Real-Time Transaction Processing: Design, Implementation and Performance Evaluation

J. Huang, J. Stankovic
D. Towsley and K. Ramamritham
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 90-43
May, 1990

Real-Time Transaction Processing: Design, Implementation and Performance Evaluation *

Jiandong Huang
Department of Electrical and Computer Engineering

John A. Stankovic
Don Towsley
Krithi Ramamritham
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

May, 1990

Abstract

In addition to satisfying database consistency requirements, as in traditional database systems, real-time transaction processing systems must also satisfy timing constraints. To support real-time transaction processing, some new criteria and issues should be considered in design and implementation of real-time database systems. In this paper, we design algorithms for handling CPU scheduling, data conflict resolution, deadlock resolution, transaction wakeup, and transaction restart, based on a locking scheme for concurrency control. We implement a real-time database testbed containing these algorithms and evaluate their performance. The performance data indicates that (1) for a CPU-bound system, the CPU scheduling algorithm is the most significant of all the algorithms in improving the performance of real-time transactions, (2) conflict resolution protocols which directly address deadlines and criticalness can have a substantial impact on performance compared to protocols that ignore such information, and regardless of whether the system bottleneck is the CPU or the I/O, the protocols always improve performance for the most critical transactions, (3) both criticalness and deadline distributions strongly affect transaction performance, and (4) overheads such as locking and message communication are non-negligible and can not be ignored in real-time transaction analysis. We believe that these empirical results represent the first experimental results for real-time transactions on a testbed system.

Index terms - concurrency control, operating systems, performance evaluation, real-time databases, real-time systems, system design and implementation.

---

*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 IRI-8908693, 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 are becoming increasingly important in a wide range of applications. One example of real-time database systems is a computer integrated manufacturing system where a database keeps track of the state of physical machines, manages various processes in the production line, and collects statistical data from manufacturing operations. Transactions executing in the database may have deadlines in order to reflect, in a timely manner, the state of manufacturing operations or to respond to the control messages from operators. For instance, the information describing the current state of an object may need to be updated before a team of robots can work on the object. The update transaction is considered successful only if the data (the information) is changed consistently (in the view of all the robots) and the update operation is done within the specified time period. Other applications of real-time database systems can be found in program trading in the stock market, radar tracking systems, command and control systems, and air traffic control systems.

Most research 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 has largely ignored the problem of guaranteeing the consistency of shared data, especially for data that resides on a disk. These 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 or entirely new algorithms are needed in order to handle the requirements of real-time transactions in a unified manner. To study real-time database systems, new techniques are required to specify the properties of transactions, to define correctness, and to specify proper metrics.

Some papers have recently been published in the area of time-critical database systems. The topics covered in these papers include the identification of the characteristics of real-time transactions as well as a model of the underlying real-time operating system primitives [17,1,5], consistency issues[15,17,12], access control and conflict resolution [17,2], transaction scheduling [2,5], I/O scheduling [3,6], deadlock prevention [14], knowledge modelling [7,8,10] and data buffering [6]. 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, the results reported in the literatures are incomplete. For example, timing constraints and criticalness are two important 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. In addition, some of the previous work concentrates on only one or two specific issues, thus lacking an integrated systemwide approach. Furthermore, none of the ideas or proposed algorithms in the previous studies, though well thought out, have been evaluated in real systems. Indeed, only [2,6] present any experimental results. However, these results are based on simulation and the effect of many important overheads, such as locking and process

communication, are ignored.

The work reported here uses an integrated approach to study real-time transaction processing on a testbed system. We consider a real-time database system where data consistency based on the notion of serializability is preserved while the timing constraints are soft real-time.[1] Using a basic locking scheme for concurrency control, we investigate several algorithms for handling CPU scheduling, data conflict resolution, deadlock resolution, transaction wakeup, and transaction restart. The focus of this study is to understand the effect of these processing components on real-time databases and to identify the dominant components in real-time transaction processing. Another aspect of this work is to study the relationship between transaction timing constraints and criticalness and their combined effects on system performance. To provide deeper insight into our performance studies, we built a real-time database testbed, called RT-CARAT. The testbed captures the system overheads, which are largely ignored in simulation studies, and provides an improved understanding of the functional requirements and operational behavior of real-time database systems.

The rest of this paper is organized as follows. In section 2, we describe our real-time database model and a model for real-time transactions. 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, briefly showing how the proposed protocols and policies are implemented and integrated as a whole. The performance results are demonstrated and discussed in section 5. We make concluding remarks in Section 6.

2 A Real-Time Database Model

In this study we investigate a centralized, secondary storage real-time database. 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. The property of data consistency may be relaxed in some real-time database systems, depending on the application environment and data properties [15,17,12]. Relaxation of consistency is not considered in this paper. In our system, serializability is enforced by using the two-phase locking protocol.

Figure 1 depicts our system model from the perspective of transaction flow. This model is an extended version of the model used in [4]. The system is a closed queueing network, where a fixed number of users submit transaction requests one after another. Any new or re-submitted transaction will enter a scheduling queue first. The *scheduling* component performs dynamic CPU scheduling according to a certain policy. Before a transaction performs operation on a data object, it must go through the *concurrency control* component (CC) to obtain a lock on that object. If the request for the lock is denied, the transaction

[1] Soft real-time transactions are those that should meet their deadlines, but there may still be some (but diminishing) value for completing the transactions after their deadlines.

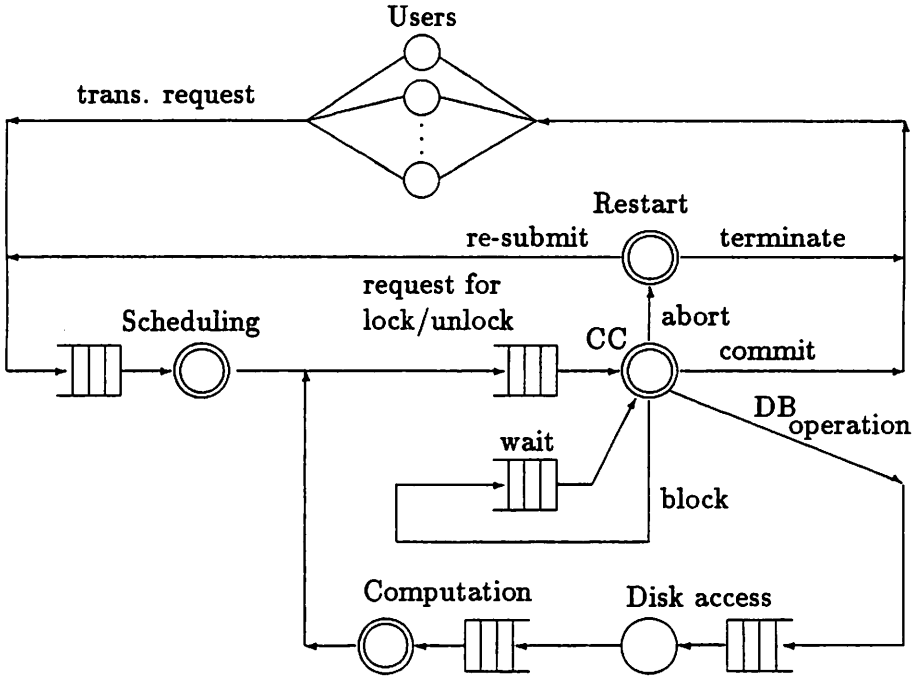

Figure 1: The system model

will be placed into a wait queue. The waiting transaction will be awakened when the requested lock is released. If the requested lock is granted, the transaction will perform the operation which consists of *disk access* and *computation*. A transaction may continue this "lock request – operation cycle" many times until it commits. At its commit stage, the transaction releases (unlocks) all the locks it has been holding. The concurrency control algorithm may abort a transaction for any number of reasons (to be discussed later). In that case, the *restart* component will decide, according to its current policy, whether the aborted transaction should be re-submitted or terminated. Note that this model only reflects the logical operations involved in transaction processing and it does not show the interaction of the processing components with physical resources. In practice, all of the processing components depicted by a double circle in Figure 1 compete for the CPU. Access to the CPU resource is determined by the *scheduling* component.

A real-time transaction is characterized by its length and a value function. [2] The transaction length is dependent on the number of data objects to be accessed and the amount of computation to be performed, which may not be always known. In this study, some of the protocols 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.

[2]Note that there are no standard workloads for real-time transactions, but a value function has been used in other real-time system work [13,1].

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). In general, the selection of a value function depends on the application [1]. In this work, we model the value of a transaction as a function of its criticalness, start time, deadline, and the current system time. Here criticalness represents the importance of transactions, while deadlines constitute the time constraints of real-time transactions. Criticalness and deadline are two characteristics of real-time transactions and they are not necessarily related. A transaction which has a short deadline does not imply 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. 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. We use the following formula to express the value of transaction T:

$$V_T(t) = \begin{cases} c_T, & s_T \le t < d_T \\ c_T \times (z_T - t)/(z_T - d_T), & d_T \le t < z_T \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

where t - current time;
s_T - start time of transaction T;
d_T - deadline of transaction T;
c_T - criticalness of transaction T, $1 \le c_T \le c_{Tmax}$;
c_{Tmax} - the maximum value of criticalness.

In this model, a transaction has a constant value, i.e. its criticalness value, before its deadline. The value decays when the transaction passes its deadline and decreases to zero at time z_T. We call z_T the *zero-value point.* As an example, Figure 2 shows the value functions of two transactions T_1 and T_2.

The decay rate, i.e. the rate at which the value of a transaction drops after its deadline, is dependent on the characteristics of the real-time transaction. To simplify the performance study, we model the decay rate as a linear function of deadline and criticalness. Here we study two models with z_T expressed by the following two formulas.

$$z_T = d_T + (d_T - s_T)/c_T \quad (2)$$

$$z_T = d_T + (d_T - s_T)/(c_{Tmax} - c_T + 1) \quad (3)$$

For a given c_{Tmax}, when c_T increases, under Eq. (2), z_T decreases, whereas under Eq. (3), z_T increases. With Eq. (2), if a transaction is extremely critical ($c_T \to \infty$), its value drops to zero immediately after its deadline. This is the case that we can see in many hard real-time systems. In this work, we use Eq. (1) and Eq. (2) as the base model, and we consider Eq. (3) as an alternative for Eq. (2) as a sensitivity study.

The transactions considered here are solely soft real-time. Given the value function, real-time transactions should be processed in such a way that the total value of completed

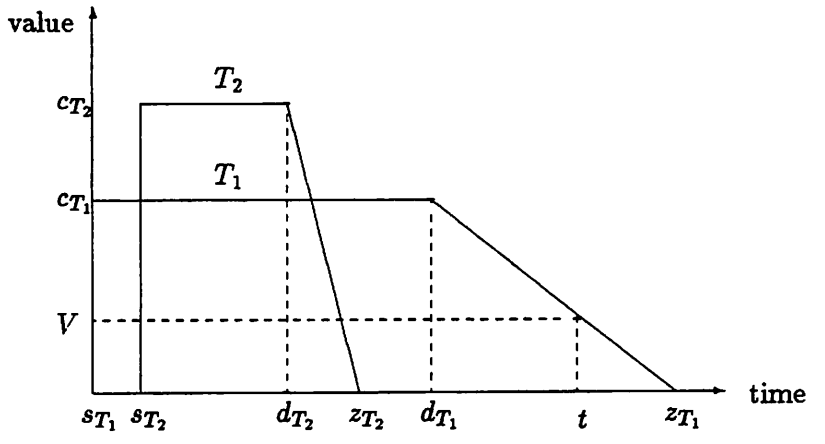

Figure 2: Value functions for transaction T_1 and T_2

transactions is maximized. In particular, a transaction should abort if it does not complete before time z_T (see Figure 2), since its execution after z_T does not contribute any value to the system at all. On the other hand, a transaction aborted because of deadlock or data conflict may be restarted if it may still impart some value to the system. [3]

Finally, at times, the estimated execution time of a transaction, r_T, may be known. This information might be helpful in making more informed decisions regarding which transactions are to wait, abort, or restart. This hypothesis is tested in our experiments by using algorithms that make use of r_T.

