Real-Time Transaction Scheduling: A Cost Conscious Approach*

D. Hong T. Johnson S. Chakravarthy

Database Systems Research and Development Center
Computer and Information Sciences Department
University of Florida, Gainesville, FL 32611
dh2,ted,sharma@snapper.cis.ufl.edu

Real-time databases are an important component of embedded real-time systems. In a real-time database context, transactions must not only maintain the consistency constraints of the database but must also satisfy the timing constraints specified for each transaction. Although several approaches have been proposed to integrate real-time scheduling and database concurrency control methods, none of them take into account the dynamic cost of scheduling a transaction. In this paper, we propose a new cost conscious real-time transaction scheduling algorithm which considers dynamic costs associated with a transaction. Our dynamic priority assignment algorithm adapts to changes in the system load without causing excessive numbers of transaction restarts. Our simulations show its superiority over EDF-HP algorithm.

1 Introduction

The main focus of attention in the real-time systems area has been the problem of scheduling tasks with time constraints, while the active area of research in databases has been concurrency control and recovery to guarantee database consistency. In designing a transaction scheduling policy for a real-time database system (RTDBS), an integrated approach is required to maintain the consistency constraints and at the same time satisfy transaction timing constraints. This dual requirement makes real-time transaction scheduling more difficult than task scheduling in real-time systems or transaction scheduling in database systems. Scheduling algorithms [ZRS87b, ZRS87a] used in current real-time systems assume *apriori* knowledge of tasks (arrival time, deadline, resource requirement, worst case execution time). For database applications, however, only part of such knowledge (arrival time, deadline, *conservative* data access pattern) is available. As a result, transaction scheduling in real-time database systems needs a different approach than that used in scheduling tasks in real-time systems.

*This work was supported by the National Science Foundation Research Initiation Grant IRI-9011216 and in part by the Florida High Technology and Industry Council.

SIGMOD /5/93/Washington, DC,USA
© 1993 ACM 0-89791-592-5/93/0005/0197...$1.50

In this paper, we view a real-time database system as a transaction processing system whose workload is composed of transactions with individual timing constraints. A timing constraint is expressed in the form of a deadline, and we schedule soft real-time transactions (i.e, they have timing constraints, but catastrophic results do not occur if the transaction misses its deadline [SZ88]). We assume that the transactions do not have firm deadlines [Har91], so transactions that do not meet their deadlines are not dropped. We propose a cost conscious dynamic scheduling algorithm to minimize the number of transactions that miss their deadlines and mean lateness.

2 Previous work

There are several classes of real-time database (time-critical database) scheduling algorithms in which various properties of time-critical schedulers are combined with properties of concurrency control algorithms [AGM88a, AGM88b, AGM89, BMH89, C+89, Har91, Sha88, SRSC91, SZ88, HSRT91]. Priority scheduling without knowing the data access pattern is presented as a representative of algorithms with incomplete knowledge of resource requirements. The works in [AGM88a, AGM88b, AGM89, HSRT91, Har91, SZ88] fall into this category. Earliest Deadline First (EDF) [LL73], Least Slack First (LSF) priority assignment policy with High Priority (HP) concurrency control method and Conditional Restart with estimated execution time on the main memory resident databases were proposed in [AGM88a, AGM88b]. [AGM89] extends the above to disk resident database environments. An optimistic concurrency control scheme with a deadline and transaction length based priority assignment scheme [HSRT91] and an optimistic concurrency control with Adaptive EDF have also been proposed [Har91]. Optimistic concurrency control scheme, however, shows better performance only for firm real-time transactions.

Priority scheduling with transaction pre-analysis is introduced as another approach with more knowledge of resource requirements [BMH89, Sha88, SRSC91]. Conflict avoiding nonpreemptive method and Hybrid algorithms which use conflict avoiding scheme in the non-overload case and Conditional Restart conflict resolution method in the overload case have been proposed in [BMH89]. Static priority assignment based Priority Ceiling protocol using priority inheritance with exclusive lock and read/write Priority Ceiling protocol have been proposed in [Sha88] and [SRSC91].

The rest of the paper is structured as follows: Section 3 provides motivation for our approach and presents details of transaction pre-analysis and the scheduling algorithm that uses the dynamic priority assignment policy. Sections 4 and 5 compare our approach to EDF-HP (EDF priority assignment policy with High Priority conflict resolution method [AGM88b]) for main memory and disk resident database respectively through a simulation. Section 6 contains conclusion and future research.

3 Cost Conscious Approach (CCA)

A static priority assignment is not adequate in a real-time transaction processing system because it can't consider the urgency of deadline. The standard transaction pre-analysis method also is inadequate because it is too pessimistic to use in real-time systems. The EDF and LSF priority assignment policies are restrictive for real-time transaction scheduling because they ignore the blocking among transactions, the rollback, and restart effects. To the best of our knowledge, the effects of transaction rollback and restart overhead have not been used in conjunction with finer analysis of conflicts among transactions, which is the main contribution of this paper.

3.1 Assumptions

We assume that our system contains a single CPU that manages disk or main memory resident data. Every transaction that the system executes is assumed to be an instance of a predefined class of transactions. Further, we assume that we have pre-analyzed these classes. We allow only write locks in our current analysis (shared locks will make the dynamic cost an even more important factor in real-time transaction scheduling). When a transaction arrives, we assume that we know its deadline.

3.2 Motivation

Consider the well known real-time priority assignment policies, EDF and LSF. In the context of real-time task scheduling, these policies are known to be acceptable [XP90, ZRS87b, ZRS87a]. However, these policies are not acceptable in the context of real-time transaction scheduling.

LSF is not appropriate for RTDBS because it is not easy to estimate the worst case execution time of a transaction, due to the existence of disk IO and blockings among transactions. In addition priority reversal may happen if we use LSF as a dynamic priority assignment with continuous evaluation policy.

Under high level of resource and data contention, EDF causes more transactions to miss their deadlines since they receive high priority only when they are close to missing their deadline [Har91]. EDF-HP causes many transaction aborts. The time spent on transaction aborts delays the start of other transactions.

