transaction, or some combination of these factors.

We have implemented four simple CPU scheduling algorithms. Since CPU scheduling turns out to be the most important factor, we expect to study other CPU scheduling algorithms in the near future. The four scheduling algorithms are as follows:

- **Scheduling the most critical transaction first** (MCF)
- **Scheduling by earliest deadline first** (EDF)
- **Scheduling by criticalness and deadline** (CDF): In this algorithm each transaction is assigned a priority at the time of arrival based on the factor *relative_deadline/criticalness*, where *relative_deadline* is the difference of the transaction deadline and its start time.
- **Scheduling the longest execution first** (LEF): In this algorithm, each transaction is assigned a priority based on the amount of database operations that the transaction has performed. Transaction priority is increased by one after a certain number of operations so that the longer a transaction executes the higher its priority. Here the point is that in general, the longer a transaction holds its locks on data objects, the more chance it will block other transactions because of data conflict. If we let the transaction priority increase as it gets close to its end, then the transaction will release its locks more quickly, thus reducing the blocking time for other transactions and, in turn, reducing the chance for deadlock.

Note that for each of these four algorithms, as a transaction is committing, its priority is raised to the highest value among all the active transactions. This enables a transaction in its final stages of processing to complete as quickly as possible so that it will not be blocked by other transactions. This policy also reduces the chance for the committing transaction to block other transactions. The idea behind this was discussed in the LEF algorithm. In all four algorithms, the transactions are preemptable, i.e. an executing transaction can be preempted by a transaction with a higher priority.

3.2 Conflict Resolution Protocols (CRP)

Two or more transactions have a data conflict when they require the same data in non-compatible lock modes. The conflict should be resolved according to the characteristics of the conflicting transactions. In this study, we only consider data conflict over exclusive locks. Here we present five protocols for conflict resolution. The first protocol is based on the notion of a *virtual clock*, the second on a pairwise comparison of the results from combining transaction parameters, the third on separating deadlines and criticalness, the fourth on deadline, criticalness, and a dynamic estimation of remaining execution time, and the fifth is only based on criticalness.

In the following descriptions, T_R denotes the transaction which is requesting a data item D and T_H the transaction that is holding a lock on D. d_T and c_T represent the deadline and the criticalness of a transaction T, respectively. The five protocols have the same algorithmic structure. The structure is as follows:

```
T_R requests a lock on the data item D
if no conflict
  then T_R accesses D
  else call CRPi    (i = 1,2,3,4,5)
end if
```

3.2.1 Protocol 1 (CRP1): Based on a virtual clock

Each transaction, T_i, has a virtual clock [8,12] associated with it. The virtual clock value, $VT(T_i)$, for transaction T_i is calculated by the following formula.

$$VT(T_i) = t_{s_i} + \beta_i * (t - t_{s_i})$$

where t_{s_i} is the start time of transaction T_i; t is the current time and β_i is the clock running rate which is proportional to transaction T_i's criticalness. The more critical the transaction, the larger the value β. The protocol controls the setting and running of the virtual clocks. When transaction T_i starts, VT(T_i) is set to the current real time t_{s_i}. Then, the virtual clock runs at rate β_i. That is, the more critical a transaction is, the faster its virtual clock. In this work, $\beta_i = c_{T_i}$. The protocol is given by the following pseudo code.

```
if d_TR > d_TH
  then T_R waits
  else
       if VT(T_H) < d_TH
         then T_R aborts T_H
         else  T_R waits
       end if
end if
```

In this protocol, transaction T_R may abort T_H based on their relative deadlines, and on the criticalness and elapsed time of transaction T_H. Also note that in this protocol, when the virtual clock of transaction T_H has surpassed its deadline, it is never aborted.

In general, this protocol synthesizes a number of considerations.

1. Information on the amount of elapsed time for transactions $(t - t_s)$ is used in the abort decision. That is, if a transaction has existed for only a short time it is more likely to be aborted, and vice versa. Note that the amount of computation time used by the transactions is implicitly taken into account here. The larger the computation time used, the larger the elapsed time is likely to be.

2. The deadlines of transactions are used. Only if the transaction T_R has a earlier deadline is abortion considered. Further, the larger the deadline of the transaction T_H, the more likely it will be aborted.

3. The criticalness of the transaction T_H is also used in making the decision. However, the above protocol does not consider the criticalness of the requesting transaction, T_R. This is a weakness of this protocol and will be overcome by an alternative protocol described in the next section.

For further details about this protocol, the reader is refered to [11].

3.2.2 Protocol 2 (CRP2): Based on combining transaction parameters

This protocol takes into account many kinds of information of the involved transactions. Let $CP(T_i)$ denote the combined value from the set of parameters for transaction T_i. Let t_{s_i} be the transaction start time, then at any time $t \geq t_{s_i}$,

$$CP(T_i) = c_{T_i} * (w_1 * (t - t_s) - w_2 * d_{T_i} + w_3 * p_{T_i} + w_4 * io_{T_i} - w_5 * l_{T_i})$$

where p_{T_i} - cpu time consumed by transaction T_i
io_{T_i} - I/O time consumed by transaction T_i
l_{T_i} - approximate laxity[3] of transaction T_i, if known

and the w_k's are weights.

The protocol is described by the following pseudo code.

if $CP(T_R) \leq CP(T_H)$
  then T_R waits
  else T_R aborts T_H
end if

In this protocol, one transaction aborts another one based on the above general decision rule. By appropriately setting weights to zero it is easy to create various outcomes, e.g., where a smaller deadline transaction always aborts a larger deadline transaction. The reader may refer to [11] for further discussion.

In a disk resident database system, it is difficult to determine the computation time and I/O time of a transaction. For some of the protocols in the study, we assume that the transaction length is known when the transaction is submitted to the system. This assumption is justified by the fact that in many application environments like banking and inventory management, the transaction length, i.e. the number of records to be accessed and the number of computation steps, is likely be known in advance. We further assume that

[3]Lexity is the maximum amount of time that a transaction can afford to wait but still make its deadline.

a transaction consists of a sequence of I/O-computation operations and the computation time and I/O time are proportional. In our experiments, we simplify the above formula for CP calculation as follows:

$$CP(T_i) = c_{T_i} * [w1 * (t - t_s) + w2 * (R\_accessed_{T_i} / R\_total_{T_i}) - w3 * d_{T_i}]$$

where $R\_total_{T_i}$ is the total number of records to be accessed by T_i; $R\_accessed_{T_i}$ is the number of records that have been accessed.

3.2.3 Protocol 3 (CRP3): Based on deadline-first-then-criticalness

We anticipate that criticalness and deadlines are the most important factors for real-time transactions. This protocol only takes these two factors into account. Unlike the scheme of combining transaction parameters discussed above, here we separate deadline and criticalness by checking the two parameters one after another. The algorithm for this protocol is as follows:

```
if d_TR > d_TH
  then T_R waits
  else
       if c_TR ≤ c_TH
         then T_R waits
         else  T_R aborts T_H
       end if
end if
```

3.2.4 Protocol 4 (CRP4): Based on deadline, criticalness and estimation of remaining execution time

CRP4 is an extension of CRP3. The protocol is as follows:

```
if d_TR > d_TH
  then T_R waits
  else
       if c_TR < c_TH
         then T_R waits
         else
              if c_TR = c_TH
                then
                     if (time_needed_TR + t) > d_TR
                       then T_R waits
                       else T_R aborts T_H
                     end if
```

```
                    else  T_R aborts T_H
                  end if
            end if
      end if
```

where

$$time\_needed_T = (t - t_s) \times (R\_total_T - R\_accessed_T)/R\_accessed_T$$

and R_total_T and $R\_accessed_T$ are the same as defined in CRP2.

3.2.5 Protocol 5 (CRP5): Based on criticalness only

This protocol is a simplification of CRP3. It only takes criticalness into account.

```
if c_{T_R} < c_{T_H}
  then T_R waits
  else  T_R aborts T_H
end if
```

Note that this protocol is a deadlock-free protocol, since waiting transactions are always considered in criticalness order. In addition, this protocol implements an *always-abort* policy if there is a system in which all the transactions have the same criticalness.

In summary, the five protocols resolve the data conflict by either letting the lock-requesting transaction wait or aborting the lock holder, depending on various parameters of the conflicting transactions.

3.3 Policies for Transaction Wakeup

According to the locking protocol, no more than one transaction is allowed to hold an exclusive lock on a data object. Thus as an exclusive lock holder releases the lock, it is possible that more than one transaction is waiting for the lock. At this point, it is necessary to decide which waiting transaction should be granted the lock. The decision should be based on transaction parameters, such as deadline and criticalness, and also should be consistent with the conflict resolution protocols (CRP) discussed in the previous section. Here we give the policies for transaction wakeup operation which correspond to each CRP.

- For CRP1, wake up the waiting transaction with maximum $VT()$ - the value of the virtual clock.

- For CRP2, wake up the waiting transaction with maximum $CP()$ - the value of the combined transaction parameters.
- For CRP3 and CRP4, wake up the waiting transaction with the minimum deadline.
- For CRP5, wake up the waiting transaction with the highest criticalness.

By applying these policies at the point of wake-up, real-time transactions are treated based on their timing requirements or/and criticalness.

3.4 Deadlock Resolution

The use of a locking scheme may cause deadlock. This problem can be solved by using deadlock detection, deadlock prevention, or deadlock avoidance. For example, the conflict resolution protocol 5, presented in the previous section, is a kind of scheme for deadlock prevention. In this section we focus on the scheme of deadlock detection and describe the policies for deadlock resolution.

A deadlock detection routine is invoked when a transaction is to be queued for a locked data object. If a deadlock cycle is detected, one of the transactions involved in the cycle must be aborted in order to break the cycle. Choosing a transaction for abort is a policy decision. For real-time transactions, we want to abort such that the timing constraints of these transactions can be met as much as possible, and at the same time the abort operation will incur the minimum cost. Here we present five deadlock resolution policies which take into account the timing properties of the transactions, the cost of abort operations, and the complexity of the protocols.

Deadlock resolution policy 1 (DRP1): *Always abort the transaction which invokes the deadlock detection.* This policy is simple and efficient since it does not need any information from the transactions in the deadlock cycle.

Deadlock resolution policy 2 (DRP2): *If a transaction has passed its zero-value point, abort it; otherwise abort the transaction with the longest deadline.*

Recall that a transaction which has passed its zero point may not yet be aborted because it may not have executed since passing the zero point, and because preempting another transaction execution to perform the abort may not be advantageous. Consequently, in DRP 2 and DRP 3 we first abort any waiting transaction that has passed its zero point.