3 Real-Time Transaction Processing

Given the above system model and the characteristics of real-time transactions, the objective of our work is to develop and evaluate policies that provide the necessary support for real-time transactions. In this section, we explicitly address the problems of CPU scheduling, conflict resolution, transaction wakeup, deadlock resolution and transaction restart.

3.1 CPU Scheduling

There is a wide variety of algorithms for scheduling the CPU in traditional database systems. Such algorithms usually emphasize fairness and attempt to balance CPU and I/O bound transactions. These scheduling algorithms 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

[3] In some situations, a transaction may have to be completed even if time is past z_T. We do not consider such transactions here.

through transaction preanalysis [5]. On the other hand, in many cases complete knowledge may not be available. Then a priority based scheduling algorithm may be used, where the priority is set based on deadline, criticalness, length of the transaction, or some combination of these factors.

We consider three simple CPU scheduling algorithms. The first two algorithms are commonly found in real-time systems, and the third is an attempt to combine the first two so as to achieve the benefits of both.

- **Scheduling the most critical transaction first** (MCF)
- **Scheduling by earliest deadline first** (EDF)
- **Scheduling by criticalness and deadline** (CDF): In this algorithm, when a transaction arrives, it is assigned a priority based on the formula $(d_T - s_T)/c_T$. The smaller the calculated value, the higher the priority.

Under all of these three algorithms, when a transaction begin its commit phase, its priority is raised to the highest value among all the active transactions. This enables a transaction in its final stage 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. In all three algorithms, the transactions are preemptable, i.e., an executing transaction (not in its commit phase) can be preempted by a transaction with 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 (i.e. *write-write* and *write-read*). The conflict should be resolved according to the characteristics of the conflicting transactions. Here we present five protocols for conflict resolution.

In the following descriptions, T_R denotes the transaction which is requesting a data item D, and T_H is another transaction that is holding a lock on D. The five protocols have the same algorithmic structure as follows:

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

We start with the simple protocols in terms of complexity and the amount of information required.

3.2.1 Protocol 1 (CRP1): Based on criticalness only

This simple protocol only takes criticalness into account.

```
if c_{T_R} < c_{T_H} for all T_H
  then T_R waits
  else
       if c_{T_R} > c_{T_H} for all T_H
         then T_R aborts all T_H
         else T_R aborts itself
       end if
end if
```

Note that protocol 1 is a deadlock-free protocol, since waiting transactions are always considered in criticalness order. In addition, this protocol implements an *always-abort* policy in a system where all the transactions have the same criticalness.

3.2.2 Protocol 2 (CRP2): Based on deadline-first-then-criticalness

We anticipate that criticalness and deadlines are the most important factors for real-time transactions. Protocol 2 only takes these two factors into account. Here we separate deadline and criticalness by checking the two parameters sequentially. The algorithm for this protocol is:

```
if d_{T_R} > d_{T_H} for any T_H
  then T_R waits
  else
       if c_{T_R} ≤ c_{T_H} for any T_H
         then T_R waits
         else T_R aborts all T_H
       end if
end if
```

3.2.3 Protocol 3 (CRP3): Based on deadline, criticalness and estimation of remaining execution time

CRP3 is an extension of CRP2. Besides deadline and criticalness, we further examine the remaining execution time of the conflicting transactions. Here we assume that the computation time and I/O operations of a transaction are known and they are proportional. Then the remaining execution time of transaction T can be estimated by the following formula:

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

where R_total_T is the total number of records to be accessed by T; $R\_accessed_T(t)$ is the number of records that have been accessed as of time t. The protocol is as follows:

```
if d_{T_R} > d_{T_H} for any T_H
  then T_R waits
  else
       if c_{T_R} < c_{T_H} for any T_H
         then T_R waits
         else
              if c_{T_R} = c_{T_H} for any T_H
                then
                     if (time_needed_{T_R}(t) + t) > d_{T_R}
                       then T_R waits
                       else T_R aborts all T_H
                     end if
                else T_R aborts all T_H
              end if
       end if
end if
```

3.2.4 Protocol 4 (CRP4): Based on a virtual clock

Each transaction, T, has a virtual clock associated with it. The virtual clock value, $VT_T(t)$, for transaction T is calculated by the following formula.

$$VT_T(t) = s_T + \beta_T * (t - s_T), \quad t \geq s_T$$

where β_T is the clock running rate which is proportional to transaction T's criticalness. The higher the c_T, the larger the value β_T. The protocol controls the setting and running of the virtual clocks. When transaction T starts, $VT_T(t)$ is set to the current real time s_T. Then, the virtual clock runs at rate β_T . That is, the more critical a transaction is, the faster its virtual clock runs. In this work, $\beta_T = c_T$. The protocol is given by the following pseudo code.

```
if d_{T_R} > d_{T_H} for any T_H
  then T_R waits
  else
       if any VT_{T_H}(t) ≥ d_{T_H}
         then T_R waits
         else T_R aborts all T_H
       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. When the virtual clock of an executing transaction has surpassed its deadline, it cannot be aborted. Intuitively, this means that for the transaction T_H to make its deadline, we are predicting that it should not be aborted. For further details about this protocol, the reader is referred to [17].

3.2.5 Protocol 5 (CRP5): Based on combining transaction parameters

This protocol takes into account a variety of different information about the involved transactions. It uses a function $CP_T(t)$ to make decisions.

$$CP_T(t) = c_T * (w_1 * (t - s_T) - w_2 * d_T + w_3 * p_T(t) + w_4 * io_T(t) - w_5 * l_T(t))$$

where $p_T(t)$ and $io_T(t)$ are the CPU time and I/O time consumed by the transaction, $l_T(t)$ is the approximate laxity[4] (if known), and the w_k's are non-negative weights. The protocol is described by the following pseudo code.

if $CP_{T_R}(t) \leq CP_{T_H}(t)$ for any T_H
  then T_R waits
  else T_R aborts all T_H
end if

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 is referred to [17] for further discussion of this protocol.

In a disk resident database system, it is difficult to determine the computation time and I/O time of a transaction. In our experiments, we simplify the above formula for CP calculation as follows:

$$CP_T(t) = c_T * [w1 * (t - s_T) - w2 * d_T + w3 * (R\_accessed_T(t)/R\_total_T)]$$

where R_total_T and $R\_accessed_T(t)$ are the same as defined in CRP3.

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

3.3 Policies for Transaction Wakeup

When a 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 wake-up operation which correspond to each CRP.

- For CRP1, wake up the waiting transaction with the highest criticalness.

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

- For CRP2 and CRP3, wake up the waiting transaction with the minimum deadline.
- For CRP4, wake up the waiting transaction with maximum $VT_T(t)$ - the value of virtual clock.
- For CRP5, wake up the waiting transaction with maximum $CP_T(t)$ - the value of combined transaction parameters.

3.4 Deadlock Resolution

The use of a locking scheme may cause deadlock. This problem can be resolved by using deadlock detection, deadlock prevention, or deadlock avoidance. For example, CRP1 presented in the previous section is a kind of scheme for deadlock prevention. In this study, we focus on the problem of deadlock detection as it is required by the remaining concurrency control algorithms.

With the deadlock detection approach, 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 choose a victim so that the timing constraints of the remaining 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): *Trace the deadlock cycle. Abort the first transaction T with $t > z_T$; otherwise abort the transaction with the longest deadline.*

Recall that a transaction which has passed its zero-value point, z_T, may not have been aborted yet because it may not have executed since passing z_T, and because preempting another transaction execution to perform the abort may not be advantageous. Consequently, in this and the following protocols we first abort any waiting transaction that has passed its zero-value point.

Deadlock resolution policy 3 (DRP3): *Trace the deadlock cycle. Abort the first transaction T with $t > z_T$; otherwise abort the transaction with the earliest deadline.*

Deadlock resolution policy 4 (DRP4): *Trace the deadlock cycle. Abort the first transaction T with $t > z_T$; otherwise abort the transaction with the least criticalness.*

Deadlock resolution policy 5 (DRP5): Here we use $time\_needed_T(t)$ as defined in CRP3. A transaction T is *feasible* if $(time\_needed_T(t) + t) < d_T$ and *tardy* otherwise. 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: set tardy_set to empty
        set feasible_set to empty

Step 2: trace deadlock cycle
        for each T in the cycle do
        if t > z_T
          then abort T
               return
          else
               if T is tardy
                 then add T to tardy_set
                 else add T to feasible_set
               end if
        end if

Step 3: if tardy_set is not empty
          then search tardy_set for T with the least criticalness
          else search feasible_set for T with the least criticalness
        end if
        abort T
        return
```

3.5 Transaction Restart

A transaction may abort for any number of reasons. Basically, there are two types of aborts.

- *termination abort:* This type of abort is used to terminate a transaction. For example, a transaction may abort itself due to some execution exception. Also, a transaction could be aborted by the system if it has a zero value. Such aborts always lead to transaction termination.
- *concurrency abort:* This type of abort results from concurrency control. For instance, a transaction may be aborted in order to resolve a deadlock, or, a transaction may be aborted by another transaction because of a data access conflict. These aborted transactions should be restarted as long as they may still contribute a positive value to the system.

Based on our transaction model, we propose three policies for transaction restart from *concurrency abort.*

Transaction restart policy 1 (TRP1): *Restart an aborted transaction T if $t < z_T$.* In other words, an aborted transaction will be restarted as long as it may still have 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): *Restart an aborted transaction T if $r_T + t < z_T$*. Here we assume that the runtime estimate r_T of transaction T is known. The decision on transaction restart is based on the estimate of whether the transaction can complete by time z_T, if it is restarted.

Transaction restart policy 3 (TRP3): TRP3 is an extension of TRP2 and is given by the following algorithm.

```
if z_T − t < r_T
  then terminate T
  else
       if d_T − t > r_T
         then restart T
         else increase c_T by one
              restart T
       end if
end if
```

Here if it is estimated that T can not complete by d_T, then T's criticalness is increased by one. This restarted transaction has a higher priority than it did in its previous incarnation. 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 criticalness. The impact of this strategy on system performance is examined through the performance tests.

4 The Testbed System

We implemented and used a real-time database testbed named RT-CARAT. It is an extension of a previously built testbed, called CARAT (Concurrency and Recovery Algorithm Testbed) [11]. RT-CARAT was designed to be a flexible tool for the testing and performance evaluation of real-time database protocols. Currently, RT-CARAT testbed is a centralized, secondary storage real-time database system. The testbed is complete. It contains all the major functional components of a transaction processing system, such as transaction management, data management, log management, and communication management. The testbed is build on top of the VAX/VMS operating system. By appropriate settings of various VMS system parameters, non-essential overheads of VMS, such as memory paging and process swapping, are eliminated.

The testbed is implemented as a set of cooperating server processes which communicate via efficient message passing mechanisms. Figure 3 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

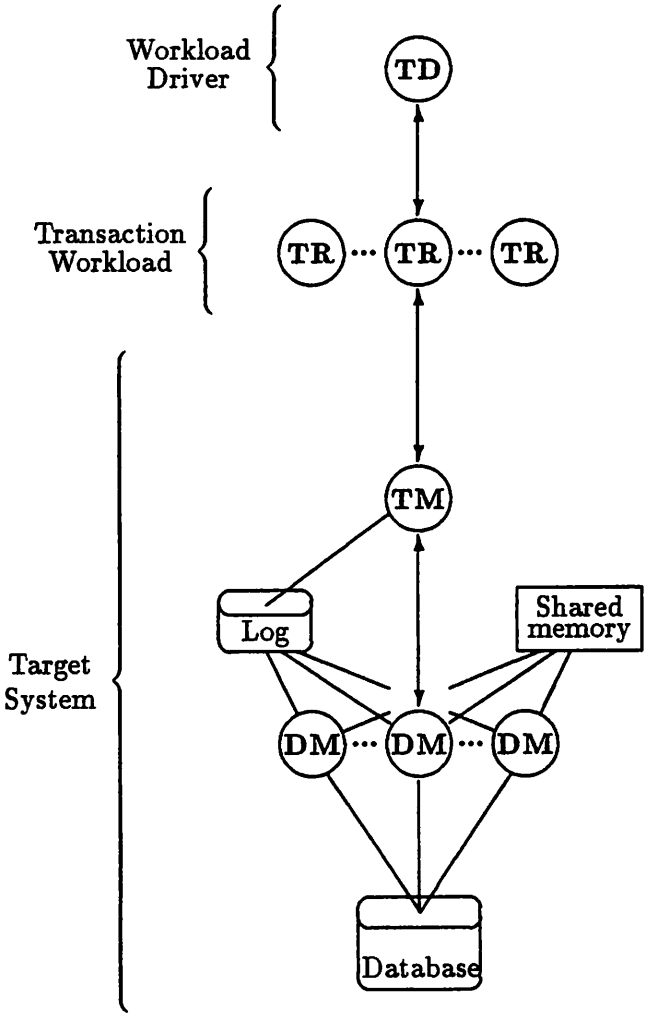

Figure 3: RT-CARAT processes and message structure

through the mailbox, a facility provided by VAX/VMS. In order to speed up the processing of real-time transactions, the communication among DM processes is implemented using a shared memory space, called a global section in VAX/VMS.

