# SCHEDULING REAL-TIME TRANSACTIONS:
# A PERFORMANCE EVALUATION

Robert Abbott
Hector Garcia-Molina

CS-TR-146-88

February 1988

# Scheduling Real-time Transactions: a Performance Evaluation

*Robert Abbott and Hector Garcia-Molina*

Department of Computer Science
Princeton University
Princeton, NJ 08544

**Abstract**

Managing transactions with real-time requirements presents many new problems. In this paper we focus on two: How can we schedule transactions with deadlines? How do the real-time constraints affect concurrency control? We describe a new group of algorithms for scheduling real-time transactions which produce serializable schedules. We present a model for scheduling transactions with deadlines on a single processor memory resident database system, and evaluate the scheduliing through detailed simulation experiments.

## 1. Introduction

Transactions in a database system can have real-time constraints. Consider for example program trading, or the use of computer programs to initiate trades in a financial market with little or no human intervention [8]. A financial market (e.g., a stock market) is a complex process whose state is partially captured by variables such as current stock prices, changes in stock prices, volume of trading, trends, and composite indexes. These variables and others can be stored and organized in a database to model a financial market.

One type of process in this system is a sensor/input process which monitors the state of the physical system (i.e. the stock market) and updates the database with new information. If the database is to contain an accurate representation of the current market then this monitoring process must meet certain real-time constraints.

A second type of process is an analysis/output process. In general terms this process reads and analyzes database information in order to respond to a user query or to initiate a trade in the stock market. An example of this is a query to discover the current bid and ask prices of a particular stock. This query may have a real-time response requirement of say 3 seconds. Another example is a program that searches the database for arbitrage opportunities. Arbitrage trading involves finding discrepancies in prices for objects, often on different markets. For example, an ounce of silver might sell for $10 in London and fetch $10.50 in Chicago. Price discrepancies are normally very short-lived and to exploit them one must trade large volumes on a moments notice. Thus the detection and exploitation of these arbitrage opportunities is certainly a real-time task.

Another kind of real-time database system involves threat analysis. For example, a system may consist of a radar to track objects and a computer to perform some image processing and control. A radar signature is collected and compared against a database of signatures of known objects. The data collection and signature look up must be done in real-time.

A *real-time database system* (RTDBS) has many similarities with conventional database management systems and with so called real-time systems. However, a RTDBS lies at the interface and is not quite the same as either type of conventional system. Like a database system, a RTDBS must process *transactions* and guarantee that the database consistency is not violated. However, conventional database systems do not emphasize the notion of time constraints or deadlines for transactions. The performance goal of a system is usually expressed in terms of desired *average* response times rather than constraints for individual transactions. Thus, when the system makes scheduling decisions (e.g., which transaction gets a lock, which transaction is aborted), individual real-time constraints are ignored.

Conventional real-time systems do take into account individual transaction constraints but ignore data consistency problems. Furthermore, real-time systems typically deal with simple transactions (called processes) that have simple and predictable data (or resource) requirements. For a RTDBS we assume that transactions make unpredictable data accesses (by far the more common situation in a database system). This makes the scheduling problem much harder, and this leads to another difference between a conventional real-time system and a RTDBS. The former usually attempts to

ensure that *no* time constraints are violated, i.e., constraints are viewed as "hard." [7] In a RTDBS, on the other hand, it is very difficult to guarantee all time constraints, so we strive to *minimize* the ones that are violated.

In the previous paragraphs we have "defined" what we mean by a RTDBS (our definition will be made more precise in Section 2). However, note that other definitions and assumptions are possible. For instance, one could decide to have hard time constraints and instead minimize the number of data consistency violations. However, we believe that the type of RTDBS that we have sketched better matches the needs of applications like the ones mentioned earlier. For instance, in the financial market example, it is probably best to miss a few good trading opportunities rather than permanently compromise the correctness of the database, or restrict the types of transactions that can be run.

We should at this point make two comments about RTDBS applications. It may be argued that real-time applications do not access databases because they are "too slow." This is a version of the "chicken and the egg" problem. Current database systems have few real-time facilities, and hence cannot provide the service needed for real-time applications. The way to break the cycle is by studying a RTDBS, designing the proper facilities, and evaluating the performance (e.g., what is the price to be paid for serializability?).

It is also important to note that with good real-time facilities, even applications one does not typically consider "real-time" may benefit. For example, consider a banking transaction processing system. In addition to meeting average response time requirements, it may be advantageous to tell the system the urgency of each transaction so it can be processed with the corresponding priority. As a matter of fact, a "real" banking system may already have some of these facilities, but not provided in a coherent fashion by the database management system.

The design and evaluation of a RTDBS presents many new and challenging problems: What is the best data model? What languages can we use to specify real-time constraints? What mechanisms are needed for describing and evaluating triggers (a trigger is an event or a condition in the database that causes some action to occur)? How are transactions scheduled? How do the real-time constraints affect concurrency control?

In this paper we focus on the last two questions. In particular, if several transactions are ready to execute at a given time, which one runs first? If a transaction requests a lock held by another transaction, do we abort the holder if the requester has greater urgency? If transactions can provide an estimate of their running time, can we use it to tell which transaction is closest to missing a deadline and hence should be given higher priority? If we do use run time estimates, what happens if they are incorrect? How are the various strategies affected by the load, the number of database conflicts, and the tightness of the deadlines?

In the next section we summarize our transaction model and basic assumptions. In Section 3 we develop a group of new scheduling/concurrency control algorithms for RTDBS. The performance of the various algorithms has been studied via detailed event driven simulations. Section 4 contains the results as well as some answers to the questions posed in the previous paragraph.

## 2. Model and Assumptions

In this section we describe our basic assumptions and real-time transaction model. We assume that transactions are scheduled dynamically on a *single processor* machine with enough main memory to accommodate the entire database. Our main justification for studying a single processor and a memory resident database is that this reduces the number of parameters (no multiple processors, buses, or disks to model) and makes it easier to understand the scheduling options and their impact on performance. However, we believe that these are reasonable restrictions for a first study of real-time database scheduling. Most existing real-time systems currently hold all their data in memory. Furthermore, since memory prices are steadily dropping, memory sizes are growing and memory residence becomes less of a restriction. Along similar lines, multiple processor real-time systems do exist, but it is important to understand the single processor case first.

A transaction is characterized by its timing constraints and its data and computation requirements. The timing constraints are a release time $r$ and a deadline $d$. A computation requirement is represented by a run time estimate $E$ which approximates the amount of computation required by the transaction. These characteristics, release time, deadline and run time estimate are known to the scheduler when a task enters the system. The last characteristic, data requirements, is not known beforehand but is discovered dynamically as the transaction executes. Our decision to assume knowledge of computation requirements but no knowledge of data requirements is justified because it is easier to estimate the execution time of a transaction than to predict its data access pattern. And in any case, $E$ is simply an estimate that could be wrong or not given at all.