In order to solve the problem of too many transaction aborts of EDF-HP, EDF-WP (EDF priority assignment policy with Wait Promote conflict resolution method [AGM89]) has been proposed. However, EDF-WP causes too much waiting due to its nonabortive conflict resolution method, furthermore EDF-WP has deadlock problems.

Several hybrid methods that use combinations of abortive and nonabortive methods have been proposed [AGM88a, SZ88]. These methods make decisions about transaction blocking and rollback using additional information like slack time or estimated execution time. However, they still have deadlock problems.

In this paper, we introduce the *cost conscious* approach (CCA) to real-time transaction processing. The CCA includes the cost of aborted transactions in its priority calculation to solve EDF-HP's problem of excessive aborts. Furthermore CCA uses a transaction pre-analysis to avoid the deadlock problems of the hybrid methods.

3.2.1 Dynamic cost

The transaction response time consists of the time needed to execute a transaction in an isolated environment, T_{static}, and the blocking (both IO and concurrency related) and restart overhead, $T_{dynamic}$. T_{static} is dependent on the semantics (data value and branches) in the transaction. $T_{dynamic}$ is dependent on the current status of the system and on future events, i.e, on the transactions that are currently in the system and the transactions that will arrive in the future. In the real-time database context $T_{dynamic}$ is very difficult to compute because we don't know the future events. However it would help to improve the performance of RTDBS if we could use an appropriate approximation of dynamic cost.

If a newly arrived transaction, T_a, has earlier deadline than that of the currently running transaction, T_r, and does not cause rollback (and restart) of partially executed transactions, then the newly arrived transaction is a good choice for immediate execution. If T_a has earlier deadline than that of the running transaction and conflicts with some or all of the partially executed transactions, we have to consider several choices. If we use EDF priority assignment policy, several partially executed transactions that conflict with T_a might have to be rolled back. If we consider the dynamic cost, we might find that we lose too much time for the execution of the highest priority transaction. In order to calculate the approximate dynamic cost which will be used in our priority assignment policy, we analyze transaction programs.

3.2.2 Transaction Pre-analysis

The set of data items that a transaction of some transaction type *might* access is called its *data set*. A particular execution of a transaction is likely to actually access only a small fraction of its data set. If we have no information about a transaction's execution, we must make the pessimistic assumption that it will access all items in its data set. In order to make a finer analysis of the conflict relations between transactions, we assume that as the transaction executes, it makes decisions that restricts the set of data items that it will access. Consider, for example, the two transaction programs in Figure 1:

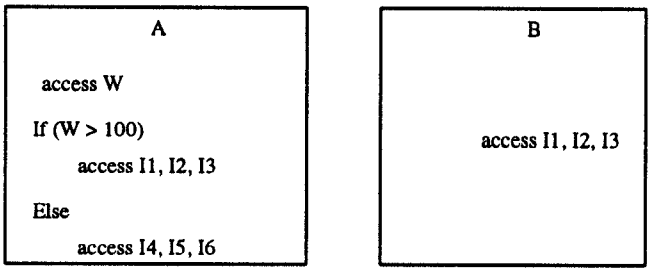

Figure 1: Transaction programs

Suppose that T_{A1} executes program A and T_{B1} executes program B. If T_{A1} executes the `If` statement and finds $w > 100$, T_{A1} and T_{B1} conflict. Otherwise, T_{A1} finds that $w <= 100$, T_{A1} and T_{B1} don't conflict. Before T_{A1} executes the `If` statement, T_{A1} and T_{B1} might conflict, so we must make the pessimistic assumption that they do conflict. We call the statements in the transaction program where the transaction commits itself (by executing a conditional statement) to accessing a subset of its data set the *decision points*. We can model each transaction as a tree[1], (i.e, the *transaction tree*) with the root labeled by the name of the transaction program. At each decision point, the tree branches, and those nodes are given unique labels related to the program name. These nodes represent refinements of what we know about the transaction's execution, and in particular about the data set it accesses. The decision points in a program can be identified by a programmer, or by a compiler. Figure 2 shows the transaction trees of transaction programs A and B. Program A's decision point splits the transaction tree into node Aa and node Ab, which have different data sets. Since program B contains no decision points, its transaction tree consists of a single vertex. When we analyze the transaction programs, we find that T^A_{A1} (subscript represents the identifier of a transaction and superscript represents its label) conflicts with T^B_{B1}, T^{Aa}_{A1} conflicts with T^B_{B1}, but that T^{Ab}_{A1} doesn't conflict with T^B_{B1}.

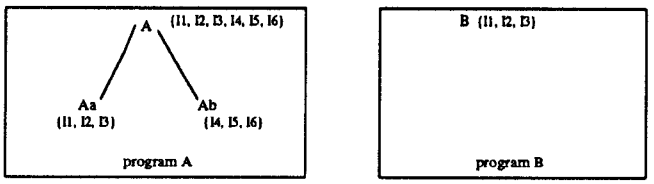

Figure 2: Transaction Tree

In the Figure 2, before node A reaches the decision point, it might conflict with node B (if node A takes the branch to node Aa), or it might not conflict with node B (if node A takes the other branch). Suppose node A makes the branch to node Aa. At this point we are certain that node A conflicts with node B. So, our pre-analysis system has several different flavors of data conflict. We say that two transactions *don't conflict* if, given their current state, they won't access overlapping data sets for all possible execution paths. Two transactions *conflict* if, no matter what their execution paths, they will access overlapping datasets. If two transactions might or might not conflict based on their future execution, then they *conditionally conflict*.

Suppose that transaction T^P_N conflicts with transaction T^Q_M, and T^P_N is scheduled to execute. If T^Q_M has not yet accessed any data items that T^P_N might access, then there is no need to roll back T^Q_M, we only need to block it. In this case, we say that T^Q_M is *safe* with T^P_N. If T^Q_M has accessed a data item that T^P_N will access, then T^Q_M is *unsafe* with T^P_N and needs to be rolled back when T^N_P accesses the conflicting item. Finally, T^Q_M is *conditionally unsafe* with T^P_N if T^Q_M might be safe or unsafe with T^P_N, depending on T^P_N's execution. We will soon define these concepts rigorously.