Deadlock resolution policy 3 (DRP3): *If a transaction has passed its zero-value point, abort it; otherwise abort the transaction with the earliest deadline.*

Deadlock resolution policy 4 (DRP4): Here we assume that the average execution time of a transaction is known. Let dr_T denote the relative deadline of transaction T, RE_T the runtime estimate, t_s the start time, t the current time, and et_T the elapsed time ($et_T = t - t_s$). A transaction T is *feasible* if $dr_T \geq RE_T - et_T$ and *tardy* if $dr_T < RE_T - et_T$. This policy aborts a tardy transaction with the least criticalness if one exists, otherwise it aborts a feasible transaction with the least criticalness. The following algorithm describes this policy.

```
Step 1: trace deadlock cycle;
        for each T_i in the cycle do:
        if T_i has passed its zero-value point
          then abort T_i;
               return
          else
               if T_i is tardy
                 then tardy_list ← T_i
                 else feasible_list ← T_i
               end if
        end if

Step 2: if tardy_list is not empty
          then search tardy_list for T_i with the least criticalness
          else search feasible_list for T_i with the least criticalness
        end if
        abort T_i;
        return
```

Deadlock resolution policy 5 (DRP5): *If a transaction has passed its zero-value point, abort it; otherwise abort the transaction with the least criticalness.*

3.5 Transaction Restart

A transaction may abort for any number of reasons. A user may abort the transaction. The transaction can abort itself because of some execution exception, e.g. an arithmetic overflow. Also, the transaction could be aborted by the system if it has value 0. We call these kinds of aborts *termination aborts*. There is another kind of transaction abort, called *non-termination abort*, where transaction restart should be considered. For instance, a transaction may abort from a deadlock, or, a transaction may be aborted by another transaction because of a data conflict. These aborted transactions should be restarted as long as they may still meet their deadlines or contribute a positive value to the system. Based on our transaction model, we propose three policies for transaction restart.

Transaction restart policy 1 (TRP1): *Restart an aborted transaction if* $t < z$. In other words, an aborted transaction will be restarted as long as it still has some value to the system. Note that the transaction may have already passed its deadline at this point. This policy is intended to maximize the value that the transaction may contribute to the system.

Transaction restart policy 2 (TRP2): *If* $t < z$, *increase the CPU scheduling priority of the transaction by one and restart it.* This policy dynamically adjusts the CPU scheduling priority of the transaction. This adjustment makes the restarted transaction have a higher priority than its previous run. Thus the transaction will have a greater chance to meet its timing constraint after its restart. Note that the performance of other concurrent

transactions may be affected by this dynamic change of transaction priority and the priorities no longer accurately reflect the relative values of transactions. The impact this strategy makes on the system is examined through the performance tests.

Transaction restart policy 3 (TRP3): Assume that the runtime estimate of a transaction is known and is denoted by RE_{c_T}. The decision on transaction restart is based on the estimation of whether the transaction can complete by the time z, if it is restarted. We give the details of the policy by the following algorithm.

```
if z − t < RE_cT
  then terminate T;
  else
       if d_T − t > RE_cT
         then restart T
         else increase T's priority by one;
              restart T
       end if
end if
```

With the knowledge of the estimated execution time of a transaction, a better decision can be made and better system performance is expected than for TRP1 and TRP2 where it is assumed that no knowledge of execution time is known.

4 The Implementation of a Testbed System

In this section, we first provide an overview of CARAT testbed system - a distributed database testbed from which our single node implementation originates. Then we describe our real-time database testbed, named RT-CARAT.

4.1 CARAT: A Distributed Database Testbed

CARAT is a simplified but complete transaction processing environment [7]. It was designed to be a flexible tool for the testing and performance evaluation of distributed concurrency control, deadlock detection and avoidance, and recovery mechanisms used in distributed database systems. CARAT contains all the major functional components of a distributed transaction processing system (transaction management, data management, log management, communication management, and catalog management) in enough detail so that the performance results will be realistic. Some of the protocols previously implemented in CARAT are: a two-phase locking protocol with distributed deadlock detection, several distributed versions of an optimistic concurrency control protocol, before-image and after-image journaling mechanisms for transaction recovery, and a two-phase commit protocol for global consistency of distributed transactions. Some system utilities were also developed to

help dynamically monitor the system and collect data. The CARAT testbed now runs on a local area network of five DEC MicroVAX computers under the VMS operating system with DECnet network support. Some of our previous results in this area include [7,6,5,10].

4.2 RT-CARAT: A Real-Time Database Testbed

The current implementation of the RT-CARAT testbed is a centralized, secondary storage real-time database system. With respect to the process structure and communication, RT-CARAT is basically the same as CARAT. It is implemented as a set of cooperating server processes which communicate via efficient message passing mechanisms. Figure 2 illustrates the process and message structure of RT-CARAT.

A pool of transaction processes (TR's) simulate the users of the real-time database. Accordingly, there is a pool of data managers (DM's) which service transaction requests from the user processes (the TR's). There is one transaction manager, called the TM server, acting as the inter-process communication agent between TR and DM processes. The communications between TR, TM and DM processes are carried out through the mailbox, a facility provided by the VAX/VMS operating system. In order to speed up the processing of real-time transactions, the communication among DM processes is implemented using a shared memory space, called global section in the VAX/VMS.

For concurrency control and recovery, RT-CARAT simply adopts the two-phase locking protocol (2PL) and after-image (AI) journaling mechanism from the CARAT implementation.

One of the important features of RT-CARAT is the ability to schedule real-time transactions. RT-CARAT is not a time-sharing system. Instead, CPU is scheduled based on transaction priority with preemption. The scheduling process of RT-CARAT is carried out through the underlying VAX/VMS operating system. In the VAX/VMS, process scheduling priority is used in determining the relative precedence of processes for execution. Priority is a value in the range from 0 to 31, with 31 being the highest priority. The range of 32 priority levels is divided evenly between the normal process levels of 0 to 15 and the real-time process levels of 16 to 31. The execution behavior of a process is significantly affected by the type of process (normal or real-time) and its assigned priority level. With real-time priority scheduling, an executing process controls the CPU until one of the following events occurs:

- It is preempted by a higher or equal priority computable process.
- It enters a resource or event wait state.

In RT-CARAT, to reflect the nature of priority scheduling for real-time transactions, we implement TR's, TM and DM's as real-time processes with priority levels 16 to MAX_PRI ($16 < MAX\_PRI < 31$). TM, the transaction manager, is in charge of CPU scheduling as well as the communication between TR and DM processes. To avoid bottleneck, the priority of TM process is set to MAX_PRI. DM processes execute on behalf of the transactions. Their priority varies from 16 to MAX_PRI, which is determined and set by the CPU scheduler

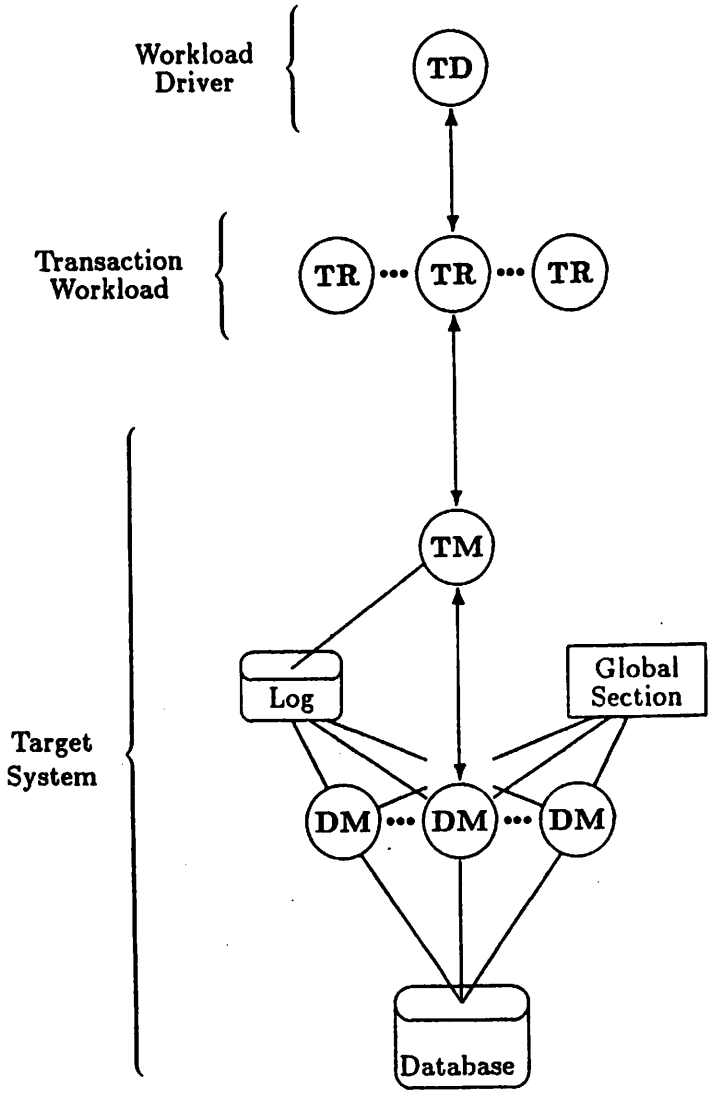

Figure 2: RT-CARAT processes and message structure

embedded in TM (the details are discussed in the following paragraph). Since the system performance should not be affected by the behavior of its user processes, we set the priority of all TR's at MAX_PRI. Being on top of VAX/VMS, the process priority setting scheme of RT-CARAT enables the real-time transactions to be processed in the order of their priorities. Note that an executing transaction with high priority can be blocked by a low priority transaction because of data conflict. The blocking is resolved by the conflict resolution protocols embedded in the DM.

In RT-CARAT, the functional components associated with real-time transaction processing are *the transaction generator, the CPU scheduler, the conflict resolution protocol, the deadlock resolution policy, the transaction wake-up policy* and *the transaction restart policy*. The block diagram in Figure 4.2 shows these functional components and their interactions. In the following, we describe the functionality of two components: *the transaction generator* and *the CPU scheduler*. The functionality of other components can be understood by reading the algorithms and policies discussed in Section 3.

- **Transaction generator**. The transaction generator generates transactions according to a configuration file where transaction type (read, write, or read/write), length (the number of records to be accessed and the length of computation time), the range of criticalness and deadline are specified. It submits a transaction for execution by synchronously sending/receiving a sequence of messages to/from TM process. A transaction performs a certain number of predefined operations, called *tdo* steps, and each operation may access a certain number of records and do a certain amount of computation. A transaction terminates upon completion or abort (self-abort or zero-value abort). In the tests described here, the transaction generator submits a new transaction immediately after the previous transaction has terminated, i.e. the generator generates real-time transactions in the form of a closed network.