For concurrency control and recovery, RT-CARAT has adopted the two-phase locking protocol (2PL) and after-image (AI) journaling mechanism from the CARAT implementation. These operations, plus deadlock detection and conflict resolution, are carried out in the DM processes.

In RT-CARAT, the CPU is scheduled based on transaction priority with preemption using the underlying VAX/VMS operating system real-time priorities. The CPU scheduler is embedded in the TM. Upon receiving a transaction execution request from a TR, the scheduler assigns a priority to the transaction according to the CPU scheduling policy (see Section 3.1). The scheduling operation is done by mapping the assigned transaction priority to the real-time priority of the DM process which carries out the transaction execution. At this point, an executing DM will be preempted if it is not the highest priority DM process at the moment, otherwise it will continue to run. 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 (see Section 3.2) embedded in the DM.

In a secondary storage database system, disk I/O is an important operation. From the point of view of real-time, especially for I/O bound real-time database systems, these I/O operations should be scheduled according to the characteristics of real-time transactions [3,6]. In our testbed, unfortunately, disk access is under the control of disk controllers instead of the operating system, i.e. there is no way to directly manipulate disk access through the system utilities. Thus, in the current implementation, there is no component dealing with real-time I/O scheduling. However, through careful design of the experiments, we are able to determine the impact of not doing real-time I/O scheduling on system performance.

5 Experimental Results

5.1 The Test Environment

In our experiments, the database consists of 3000 physical blocks (512 bytes each) with each block containing 6 records for a total of 18,000 records. Two separate disks are used, one for the database and the other for the log. In all the experiments, the multi-programming level in the system is 8.

A transaction generator in each TR process generates transactions according to a configuration file where transaction type (write or read only), length (the number of records to be accessed and the amount of computation time), criticalness, and deadline are specified. A transaction performs a certain number of predefined operations, called *steps*, and each operation may access a certain number of records and do a certain amount of computation. A transaction terminates upon completion or a termination abort. In the tests described

here, the transaction generator submits a new transaction immediately after the previous transaction has terminated, i.e. there is no external think time between consecutive transactions.

In the experiments presented in this paper, each transaction is either a *read only* transaction or *a write* transaction. To achieve a mix of read and write operations in the system, we specify the number of *read* transactions and *write* transactions, which is represented by w/r - the ratio of *write* transactions to *read* transactions.

We express the length of a transaction by the form $T(x, y, u)$, where x is the number of steps, y the number of records accessed in each step, and u the amount of computation units per step with 1 unit = 50 ms. 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. For each workload in the experiments, *d_base* is specified first by the formula:

$$d\_base = avg\_rsp - stnd\_dvi$$

where *avg_rsp* is the average response time of the same real-time transactions when executed 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 from a uniform distribution. In the experiments, there are up to 8 levels of criticalness and accordingly, the transactions are classified into 8 classes. We specify the criticalness of transaction T as a function of its class, i.e.

$$c_T(class) = 8 - class + 1, \quad class = 1, 2, ..., 8$$

The smaller the class number, the higher the corresponding criticalness, or vice versa. Once 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).

5.2 Baseline and Metrics

For these tests, the performance baseline is a non real-time transaction processing system (NRT), which is defined by the following baseline algorithms:

- CPU scheduling policy: *schedule transactions by a multi-level feedback queue* (MFQ);
- conflict resolution: *place the lock-requesting transaction into a FIFO wait queue* (CRP0);
- deadlock resolution: DRP1;
- transaction restart: TRP1.

We use the following metrics to evaluate the proposed algorithms and protocols.

- Deadline guarantee ratio (DGR_{class}) - the percentage of transactions in a class that complete by their deadline.

- Average deadline guarantee ratio (ADGR) - the percentage of transactions in all classes that complete by their deadline, i.e.

$$ADGR = \frac{1}{8} \sum_{class=1}^{8} DGR_{class}$$

- Weighted value (WV_{class})- the total value of all transactions in a class that complete by their zero-value points (z_T) divided by the total maximum value of the invoked transactions in all classes.

- Total weighted value (TWV) - the sum of weighted values in all classes, i.e.

$$TWV = \sum_{class=1}^{8} WV_{class}$$

- Abort ratio (AR_{class})- the percentage of aborted transactions in a class.

- Total abort ratio (TAR) - the percentage of aborted transactions in all classes, i.e.

$$TAR = \sum_{class=1}^{8} AR_{class}$$

Our data collection is based on the method of *replication*. 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.

5.3 Experiments

We present results from the following six *sets* of experiments:

- **System performance measurements**: The purpose of this experiment is to study the supporting system software performance and to identify the dominant overheads and their magnitudes.
- **CPU scheduling**: The effect of CPU scheduling on the performance of real-time transactions was studied by varying transaction length, deadline setting, write/read ratio, and mixing transactions with different lengths.
- **Conflict resolution**: In these experiments we compared the performance of all the conflict resolution policies by varying transaction length, deadline setting, and write/read ratio.

- **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.
- **CPU bound vs. I/O bound systems**: This experiment was intended to identify the degree of the system performance degradation due to the lack of real-time based I/O scheduling.
- **Sensitivity study for value functions**: Based on the value function model proposed in Section 2, we investigated the impact of two different value functions on the experimental results.

Besides the above studies, we also investigated the proposed *deadlock resolution policies*. The results show that for the wide range of workloads we tested, the performance of all the policies are similar because the deadlock cycle involved only two transactions most of the time. We also did experiments for the three *transaction restart policies*, and found no significant differences between them. This is because the runtime estimate (r_T) used in the algorithms was based on the transaction average response time. Due to the large deviation of the response time, r_T was not accurate enough to support algorithms TRP2 and TRP3. Thus, to save space, we do not show these experimental results here. For all the experiments discussed below, DRP1 and TRP1 were used for deadlock resolution and transaction restart, respectively.

Table 1 summarizes the parameter settings and protocol selections in the experiments.

5.3.1 System Performance Measurements

To study various system overheads, several experiments are conducted with different workloads and different combinations of the protocols. To focus on the overheads, the user computation time u is set to zero. This has the effect of maximizing number of data requests and the subsequent overheads. Table 2 shows the measurement results with respect to (average) CPU utilization over all the runs. In the table, the data for RT-CARAT is the overall system CPU utilization which is the sum of the CPU utilizations of the TR, TM and DM processes. *Locking* is a processing component of DM, which in addition includes *deadlock detection* and *conflict resolution*. *CPU scheduling* is part of the TM process. The item *communication* includes all the overhead involved in message communication through mailboxes in the system.

It can be seen that locking and message communication are the main overheads of the system, while the overhead from CPU scheduling, deadlock detection and conflict resolution is negligible. These measures indicate that in the analysis of real-time transaction processing, some overheads such as locking and process communication cannot be simply ignored, while some overheads such as conflict resolution should not be over-emphasized.

Table 1: Experimental Settings

Parameter	Setting
Disks	disk1: database; disk2: log.
Database size	3000 blocks (18000 records)
Multiprogramming level	8
Transaction class	8 levels of criticalness
α (deadline window factor)	2.0 - 5.0
w/r (ratio of write/read trans.)	2/6, 4/4, 6/2, 8/0
x (steps per transaction)	4 - 20 steps
y (records accessed per step)	4 records
u (computation time per step)	0 or 10 units
Protocol	**Selection**
CPU scheduling	MFQ, MCF, EDF, CDF
Conflict resolution	CRP$i, i = 0, 1, 2, 3, 4, 5$
Deadlock resolution	DRP1
Transaction restart	TRP1
Wakeup policy	A function of conflict resolution policy (see 3.3)

Table 2: System Performance Measurements

Process/Component	Avg. CPU util.	Standard. devi.
RT-CARAT	0.944	0.012
TRs	0.080	0.001
TM	0.167	0.017
DMs	0.697	0.019
Locking (lock/unlock)	0.438	0.021
Deadlock detection	0.003	0.002
Conflict resolution	0.001	0.000
CPU scheduling	0.006	0.003
Communication	0.213	0.017

5.3.2 CPU Scheduling

In this experiment, in order to observe the effects that different CPU scheduling policies have on performance, we used CRP0 for conflict resolution, i.e. in case of data access conflict, it is always the lock-requesting transaction that is queued.

Figures 4-6 compare three scheduling schemes with respect to deadline guarantee ratio, weighted value, and total abort ratio, respectively. In this experiment, all the transactions were of equal length, $T(12,4,10)$, with w/r=2/6 and α=3. Figure 4 plots the deadline guarantee ratio versus the transaction class. Our first observation is that CPU scheduling algorithms that make use of transaction information perform better than the baseline NRT, except for the transactions in classes 7 and 8 when executed under the scheduling algorithms MCF and CDF. As compared with the baseline and the scheduling algorithms EDF, both MCF and CDF result in a higher deadline guarantee ratio for the transactions with high criticalness, but a lower deadline guarantee ratio for the transactions with low criticalness. This is because the two algorithms take the criticalness of transactions into account and hence 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. Note that for transaction class 1, CDF improves the deadline guarantee ratio from 60% for the baseline to 97% for CDF. 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.

Figure 5 depicts the weighted value that each class of transactions contributed to the system using a linear weighting scheme where each of the 8 criticalness levels differs in value by 1 unit. 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. Since the current value weighting scheme is linear, other weighting schemes, such as exponential, would result in even higher gains by the real-time CPU scheduling algorithm.

The transaction total abort ratio is shown in Figure 6. These plots are basically the inverse of plots for 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 criticalness transactions is achieved by aborting more low critical transactions. The abort ratio with EDF is low over all classes of transactions.

The low transaction abort ratio under EDF scheduling policy results from the fact that under EDF most of the time transactions execute in a kind of sequential order. This interesting result can be explained as follows. First, transaction deadline and transaction arrival time are highly correlated in the experiments. Later arriving transactions usually have farther deadlines. Figure 7 shows the transaction deadline distributions under scheduling policy EDF. [5] Here the horizontal axis represents the deadline order of arriving transactions. Upon arrival of every transaction, we compare its deadline value with the deadlines

[5]Note that we also measured the transaction deadline distributions under NRT, MCF and CDF. The results are similar to what we show in Figure 7.

of those transactions executing in the system. As a result, the smaller its deadline value, the smaller its deadline order. For instance, an arriving transaction will be in order 1 if its deadline value is the smallest compared to the transactions already in the system, and will be in order 8 if its deadline value is the largest. The vertical axis represents the probability of the occurrence of each order. Clearly, the large percentage of arriving transactions have the longer deadline values, compared to the transactions executing in the system. Thus, under EDF most of the later arriving transactions have lower priority than the transactions already in their execution. Therefore, transaction preemption due to transaction arrivals seldom occurs. Second, the system is highly CPU-bound and the computation time in each transaction step is very long (500ms for u=10), compared to I/O time (about 80ms for y=4). Under such a system, with the given deadline distribution as just discussed, EDF actually reduces the concurrency of transaction execution. This is corroborated by the measures of the average number of granted locks at any instant in the system. Figure 8 plots the average number of granted locks as a function of transaction length x, under NRT, MCF, EDF and CDF, respectively. The measures indicate that EDF results in the lowest concurrency among the four scheduling policies. The degree of concurrency under the baseline NRT is the highest, since NRT uses a multi-level feedback queue scheduling policy which emphasizes equality for concurrent transactions. After examining these performance results in detail, it is clear now that under EDF, transaction execution is almost sequential.