If we know what data items a transaction accesses between decision points, we can calculate the conflict and safety relations in a straightforward manner. Towards this end, we define:

Leaf The label of a transaction that will execute no further decision points.

accesses(T^P_N) Set of data items that a transaction N of label P accesses before it reaches its next decision point.

hasaccessed(T^P_N) Set of data items that a transaction N of label P has accessed up to this point.

mightaccess(T^P_N) Set of data items that a transaction N of label P might access.

leaves(T^P_N) Set of leaves of the subtree rooted at label P of a transaction N.

We now give precise definitions of the conflict and safety relationships, which also provide a method to calculate the relations. Suppose we are given accesses(T^P) for every node P in the transaction tree. If K is the set of nodes on the path from the root to P, inclusive, then

$$\begin{aligned} \text{hasaccessed}(T^P) &= \cup_{k \in K} \text{accesses}(T^k) \\ \text{mightaccess}(T^P) &= \text{hasaccessed}(T^P) \quad P \text{ a leaf} \\ &= \cup_C \text{mightaccess}(T^C) \quad P \text{ not a leaf} \\ &\quad (C \textit{ is a child of } P) \end{aligned}$$

With *mightaccess* and *hasaccessed* calculated at every node, we can calculate the conflict and safety relations:

- Leaf transactions T^P_N and T^Q_M *conflict* if and only if $\text{mightaccess}(T^P_N) \cap \text{mightaccess}(T^Q_M) \neq \phi$
- Transactions T^P_N and T^Q_M *conflict* iff $\forall_{p \in leaves(T^P_N)} \forall_{q \in leaves(T^Q_M)} \text{mightaccess}(T^p_N) \cap \text{mightaccess}(T^q_M) \neq \phi$.
- Transactions T^P_N and T^Q_M *conditionally conflict* if and only if $\exists_{i,j \in leaves(T^P_N)}$, $\exists_{m,n \in leaves(T^Q_M)}$ such that $\text{mightaccess}(T^i_N) \cap \text{mightaccess}(T^m_M) \neq \phi$ and $\text{mightaccess}(T^j_N) \cap \text{mightaccess}(T^n_M) = \phi$.

[1] Although, a loop-free program is a directed acyclic graph, we use a tree representation for the sake of simplicity

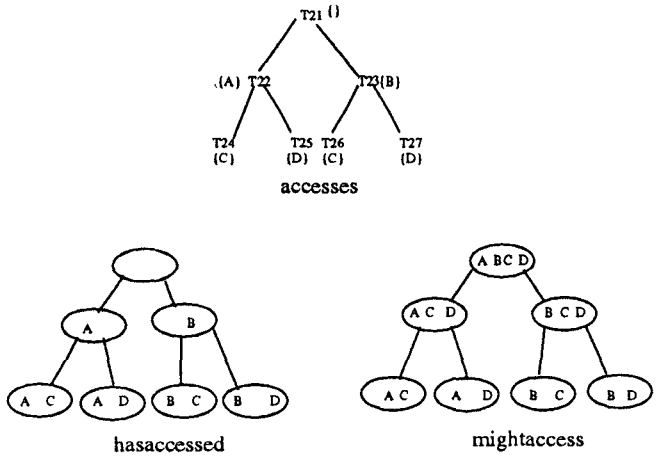

Figure 3: Auxiliary Transaction Tree

- Transactions T_N^P, T_M^Q *don't conflict* if and only if they neither conflict nor conditionally conflict.
- Transaction T_N^P is *safe* with respect to T_M^Q if and only if hasaccessed(T_N^P) $\cap$ mightaccess(T_M^Q) $= \phi$.
- Transaction T_N^P is *unsafe* with respect to T_M^Q if and only if $\forall_{q \in leaves(T_M^Q)}$, hasaccessed($T_N^P$) $\cap$ mightaccess(T_M^q) $\neq \phi$.
- Transaction T_N^P is *conditionally unsafe* with respect to T_M^Q if and only if hasaccessed(T_N^P) $\cap$ mightaccess(T_M^Q) $\neq \phi$, and $\exists_{q \in leaves(T_M^Q)}$ such that hasaccessed(T_N^P) $\cap$ mightaccess(T_M^q) $= \phi$.

Note that safety relationships are computed based on the assumption that a transaction accesses its data items when it begins and immediately after its decision points. These transaction relationships will be used to calculate transaction priorities more accurately in the following section. Even though maintaining the transaction relationship information requires additional space, it is a reasonable approach for RTDBS to trade-off space for better performance.

3.3 Scheduling Algorithm

A real-time transaction scheduling algorithm consists of one or more priority assignment policies and conflict resolution methods. The system might use different priority assignment policies for different resource types. Whenever a resource conflict occurs, a priority is used to resolve the conflict. In [AGM88b, SZ88] they use different priority assignment policies for CPU and data conflict in order to cope with their different restrictions. But if we use different priority assignment policies for different resource types we might have a priority reversal that causes deadlock problems [BMH89]. In CCA we use only one priority assignment policy for CPU and data conflicts.

3.3.1 Priority Assignment

CCA uses a dynamic priority assignment policy with a continuous evaluation method which evaluates the priority several times during the execution of a transaction in order to capture all the dynamic features of database transactions.

If the transaction T_a which is selected to be run next conflicts with m transactions that are *unsafe* or *conditionally unsafe* with T_a, we might lose

$$Timelost(T_a) = \Sigma_{t \in M} \ (rollback_t + exec_t)$$

$$M = \{t \mid t \ is \ unsafe \ or \ conditionally \ unsafe \ with \ T_a\}$$

where $exec_t$ is the effective service time of T_t and $rollback_t$ is the time required to roll back T_t.