- **CPU scheduler**. Upon receiving a transaction execution request, the CPU scheduler schedules the transaction based on the transaction priority. The scheduling operation consists of three steps: assigning transaction priority, mapping transaction priority to DM process priority, and setting the priority of DM process(es). First, the scheduler assigns a priority to the transaction. The priority assignment is determined by the parameters and the current state of that transaction (see the scheduling policies discussed in Section 3.1). Then, the scheduler maps the assigned transaction priority to a certain level of VAX/VMS real-time process priority. Finally, according to the mapped real-time priority, the scheduler sets, through a system call, the priority of the DM process which serves the transaction execution. At this point, the underlying VAX/VMS operating system schedules the transaction execution by the newly set priority of DM process(es). An executing transaction will be preempted if its DM process priority is not the highest at the moment, otherwise it will continue to run.

  The scheduler considers not only the incoming transaction but also the transactions that are currently running in the system. For example, under the policy of earliest deadline first scheduling, the priority of all the concurrent transactions is re-calculated upon arrival of a new transaction, and the priority of corresponding DM processes

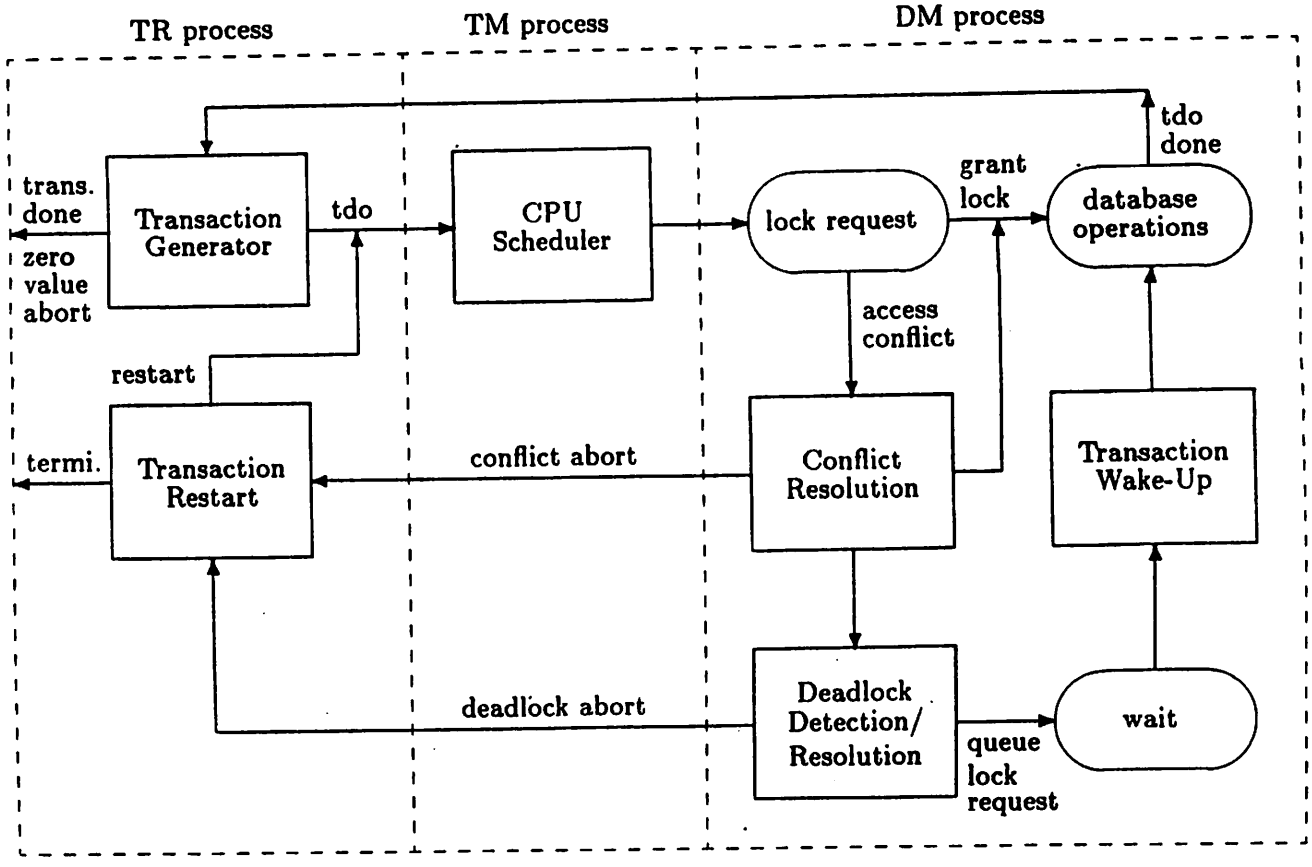

Figure 3: The real-time related functional components in RT-CARAT

is re-mapped. According to the adjustment, the scheduler resets the priority of DM processes one by one if it is different from the previous setting.

During its execution, a transaction may be aborted by another concurrent transaction because of data conflict or in order to resolve deadlock. This kind of abort is detected by the aborted transaction itself. Each transaction checks its status flag in the global section at certain points along the course of its execution. The transaction immediately aborts once it discovers that it has been asked to abort by another transaction. In that case, the DM process that is executing the transaction will release all the locks it holds, rollback the transaction by discarding the after-image and notify its corresponding TR process. Note that before the rollback operation, the exclusive locks held by the aborted transaction can be granted to other transactions. In other words, the aborted transaction will not block other transactions even though it has not really released its locks.

One issue should be mentioned before we close this section. An important operation in the secondary storage database system is the file I/O which supports both synchronous and asynchronous write to non-volatile disk storage. Asynchronous write is used for database read/write operations while synchronous write, also called force-write, is primarily used for journaling operations to ensure that the log records are safely stored on disks before a process can continue executing other operations [4]. From the real-time point of view, especially for I/O bound real-time database systems, these I/O operations should be scheduled according to the characteristics of real-time transactions. But, in our testbed, the disk access is under the control of disk controllers instead of the operating system, i.e. there is no way to directly manipulate the disk access through the system utilities. Thus in the current implementation, there are no components dealing with disk I/O operations. One scheme for controlling the disk access is to associate an I/O queue to each disk. The I/O requests will go through the queue before reaching the disk controller. In this case, we can schedule the I/O operations by manipulating the I/O queue. This or other schemes remain for future work. In this work, the impact of lacking control over I/O operation on system performance is studied through the experiments.

5 Experimental Results

5.1 The Test Environment

The performance tests are carried out on the RT-CARAT testbed. The database consists of 3000 physical blocks (512 bytes each) with each block containing 6 records for a total of 18,000 records. In all the experiments, the maximum multi-programming level in the system is 8, i.e., there are up to 8 transactions running concurrently. We express the length of a transaction T by the notation $T(x, y, z)$, where x is the number of *tdo* steps, y the number of records accessed in each *tdo* step, and z the amount of computation units per *tdo* step with 1 unit = 50 ms. To stress the real-time protocols and policies, a high access conflict situation was modelled by letting all the transactions perform write operations.

[4] In RT-CARAT, two separate disks are used, one for the database and the other for the log.

The transaction deadline is randomly generated from a uniform distribution within a deadline window, $[d\_base, \alpha \times d\_base]$, where *d_base* is the window baseline and α is a variable determining the upper bound of the deadline window. To model different workloads, *d_base* was specified by either of the two formulas:

$$d\_base = avg\_rsp \text{ or } d\_base = avg\_rsp - stnd\_dvi$$

where *avg_rsp* is the average response time of non-real-time transactions which run in a non-real-time database environment, and *stnd_dvi* is the standard deviation of the response time.

Besides the deadline, each transaction, when initiated, is randomly assigned a criticalness. In the experiments, there were up to 8 levels of criticalness and accordingly, the transactions were classified into 8 classes (with class 1 being the most critical). Based on a simple linear weighting scheme, the criticalness of a transaction is inversely proportional to its class number, i.e. the criticalness of transactions in class 1 has value 8, the criticalness in class 2 has value 7, etc. As the deadline and criticalness are specified, the value function of the transaction is fixed and the transaction value can be computed at any time (see Section 2).

In the experiments each test consists of two to six runs where each run was two hours long. The data was collected and averaged over the total number of runs. The number of runs for each test depends on the stability of the data. Our requirement on the statistical data is to generate 95% confidence intervals for the deadline guarantee ratio whose width is less than 10% of the point estimate of the deadline guarantee ratio.

5.2 Baseline and Metrics

For these tests, the performance baseline is a non real-time transaction processing system. Here transactions still possess the real-time properties (i.e., the deadline, criticalness and value function), but are not processed by any real-time related protocols and policies. In this non real-time transaction baseline, the standard two-phase locking protocol is used. In the baseline there is no access conflict abort. For deadlock detection, it is always the transaction requesting the new lock that is aborted. It should be noted that there do exist zero-value aborts, since a value function is attached to each transaction. In addition, an aborted transaction will restart as long as it has not passed its zero-value point.

For the results presented in this paper we use the following metrics.

- Deadline guarantee ratio - the percent of transactions that complete by their deadline.
- Average deadline guarantee ratio - an average value of deadline guarantee ratio over all transactions in different classes of criticalness.
- Weighted value - the total value of all transactions in each class of criticalness that complete by their deadline divided by the total value of all invoked transactions.

- Total weighted value - the total value of all transactions in all classes of criticalness that complete by their deadline divided by the total value of all invoked transactions.
- Total abort ratio - percent of transactions aborted for any reason.
- Throughput - the number of transactions that complete by their deadline per minute.

5.3 Experiments

The following four *sets* of experiments were conducted.

- **CPU scheduling**: The effect of CPU scheduling on the performance of real-time transactions was studied by varying transaction length, varying deadline setting, mixing transactions with different lengths, and changing the system from CPU bound to I/O bound.
- **Conflict resolution**: In these experiments we compared the performance of all the conflict resolution policies by varying workloads, varying deadline settings, and changing the level of criticality.
- **CPU scheduling vs. conflict resolution**: In these experiments we studied the impact of CPU scheduling versus conflict resolution on the performance of real-time transactions.
- **Deadline setting**: In these experiments the effect of deadline distributions on transaction performance was examined.