The scheduling algorithms were also tested for transactions of different lengths and the performance results were basically the same as in the figures presented above [9].

We next study the sensitivity of the scheduling algorithms to transaction deadline settings. α is varied from 2.0 to 4.0 in steps of 0.5 with $w/r = 2/6$ and $T(12, 4, 10)$. Figure 9 shows the average deadline guarantee ratio over 8 classes of transactions. Among the scheduling algorithms EDF is most sensitive to deadline setting. This is because EDF uses only the information about deadline for scheduling. Also note that EDF performs best when the deadline is loose, and worst when the deadline is tight. As we mentioned above, under EDF, transactions execute almost in sequential order most of the time. Thus, when the deadline is tight, it performs worse than the baseline which is scheduled using a multi-level feedback queue. Both MCF and CDF are not sensitive to the deadline distributions. This is obvious for MCF, since it does not use deadline information. For CDF, criticalness is a dominant factor even though it uses deadline information.

We further examine the overall effects of the scheduling algorithms by varying the transaction length. Figure 10 presents the total weighted value versus the transaction length x, with $w/r = 2/6$ and $\alpha = 3$. The reader can see that the total value that the system gains under CPU scheduling is far more than the value gained by the baseline. However, when transactions become longer, the performance degrades as much as 20% because of higher data conflict rate, higher deadlock rate, and relatively tighter deadline windows (due to the larger *stnd_dvi* value).

All the above results are for workloads with $w/r = 2/6$. The CPU scheduling algorithms are also studied by varying the ratio of write/read transactions. Figure 11 and 12 depicts the performance results for different w/r ratio. Under the baseline NRT and the scheduling algorithms MCF and CDF, the data access conflict increases as w/r ratio increases, thus

increasing the transaction abort ratio (due to deadlock) and lowering the deadline guarantee ratio. Owing to its sequential execution nature discussed above, EDF is not sensitive to the change of w/r ratio.

These observations and discussions lead to the following points:

- CPU scheduling by MCF and CDF significantly 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 criticalness as well as timing constraints. Moreover, in order to get the best performance, both criticalness and deadline of a transaction are needed for CPU scheduling.
- EDF is most sensitive to deadline distributions and relatively independent of data access conflict. It performs well only when deadlines are not tight.

5.3.3 Conflict resolution

In this experiment, we study the performance of conflict resolution protocols by varying transaction length, deadline settings, and w/r ratio. The CPU scheduling algorithm used in the experiment is CDF, since it was shown to be the best in the previous section. Unlike the other experiments, the baseline compared in this experiment was chosen to be NRT_CDF - non real-time, applying CPU scheduling (CDF) only. This baseline enables us to isolate the performance differences due to the use of conflict resolution protocols. To create a high conflict rate, we first exercised the workloads with all write transactions (Figures 13-15) and then the workloads with the mix of read and write transactions (Figure 16).

Figure 13 presents the performance results with respect to total weighted value versus x, with w/r=8/0 and $\alpha = 3$. With short transactions ($x = 4, 8$), all the protocols perform basically the same, and there is no significant performance improvement as compared with NRT_CDF. 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. The performance difference can be seen as the conflict rate becomes high with long transactions ($x = 12, 16, 20$), where all the protocols outperform NRT_CDF.

Figure 14 provides a detailed examination of the performance results of long transactions with w/r=8/0, $x = 16$ and $\alpha = 3$. Among the five protocols, CRP1, the simplest one, performs best. This is largely due to the fact that CRP1 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 (e.g. CPU, I/O, critical section and data). Since the conflict resolution is applied before deadlock resolution in the course of transaction execution, an early abort from conflict resolution decreases the amount of resources that would be wasted if the transaction is aborted later from deadlock resolution.

The performance of CRP2 and CRP3 is almost identical, since CRP3 checks only one more condition than CRP2, namely the amount of time that the transaction needs to finish

before its deadline. It is clear now that this additional estimated information does not substantially improve the performance.

CRP4 only outperforms NRT_CDF for transactions with high criticalness, but it performs slightly better than CRP2 and CRP3, as well as CRP1, for low critical transactions. This is because CRP4 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, CRP4 allows the lock requester with high criticalness to wait for the lock holder with low criticalness, thus lowering the performance for higher criticalness transactions. This situation never occurs with the other conflict resolution protocols.

The performance of CRP5 is not as good as CRP1 but is better than that of other protocols. [6] This is because the dominant factor in CRP5 is criticalness (c_T is a multiplicand in calculation of $CP_T(t)$), which results in transaction aborts similar to those due to CRP1, i.e. the large percentage of transaction aborts come from conflict resolution. But because CRP5 is not a deadlock-free protocol, there are still some aborts due to deadlock.

The conflict resolution protocols were also studied with respect to transaction deadline distribution. The performance results illustrated in Figure 15 show that the protocols CRP2, CRP3 and CRP4, which explicitly use deadline information, are sensitive to deadline settings. Note that the three protocols do not provide better performance than the baseline when the deadline is tight, and as the deadline window increases, the total weighted value becomes "saturated" (no relative gain occurs as compared with the baseline). This indicates that the conflict resolution plays an important role only for a certain range of deadline distributions.

Thus far, we have shown the performance of conflict resolution protocols under the situation where only write-write conflicts exist. Now we consider the workloads which lead to write-read as well as write-write conflicts. Figure 16 presents the performance results in terms of total weighted value versus the ratio of write/read transactions. As we can see, the performance difference among the proposed protocols and the baseline is small when the w/r ratio is equal to 2/6. This is because the majority of concurrent transactions are executing *read* operations, and therefore the conflict rate is low. As the w/r ratio increases, the transaction performance degrades sharply under all the conflict resolution protocols, except CRP1. There are two reasons for the good performance of CRP1. First, under our value weighting scheme, CRP1 maximizes the total value that transactions impart to the system. Second, as we mentioned previously, the implementation makes CRP1 most efficient in using the system resources.

We also studied the conflict resolution protocols with the mix of write/read transactions by varying the deadline window factor α. The results are similar to what we have shown in Figure 15, where only write transactions were considered.

The overall results from this set of experiments show that

- the conflict resolution protocols which directly address deadlines and criticalness produce better performance than protocols that ignore such information;

[6] Different sets of weights $w1$, $w2$ and $w3$ were exercised in testing of CRP5. There was very little difference in terms of CRP5's performance.

- the relative performance among the proposed conflict resolution protocols is consistent, under different deadline distributions, transaction lengths and the ratio of write/read transactions;
- among the five protocols, CRP1 provides the best performance. This is due to two factors: 1) it is a simple protocol which maximizes the value and 2) its implementation leads to the efficient use of system resources.

5.3.4 CPU scheduling vs. conflict resolution

To explicitly distinguish between 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) CRP1 - applying conflict resolution protocol CRP1 with MFQ (the baseline of the CPU scheduling algorithm); (3) CDF - applying CPU scheduling CDF with CRP0 (the baseline of conflict resolution protocol); and (4) CDF_CRP1 - applying both CPU scheduling CDF and conflict resolution protocol CRP1. The workload for the test presented here is $T(12, 4, 10)$ with deadline setting $\alpha = 3$.

It is observed in Figure 17 that CRP1 improves the performance only for the transactions with very high criticalness (class 1), but severely degrades the performance, much worse than NRT, as transactions become less critical. CDF, on the other hand, greatly improves the performance of transactions in most classes. CDF_CRP1, the combination of CDF and CRP1, provides the best performance. The observations that can be made from this experiment indicate that real-time CPU scheduling improves the deadline guarantee ratio of real-time transactions as much as 80% and that there is a need to combine the CPU scheduling scheme with the conflict resolution so as to achieve up to an additional 12% performance improvement.

5.3.5 CPU bound vs. I/O bound systems

In the previous experiments all the transactions examined had the same computation time - 10 units in each transaction step. Under such workloads, the system is highly CPU bound with the (database) disk utilization being around 15% and the CPU utilization reaching as high as 97%. In this experiment we examine the behavior of workloads which yield systems which are not CPU bound. We created an I/O bound system by reducing the computation time to zero and further increasing the number of I/O operations in each transaction step. Figure 18 illustrates a performance result from the I/O bound system where the disk utilization was about 94% and the CPU utilization was approximately 50%. As compared with the baseline NRT, the CPU scheduling scheme (CDF), which performs very well in the CPU bound system (see Figure 17), does not improve the performance at all. This is understandable, since now the system bottleneck is at I/O, not at CPU. Here the interesting observation is that in the I/O bound system, the conflict resolution protocols, combined with CPU scheduling, still improve performance in terms of meeting deadlines for high critical transactions.

Clearly, I/O scheduling sensitive to the real-time nature of transactions is needed for I/O bound systems. We hypothesize that by applying real-time based I/O scheduling algorithms, together with the conflict resolution protocols, the system performance will be improved. For Figure 18, in particular, we expect that the deadline guarantee ratio of transactions with higher criticalness will increase through I/O scheduling, similar to what we have achieved from CPU scheduling for a CPU bound system (see Figure 17).

5.3.6 Sensitivity of different value functions

As we discussed in Section 2, for all of the above experiments, we used Eq. (1) and Eq. (2) as the basic value function. To study the sensitivity of our value function model, we now use Eq. (3) as an alternative to Eq. (2) and hence combine Eq. (1) and Eq. (3) to produce value function VF2.

Figure 19 illustrates the performance comparisons between the two different value functions used for CPU scheduling. Here we plot the total weighted value with respect to the deadline window factor α. For the baseline NRT, the total weighted value gained under VF2 is larger than that under VF1. This is what one can expect, since under VF2, the transaction's zero-value point (z_T) is proportional to its criticalness value, and thus, in the case of missing deadline, transactions with high criticalness, modeled by VF2, may impart a larger value to the system than the transactions modeled by VF1. For the same reason, we see the performance difference between VF1 and VF2 for the CPU scheduling scheme EDF. Because the deadline guarantee ratio of EDF is lower than that of NRT when deadlines are tight, and higher when deadlines are loose (see Figure 9), the change of the performance difference between VF1 and VF2 is greater with EDF than with NRT. For the CPU scheduling schemes MCF and CDF, there is no large performance difference between VF1 and VF2. This is because both algorithms consider transaction criticalness for scheduling. According to our value weighting scheme, the large percentage of the total weighted value results from the committed transactions with high criticalness. Since for the tested workloads, the deadline guarantee ratio of the transactions with high criticalness is very high (above 95%, see Figure 4), the difference on modeling the decay value will not affect the overall performance results.

VF2 was also exercised with the proposed conflict resolution policies. The observations from these experiments were basically the same as what we just discussed for the results from CPU scheduling.

In general, our experimental results show that the two value functions may result in some performance difference for each protocol. In any case, however, the relative performance among the different protocols is the same.

6 Conclusions

Real-time transaction processing is complex. Many issues arise as data consistency has to be maintained and timing constraints for the transactions have to be met. In this paper,

we have presented a real-time database model where the general notion of value function is used to characterize real-time transactions. Based on a basic locking scheme for concurrency control, we have developed several algorithms with regard to the issues of CPU scheduling, data conflict resolution, transaction wakeup, deadlock resolution, and transaction restart. In addition, we have briefly presented the RT-CARAT testbed where the proposed algorithms are integrated and implemented.

In general, our experimental results from the testbed indicate the following:

- In a CPU-bound system, the CPU scheduling algorithm has a significant impact on the performance of real-time transactions, and dominates all the other types of protocols. In order to obtain good performance, both criticalness and deadline of a transaction are needed for CPU scheduling;
- Various conflict resolution protocols which directly address deadlines and criticalness produce better performance than protocols that ignore such information. In terms of transaction's criticalness, regardless of whether the system bottleneck is the CPU or the I/O, criticalness-based conflict resolution protocols always improve performance;
- Both criticalness and deadline distributions strongly affect transaction performance. Under our value weighting scheme, criticalness is a more important factor than the deadline with respect to the performance goal of maximizing the deadline guarantee ratio for high critical transactions and maximizing the value imparted by real-time transactions;
- Overheads such as locking and message communication are non-negligible and can not be ignored in real-time transaction analysis.