If the value of $Timelost(T_a)$ is large, executing T_a wastes system resources. We characterize the time lost as the *penalty of conflict*.

penalty of conflict is the value $Timelost(T_a)$, which is the sum of the effective service time and rollback time of the transactions that must be rolled back to execute T_a to its commit point without interruption.

The notion of the penalty of conflict, described above, can be introduced into the our CCA dynamic priority computation formula. If $\Pr(T_i)$ is the priority of transaction T_i and d_i is the deadline of transaction T_i, then

$$Pr(T_i) = -(d_i + \omega Timelost(T_i))$$

Thus larger value means higher priority. The value of ω will be readjusted accordingly to get the best performance.

3.3.2 Conflict Resolution

There are 3 types of resources in the system: CPU, disk and data. The main active resources in real-time database systems are the CPU and disk, and the passive resource is the data. They require different scheduling disciplines [Har91].

Data conflict Whenever a data conflict occurs, the running transaction aborts the conflicting transactions. The priority of the running transaction is always higher than that of the conflicting transactions because only the highest priority transaction (main memory database) or the transaction that doesn't have the highest priority but is compatible with partially executed transactions (disk resident database) gets the CPU.

CPU conflict If we assume that we have a single CPU system, there are many opportunities for CPU scheduling. Whenever a new transaction arrives or a running transaction finishes, the scheduler is invoked.

IO conflict If the real-time database contains disk resident data, a transaction might perform many IO waits during its execution. Several real-time IO scheduling methods have been proposed [AGM89, C+89] in order to reduce IO wait. Disk IO introduces new problems in real-time transaction scheduling. Consider the following scenario: Transaction T_1 is blocked and is waiting for an IO completion. The next highest priority transaction, T_2, gets the CPU and starts executing so as not to waste the CPU. If T_2 conflicts with T_1, then T_2 performs a *noncontributing execution*, lower priority transaction's execution during the IO wait of higher priority transactions that is

rolled back when a higher priority transaction finishes IO, because it must be rolled back when T_1 unblocks. This situation is worse than the situation in which no transaction is selected to execute during T_1's IO wait time, because of the cost incurred in rolling T_2 back. If the third highest priority transaction, T_3, accesses a data set disjoint with that of T_1 and T_2, then T_3 is the better choice. In our approach we select T_3 rather than T_2 during T_1's IO wait using the pre-analyzed information.

3.3.3 Algorithm

The following is the pseudo code for the scheduling algorithm proposed in this paper and is based on the notion of cost incurred due to conflicts. The function "IOwait-sched" is invoked whenever a transaction blocks waiting for IO completion. This function reduces the noncontributing execution and hence avoids rollback by using transaction conflict relations.

```
Function IOwait-sched
begin
    if ready queue is empty
    then
        return NIL;
    else
        IF there are transactions in the ready queue
           that don't conflict or conditionally conflict
           with partially executed transactions
        then
           return the one with
           the highest priority among them;
        else
           return NIL;
end
```

The procedure "tr-arrival-sched" is called whenever a new transactions arrives and the procedure "tr-finish-sched" is invoked whenever the running transaction finishes. These two procedures use the penalty of conflict (approximation of dynamic cost) of transactions in order to improve the performance of RTDBS. The sleep queue holds transactions that are blocked and the partially executed transaction list (*P list*) links all transactions that are executed partially. We introduce a parameter penalty-weight (ω in our previous priority formula) that can be used to weigh the contribution of penalty of conflict on the value of the priority value computed. We use a value of penalty-weight between 0 and ∞ for our algorithm.

```
Function Pr
begin
    calculate
    (deadline + (penalty-weight * penalty of conflict));
    return(take negative value);
end
```

T_A is a new transaction and T_H is the highest priority transaction in the following procedures.

```
Procedure tr-arrival-sched
begin
    IF Pr(T_H) < Pr(T_A)
    then
        make T_A as a new T_H;
        schedule T_H;
    else
        add T_A to the ready queue;
        schedule T_H;
end


Procedure tr-finish-sched
begin
    foreach transaction in the ready queue
    begin
        assign new priority;
        Choose the highest priority transaction
        and make it t_H;
    end
end
```

3.3.4 Properties of CCA

CCA uses a dynamic priority assignment with continuous evaluation method in order to adapt to the changes of systems load effectively. However a dynamic priority assignment with continuous evaluation method might have two potential problems: deadlock and circular abort.

The HP conflict resolution scheme is a deadlock prevention mechanism if it is combined with a fixed or dynamic with static evaluation priority assignment. If it is combined with some dynamic assignment with a continuous evaluation methods (e.g., LSF) it can cause deadlock due to *priority reversal*. Our approach uses a dynamic priority assignment with a continuous evaluation method and HP conflict resolution scheme. However CCA is still deadlock free because there is no lock wait in CCA.

Theorem 1 *There exist no deadlock under CCA.*

In order to prove Theorem 1 we define following terms;

A primary transaction is defined as the transaction T_H that is scheduled by the procedure "tr-arrival-sched" or "tr-finish-sched". Only one primary transaction exists in the system. Although it is possible to have more than one transaction that have the highest priority value, only one is designated as T_H.

A secondary transaction is defined as the transaction T_S that is scheduled by the function "IO-wait-schedule". By definition, the transaction T_S doesn't *conflict* or *conditionally conflict* with any partially executed transaction. During the IO wait, only T_S can be executed.

Proof :

case1 The transaction T_H accesses the data item that is held by some transaction in the P list then the transaction in P list will get aborted.

case2 The transaction T_S accesses the data item that is held by a transaction in P list. This case cannot happen by the definition of T_S.

Lemma 1 *There exist no priority reversal between T_H and any other transaction in the system under CCA scheduling.*

Proof : Note that when T_S is being executed the priority of any transaction in the system does not change by definition (as it does not conflict with any transaction in the P list). Now when T_H (either an incoming transaction or a transaction picked from the ready queue) is running $\Pr(T_H)$ is greater than or equal to any transaction in the ready queue. Without loss of generality, if T_H aborts any transaction (say T_X), then the priority of T_H increases (because the penalty of conflict decreases by the effective service time of T_X) and so will the priority of any transaction T_Y in the ready queue that conflicts with T_X by the same amount.[2] Also, the priority of any transaction in the ready queue that doesn't conflict with T_X does not change. Hence the priority relationship between T_H and every other transaction in the ready queue that is not aborted remains the same. Hence there is no priority reversal. However there could be priority reversal between non-conflicting transactions.