Our goal is to minimize the number of transactions that miss their deadlines, i.e., that finish after time $d$. If transactions can miss their deadlines, one must address the question of what happens to transactions that have already missed their deadlines but have not finished yet.

There are two alternatives. One is to assume that a transaction that has missed its deadline, i.e., is tardy, is worthless and can be aborted. This may be reasonable in our arbitrage example. Suppose that a transaction is submitted to buy and sell silver by 11:00am. If the deadline is missed, it may be best not to perform the operation at all; after all the conditions that triggered the decision to go ahead may have changed. The user who submitted the transaction may wish to reconsider the operation.

A second option is to assume that all transactions must be completed eventually, regardless of whether they are tardy or not. This may be the correct mode of operation in, say, a banking system where customers would rather do the transaction late than not at all. (Of course, the user may on his own decide to abort his transaction, but this is another matter.) If tardy transactions must be executed, there is still the question of their priority. Tardy transactions could receive higher and higher urgency as their tardiness increases. On the other hand, since they already missed the deadline anyway, they may simply be postponed to a later, more convenient time (e.g., execute at night).

In this paper we will study both cases, when tardy transactions must be completed and when they can be aborted. If they must complete, we will assume that their priority increases as the tardiness increases (they are not put off). (Incidentally, [1] discusses a more detailed deadline model where users can specify how the "value" of a transaction changes over time, both as the deadline approaches and passes.)

We assume that transaction executions must be serializable [2]. For most applications we believe that it is desirable to maintain database consistency. It is possible to maintain consistency without serializable schedules but this requires more specific information about the kinds of transactions being executed [3]. Since we have assumed very little knowledge about transactions, serializability is the best way to achieve consistency.

Finally, we assume that serializability is enforced by using a locking protocol. Our purpose is not to do a comparative study of concurrency mechanisms. Instead we have chosen a well-understood and widely-used mechanism and explored the different ways that transactions can be scheduled using this mechanism. Of course, it is conceivable that some other algorithm, like an optimistic protocol, may be better for a RTDBS, but this will have to be addressed by further research.

## 3. Some Scheduling Algorithms

Our scheduling algorithms have three components: a policy to determine which tasks are eligible for service, a policy for assigning priorities to tasks and a concurrency control mechanism. The concurrency control mechanism can be thought of as a policy for resolving conflicts between two (or more) transactions that want to lock the same data object. Some concurrency control mechanisms permit deadlocks to occur. For these a deadlock detection and resolution mechanism is needed.

Each component may use only some of the available information about a transaction. In particular we distinguish between policies which do not make use of $E$, the run time estimate, and those that do. A goal of our research is to understand how the accuracy of the run time estimate affects the algorithms that use it.

### 3.1. Determining Eligibility

The scheduler is invoked whenever a transaction terminates and, for preemptive scheduling, whenever a new transaction arrives. The concurrency control mechanism is invoked to resolve lock conflicts whenever one occurs. The first action of the scheduler is to divide the set of ready transactions into two lists, those that are eligible for scheduling and those that are not. All ineligible transactions are aborted and their locks released. Eligible transactions remain in the system and are eligible for service. If a transaction never becomes ineligible then it is eventually executed. Finally, an eligible transaction may become ineligible but an ineligible transaction cannot become eligible. We consider three different policies for determining eligibility.

*All Eligible.* All jobs are eligible for service. This means that no job is unilaterally aborted.

*Not Tardy.* All jobs which currently are not tardy are eligible for service. Jobs that have already missed their deadlines are aborted.

*Feasible Deadlines.* All jobs with feasible deadlines are eligible for service. A transaction $T$ has a feasible deadline if $t + E - P \leq d$ where $P$ is the amount of service time that $T$ has received. In other words, based on the run time estimate there is enough time to complete the transaction before its deadline. Jobs with infeasible deadlines are aborted. Note that this policy uses $E$, the run time estimate.

### 3.2. Assigning Priorities

There are many ways to assign priorities to real-time tasks. [5,6] We have studied three.

*First Come First Serve.* This policy assigns the highest priority to the transaction with the earliest release time. If release times equal arrival times then we have the traditional version of FCFS.

The primary weakness of FCFS is that it does not make use of deadline information. FCFS will discriminate against a newly arrived task with an urgent deadline

in favor of an older task which may not have such an urgent deadline. This is not desirable for real-time systems.

*Earliest Deadline*. The transaction with the earliest deadline has the highest priority. A major weakness of this policy is that it can assign the highest priority to a task that has already missed or is about to miss its deadline. One way to solve this problem is to use the eligibility policy Not Tardy or Feasible Deadlines to screen out transactions that have missed or are about to miss their deadlines.

*Least Slack*. For a transaction $T$ we define a slack time $S = d - (t + E - P)$. The slack time is an estimate of how long we can delay the execution of $T$ and still meet its deadline. If $S \geq 0$ then we expect that if $T$ is executed without interruption then it will finish at or before its deadline. A negative slack time is an estimate that it is impossible to make the deadline. A negative slack time results either when a transaction has already missed its deadline or when we estimate that it cannot meet its deadline.

Least Slack is similar to Earliest Deadline in that it can assign high priorities to tasks which have missed or about to miss their deadlines. Again we can use the eligibility policies Not Tardy and Feasible Deadlines to ameliorate this problem. Using Not Tardy disallows jobs with negative slack times that result from missed deadlines. Feasible deadlines disallows all jobs with negative slack times. Least Slack is very different from Earliest Deadline in that the priority of a task depends on how much service time it has received. Thus restarting a transaction changes its priority. We return to this issue in the next section.

### 3.3. Concurrency Control

If transactions are executed concurrently then we need a concurrency control mechanism to order the updates to the database so that the final schedule is a serializable one. We now discuss three possible solutions. Once again we distinguish between policies which make use of the runtime estimate $E$ and those that do not.

*Serial Execution*. The simplest way to resolve conflicts is not to let them happen in the first place. The way to achieve this and maintain database consistency is to execute transactions serially and without preemption. Once the highest priority transaction gains the processor it runs to completion. This method is efficient only if no transactions are forced to wait for data transfers from disk to memory and back. If the entire database is memory resident then this may be a good method for maintaining serializability at low cost. A drawback of this method is that an arriving task with an urgent deadline must wait until the current task (possibly one with a less urgent deadline and a large remaining computation) completes.

It may be possible to improve performance by executing transactions concurrently. If this is done then we can expect conflicts to occur. In the following discussions let $T_H$ denote a transaction which holds a lock on data object $X$. Let $T_R$ be a transaction which is requesting a lock on $X$. We now present two methods to resolve conflicting transactions.