An important issue we have not yet addressed is I/O scheduling. Unfortunately, most of today's disk controllers have built-in scan algorithms that do not take deadlines and criticalness into account and are not easily modified.

Our work presented in this paper can be extended in many directions. For example, optimistic concurrency control (OCC) is another scheme for enforcing data consistency in traditional database systems. If, and under which conditions, OCC is suitable for real-time databases remains to be answered. Another issue is the buffer management. Buffer management plays an important role in traditional database systems. We would like to know how important the buffer management is in real-time databases and how to improve performance through buffer management. Finally, with our developed soft real-time database system, we plan to study how to interface the system with hard real-time systems and how to integrate them as a whole.

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*, Aug. 1988.

[3] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data," *A Technical Report, CS-TR-207-89, Princeton University*, Feb. 1989.

[4] Agrawal, R., M.J. Carey and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Transaction on Database Systems, Vol.12, No.4*, Dec. 1987.

[5] Buchmann, A.P., D.R. McCarthy, M. Hsu, and U. Dayal, "Time-Critical Database Scheduling: A Framework For Integrating Real-Time Scheduling and Concurrency Control," *Data Engineering Conference*, Feb. 1989.

[6] Carey, M.J., R. Jauhari and M. Livny, "Priority in DBMS Resource Scheduling," *Proceedings of the 15th VLDB Conference*, 1989.

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

[8] Dayal, U., "Active Database Management Systems," *Proceedings of the 3rd International Conference on Data and Knowledge Management*, June 1988.

[9] Huang, J., J.A. Stankovic, D. Towsley and K. Ramamritham, "Experimental Evaluation of Real-Time Transaction Processing," *A Technical Report, COINS 89-48, University of Massachusetts*, April 1989.

[10] Hsu, M., R. Ladin and D.R. McCarthy, "An Execution Model for Active Database Management Systems," *Proceedings of the 3rd International Conference on Data and Knowledge Management*, June 1988.

[11] Kohler, W. and B.P. Jenq, "CARAT: A Testbed for the Performance Evaluation of Distributed Database Systems," *Proceedings of the Fall Joint Computer Conference*, Nov. 1986.

[12] Lin, K.J., "Consistency issues in real-time database systems," *Proceedings of the 22nd Hawaii International Conference on System Sciences*, Jan. 1989.

[13] Locke, C.D., "Best-Effort Decision Making for Real-Time Scheduling," *Ph.D. Dissertation*, Canegie-Mellon University, 1986.

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

[15] Son, S.H., "Using Replication for High Performance Database Support in Distributed Real-Time Systems," *Proceedings of the 8th Real-Time Systems Symposium*, Dec. 1987.

[16] Son, S.H. and C.H. Chang, "Priority-Based Scheduling in Real-Time Database Systems," *Proceedings of the 15th VLDB Conference*, 1989.