Theorem 2 *There exist no circular abort under CCA.*

Proof : Only the primary transaction aborts conflicting transactions and conflicting transactions cannot have higher priority than that of the primary transaction. Thus from Theorem 2 there is no priority reversal and an abort occurs between conflicting transactions only.

4 Simulation of main memory database

In order to evaluate the performance of the CCA algorithm we simulated a real-time transaction scheduler using the SIMPACK simulation package [Fis92]. In both the main memory and disk resident database experiments (with a single processor) we assume that we have transactions with no branch points. Presence of branch points will improve upon the results presented here.

Transactions enter the system according to a Poisson process with arrival rate λ (i.e., exponentially distributed inter-arrival times with mean value $1/\lambda$), and they are ready to execute when they enter the system (i.e., release time equals arrival time). Every transaction is one instance of 50 transaction types and the transaction type for arriving transaction is chosen uniformly from the range of types. The number of objects updated by a transaction type is chosen from a normal distribution and the actual database items are chosen uniformly from the range of database size. These items and the number are regenerated at each runs and the time to access data item is always the same. The assignment of a deadline is controlled by the resource time of a transaction and two parameters Min-slack and Max-slack which set a lower and upper bound of percentage of slack time compared to the execution time respectively. A deadline T_d is calculated by summing resource time and slack time which is calculated by multiplying slack percent and resource time. Slack percent is chosen uniformly from the range of Min-slack and Max-slack.

$$T_d = arrival\ time + resource\ time \times (1 + slack\ percent)$$

[2] If both T_X and T_Y are aborted by T_H then it does not pose any problem for maintaining the priority order either.

We ran the simulation with the same parameter for 10 different random number seeds and 1000 transactions are executed at each run. For each algorithm the result were collected and averaged over the 10 runs. We varied parameters shown in Table 1 in order to see the behavior of CCA on various environment.

Parameter	Value
Transaction type	50
Update per transaction(mean, std)	(20, 10)
Computation/update(ms)	4
Database size	30
Min-slack as fraction of total runtime	20(%)
Max-slack as fraction of total runtime	800(%)
abort cost(ms)	4
weight of penalty of conflict	1

Table 1: Base parameters

Effect of Arrival Rate: In this experiment, we varied arrival rate from 1 tr/sec to 10 trs/sec with the base parameters shown in Table 1. With the base parameters the capacity of the system (assuming no aborts) is:

$$\frac{4\ ms}{update} \times \frac{20\ update}{transaction} = \frac{80\ ms}{transaction} = 12.5\ trs/sec$$

We plot our results to 10 trs/sec, because past this point more than 20 % miss their deadlines. The 20 % miss rate indicates an overload condition [HCL90] requiring specialized mechanisms. Figure 4 shows the effect of arrival rate. Figure 5 shows the improvement of CCA over EDF-HP in term of lateness and the number of transactions that have missed their deadlines. CCA shows upto 30 % and 20 % improvement in terms of mean lateness and miss percent respectively. The improvement of CCA over EDF-HP is calculated by:

$$improvement = \frac{EDF - CCA}{EDF} \times 100\%$$

We observe that with the base parameters in Table 1, the number of restarts climbs steeply up to arrival rate 8 and then declines sharply from the peak point in Figure 6. The reason for sharp decline is that when the arrival rate is high, it is less likely that an arriving transaction will have an earlier deadline than the currently running transaction [AGM88a]. The improvement graph of CCA over EDF-HP is almost the same shape as the graph of transaction abort.

The average number of partially executed transactions in the experiment with base parameters is 1 to 2 with the changes of arrival rate from 1 tr/sec to 10 trs/sec. Thus scheduling overhead of the CCA does not cause a problem.

Effect of Variation of Update Time: In this experiment, we classified the 50 transaction types into 3 classes and assigned the computation time per update as 0.4, 4 and 40 according to their classes. The capacity of the system (disregarding aborts) is:

$$\frac{\frac{0.4+4+40}{3}}{update} \times \frac{20}{transaction} = \frac{296\ ms}{transaction} = 3.37\ trs/sec$$

The transaction classes create a lot of variance in the transaction execution time (the execution time of transaction varies from 4 ms to 1200 ms). Therefore, there will be more chances for transaction preemption. Figure 7 and Figure 8 show the results of the experiment. With the variation of update time there is higher possibility that an arriving transaction will have an earlier deadline than the currently executing transaction. Thus the improvement of CCA over EDF-HP is a little higher in term of miss percent and mean lateness.

Effect of Database Size: In this experiment, we fixed every parameter except database size. When the database size increases, system load decreases. Figure 9 shows the effect of DB size on arrival rate 10. The experiment shows that CCA performs better than EDF-HP in the situation where many transaction aborts occur due to heavy data contention among transactions.

Effect of Penalty-Weight: Figure 10 shows the stability of penalty-weight. This is desirable property because the performance of the system is not sensitive to the selection of penalty-weight within some range. This value is largely dependent on the lengths of transactions that are run on the system.