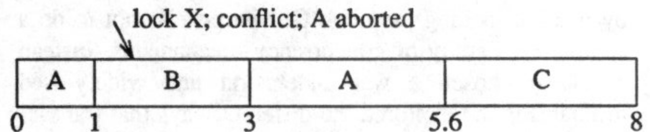
*High Priority*. The idea of this policy is to resolve a conflict in favor of the transaction with the higher priority. The favored transaction, the winner of the conflict, gets the resources, both data locks and the processor, that it needs to proceed. The loser of the conflict relinquishes control of any resources that are needed by the winner. We implement this policy by comparing transaction priorities at the time of the conflict. If the priority of $T_R$ is greater than the priority of $T_H$ then we abort $T_H$ thereby freeing the lock for $T_R$. $T_R$ can resume processing; $T_H$ is rolled back and scheduled for restart. If the priority of $T_R$ is less than or equal to the priority of $T_H$ then we let $T_H$ keep its lock and $T_R$ blocks to wait for $T_H$ to finish and release its locks.

Consider the following set of transactions with release time $r$, deadline $d$, runtime estimate $E$ and data requirements.

| Transaction | $r$ | $E$ | $d$ | updates |
|---|---|---|---|---|
| A | 0 | 2.6 | 5 | X |
| B | 1 | 2 | 4 | X |
| C | 2 | 2.4 | 8 | Y |

Example 1.

Note that transactions $A$ and $B$ both update item $X$. Therefore these transactions must be serialized. If we use Earliest Deadline to assign priority and High Priority to resolve conflicts then the following schedule is produced. (It assumes that estimates are perfect and ignores the time required to make scheduling decesions or rollback transactions.)




In this schedule, $A$ runs in the first time unit during which it acquires a lock on item $X$. Transaction $B$ gains the processor at time 1 (it has an earlier deadline) and requests a lock on item $X$. Thus a conflict is created which is resolved by rolling back $A$ thereby freeing the lock on $X$. Transaction $B$ continues processing and completes before its deadline. After $B$ completes at time 3, $A$ is restarted. Transactions $B$ and $C$ meet their deadlines but $A$ is tardy.

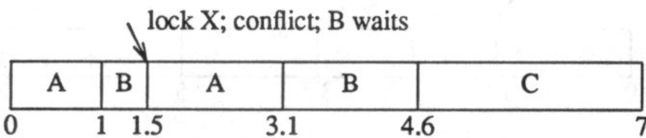
An interesting problem arises when we use Least Slack to prioritize transactions. Recall that under this policy, a transaction's priority depends on the amount of service time that it has received. Rolling back a transaction to its beginning reduces its effective service time to 0 and raises its priority under the Least Slack policy. Thus a transaction $T_H$, which loses a conflict and is aborted to allow a higher priority transaction $T_R$ to proceed, can have a higher priority than $T_R$ immediately after the abort. The next time the scheduler is invoked, $T_R$ will be preempted by $T_H$. $T_H$ may again conflict with $T_R$ initiating another abort and rollback.

Our solution to this problem is to compare the priority of $T_R$ against that of $T_H$ assuming that $T_H$ were aborted. We can write this new conflict resolution policy as follows:

*High Priority Conflict Resolution Policy.*

**if** $p(T_H) < p(T_R)$ **and** $p(T_H^A) < p(T_R)$
**then**
  Abort $T_H$
  Run $T_R$
**else**
  $T_R$ blocks
  Run $T_H$

The function $p(T_H)$ is the priority of $T_H$ and $p(T_H^A)$ is the priority of $T_H$ were it to be aborted. Using this revised policy, the schedule produced by Least Slack is as follows:




Now when the conflict occurs at time 1.5, $B$ has a slack of 1 but $A$ were it aborted has a slack of .9. So $A$ is not aborted but is assigned to the processor while $B$ waits for $A$ to finish. Transaction $B$ is unblocked when $A$ finishes at time 3.1. Transactions $A$ and $C$ meet their deadlines but $B$ is tardy.

For FCFS and Earliest Deadline policies, $p(T_H) = p(T_H^A)$, so it does not matter if we use the original High Priority resolution rule or the modified one above. Since the modified rule is clearly superior for Least Slack priority assignment, we will use it for our performance evaluations.

We make two final observations about High Priority. First, if $T_R$ waits for $T_H$ it is possible that $T_H$ is not a ready transaction. That is, $T_H$ may be blocked waiting for a lock held by another transaction $T_J$. In this case the conflict resolving mechanism is applied again except that $T_H$ is now the lock requesting transaction and $T_J$ is the lock holder. And second, because transactions wait for for locks, deadlock is a possibility. Deadlock detection can be done using one of the standard algorithms [4]. Victim selection, however, should be done with consideration of the time constraints of the tasks involved in the deadlock.

*Conditional Restart*

Sometimes High Priority may be too conservative. Let us assume that we have chosen the first branch of the algorithm, i.e., $T_R$ has a greater priority than $T_H$ and $T_H^A$. We would like to avoid aborting $T_H$ because we lose all the service time that it has already consumed. We can be a little cleverer by using a conditional restart policy to resolve conflicts. The idea here is to estimate if $T_H$, the transaction holding the lock, can be finished within the amount of time that $T_R$, the lock requester, can afford to wait. Let $S_R$ be the slack of $T_R$ and let $E_H - P_H$ be the estimated remaining time of $T_H$. If $S_R \geq E_H - P_H$ then we estimate that $T_H$ can finish within the slack of $T_R$. If so then we let $T_H$ proceed to completion, release its locks and then let $T_R$ execute. This saves us from restarting $T_H$. If $T_H$ cannot be finished in the slack time of $T_R$ then we restart $T_H$ and run $T_R$ (as in the previous algorithm). This modification yields the following algorithm:

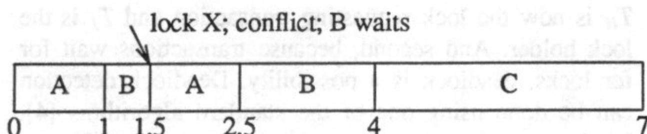
*Conditional Restart Conflict Resolution Policy.*

**if** $p(T_H) < p(T_R)$ **and** $p(T_H^A) < p(T_R)$
**then**
  **if** $E_H - P_H \leq S_R$
  **then**
    $T_R$ blocks
    Run $T_H$
  **else**
    Abort $T_H$
    Run $T_R$
**else**
  $T_R$ blocks
  Run $T_H$

The following example illustrates the idea of this policy. Again we use Earliest Deadline to assign priority. (We continue to assume that estimates are exact and scheduling decisions and rollbacks are done instantly.)

| Transaction | $r$ | $E$ | $d$ | updates |
|---|---|---|---|---|
| A | 0 | 2 | 5 | X |
| B | 1 | 2 | 4 | X |
| C | 2 | 3 | 8 | Y |

Example 2.