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

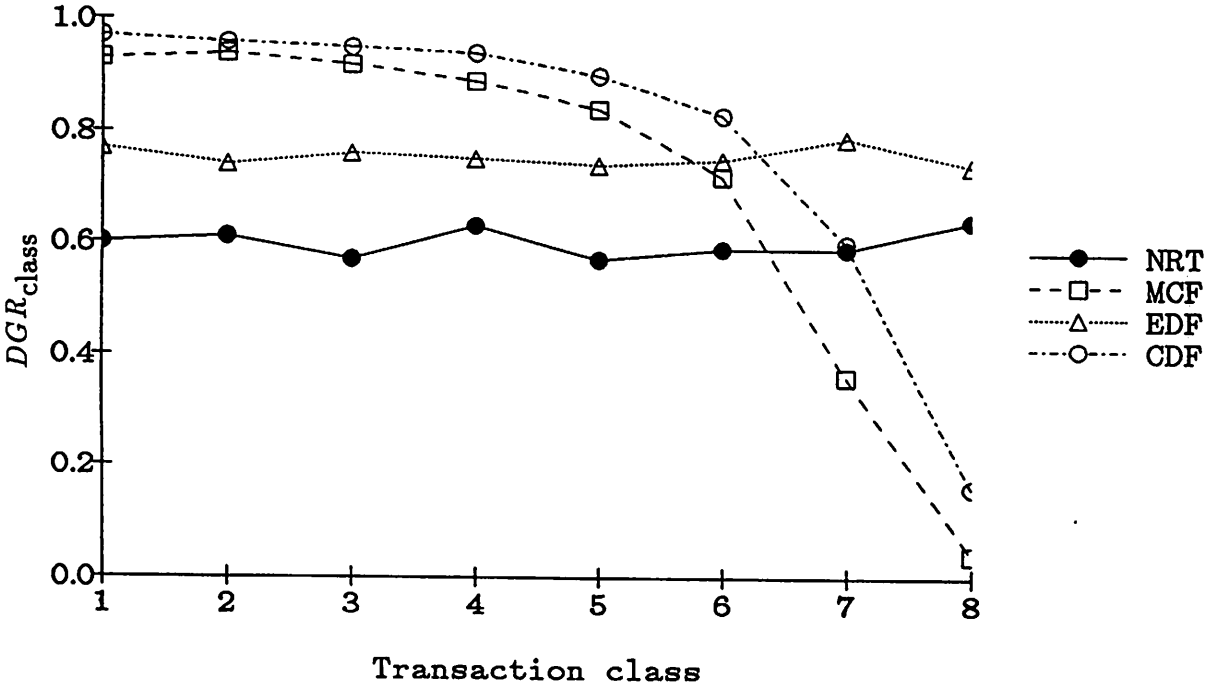

Fig. 4: CPU scheduling with w/r=2/6, T(12,4,10), a=3

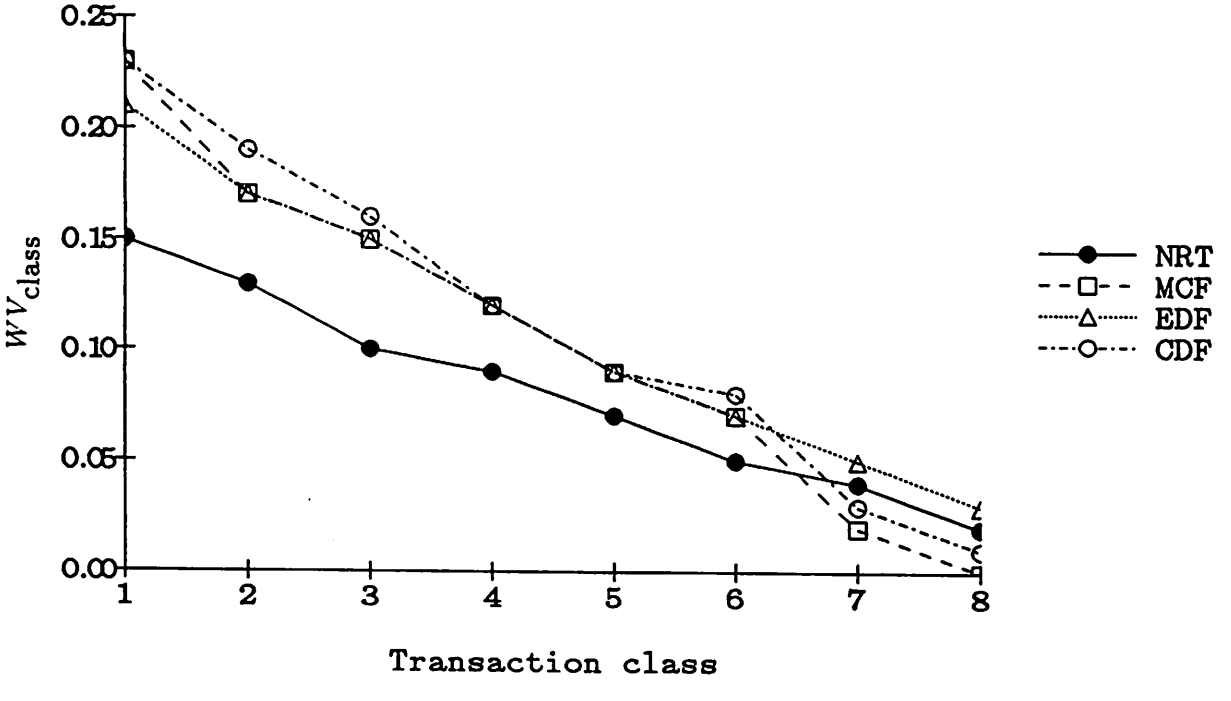

Fig. 5: CPU scheduling with w/r=2/6, T(12,4,10), a=3

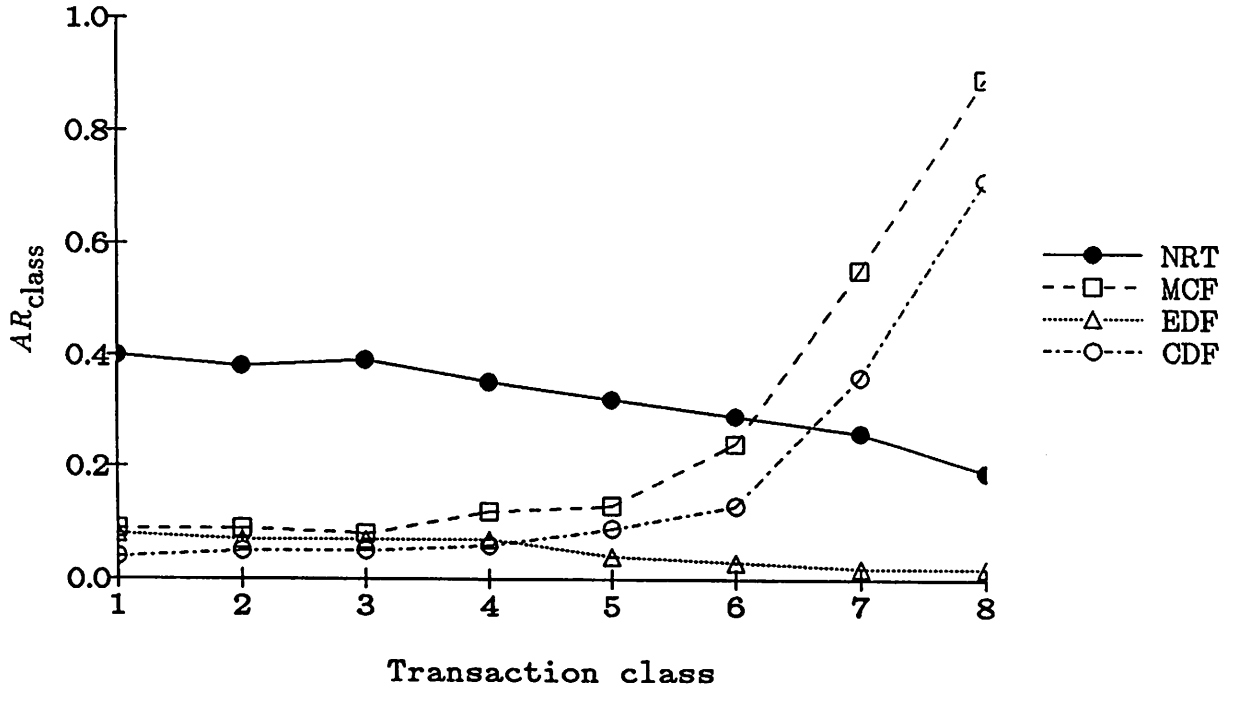

Fig. 6: CPU scheduling with w/r=2/6, T(12,4,10), a=3

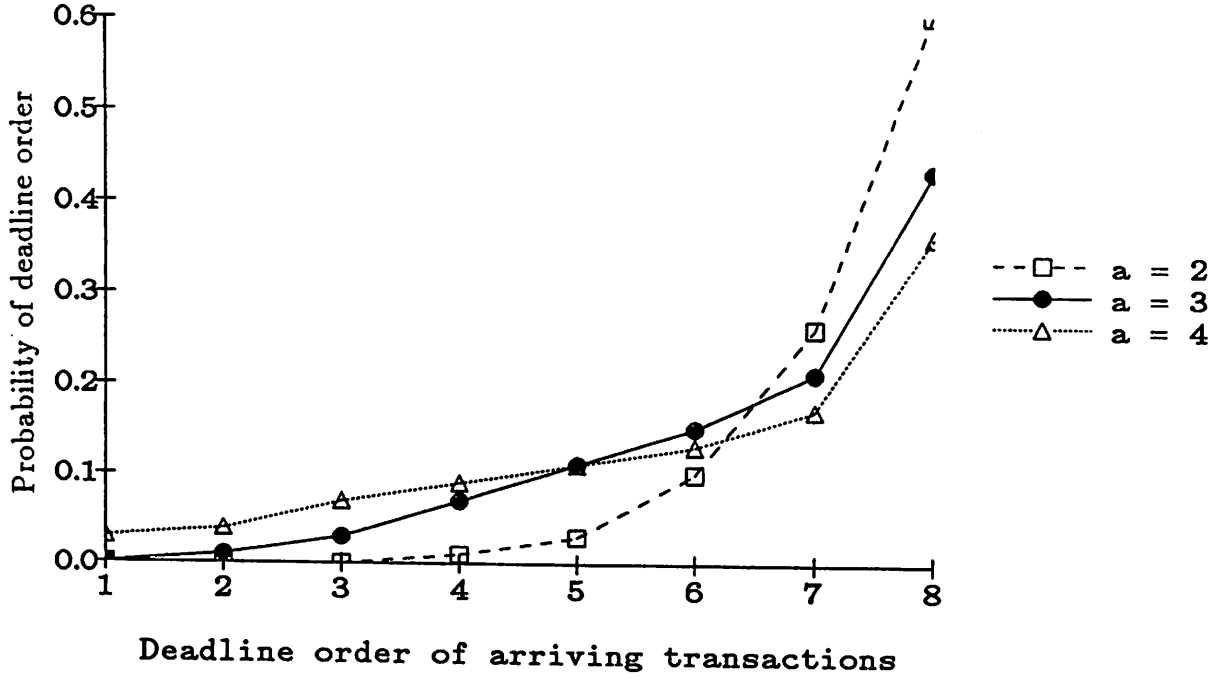

Fig. 7: Deadline distribution under EDF

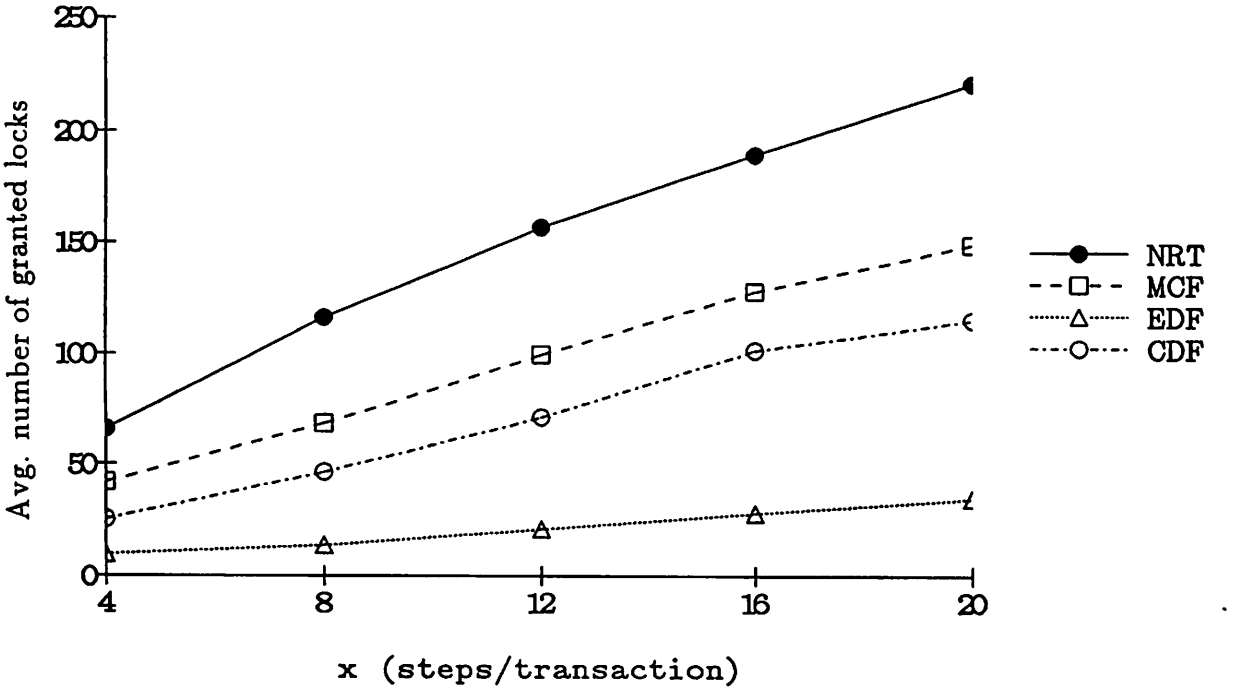

Fig. 8: Concurrency measurement with w/r=2/6, T(x,10,4), a=3

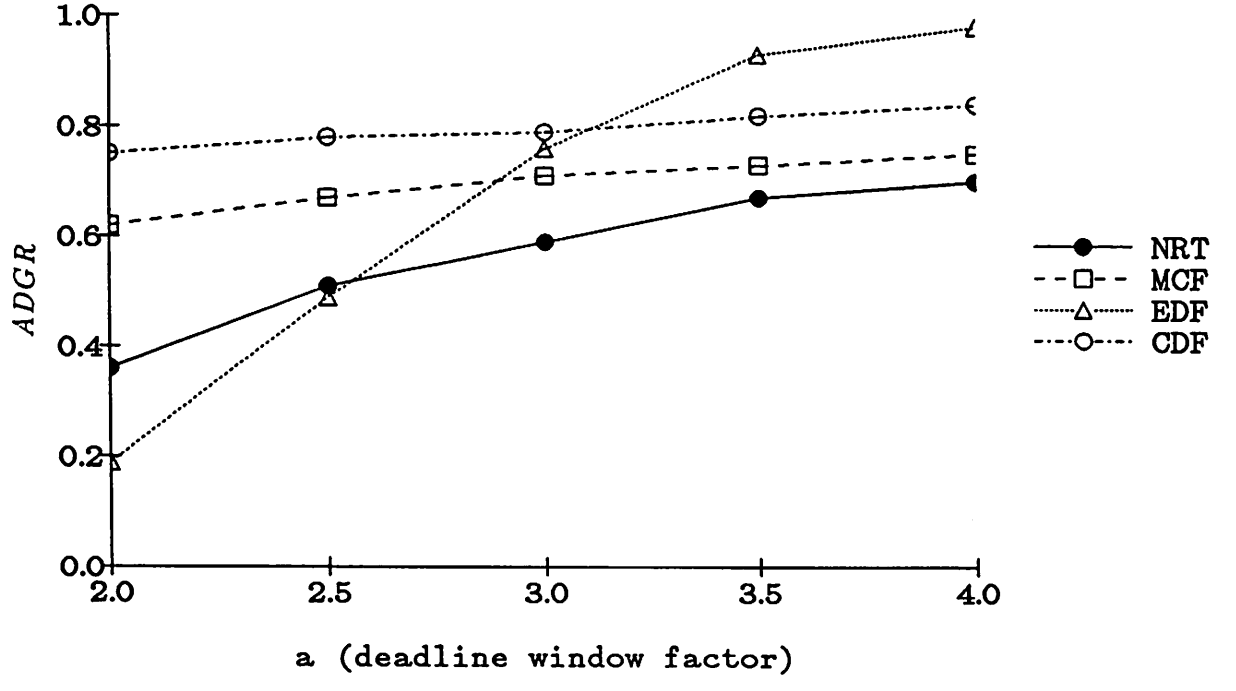

Fig. 9: CPU scheduling with w/r=2/6, T(12,4,10)