5 Simulation of Disk resident database

In order to see the performance of our algorithm on disk resident database we extended the simulation program of real-time database system. In this simulation we assumed that we have single processor, single disk and FCFS IO scheduling. If a transaction is aborted during its wait on the disk queue, the transaction is deleted from the disk queue immediately. However, if a transaction is aborted during its IO access it is not deleted until it releases the disk. The base parameters in this simulation program was selected to avoid the bottleneck at the disk access. With the base parameters in the Table 1 and Table 2 the capacity of the system is:

Parameter	Value
abort cost(ms)	5
Disk access time(ms)	25
Disk access probability	1/10

Table 2: Base parameters for disk resident database: The other parameters are the same as those of main memory database

$$\frac{20\ items}{transaction} \times \frac{4}{item} = \frac{80\ ms}{transaction} = 12.5\ trs/sec$$

This calculation is very optimistic because it doesn't include abort cost nor the cost of re-executing transactions. When the load of the system reaches its capacity the utilization of the disk is:

$$\frac{12.5 \times 20/10 \times 25}{1000\ ms} \times 100 = 62.5\ \%$$

We ran the simulation with the same parameter in Table 1 and Table 2 for 30 different random number seeds and 300 transactions are executed at each run. The number of transaction is chosen to avoid bottleneck at disk access without introducing new long term scheduling (job scheduling). In our experiment the utilization of the disk within arrival rate from 1tr/sec to 7trs/sec is below 62.5 % which is maximum possible when we schedule only compatible transactions. For each algorithm the results were collected and averaged over the 30 runs.

Effect of Arrival Rate: In this experiment, we varied arrival rate from 1 tr/sec to 7 trs/sec with the base parameters shown in Table 1 and Table 2. Figure 11 shows the effect of arrival rate and Figure 13 shows the improvement of CCA over EDF-HP. CCA shows upto 95 % and 40 % improvement in terms of mean lateness and miss percent respectively. The improvement of CCA over EDF-HP in terms of mean lateness is remarkable for soft real-time transactions. The improvement is better than that of main memory resident case. The reason for better result is that CCA not only makes better decision about transaction blocking and restart but also prevents *noncontributing executions* effectively.

Figure 12 shows the number of transaction restarts in CCA and EDF-HP with the increase of arrival rate. The plot of CCA restart is almost the same as that of main memory database. However the restart in EDF-HP increases monotonically with the increase of arrival rate. This phenomenon is different from previous simulation on main memory database. The reason for monotonic increase in EDF-HP is that when the arrival rate is high, the possibility of restarting the partially executed transactions which are selected during the IO wait is very high, while the possibility of restart of partially executed transactions which are selected by procedure "tr-arrival-sched" or "tr-finish-sched" is very low. With the longer IO wait time the more transactions are scheduled and that makes the IO wait time longer and longer. Thus the possibility of restarting the partially executed transactions increases monotonically.

Effect of DBsize: In this experiment, we fixed every parameters except database size. Figure 14 shows the effect of DB size on arrival rate 4. The experiment shows that CCA performs better than EDF-HP in the situation where many transaction aborts occur due to heavy data contention among transactions on disk resident database also.

Effect of Penalty-Weight: Figure 15 shows the stability of penalty-weight. The performance of the system is not sensitive to the selection of penalty-weight within a wide range in this experiment.

6 Conclusion and future work

To the best of our knowledge, all previous methods of real-time transaction scheduling that use transaction abort have not considered the dynamic cost, the cost of rolling back and restarting transactions. Although, this perhaps is not a key factor in real-time task scheduling that only consider timing correctness, in real-time transaction scheduling, the cost incurred to preserve trans-

action properties is substantial.

In this paper, we have proposed a new real-time transaction scheduling algorithm that has a cost conscious dynamic priority assignment policy. The distinctive features of our approach are:

First, our dynamic priority assignment policy synthesizes deadline and *penalty of conflict* together. The amount of effective service time of a transaction is taken into account as it is a part of the penalty of conflict computed for conflicting transactions. Second, our approach is deadlock free. Third, there is no circular abort in our approach. Fourth, our priority assignment policy easily adapts to the changes of system load which is caused by data contention using penalty of conflict and works well in a high data contention. Fifth, there is no starvation in our approach. If we consider the deadline of a transaction when calculating the transaction priority then we avoid starvation.

We are currently extending our approach to multiprocessor environments. Our approach seems to be more promising than simple EDF-HP approach because our approach shows better performance than EDF-HP even when data contention is high and EDF-HP which only uses deadline information looks almost impossible to get better performance on multiprocessors systems. Currently we are developing a combination of CCA and EDF-HP for shared memory multiprocessors and shared nothing multiprocessors systems.

References

[AGM88a] Robert Abbot and Hector Garcia-Molina. Scheduling real-time transactions. *SIGMOD RECORD*, 17(1):71–81, 1988.

[AGM88b] Robert Abbot and Hector Garcia-Molina. Scheduling real-time transactions: a performance evaluation. In *Proceedings of the 14th VLDB*, pages 1–12. ACM, 1988.

[AGM89] Robert Abbot and Hector Garcia-Molina. Scheduling real-time transactions with disk resident data. In *Proceedings of the 15th VLDB*, pages 385–396. ACM, 1989.

[BMH89] A. Buchmann, D.R. McCarthy, and M. Hsu. Time-critical database scheduling: A framework for integrating real-time scheduling and concurrency control. In *Proceedings of the Fifth Conference on Data Engineering*, pages 470–480, Feb 1989.

[C+89] S. Chakravarthy et al. HiPAC: A Research Project in Active, Time-Constrained Database Management, Final Report. Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, MA, Aug. 1989.

[Fis92] Paul A. Fishwick. *SIMPACK:C-based Simulation Tool Package Version 2.* University of Florida, 1992.

[Har91] Jayant Ramaswamy Haritsa. Transaction scheduling in firm real-time database systems. Technical Report TR1036, University of Wisconsin-Madison, 1991.

[HCL90] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. On being optimistic about real-time constraints. *ACM SIGMOD*, 1990.

[HSRT91] Jiandong Hyang, John A. Stankovic, Krithi Ramamritham, and Don Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *Proceedings of the 17th VLDB*, pages 35–46. ACM, 1991.

[LL73] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, pages 46–61, 1973.

[Sha88] Lui Sha. Concurrency control for distributed real-time databases. *SIGMOD RECORD*, 17(1):82–98, 1988.

[SRSC91] Lui Sha, Ragunathan Rajkumar, Sang Hyuk Son, and Chun-Hyun Chang. A real-time locking protocol *IEEE Transactions on Computers*, 40(7):793–800, 1991.