A conflict occurs when $B$ requests a lock on $X$ at time 1.5. The algorithm calculates the slack time for $B$ as $S = 4 - 1.5 - 1.5 = 1$. This equals exactly the remaining run time for $A$. Therefore $B$ waits for $A$ to finish and release its locks. $A$ finishes at time 2.5 and $B$, with 1.5 time units left to compute, regains the processor and completes at time 4. All transactions meet their deadlines.

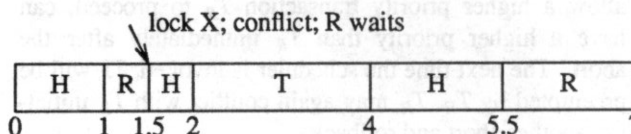
In this example the runtime estimate was an excellent approximation of the actual computation time of $A$. If the actual computation time for $A$ was a little longer than the runtime estimate then $B$ would miss its deadline. Thus the ability of this algorithm to successfully exploit slack time information in order to avoid aborting and restarting transactions is dependent on the accuracy of the runtime estimates. Note that Conditional Restart allows transactions to wait for locks, thus deadlock is a possibility.

As described the Conditional Restart policy has two problems. First, we assumed that only one job $T_H$ has to run before $T_R$. In fact $T_H$ may be waiting for a lock held by another transaction $T_J$ and we must decide how to resolve the conflict between $T_H$ and $T_J$. More generally, let $D = T_1, T_2, ..., T_n$ be a chain of tasks such that $T_1$ is waiting for a lock held by $T_2$ which is waiting for a lock held by $T_3$, ..., which is waiting for a lock held by $T_n$. (We assume that this chain is deadlock free but we make no assumptions about the relative priorities of the tasks in the chain.) Let $T_0$ with slack time $S$ be the currently executing task and say it requests a lock held by $T_1$. The idea of the conditional restart algorithm is to compute the maximum number of tasks in the chain which can be completed in the slack of $T_0$. Because of the serializability constraint we assume the $T_i$ must be either completed or aborted before $T_{i-1}$ can continue. Let $j$ be the greatest integer such that $\sum_{i=1}^{j}(E_i - P_i) \le S$. We execute in order the tasks $T_j, T_{j-1}, ..., T_1$. If $j<n$ then we must first abort $T_{j+1}$ in order to free the lock for $T_j$. When $T_1$ completes the lock is released for $T_0$.

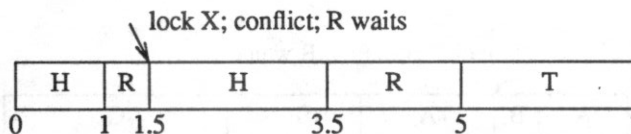
The second problem with conditional abort is illustrated by the following set of tasks and the schedule produced by using Earliest Deadline and Conditional Restart.

| Transaction | $r$ | $E$ | $d$ | updates |
|---|---|---|---|---|
| H | 0 | 3 | 12 | X |
| R | 1 | 2 | 6 | X |
| T | 2 | 2 | 7 | Y |

Example 3.




At time 1.5 the scheduler decides to run $H$ because it can be completed within the slack time of $R$. Transaction $H$ only runs for a short time before it is preempted by an arriving transaction $T$ with an earlier deadline and therefore a higher priority than $H$. ($R$ is not considered in this priority assignment because it is not a ready task.) Scheduling $T$ and $H$ before $R$ causes $R$ to miss its deadline. The problem lies in the assumption that $H$ would not be preempted while it was executing during the slack time of $R$. The way we correct this problem is by letting $R$ remain in the ready queue and be considered for scheduling. Since $T$ has a later deadline than $R$ then $R$ will be scheduled. When $R$ begins execution it requests a lock that is still held by $H$. If $H$ can still be completed within the slack of $R$ then it is executed while $R$ waits. Thus $H$ ultimately regains the processor and is not preempted by $T$. This revised algorithm is illustrated by the following schedule.




The examples we have used to illustrate the different algorithms are greatly simplified. They were presented to motivate the algorithms, not to prove that one algorithm is better than another. For instance, in reality transactions may update several items or none at all (i.e. read-only), and this will obviously affect the performance of the algorithms. In the next section we discuss a detailed simulation model that can help us to compare the various scheduling and concurrency control options.

## 4. Simulation Results

To study the algorithms we have built a program to simulate a RTDBS. To focus on the issues of concurrency control and scheduling we have simulated a single processor, memory resident database system.

Three parameters control the configuration of the database system. The first *Arr_rate* is the average arrival rate of new transactions entering the system. The second,

*DB_size* controls the number of objects in the database. The third, *Rebort*, controls the amount of time needed to abort or restart a transaction. Aborting a transaction consists of rolling it back and removing it from the system. The transaction is not executed. When a transaction is restarted it is rolled back and placed again in the ready queue. Aborts are generated by eligibility screening, restarts result from lock conflicts. The program does not explicitly account for time needed to execute the lock manager, conflict manager, and deadlock detection manager. These routines are executed on a per data object basis and we assume that the costs of these calls are included in the variable that states how much CPU time is needed per object that a transaction accesses. Context switching and the time to execute the scheduler is also ignored.

Transactions enter the system with exponentially distributed inter-arrival times and they are ready to execute when they enter the system (i.e., release time equals arrival time). The number of objects updated by a transaction (at least one, i.e., there are no read-only transactions) is chosen from a normal distribution and the actual database items are chosen uniformly from the database. A transaction has an execution profile which alternates lock requests with equal size chunks of computation, one for each object accessed. Thus the total computation time is directly related to the number of items accessed. The accuracy of a transaction's runtime estimate $E$ with respect to to the actual computation time $C$ is controlled by the parameter *Run_err* and is computed as follows: $E = C * (1 + Run_err)$. Note $E = 0$ when $Run_err = -1$, $E = C$ when $Run_err = 0$, and $E > C$ when $Run_err > 0$. The assignment of a deadline is controlled by two parameters *Min_slack* and *Max_slack* which set a lower and upper bound respectively on a transaction's slack time. A deadline is assigned by choosing a slack time uniformly from the range specified by the bounds.

In the following sections we discuss some of the results of four different experiments that we performed. Due to space considerations we cannot present all our results but have selected the graphs which best illustrate the differences and performance of the algorithms. For example, we have omitted the results of an experiment that varied the size of the database, and thus the number of conflicts, because they only confirm and not increase the knowledge yielded by other experiments. Each experiment exercises a different parameter: *Arr_rate*, *Run_err*, *Max_slack*, and *Rebort*. For each experiment we ran the simulation with the same parameters for 20 different random number seeds. Each run continued until at least 500 transactions were executed. For each algorithm tested, numerous performance statistics were collected and averaged over the 20 runs. In particular we measured the percentage of transactions which missed their deadlines, the number of restarts caused by lock conflicts and the overall throughput. The percentage of missed deadlines is calculated with the following equation: $\%missed = 100 * \frac{tardy\ jobs + aborts}{jobs\ processed}$. A job is processed if either it executes completely or it is aborted. It is these averages and 95% confidence intervals that are plotted in the following figures.