Besides the above studies, we also investigated the proposed *deadlock resolution policies* through experiments. The results show that the performance of all the policies are similar because the deadlock cycle involves only two transactions most of the time. Remember that to keep the decision consistent with the conflict resolution policies, we never abort the transaction which directly blocks the lock-requesting transaction. We also did experiments for the three *transaction restart policies*, and found no significant difference between them. Thus, to save space, we do not show these experimental results here. For all the experiments discussed below, we used DAP1 for deadlock resolution and TRP1 for transaction restart, respectively.

5.3.1 CPU Scheduling

Figures 4-7 compare four scheduling schemes with respect to the metrics of deadline guarantee ratio, weighted value, total abort ratio, and throughput. In this experiment, all the transactions were equal in length, i.e. $T(12, 4, 10)$. The deadline setting was $d\_base = avg\_rsp - stnd\_dvi$ = 65 - 30 = 35 (sec.) and $\alpha = 3$. The conflict resolution policy was CRP4. Figure 4 plots the deadline guarantee ratio versus the transaction class classified by criticalness. Our first observation is that with CPU scheduling, transactions perform

better than the baseline, except the transactions in class 7 and 8 that executed under the scheduling algorithms MCF and CDF. As compared with the baseline and the scheduling algorithms EDF and LEF, both MCF and CDF promote higher deadline guarantee ratio for the transactions with high criticalness, but incur a lower deadline guarantee ratio for the transactions with low criticalness. This is simply because the two algorithms take the criticality of transactions into account. As we expected, the high critical transactions perform better when they compete with low critical transactions. Of the two algorithms, CDF performs better than MCF, since CDF considers not only the criticalness but also the relative deadline of a transaction. With the scheduling algorithm EDF, the performance was basically the same over all classes of transactions. This is understandable since the algorithm totally ignores criticalness. Under the specified workload, LEF perform neither as well as EDF over all classes of transactions nor as well as MCF and CDF for high critical transactions, but it still better than the baseline.

Figure 5 depicts the weighted value that each class of transactions contributed to the system. The performance results indicate that the system gains more value through CPU scheduling compared with the baseline. Overall, the higher the criticalness of a transaction, the larger the value it imparts to the system. The performance relation of the four scheduling algorithms is similar to what we observed in Figure 4.

The transaction total abort ratio is shown in Figure 6. Here the performance of the scheduling algorithms is basically the inverse of the performance on the deadline guarantee ratio, i.e., the higher the deadline guarantee ratio, the lower the abort ratio. For MCF and CDF, the low abort ratio for high critical transactions is achieved by aborting more low critical transactions. The abort ratio with EDF is low over all classes of transactions. This comes from the fact that in the experiments transaction deadline and transaction arrival time are highly correlated. Thus, under EDF, transactions execute in an FCFS (*First-Come-First-Serve*) manner most of the time.

Figure 7 shows the transaction throughput of the various algorithms. The results again show that the real-time scheduling algorithms outperform the baseline.

With the same deadline setting that was used in the above tests, two more experiments were performed by varying the length of transaction, with one having a short length, $T(4, 4, 10)$ and the other a large length, $T(20, 4, 10)$. Figures 8-11 illustrate the deadline guarantee ratio and weighted value for the two experiments, respectively. Among the four CPU scheduling algorithms, EDF performs best for short transactions and worst for long transactions, with respect to deadline guarantee ratio. As we discussed above, for these workloads EDF schedules the transactions approximately in an FCFS manner. It performs well when deadline is not tight and poorly when tight (EDF performs even worse than the baseline when deadline is extremely tight. The results are not shown here due to space limitations.) For short transactions, MCF and CDF do not perform as well as EDF with the specified deadline setting and even worse than the baseline for the transactions with very low criticalness (class 8). This is understandable as MCF and CDF always attempt to preempt the low critical transactions; the preemption is not necessary in a system which runs short transactions with loose deadlines. Note that frequent preemption will slow down transaction execution, which may increase data access conflicts and the chance for dead-

lock. The performance of LEF is directly related to the length of transactions. Comparing Figures 4, 8 and 10, the reader can see that LEF improves in performance as transactions become longer.

With the workloads, $T(4,4,10)$, $T(12,4,10)$ and $T(20,4,10)$, as applied in the above experiments, we further conducted tests for a narrow deadline window setting where d_base = avg_rsp = 23 sec. for $T(4,4,10)$, 65 sec. for $T(12,4,10)$ and 97 sec. for $T(20,4,10)$, and $\alpha = 0.2$. The results (not included due to space limitations) show that the scheduling algorithms performed basically the same as what we saw and discussed above, except that overall performance of transactions are better due to the larger value of deadline base and less variation of deadlines.

Another test made in this set of experiments was to examine the overall effects of the scheduling algorithms on all classes of transactions. Figures 12 and 13 show, respectively, the total weighted value and the average deadline guarantee ratio over 8 classes of transactions versus the transaction length (*tdo* steps), where the deadline setting was $d\_base = avg\_rsp - stand\_dvi$ and $\alpha = 3$. The reader can see that the total value that the system gained under CPU scheduling is far more than the value gained under the baseline. Similarly, under CPU scheduling, the transaction average deadline guarantee ratio gets higher compared to the baseline. When transactions become longer, the performance degrades because of higher data conflict, higher chance for deadlock, and relatively tighter deadlines.

All the experiments presented thus far were carried out for workloads consisting of equal length transactions. The scheduling algorithms were also tested for the transactions of different length. The workload consisted of two the transactions with length $T(4,4,10)$, four with length $T(12,4,10)$, and two with length $T(20,4,10)$. In this case, the deadline was generated according to the length of a transaction, with $d\_base = avg\_rsp = 23$ sec. for $T(4,4,10)$, 65 sec. for $T(12,4,10)$ and 97 sec. for $T(20,4,10)$, and $\alpha = 0.2, 3$. Figures 14-17 show the performance results for this mixed workload, with a deadline setting $\alpha = 3$. Here we have two observations. First, LEF performs no better than the baseline, because the priority of long transactions can be much higher than that of short transactions, which blocks the execution of short transactions, thus lowering the performance of short transactions. Second, there is a large variation in throughput (see the curve for CDF in Figure 17) that does not reflect itself in higher deadline guarantee ratio. This means that the transaction response time under different algorithms is different even though the deadlines are met.

The previous experiments were designed to make the system CPU bound, where CPU utilization was always above 92% and the I/O utilization of the database disk ranged from 25% to 30%. In this next experiment, we changed such a CPU bound system to an I/O bound system by eliminating the computation in each tdo step of a transaction.

Figures 18 and 19 illustrate the performance of the CPU scheduling algorithms for the transactions with length $T(12,4,0)$ and deadline setting: d_base = avg_rsp - stnd_dvi = 25 - 12 = 13 (sec.) and $\alpha = 3$. The measured CPU utilization and I/O utilization are 73% and 81%, respectively. Comparing the performance results with the ones from the preceding experiments (see Figures 4 and 5 for the same workload), we see that the CPU scheduling algorithms do not perform as well in an I/O bound system as in a CPU bound system. This performance degradation is due to the lack of I/O scheduling in the current implementation

of RT-CARAT system. This experiment confirms our discussion in Section 4, i.e., for I/O bound real-time database systems, there is a need for I/O scheduling.

The observations and discussions presented above lead to the following points:

- CPU scheduling by MCF and CDF largely improves the overall performance of real-time transactions for the tested workloads. Further, MCF and CDF achieve good performance for more critical transactions at the cost of losing some transactions that are less critical. This trade-off reflects the nature of real-time transaction processing that is based on criticality as well as timing constraints. To get better performance, the information of both criticalness and deadline of a transaction is needed for CPU scheduling.
- EDF offers an even performance over all classes of transactions. It is useful only when the deadlines are loose.
- LEF outperforms the baseline only in the situation where all the transactions are long and equal in length. However, the idea used in LEF is important to real-time transaction processing.
- When the system is I/O bound, CPU scheduling does not significantly improve the performance for real-time transactions.

5.3.2 Conflict resolution

This set of tests consisted of two experiments with respect to transaction criticality. The first experiment concerned the situation where a workload consisted of transactions with different levels of criticalness, while the second considered the case of the single level of criticality.

In the first experiment, we studied the performance of conflict resolution protocols by varying transaction length and deadline settings. The CPU scheduling algorithm used in the experiment was CDF. Different from all other experiments, the baseline compared in this experiment was chosen to be NRTCDF - non real-time, applying CPU scheduling (CDF) only, i.e., no conflict resolution protocol is applied in the case of data conflict.

Figures 20 and 21 show the performance results from testing of short transactions, $T(4,4,10)$, with the deadline setting: d_base = avg_rsp = 23 (sec.) and $\alpha = 0.2$. As can be seen from the figures, all the protocols perform basically the same, and there is no significant improvement on the performance as compared with NRTCDF. This is not surprising since with short transactions, the data access conflict is low, and thus, none of the conflict resolution protocols play an important role.

To create a high conflict situation, we increased the length of transactions to $T(16,4,10)$. It was observed, as shown in Figures 22 and 23, that all the protocols improve the performance for high critical transactions. Among the five protocols, CRP5, the simplest one, performs best. This is largely due to the fact that CRP5 is a deadlock-free protocol by which all transaction aborts result from conflict resolution but not from deadlock

resolution. Here the point is that if a transaction will be aborted, then it should be aborted as early as possible in order to reduce the waste in using the resources (i.e. CPU, I/O, and data). Since the conflict resolution is applied before deadlock resolution in the course of transaction execution (see Figure 3), an early abort from conflict resolution decreases the amount of resources that would be wasted if the transaction is aborted later from deadlock resolution. With CRP5, in addition, transaction execution benefits from eliminating the overhead for deadlock detection.

The performance of CRP2 is not as good as CRP5 but is better than other protocols. This is because the dominant factor in CRP2 is criticalness, which results in the transaction abort in a way similar to that of CRP5, i.e., the large percentage of transaction aborts come from conflict resolution. But because CRP2 is not a deadlock-free protocol, there are still some aborts due to deadlock, and the operation of deadlock detection cannot be ignored.

The performance of CRP3 and CRP4 is almost identical, since CRP4 checks only one more condition than CRP3, i.e., CRP4 checks the condition on the amount of time that the transaction needs to finish before its deadline. It is clear now that this additional condition does not help in improving the performance.

CRP1 only outperforms NRTCDF for transactions with high criticalness, but it performs slightly better than CRP3 and CRP4, and as well as CRP5, for low critical transactions. This is because CRP1 does not take into account the criticalness of the lock-requesting transactions. When the deadline of lock-requesting transaction is earlier than that of lock-holding transaction, CRP1 allows the lock requester with high criticalness to wait for the lock holder with low criticalness, thus lowering the performance for high critical transactions. This situation never happens with the other conflict resolution protocols.