[SZ88] John A. Stankovic and Wei Zhao. On real-time transactions. *SIGMOD RECORD*, 17(1):4–18, 1988.

[XP90] Jia Xu and David R. Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on Software Engineering*, 16(3):360–369, 1990.

[ZRS87a] Wei Zhao, Krithi Ramamritham, and John A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, 36(8):949–960, 1987.

[ZRS87b] Wei Zhao, Krithi Ramamritham, and John A. Stankovic. Scheduling tasks with requirement in hard real-time systems. *IEEE Transactions on Software Engineering*, 13(5):225–236, 1987.

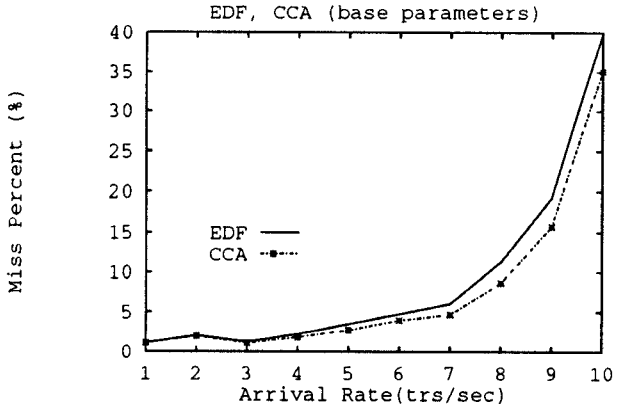

Figure 4: Miss percent of EDF,CCA

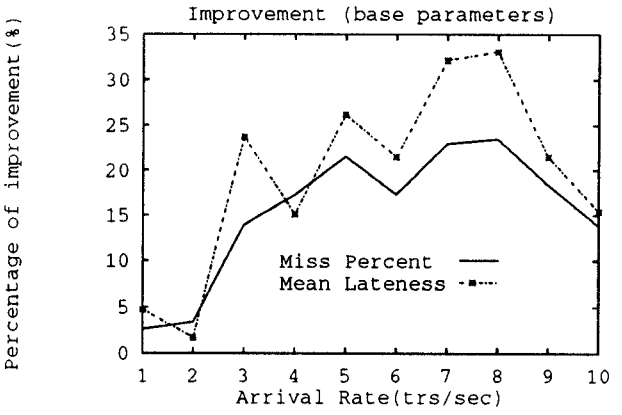

Figure 5: Improvement of CCA

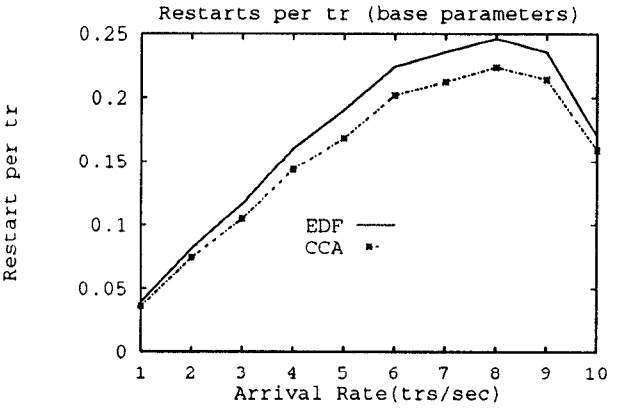

Figure 6: Restart per transaction

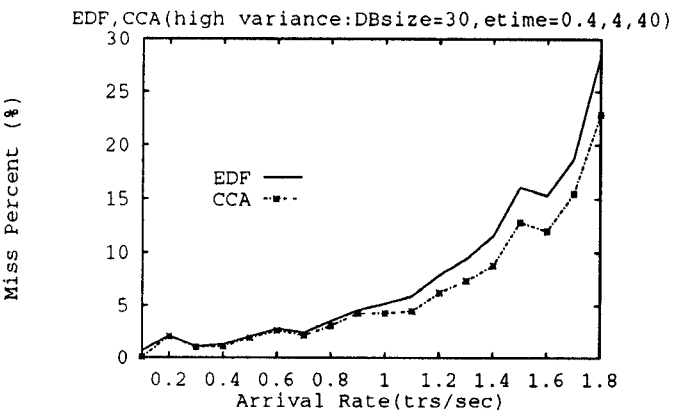

Figure 7: Effect of high variance

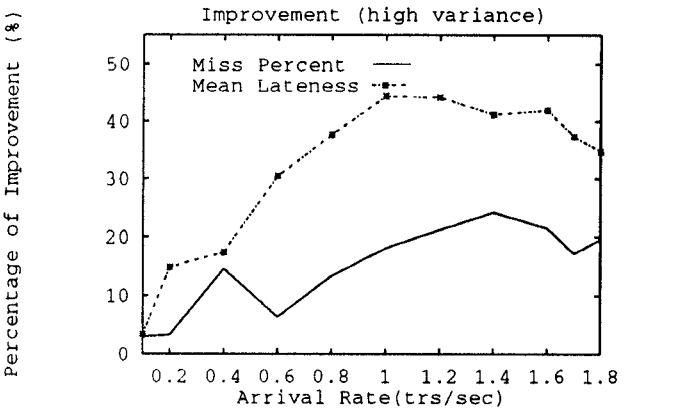

Figure 8: Improvement (high variance)