In this study we have included tardy jobs and aborted jobs together in the *%missed* metric. For some applications it may be useful to describe a separate metric for aborted jobs as they represent tasks which were never completed and as such may be more serious than simply tardy jobs. Another intersting metric is mean tardy time for transactions. For reasons of space we do not study these other metrics here. Note that we are not particularly interested in transaction response times as conventional performance evaluations of concurrency control mechanisms are. The reason is that response time is not critical as long as a transaction meets its deadline. We are interested in learning how the various strategies are affected by load, the number of database conflicts and the tightness of deadlines. Also, for the algorithms which use $E$, we are interested in learning how the accuracy of the run time estimate affects performance.

For many of the experiments the base values for parameters determining system configuration and transaction characteristics are shown in Table 1.

| Parameter | Value |
|---|---|
| Arrival rate (jobs/sec) | 18 |
| Database size | 200 |
| Restart/abort cost (ms) | 0 |
| Updates per transaction (mean) | 15 |
| Computation/update (ms) | 3 |
| Runtime estimate error | 0 |
| Min slack as fraction of total runtime | 0.5 |
| Max slack as fraction of total runtime | 5.0 |

Table 1. Base parameter values.

These values are not meant to model a specific real-time application but were chosen as reasonable values within a wide range of possible values. In particular we wanted transactions to access a relatively large fraction of the database (7.5% on average) so that conflicts would occur frequently. The high conflict rate allows the concurrency control meachanism to play a significant role in scheduling performance. We chose the arrival rate so that the corresponding computational load (an average 0.81 seconds of computation arrive per second) is high enough to test the algorithms. It is more interesting to test the algorithms in a heavily loaded rather than lightly loaded system. (We return to this issue in the conclusions section.)

Section 3 proposed three different methods each for determining eligibility, assigning priority and managing concurrency. Taking the cross product yields 27 different algorithms. However, in our model, using FCFS scheduling with High Priority or Conditional Restart is equivalent to FCFS with Serial Execution. This eliminates six algorithms, leaving 21. Table 2 summarizes the methods of Section 3 and provides the abbreviations that we will use when referring to them.

| | |
|---|---|
| Eligibility | AE - All Eligible<br>NT - Not Tardy<br>FD - Feasible Deadlines |
| Priority | FCFS - First Come First Serve<br>ED - Earliest Deadline<br>LS - Least Slack |
| Concurrency | SE - Serial Execution<br>HP - High Priority<br>CR - Conditional Restart |

Table 2. Summary of scheduling policies.

**Arrival rate experiment**

In this experiment we varied the arrival rate from 4 jobs/sec to 22 jobs/sec in increments of 2. This corresponds to a range in load of 0.18 to 0.99 seconds of computation arriving per second.

Our simulation experiments show that the eligibility tests NT and FD substantially reduce the number of deadlines missed compared with the AE policy. Aborting a few late transactions helps all other jobs meet their deadlines. This is illustrated in Figure 1 which shows the three eligibility policies for FCFS scheduling. All three algorithms perform equally well when the arrival rate is low but when the load is higher, NT and FD yield a 40-50% decrease in % missed deadlines over AE. The improvement is due entirely to the eligibility policy since the concurrency control used is SE. This same behavior holds true for the other priority assignment policies as well.

Scheduling transactions concurrently is better than serial execution because it allows transactions with more urgent deadlines to preempt transactions with less urgent deadlines. This is true for the ED scheduling policy. Figure 2 shows that CR and HP are both better than SE and CR is better than HP.

The situation is somewhat different when LS is used to assign priority. Figure 3 shows that at low arrival rates, the concurrent versions perform better than SE. However, at higher arrival rates SE is comparable to HP and CR, the concurrent versions. Under high loads the time lost to restarts that result from lock conflicts causes the concurrent versions to lose their performance edge over SE.

The relative performance of the priority policies is affected by the type of concurrency allowed. Under SE, ED and LS perform comparably, Figure 4. When the load is small they both perform better than FCFS but when the load is high all three algorithms perform the same. The reason for this is that under high load all algorithms start to fall behind. It does not pay to make smart scheduling decisions (ED, LS), since transactions with the earliest deadlines (or least slack) are likely to miss their deadlines anyway.

When non-serial execution is allowed the results are different, Figure 5. Under CR, ED is clearly the superior policy for priority assignment. It performs better than LS at both low and high load levels. At the highest load LS and FCFS are comparable and ED is better than both.

Figure 6 plots restarts against arrival rate for ED, HP. The number of restarts climbs steeply and is similar for all policies up to an arrival rate of 12. After this point the number of restarts declines sharply for AE , declines slightly for NT and flattens out for FD. The reason for the sharp decline for AE is that when the arrival rate is high many deadlines are being missed and the system falls behind. It is less likely that an arriving transaction will have an earlier deadline than the currently executing job. Thus fewer jobs are preempted and there are fewer opportunities for restarts.

Figure 7 shows that throughput is highest for AE, it declines with NT and FD. Since we are using serial execution no job is aborted once it has started, so the decreases in throughput are not due to wasted computation time. The throughput declines because the algorithms choose not to execute jobs that have missed or are about to miss their deadlines. This is true even if there is only one job (say a tardy one) in the ready queue. Ignoring this job empties the ready queue and idles the CPU until the next arrival. With increased idle time, the throughput drops.

**Run time estimate experiment**

In this experiment we varied *Run_err* from -1 to 0.8 by increments of 0.2. Recall that when $Run_err = -1$, $E = 0$ and when $Run_err = 0.8$, $E = 1.8 * C$. The other parameters had the values in Table 1. The run time estimate can affect each component of a scheduling algorithm: of the eligibility policies, FD, of the priority assignment, LS, and of the concurrency control policies, CR. We would expect the accuracy of the estimate to have a large effect on the policy FD. The eligibility policy is responsible for aborting transactions and since aborted transactions are counted as having missed their deadlines, the policy directly affects the performance.