In the second experiment we investigated the performance of the conflict resolution protocols under the situation where all the transactions have the same criticalness. Here we used EDF for CPU scheduling. Figure 24 shows the performance of the protocols on the transaction deadline guarantee ratio, with a variation of transaction length from $T(4, 4, 10)$ to $T(20, 4, 10)$. The reader can see that although all the protocols outperform the baseline, only CRP1 performs better than NRTEDF ("CPU scheduling only") as transactions become long. This is because the protocols (CRP2, CRP4 and CRP5) cause many unnecessary conflict aborts. For example, CRP5 always abort the lock holder if the two conflicting transactions are in the same class of criticalness. Clearly, this kind of abort is a negative factor for transactions which are making efforts to meet their deadlines. Note that the blind abort policy used in CRP5 is much worse than the blind wait - a conflict resolution policy used in NRTEDF.

For CRP1, the VT value of a transaction is simply equal to t, the current time, as c, the criticalness, is equal to 1. This means that the protocol implements a *minimum-deadline-first* policy before the deadline of the lock holder is surpassed. This conditional *minimum-deadline-first* policy, plus the *minimum-deadline-first* policy used by EDF, achieves the best performance.

CRP3 performs the same as NRTEDF. This is understandable since CRP3 is identical to NRTEDF when all the transaction have the same criticalness.

The experimental results indicate that

- conflict resolution protocols play an important part in real-time transaction processing only when the data conflict rate is high,
- the performance of the protocols depends on how the criticalness is specified in the workload, and
- for a workload consisting of transaction with many different values of criticalness, the criticality is the most important factor in conflict resolution.

5.3.3 CPU scheduling vs. conflict resolution

To distinguish the effects of CPU scheduling and conflict resolution on system performance, we conducted an experiment which tested four different schemes for real-time transaction processing: (1) NRT - the baseline; (2) NSCRP5 - no real-time scheduling but applying conflict resolution protocol (CRP5); (3) NRTCDF - applying CPU scheduling (CDF) only; and (4) CDFCRP5 - applying both CPU scheduling (CDF) and conflict resolution protocol (CRP5). The workload for the test presented here is $T(12,4,10)$ with deadline setting: d_base = avg_rsp = 65 (sec.) and $\alpha = 0.2$.

It was observed, as shown in Figures 25 and 26, that NSCRP5 improves the performance only for the transactions with very high criticalness (class 1 and 2), but it severely degrades the performance, much worse than NRT, as transactions become less critical. NRTCDF, on the other hand, greatly improves the performance of transactions in most classes. CDFCRP5, the combination of NSCRP5 and NRTCDF, provides the best performance. The observation indicates that CPU scheduling dominates the performance of real-time transactions and there is a need to combine the CPU scheduling scheme with the method of conflict resolution so as to achieve a better performance.

5.3.4 Deadline setting

In addition to the performance measures of CPU scheduling and deadlock resolution, we are also interested in the effect of transaction deadline distributions on the performance of transaction execution. In this experiment the transactions were equal in length $T(12,4,10)$ and had the same criticalness. We let d_base be equal to avg_rsp - stnd_dvi = 65 - 30 = 35 (sec.) and varied α from 2.0 to 3.4 in increments of 0.2. The CPU scheduling algorithm used was EDF.

Figure 27 shows the deadline guarantee ratio versus the α value for the baseline and two conflict resolution protocols, CRP2 and CRP4. It is easy to see that with a fixed workload, the deadline guarantee ratio becomes higher as the α value increases. Transactions under the control of CPU scheduling and conflict resolution protocols outperform the baseline only when α is large enough. Clearly, the deadline distributions strongly impact the performance of real-time transactions and the design of protocols for real-time transaction processing.

6 Conclusions

Real-time transaction processing is complex. Many issues arise as both data consistency and timing constraints are required for the transactions. In this paper, we have developed several algorithms with regard to the issues of CPU scheduling, data conflict resolution, transaction wakeup, deadlock resolution, and transaction restart. We have presented the integration and implementation of the proposed algorithms on the RT-CARAT testbed. Our experimental results from the testbed indicate that

- the CPU scheduling algorithm has the most significant impact on the performance of real-time transactions, and to get better performance the information of both criticalness and deadline of a transaction is needed for CPU scheduling;
- various conflict resolution protocols which directly address deadlines and criticalness produce better performance than protocols that ignore such information;
- deadlock resolution and transaction restart policies tailored to real-time constraints seem to have negligible impact on overall performance;
- criticalness is a very important factor in real-time transaction processing; and
- deadline distributions strongly affect the transaction performance.

References

[1] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions," *ACM SIGMOD Record*, March 1988.

[2] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *Proceedings of the 14th VLDB Conference*, 1988.

[3] Buchmann, A.P. et. al., "Time-Critical Database Scheduling: A Framework For Integerating Real-Time Scheduling and Concurrency Control," *Data Engineering Conference*, February 1989.

[4] Dayal, U. et. al., "The HiPAC Project: Combining Active Database and Timing Constraints," *ACM SIGMOD Record*, March 1988.

[5] Jenq, B.P., "Performance Measurement, Modeling, and Evaluation of Integrated Concurrency Control and Recovery Algorithms in Distributed Database Systems," *Ph.D. Thesis*, University of Massachusetts, Amherst, February 1986.

[6] Kohler, W. and B.P. Jenq, "Performance evaluation of Integrated Concurrency Control and Recovery Algorithms Using a Distributed Transaction Processing Testbed," *In The 6th International Conference in Distributed Computing Systems*, IEEE Computer Society, Cambridge MA, June 1986.

[7] Kohler, W. and B.P. Jenq, "CARAT: A Testbed for the Performance Evaluation of Distributed Database Systems," *Proc. of the Fall Joint Computer Conference*, IEEE Computer Society and ACM, Dallas Texas, November 1986.

[8] Molle, M. L., and Lenonard Kleinrock, "Virtual Time CSMA: Why Two Clocks are Better than One", *IEEE transactions on Communications*, Vol. COM-33, No. 9, September 1985.

[9] Sha, L., R. Rajkumar and J.P. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record*, March 1988.

[10] Shih, C.S., A. Dan, W. Kohler, J. Stankovic and D. Towsley, "Comparison of Distributed Concurrency Control Protocols Under a Testbed Environment," *submmited to IEEE Transaction on Computers*, March 1989.

[11] Stankovic, J.A. and W. Zhao, "On Real-Time Transactions," *ACM SIGMOD Record*, March 1988.

[12] Zhao, W. and K. Ramamritham "Virtual Time CSMA Protocols for Hard Real–Time Communication", *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 8, August 1987.

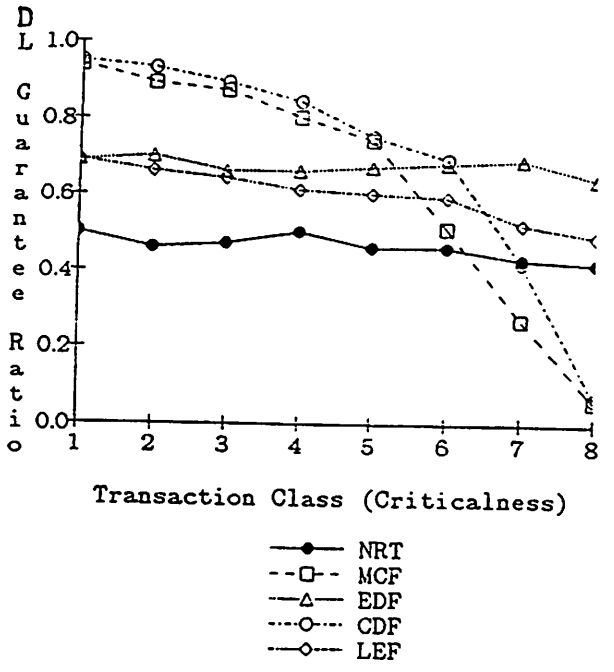

Figure 4. CPU scheduling with CRP4, $T(12,4,10)$, $d\_base = avg\_rsp - stnd\_dvi, \alpha = 3$

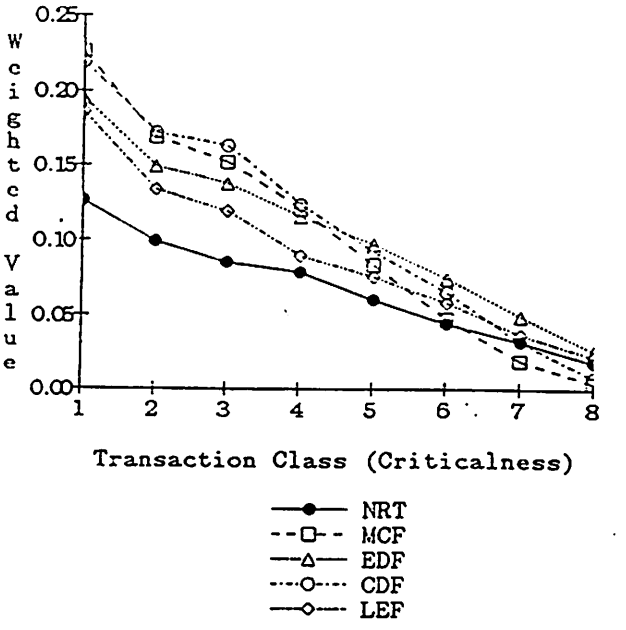

Figure 5. CPU scheduling with CRP4, $T(12,4,10)$, $d\_base = avg\_rsp - stnd\_dvi, \alpha = 3$

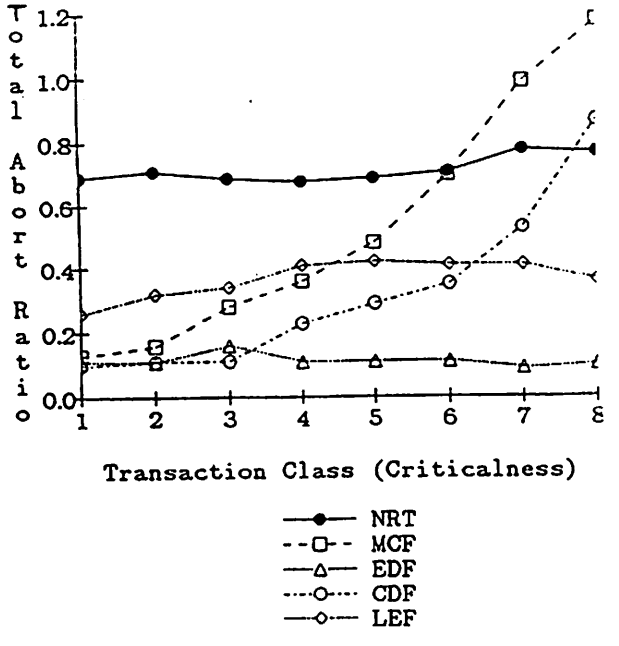

Figure 6. CPU scheduling with CRP4, $T(12,4,10)$, $d\_base = avg\_rsp - stnd\_dvi, \alpha = 3$