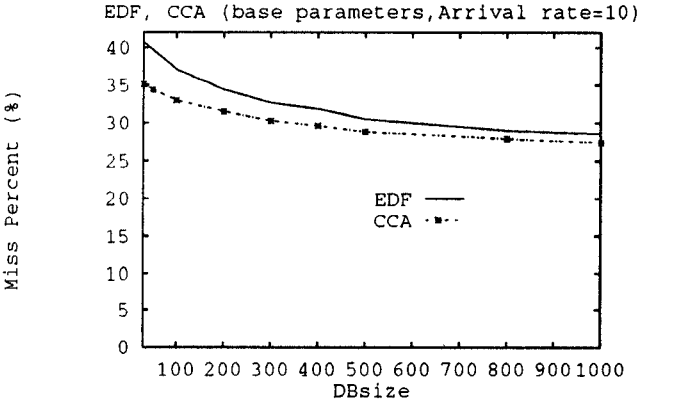

Figure 9: Effect of DB size

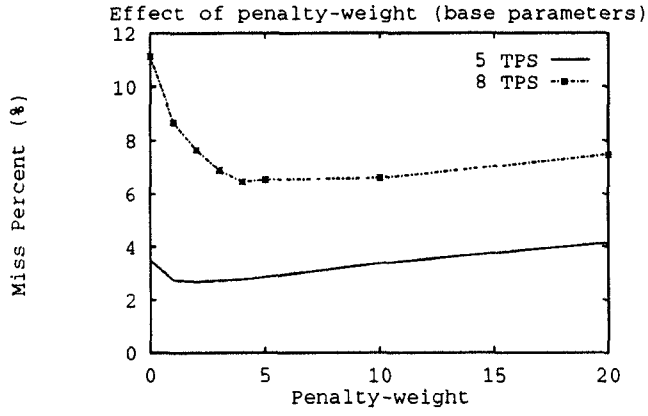

Figure 10: Effect of penalty-weight

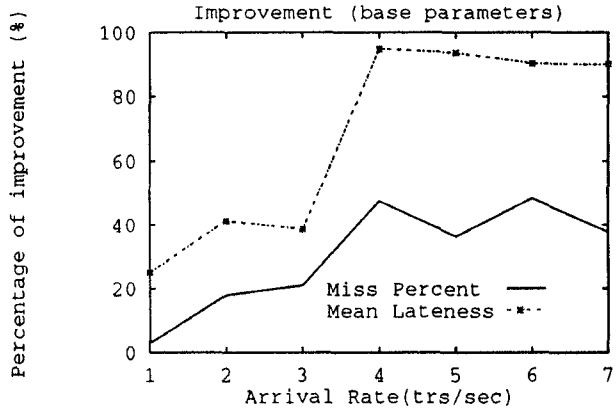

Figure 13: Improvement of CCA (disk resident)

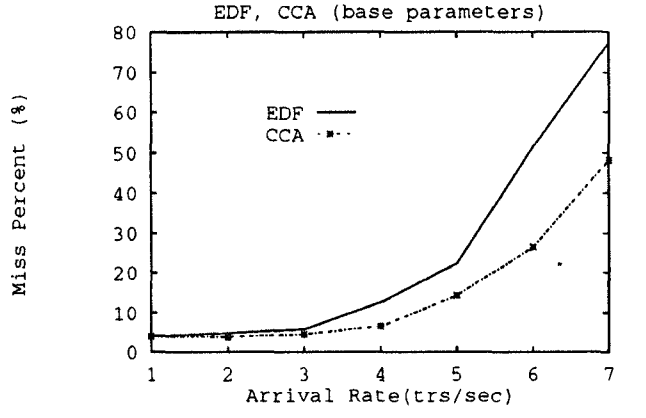

Figure 11: Miss percent of EDF, CCA (disk resident)

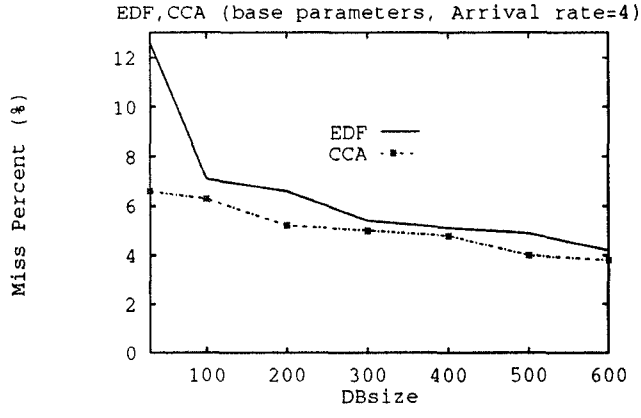

Figure 14: Effect of DB size (disk resident)

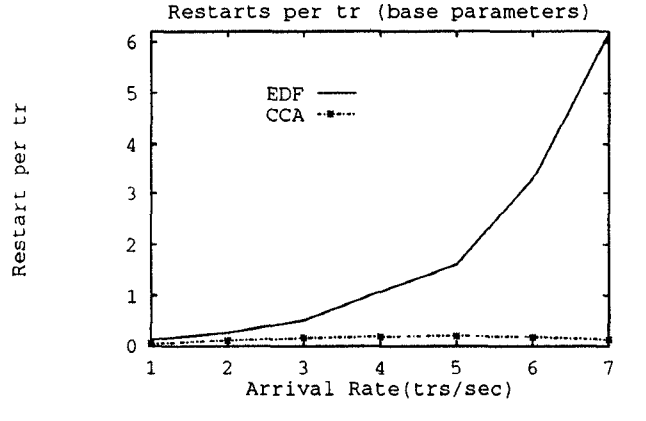

Figure 12: Restart per tr (disk resident)

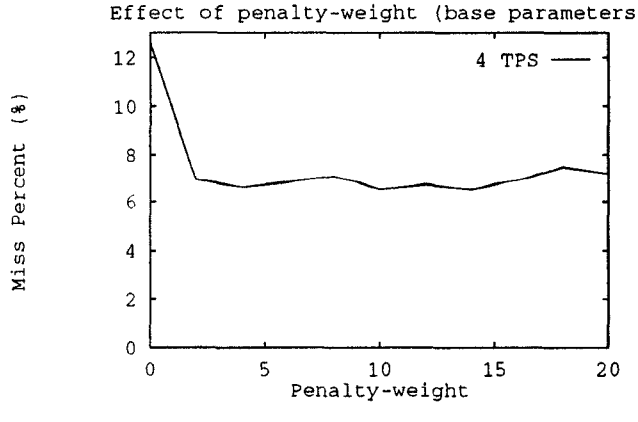

Figure 15: Effect of penalty-weight(disk resident)