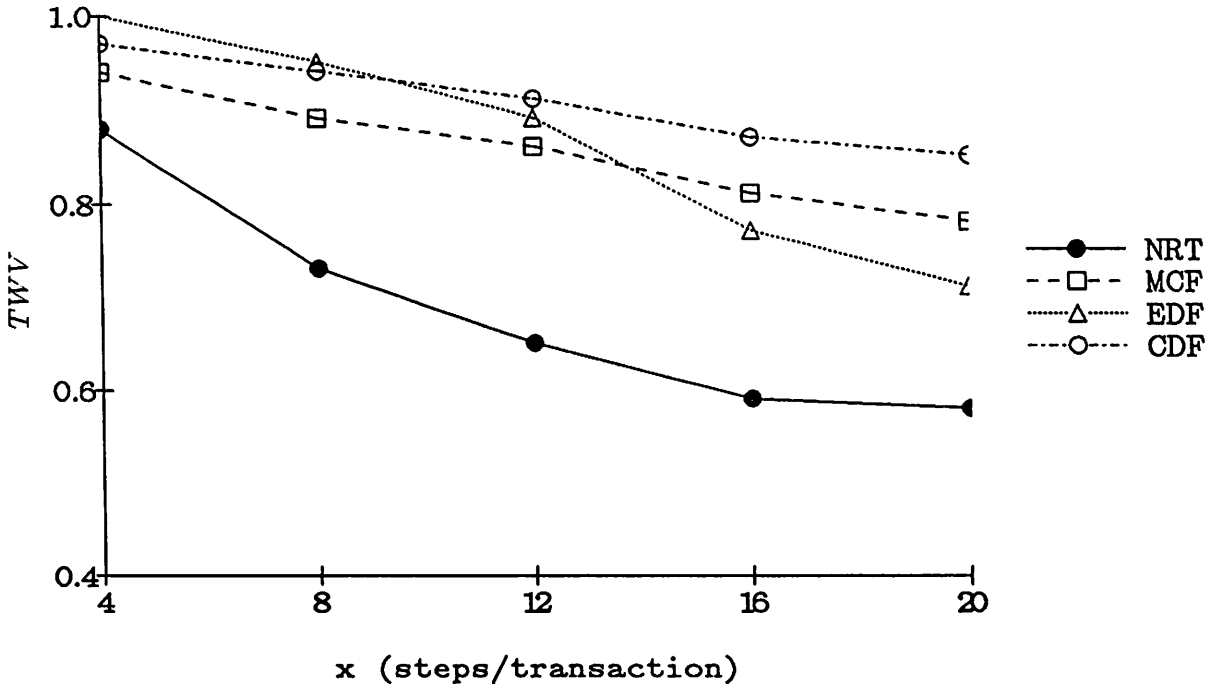

Fig. 10: CPU scheduling with w/r=2/6, T(x,4,10), a=3

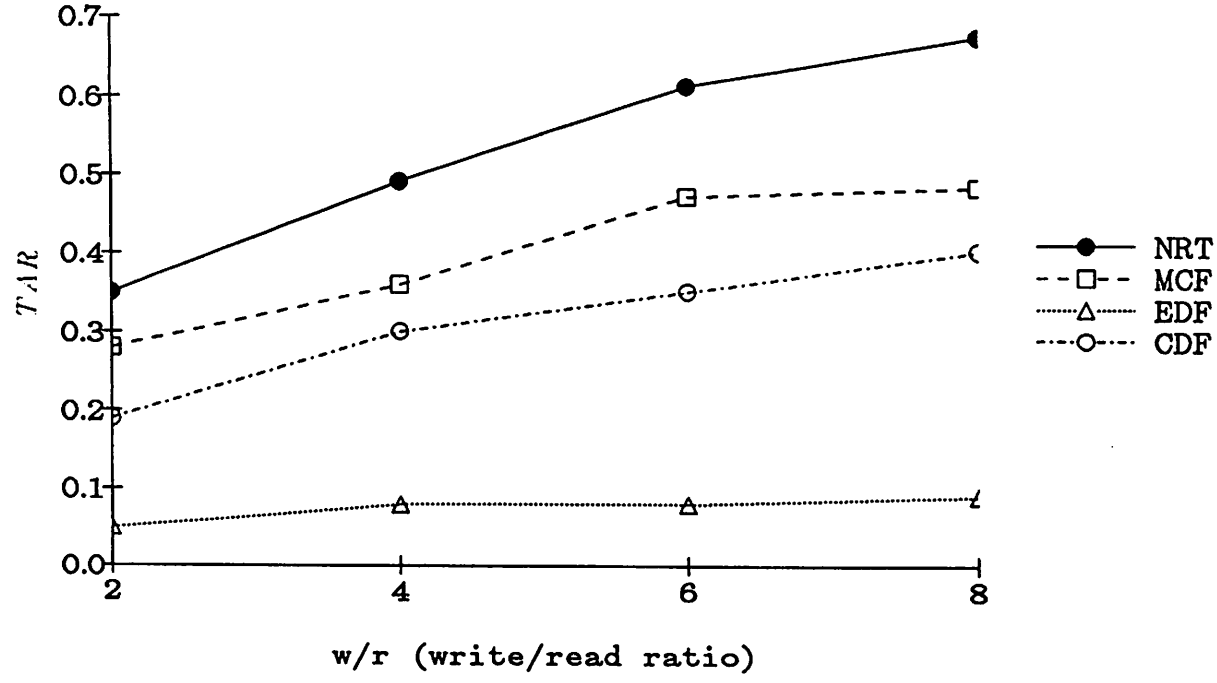

Fig. 11: CPU scheduling with T(12,4,10), a=3

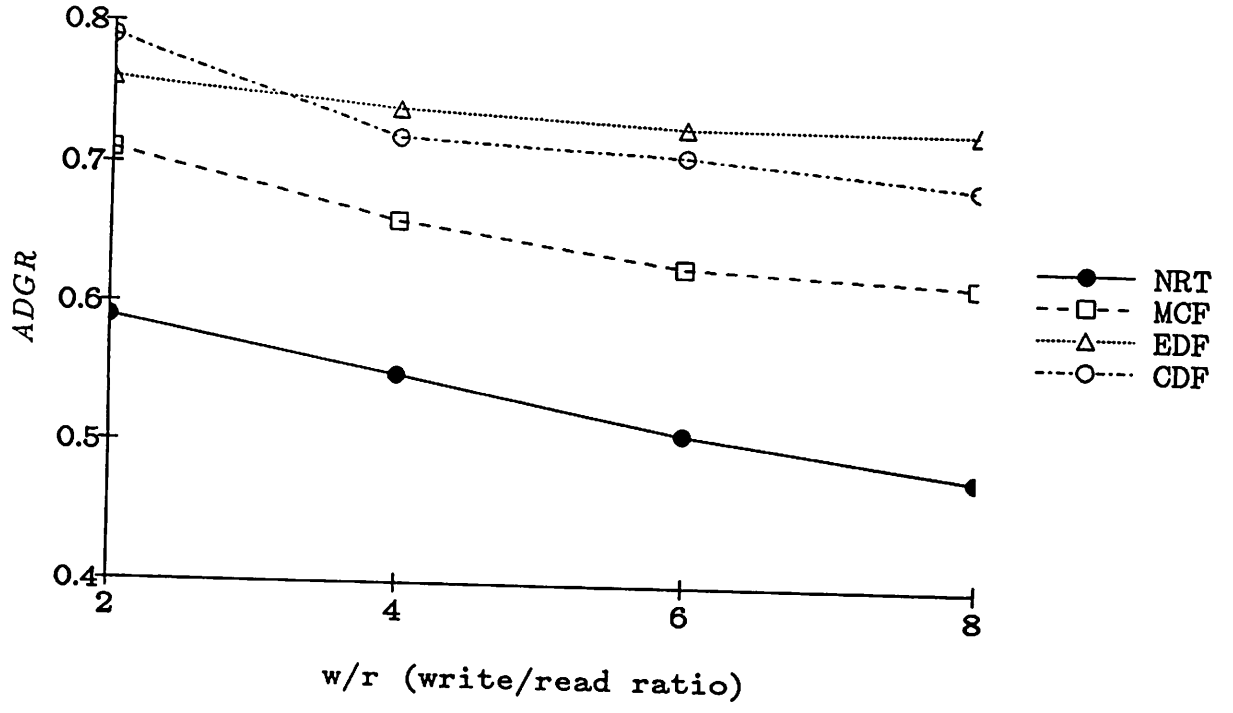

Fig. 12: CPU scheduling with T(12,4,10), a=3

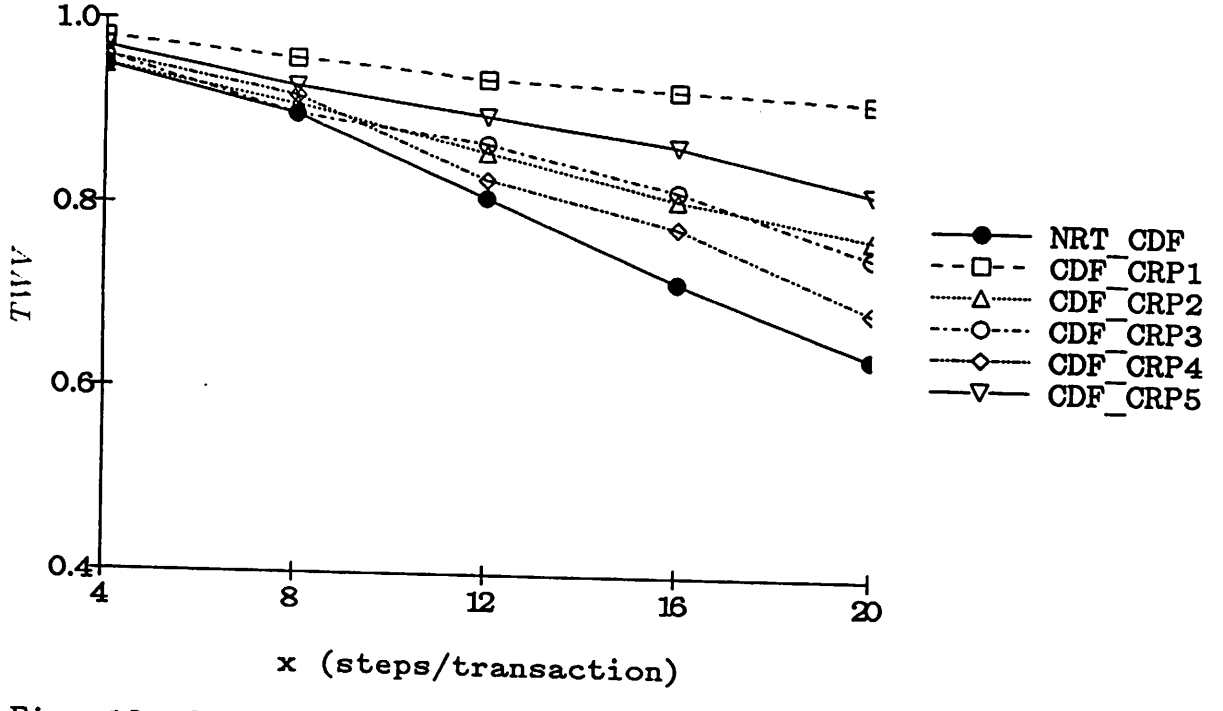

Fig. 13: Conflict resolution with w/r=8/0, T(x,4,10), a=3

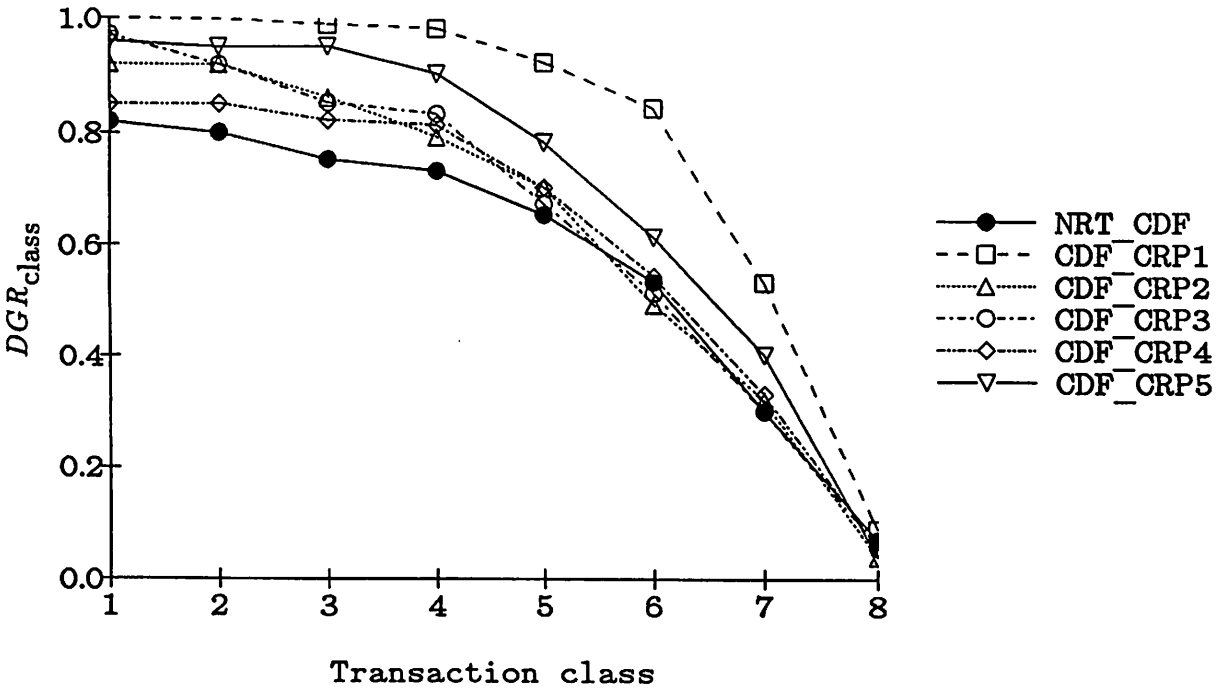

Fig. 14: Conflict resolution, with w/r=8/0, T(16,4,10), a=3

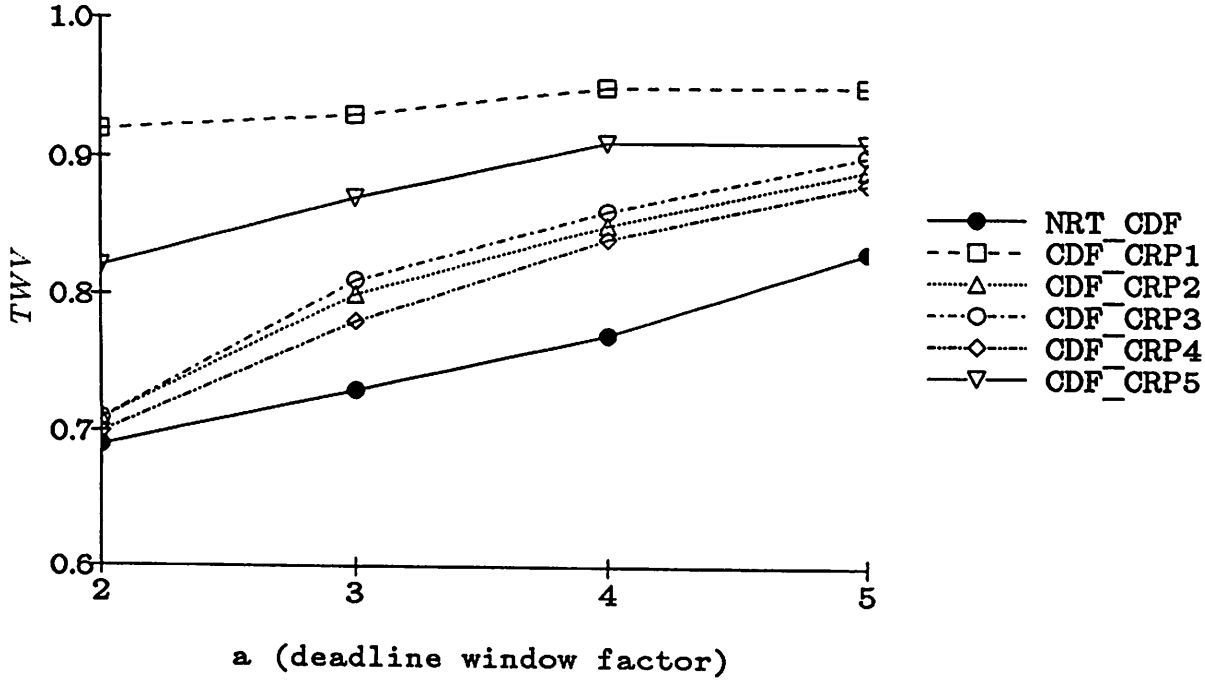

Fig. 15: Conflict resolution with w/r=8/0, T(16,4,10)