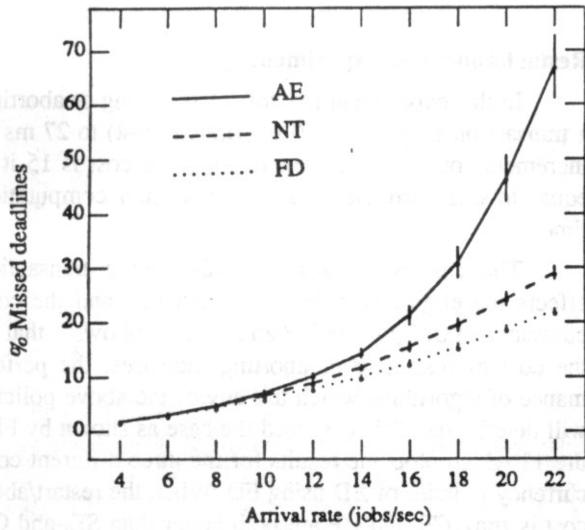



Figure 1. FCFS, SE.

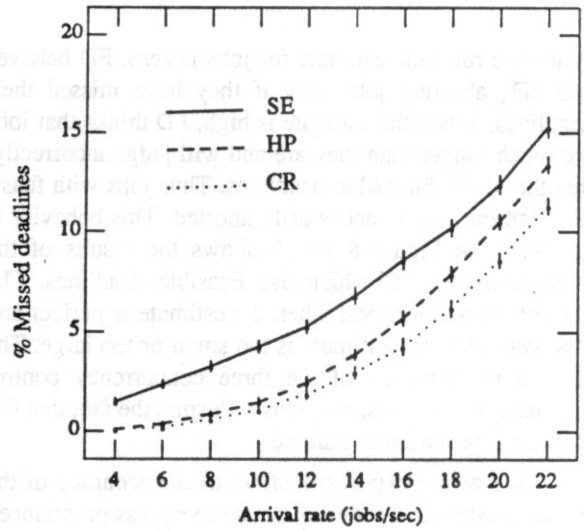



Figure 2. ED, FD.

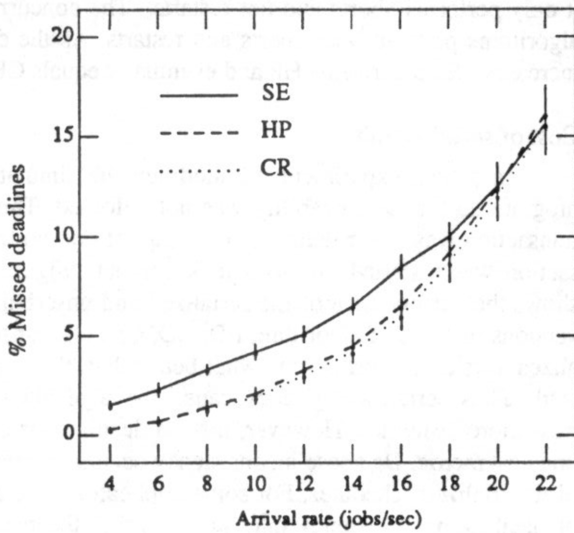



Figure 3. LS, FD.

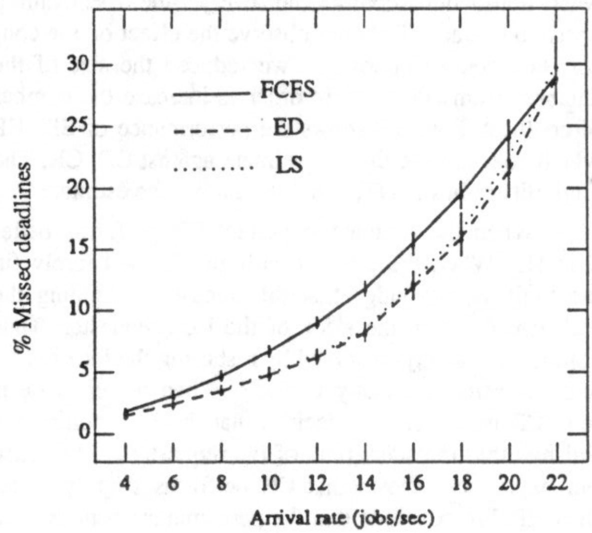



Figure 4. FCFS vs ED vs LS, NT, SE.

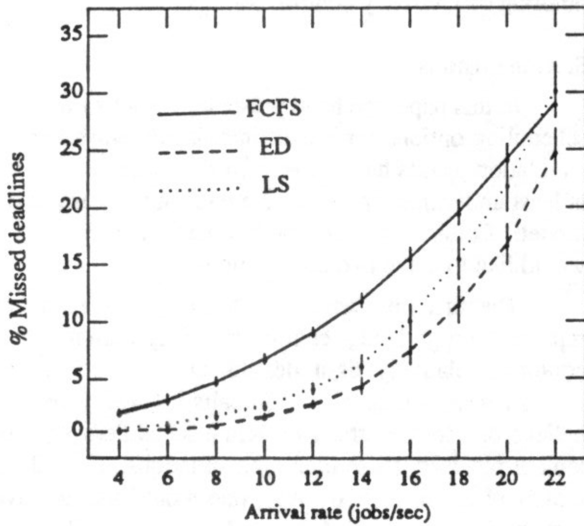



Figure 5. FCFS vs ED vs LS, NT, CR.

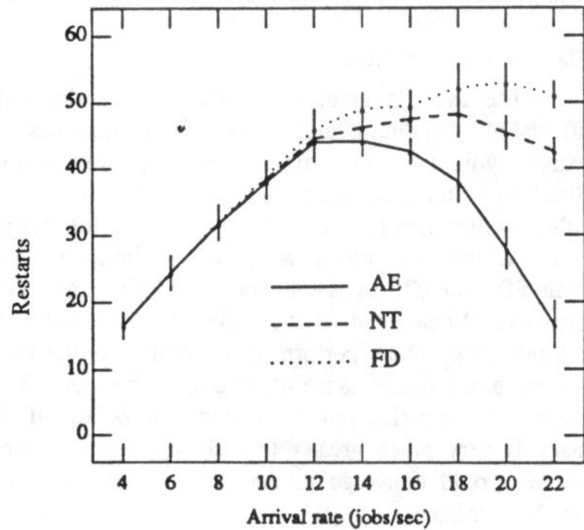



Figure 6. ED, HP.

When the run time estimate for jobs is zero, FD behaves like NT, aborting jobs only if they have missed their deadlines. When the estimate is high, FD thinks that jobs are much longer than they are and will judge incorrectly, that they have infeasible deadlines. Thus jobs with feasible deadlines are unnecessarily aborted. This behavior is confirmed in Figure 8 which shows the results of the three forms of ED which use Feasible deadlines. The algorithms perform best when the estimate is perfect and less well when the estimate is too small or too large. The relative performance of the three concurrency control policies remains the same. This is despite the fact that CR also uses the run time estimate.