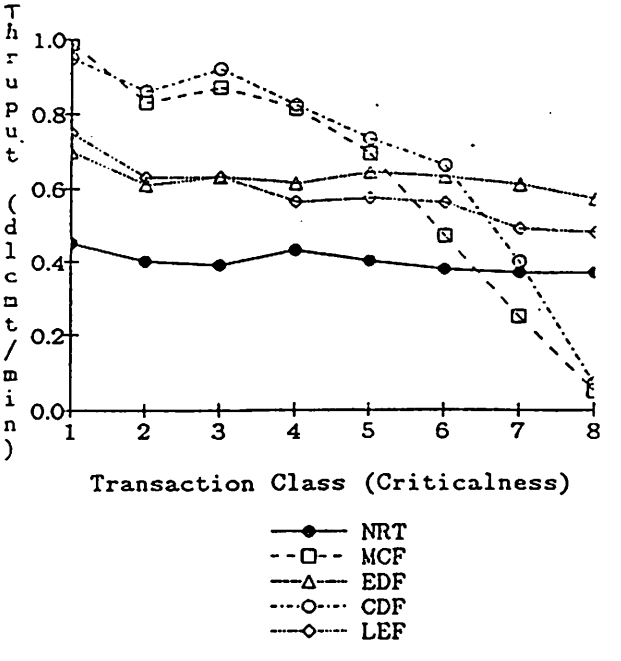

Figure 7. CPU scheduling with CRP4, $T(12,4,10)$, $d\_base = avg\_rsp - stnd\_dvi, \alpha = 3$

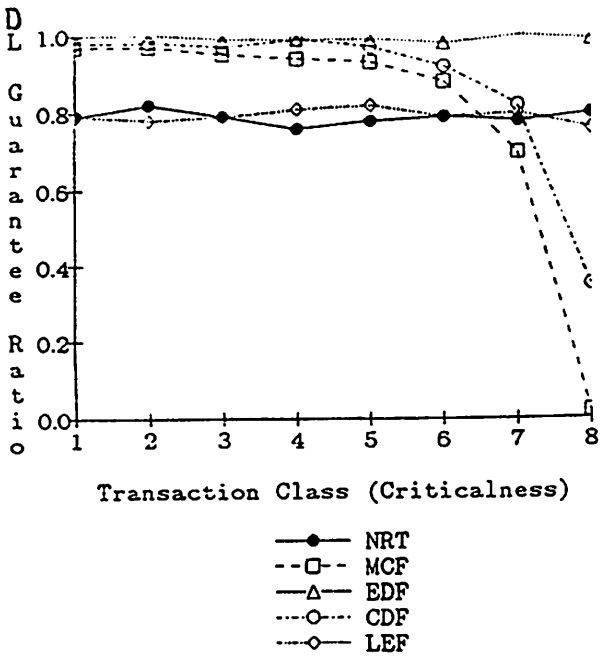

Figure 8. CPU scheduling with CRP4, $T(4,4,10)$, $d\_base = avg\_rsp - stnd\_dvi, \alpha = 3$

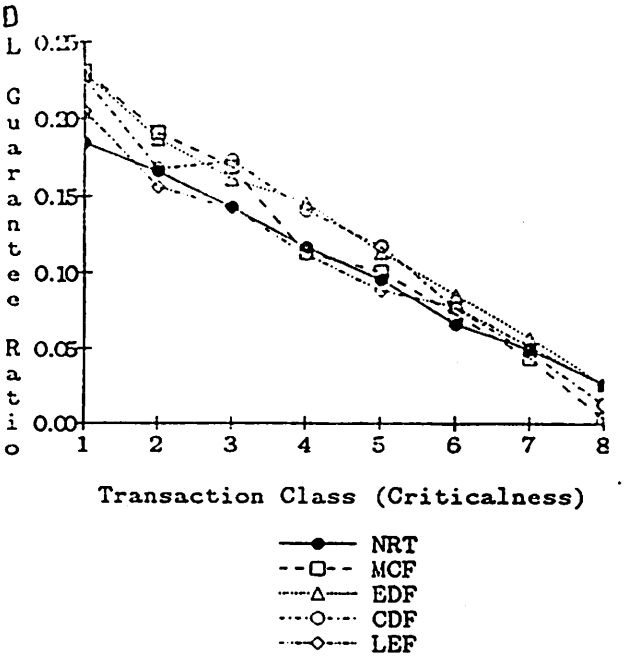

Figure 9. CPU scheduling with CRP4, $T(4,4,10)$, $d\_base = avg\_rsp - stnd\_dvi, \alpha = 3$

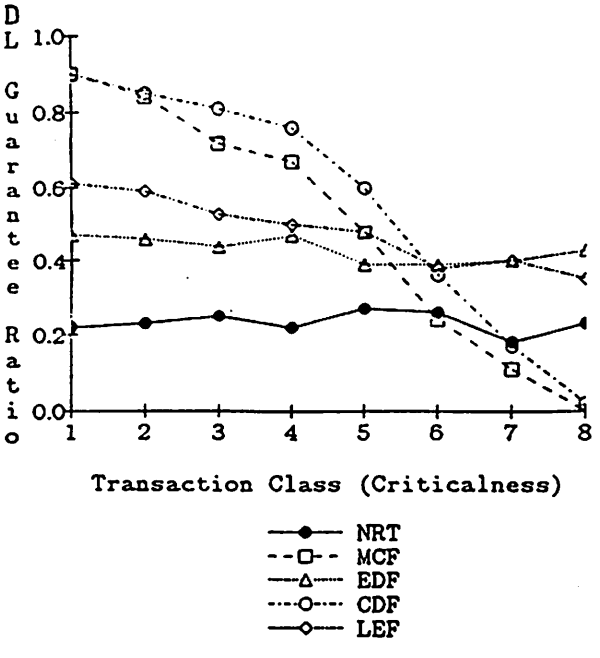

Figure 10. CPU scheduling with CRP4, $T(20,4,10)$, $d\_base = avg\_rsp - stnd\_dvi, \alpha = 3$

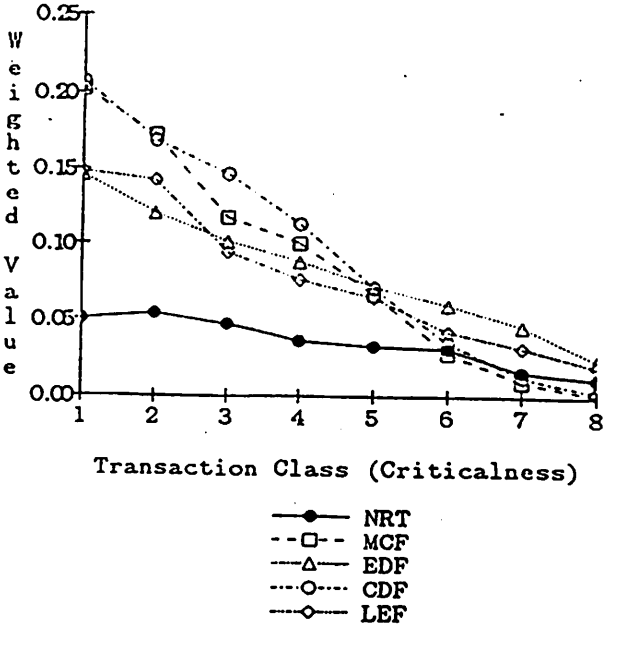

Figure 11. CPU scheduling with CRP4, $T(20,4,10)$, $d\_base = avg\_rsp - stnd\_dvi, \alpha = 3$

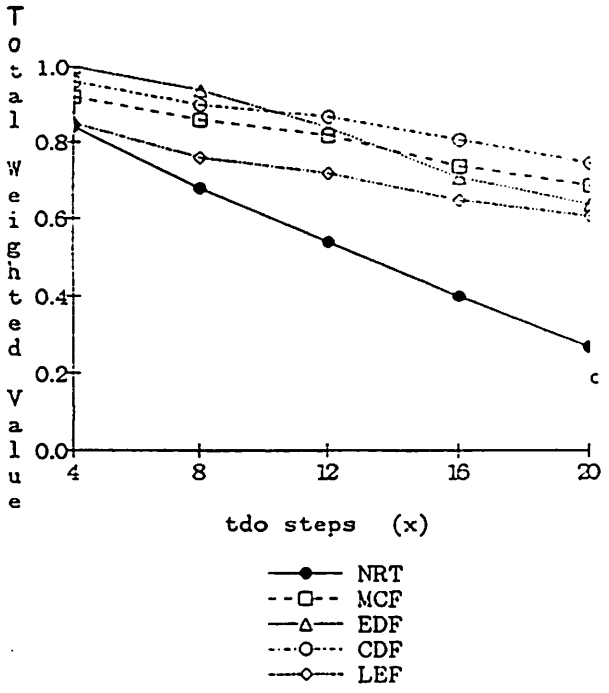

Figure 12. CPU scheduling with CRP4, $T(x, 4, 10)$, $d\_base = avg\_rsp - stnd\_dvi, \alpha = 3$

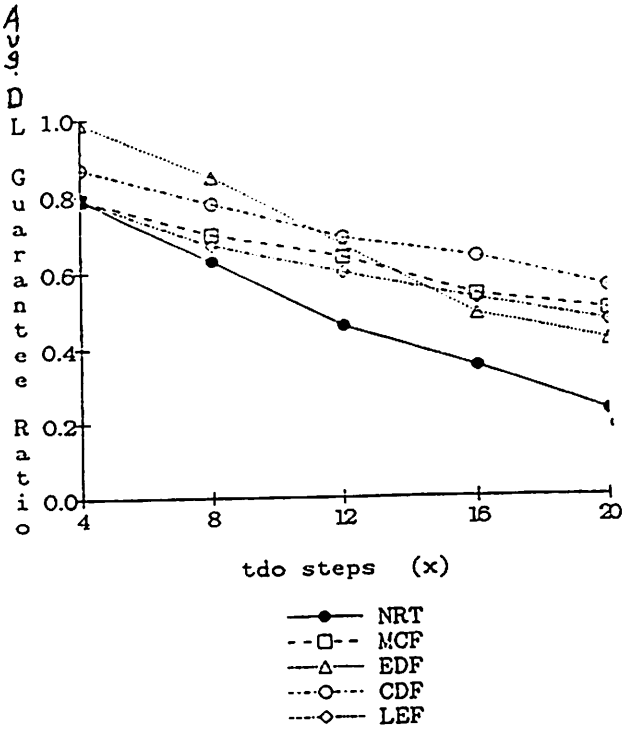

Figure 13. CPU scheduling with CRP4, $T(x, 4, 10)$, $d\_base = avg\_rsp - stnd\_dvi, \alpha = 3$