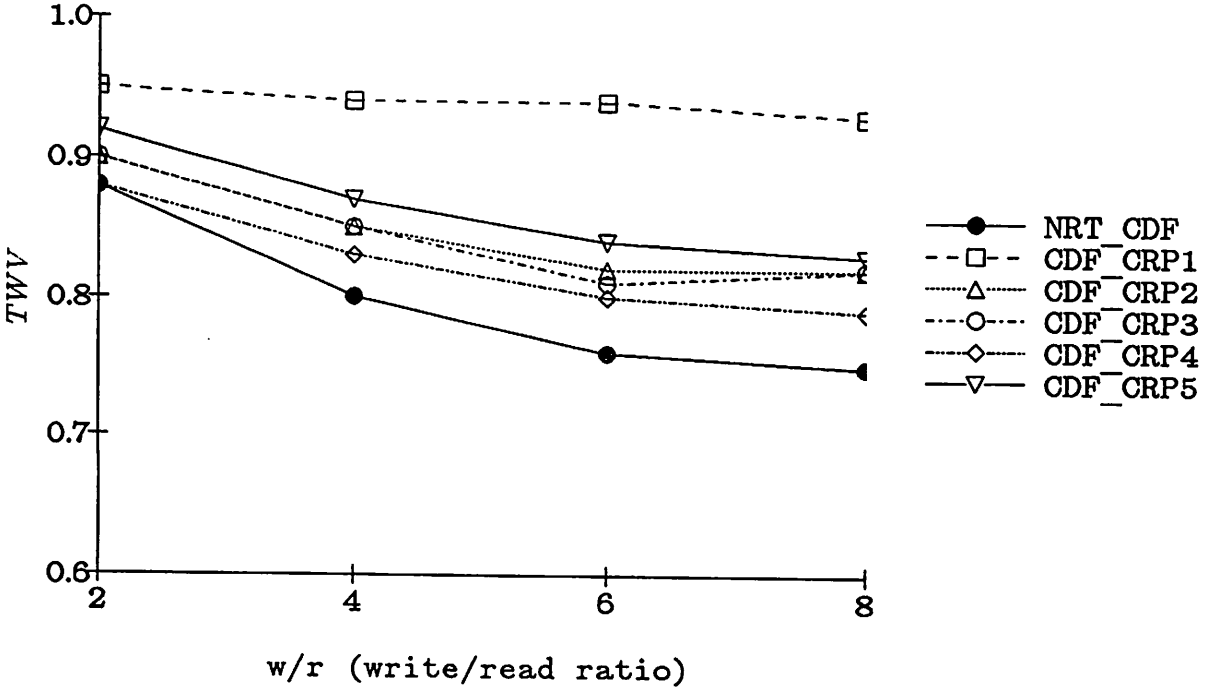

Fig. 16: Conflict resolution with T(16,4,10), a=3

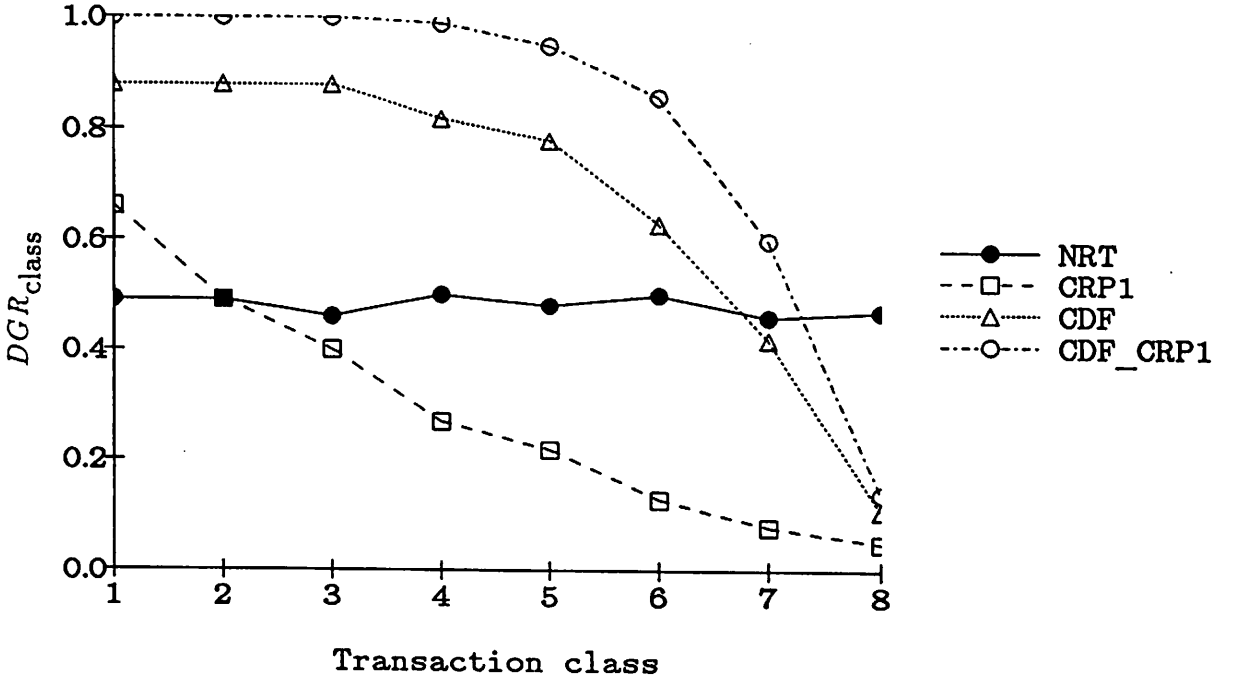

Fig. 17: CPU scheduling vs. conflict resolution, with T(12,4,10), w/r = 8/0

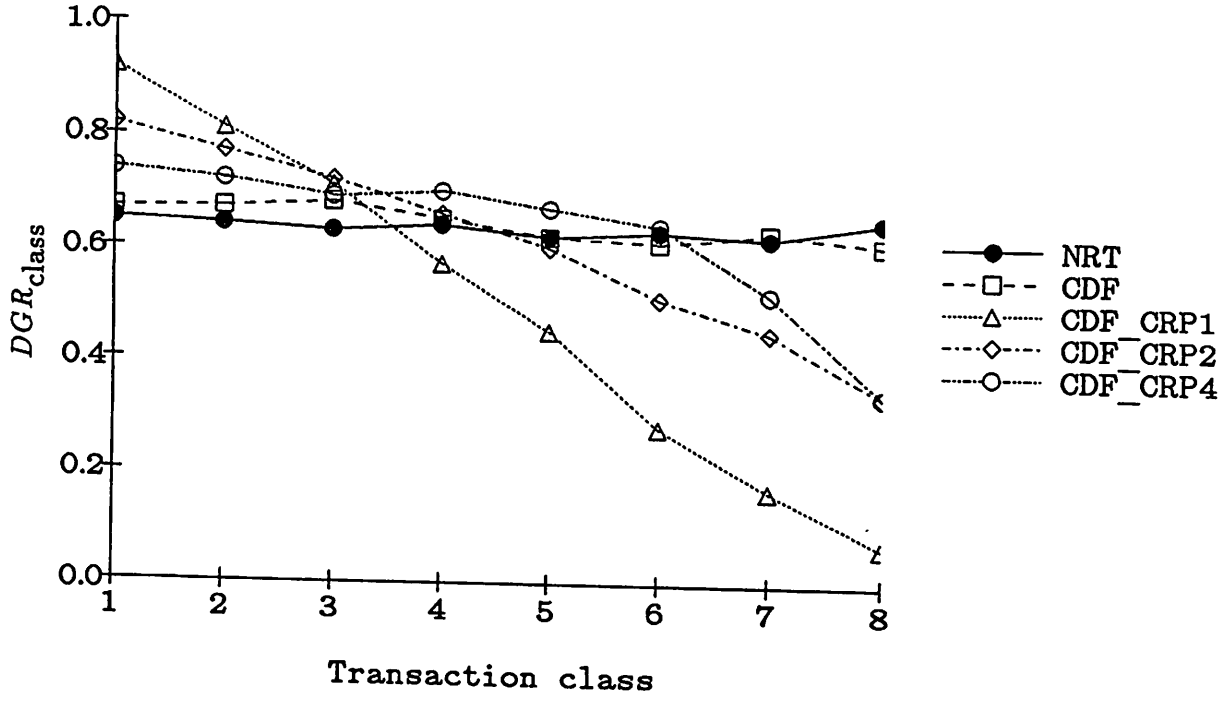

Fig. 18: An I/O bound system, with w/r=8/0, T(12,4,0), a=4

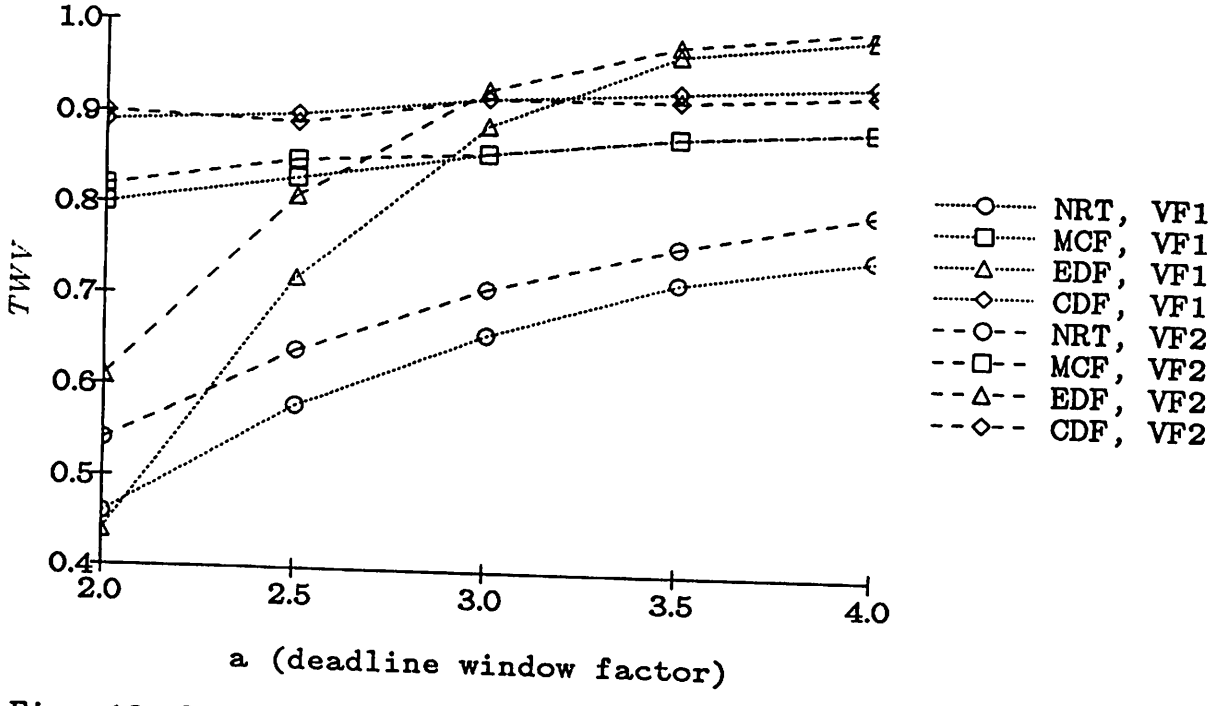

Fig. 19: Value functions with w/r=2/6, T(12,4,10)