We would expect the effect of the accuracy of the run time estimate on the CR policy to be less pronounced than the effect on FD because the estimate is used only when transactions conflict, not every time a scheduling decision is made. To better observe the effect on the concurrency control policy CR we reduced the size of the database from 200 to 40 in order to increase the number of conflicts. Figure 9 shows the performance of ED, HP which does not use the the estimate against ED, CR. The eligibility policy is NT, and does not use the estimate.

When the estimate is perfect CR performs better than HP. When the estimate is high, CR will rarely (in the limit, never) judge that the transaction holding the lock can finish in the slack of the lock requester. Thus conflicts are always resolved by restarting the lock holder and CR behaves exactly like HP. When the estimate is low, CR nearly always decides that the lock holder can finish within the slack time of the requester. In this case our experiment shows that CR performs slightly better than HP. We conclude that the performance penalty paid for always waiting for locks is comparable to the penalty paid in lost computation time that results from using HP.

**Slack time experiment**

The average amount of slack in job deadlines affects all scheduling algorithms. If all deadlines are extremely tight then most algorithms will perform poorly. Similarly, if all deadlines are very loose than most algorithms will perform well. Figure 10 shows the performance of the three priority assignment polices combined with FD and CR as the upper bound for slack time increases. When the slack is 0.5 (and deadlines are tight) the three algorithms perform comparably. As the slack increases, the performance of all three betters but ED is clearly the superior priority assignment policy. If the slack is very much greater (not shown) then the three curves would converge to zero because no deadlines would be missed.

**Restart/abort cost experiment**

In this experiment the cost of restarting or aborting a transaction ranges from 0 ms (i.e. no cost) to 27 ms in increments of 3. For reference, when the cost is 15 it is equal to one-third the average transaction computation time.

The cost of restarting or aborting a transaction affects the eligibility polices NT and FD, and the concurrency control polices HP and CR. It follows that as the cost of restarting or aborting increases, the performance of algorithms which use any of the above policies will deteriorate. This is indeed the case as shown by Figure 11 which plots the results for the three different concurrency policies of ED using FD. When the restart/abort cost is zero, CR and HP are both better than SE, and CR is best. The performance of SE decreases slowly because it only performs aborts and not restarts. The concurrent algorithms perform both aborts and restarts. As the cost increases SE outperforms HP and eventually equals CR.

**Cost of serializability**

In a final experiment we modified the simulation program so that serializability was not enforced. Thus a transaction was never denied a lock request and no transaction was restarted due to a lock conflict. Figure 12 shows the performance of the serialized and unserialized versions of the best algorithm, FD, ED, CR. The unserialized version performs somewhat better than the serialized. Thus serializabilty does cause the algorithms to miss more deadlines. However, missed deadlines is only one cost metric. Database inconsistency occurs as a result of unserialized schedules. For some applications the cost of database inconsistency may far outweigh the performance benefit in terms of missed deadlines gained by ignoring concurrency control.

## 5. Conclusions

In this paper we have presented various transaction scheduling options for a real-time database system. Our simulation results have illustrated the tradeoffs involved, at least under one representative database and transaction model. Before reaching some general conclusions, we would like to make two observations.

The first observation is that our base parameters represent a high load scenario (relatively high number of conflicts, relatively tight deadlines). One could argue that such a scenario is "unrealistic." However, we believe that for designing real-time schedulers, one must look at precisely these high load situations. Even though they may not arise frequently, one would like to have a system that misses as few deadlines as possible when these peaks occur. In other words, when a "crisis" hits and the database system is under pressure is precisely

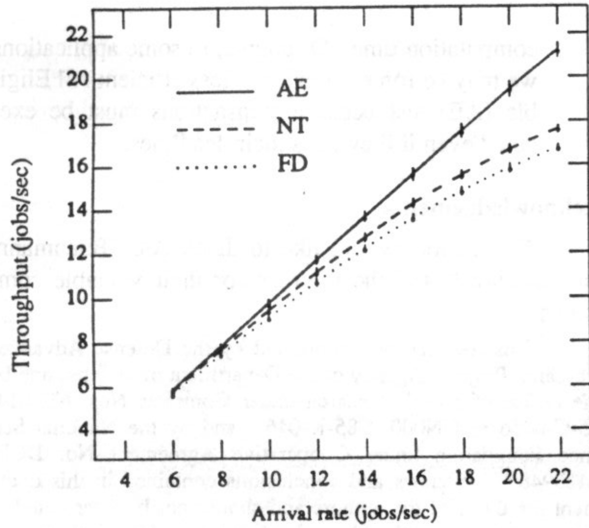



Figure 7. FCFS, SE.

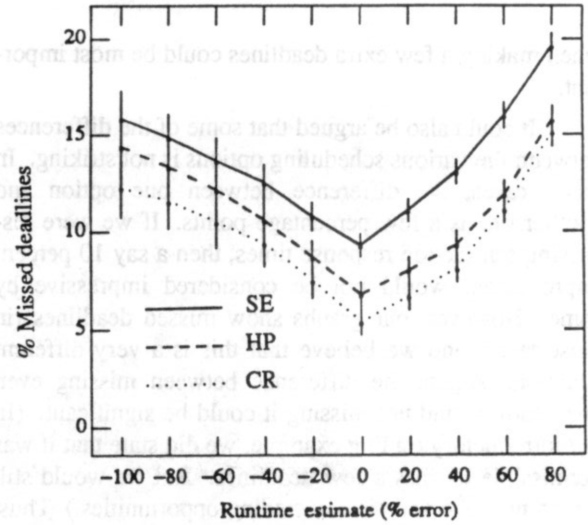



Figure 8. ED, FD.

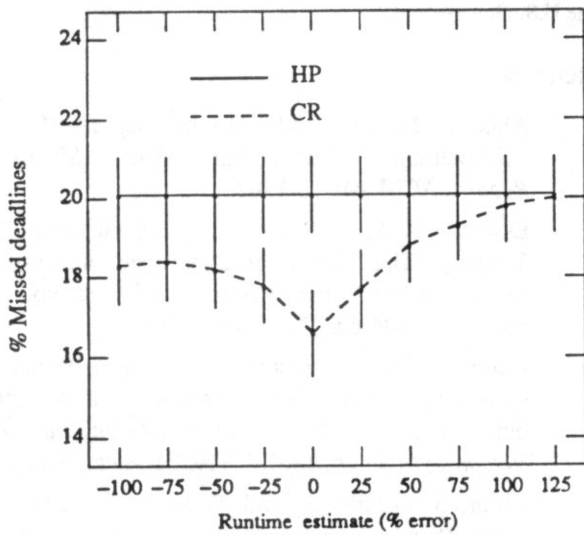



Figure 9. ED, NT.