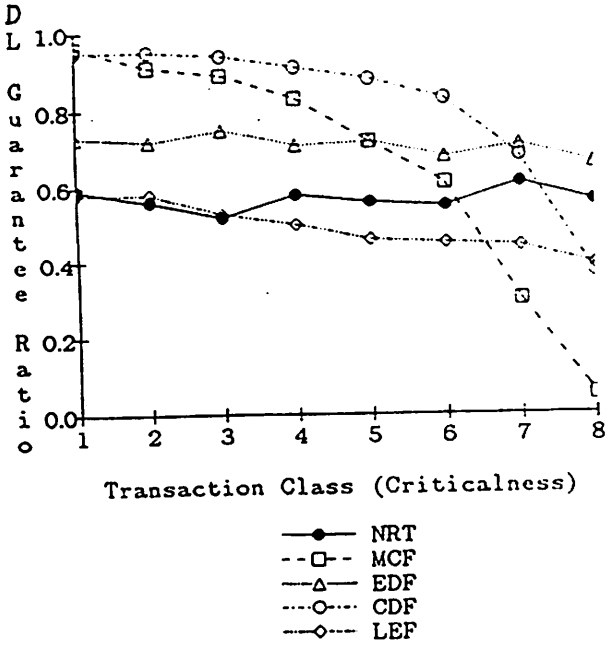

Figure 14. CPU scheduling with CRP4, $T(4\text{-}12\text{-}20, 4, 10)$, $d\_base = avg\_rsp - stnd\_dvi, \alpha = 3$

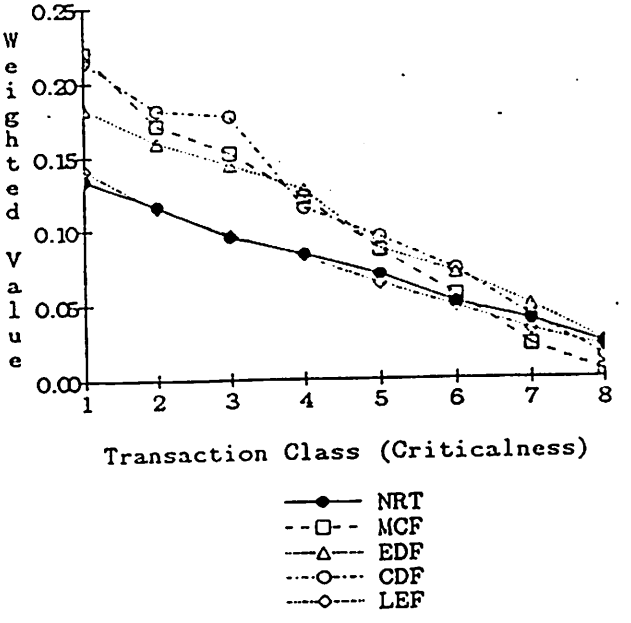

Figure 15. CPU scheduling with CRP4, $T(4\text{-}12\text{-}20, 4, 10)$, $d\_base = avg\_rsp - stnd\_dvi, \alpha = 3$

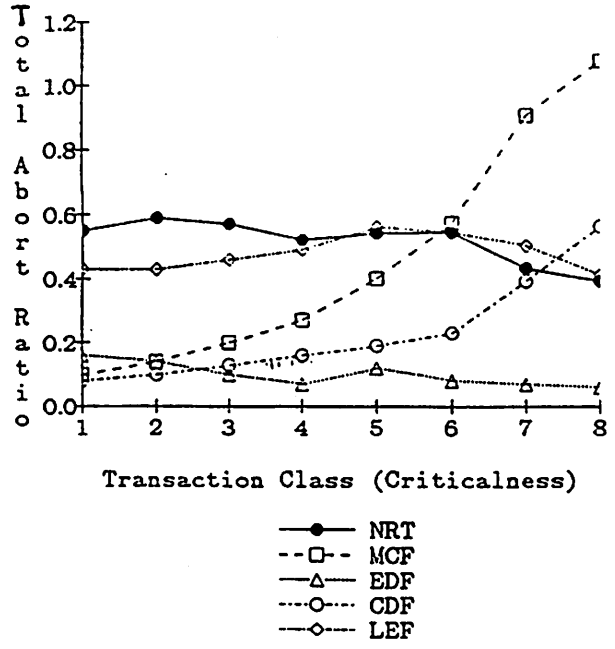

Figure 16. CPU scheduling with CRP4, $T(4\_12\_20, 4, 10)$ $d\_base = avg\_rsp - stnd\_dvi, \alpha = 3$

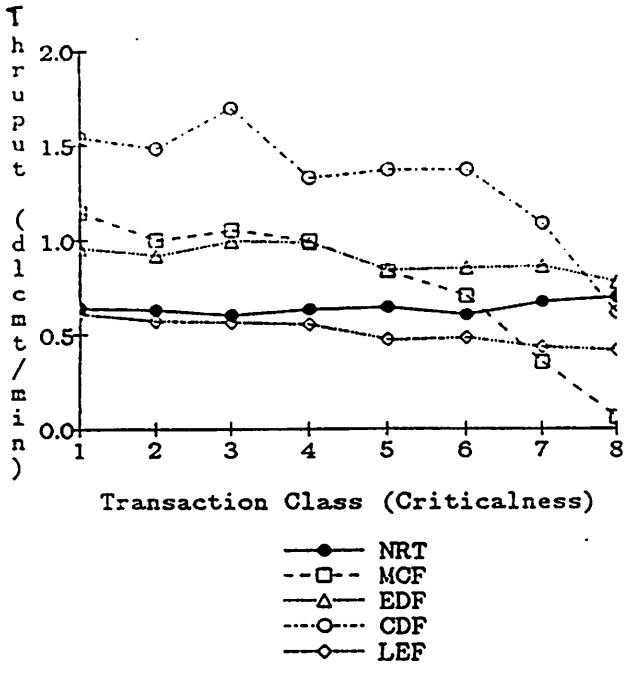

Figure 17. CPU scheduling with CRP4, $T(4\_12\_20, 4, 10)$, $d\_base = avg\_rsp - stnd\_dvi, \alpha = 3$

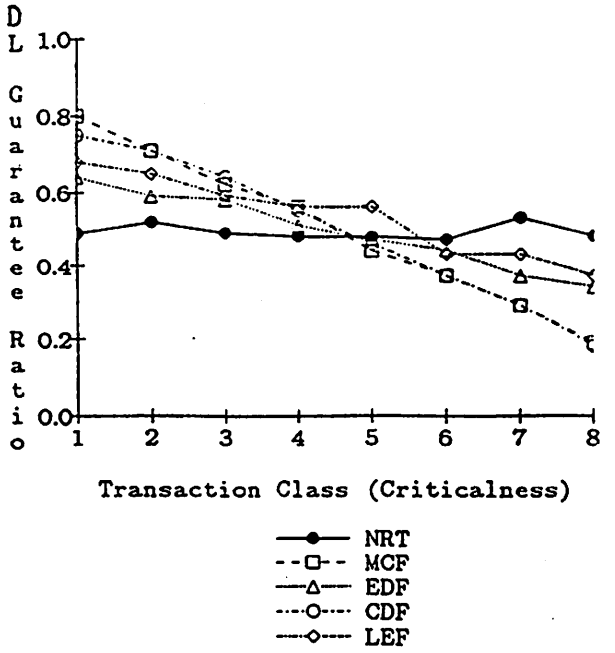

Figure 18. CPU scheduling under I/O bound system with CRP4, $T(12;4,0), d\_base = avg\_rsp - stnd\_dvi, \alpha = 3$

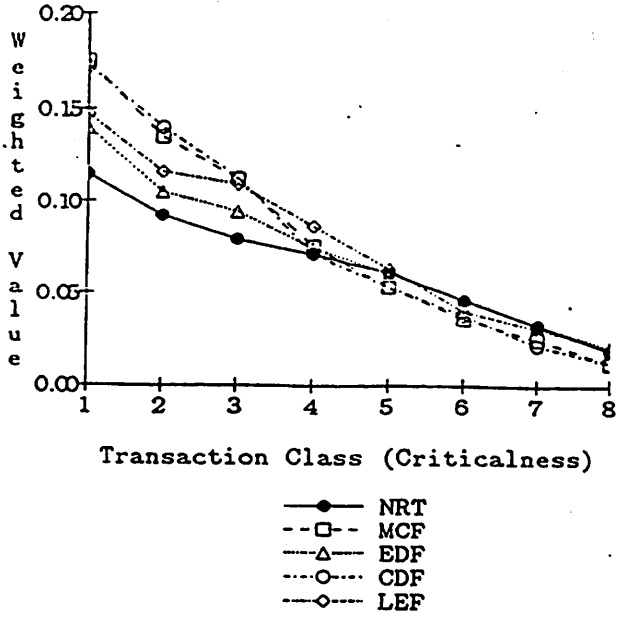

Figure 19. CPU scheduling under I/O bound system with CRP4, $T(12, 4, 0), d\_base = avg\_rsp - stnd\_dvi, \alpha = 3$

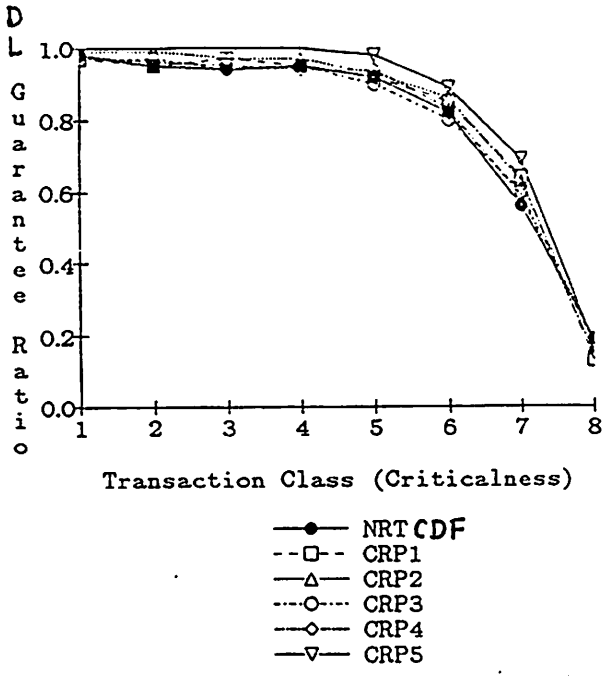

Figure 20. Conflict resolution under CDF, $T(4,4,10)$, $d\_base = avg\_rsp, \alpha = 0.2$

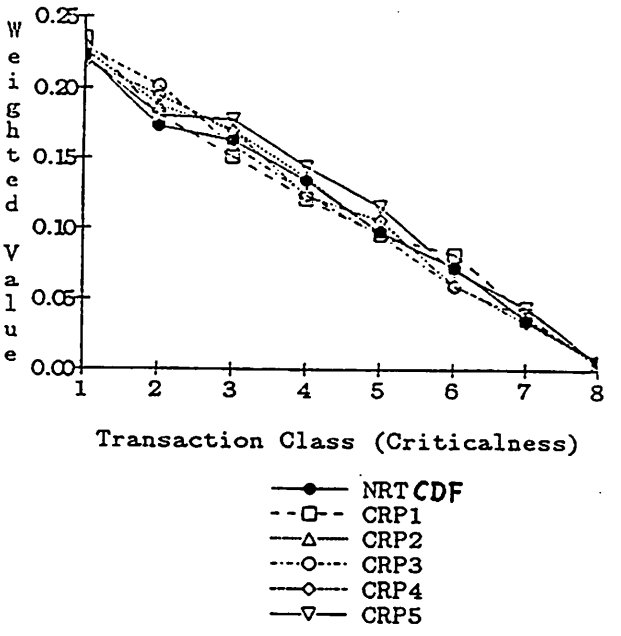

Figure 21. Conflict resolution under CDF, $T(4,4,10)$, $d\_base = avg\_rsp, \alpha = 0.2$

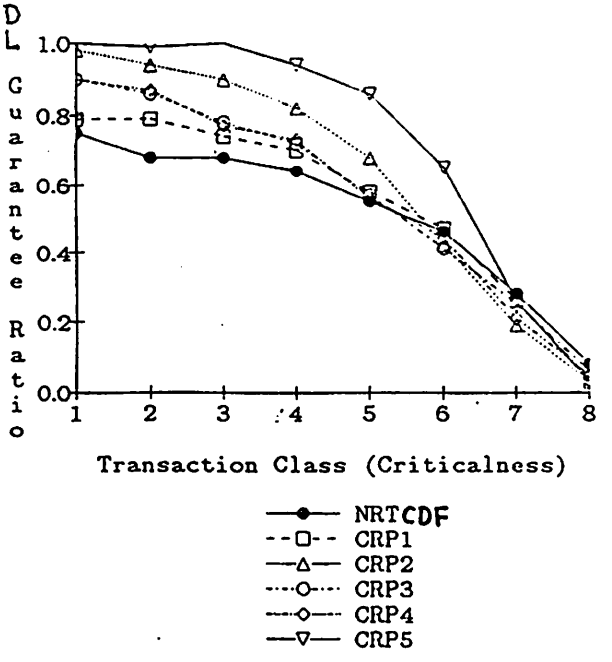

Figure 22. Conflict resolution under CDF, $T(16,4,10)$, $d\_base = avg\_rsp, \alpha = 0.2$

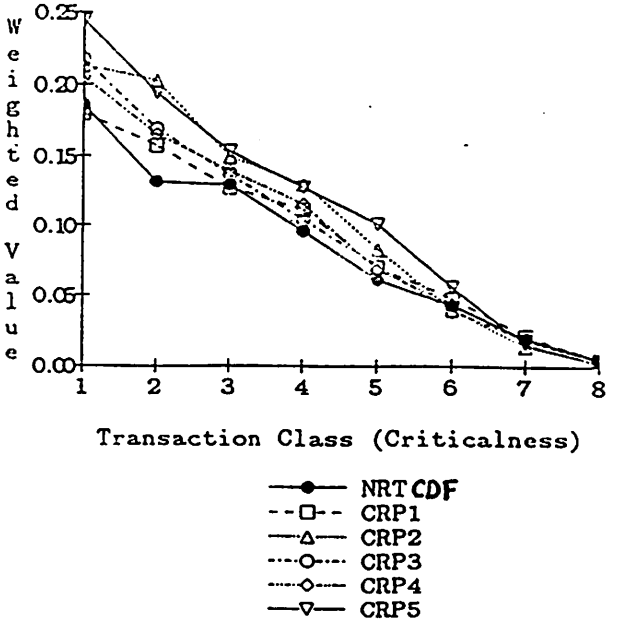

Figure 23. Conflict resolution under CDF, $T(16,4,10)$, $d\_base = avg\_rsp, \alpha = 0.2$

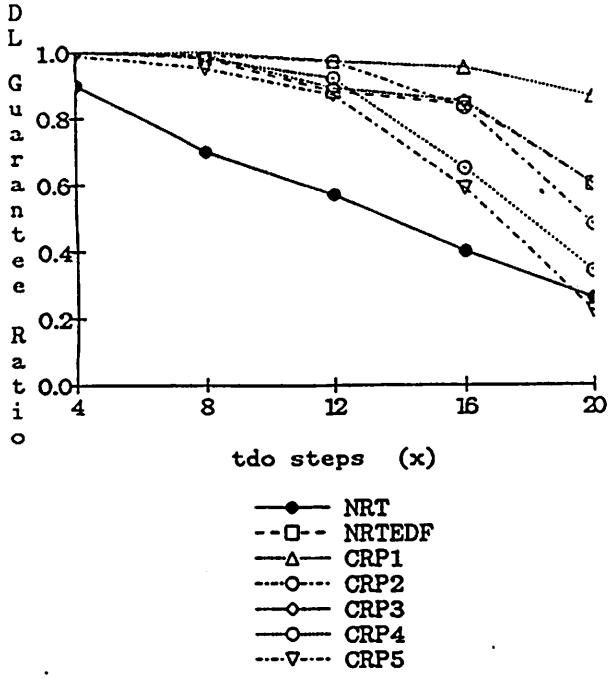

Figure 24. Conflict resolution with single level criticality, under EDF, $T(x,4,10), d\_base = avg\_rsp, \alpha = 0.2$

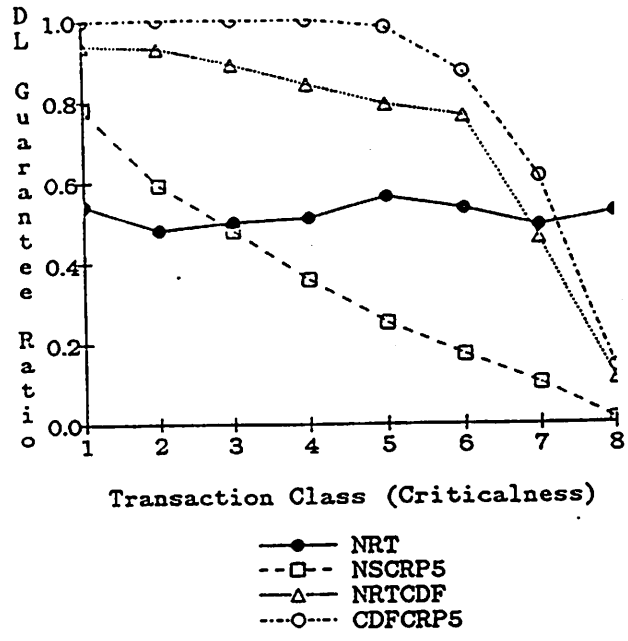

Figure 25. CPU scheduling vs. conflict resolution, $T(12,4,10), d\_base = avg\_rsp, \alpha = 0.2$

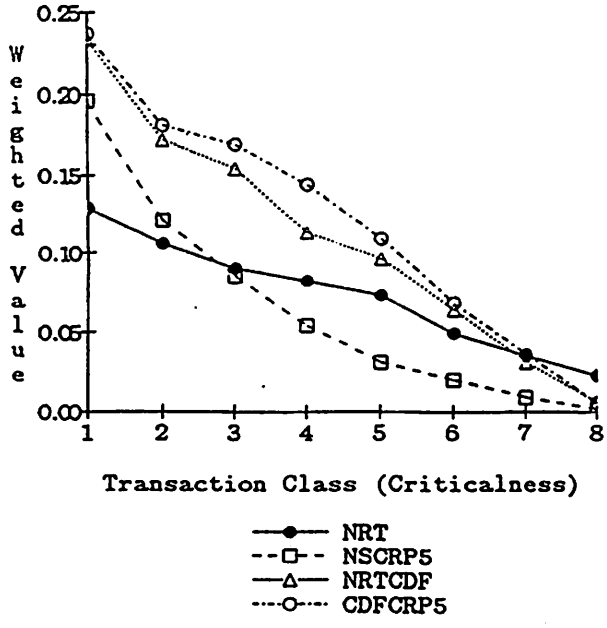

Figure 26. CPU scheduling vs. conflict resolution, $T(12,4,10), d\_base = avg\_rsp, \alpha = 0.2$

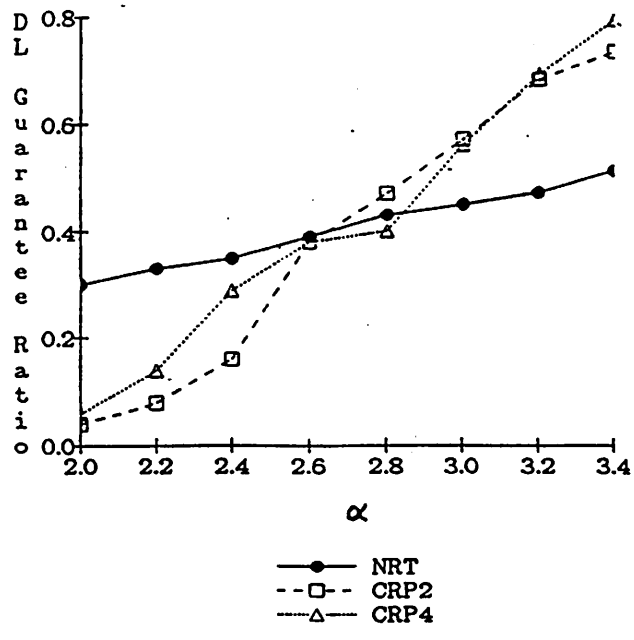

Figure 27. Deadline setting, with single level criticality, under EDF, $T(12,4,10), d\_base = avg\_rsp$