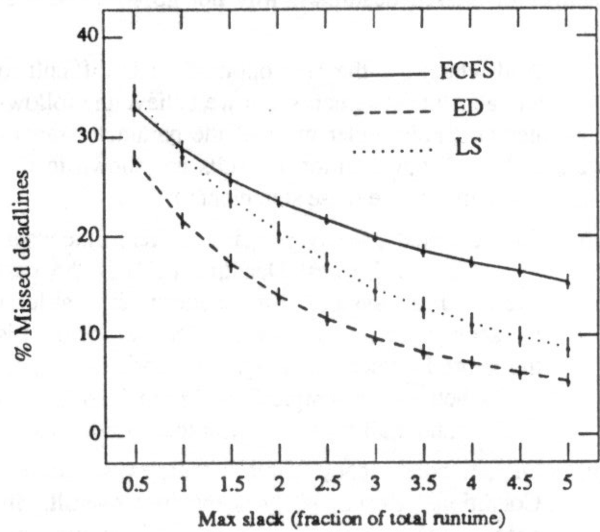



Figure 10. FCFS vs ED vs LS, FD, CR.

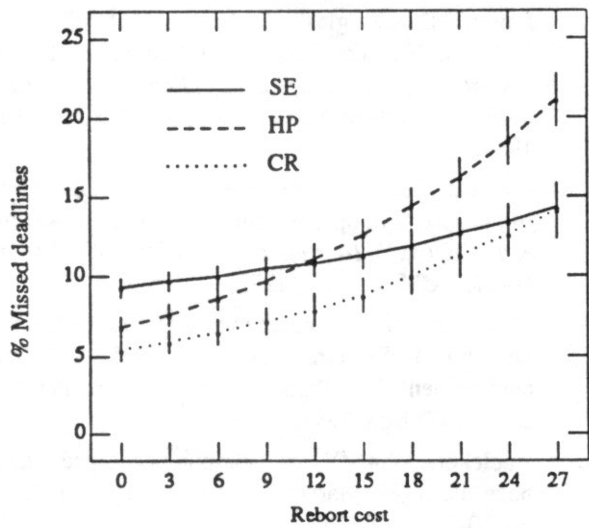



Figure 11. ED, FD.

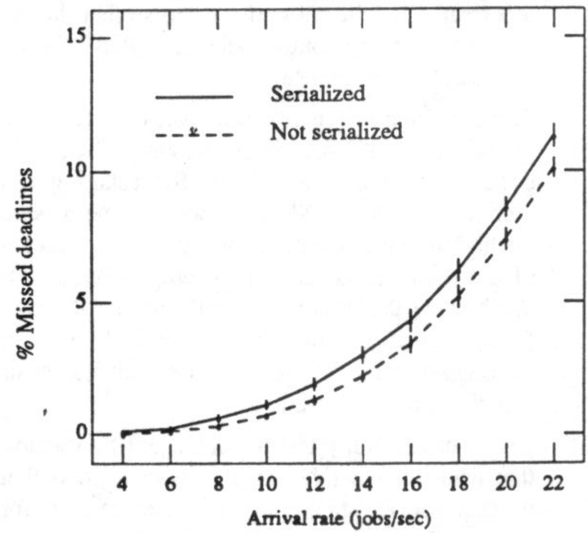



Figure 12. ED, FD, CR.

when making a few extra deadlines could be most important.

It could also be argued that some of the differences between the various scheduling options is not striking. In many cases, the difference between one option and another one is a few percentage points. If we were discussing transaction response times, then a say 10 percent improvement would not be considered impressive by some. However, our graphs show missed deadlines (in most cases) and we believe that this is a very different situation. Again, the difference between missing even one deadline and not missing it could be significant. (In our introductory trading example, we did state that it was permissible to miss a few deadlines. But we would still like to miss the very fewest trading opportunities.) Thus, if we do know that some scheduling options reduce the number of missed deadlines, why not go with the best one?

And which are the best options? It is difficult to make any absolute statements, but we believe the following statements hold under most of the parameter ranges we tested. (All our additional results not shown in this paper also substantiate these statements.)

(a) Of the tested priority policies for real-time database systems, Earliest Deadline (ED) is the best overall. It always performed better than or at least the same as the other policies. Least Slack (LS) is the second choice for assigning priorities. It performs better than simple First Come First Served (FCFS) under all but the highest load conditions.

(b) Of the concurrency control policies we tested, Conditional Restart (CR) is the best overall. Its success depends on the accuracy of the run time estimate, but even when the estimate is totally incorrect, CR will not perform worse that the two other concurrency control policies (HP, SE). It has very stable performance.

(c) There is one case where Serial execution (SE) may be superior to Conditional Restart (CR). This occurs when there is a high cost for restarting transactions. However, SE does not become a better method until the cost of restarting is more than half of the computation time of average transactions. We believe that this will usually not be the case. (Note that if the database is on disk, not our assumption here, then SE may be even less desirable that what our results show.)

(d) Using an eligibility test to screen out transactions that have missed (NT) or are about to miss their deadlines (FD) greatly improves system performance. FD performs better than or the same as NT, unless the execution estimates for transactions are more than 60 percent greater that the actual computation time. Of course, in some applications we may be forced to use the less efficient All Eligible (AE) test because transactions must be executed even if they miss their deadlines.

**Acknowledgements**

The authors would like to thank Alex Buchmann, Umesh Dayal, and the referees for their valuable comments.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Office of Naval Research under Contracts Nos. N00014-85-C-0456 and N00014-85-K-0465, and by the National Science Foundation under Cooperative Agreement No. DCR-8420948. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

**References**

1. Abbott, Robert and Hector Garcia-Molina, "Scheduling Real-time Transactions," *SIGMOD Record*, ACM, March 1988.

2. Eswaran, K. P., J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *CACM*, vol. 19, no. 11, pp. 624-633, November 1976.

3. Garcia-Molina, Hector, "Using semantic knowledge for transaction processing in a distributed database," *ACM Transactions on Database Systems*, vol. 8, pp. 186-213, ACM, June, 1983.

4. Isloor, Sreekaanth S. and T. Anthony Marsland, "The Deadlock Problem: An Overview," *IEEE Computer*, pp. 58-78, IEEE, September, 1980.

5. Jensen, E. Douglas, C. Douglass Locke, and Hideyuki Tokuda, "A time-driven scheduler for real-time operating systems," *Proceedings IEEE Real-time Systems Symposium*, pp. 112-122, IEEE, 1986.

6. Liu, C.L. and J.W. Wayland, "Scheduling algorithms for multiprogramming in hard real-time environment," *Journal of the ACM*, vol. 20, pp. 46-61, ACM, January, 1973.

7. Mok, Aloysius, "Fundamental Design Problems of Distributed Systems for the Hard Real-time environment," MIT Laboratory for Computer Science, MIT, May 1983.

8. Voelcker, John, "How Computers Helped Stampede the Stock Market," *IEEE Spectrum*, vol. 24, pp. 30-33, IEEE, December 1987.