Contents

Real-Time Transaction Scheduling : A Framework for Synthesizing Static and Dynamic Factors

S. Chakravarthy **D. Hong** **T. Johnson**

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

March 14, 1994

Abstract

Real-time databases are poised to be an important component of complex embedded real-time systems. In *real-time databases* (as opposed to *real-time systems*), transactions must satisfy the ACID properties in addition to satisfying the timing constraints specified for each transaction (or task). Although several approaches have been proposed to combine real-time scheduling and database concurrency control methods, to the best of our knowledge, none of them provide a framework for taking into account the dynamic cost associated with aborts, rollbacks, and restarts of transactions.

In this paper, we propose a framework in which both static and dynamic costs of transactions can be taken into account. Specifically, we present: i) a method for pre-analyzing transactions based on the notion of branch-points for data accessed *up to a branch point* and predicting expected data access *to be incurred* for completing the transaction, ii) a formulation of cost that include static and dynamic factors for prioritizing transactions, iii) a scheduling algorithm which uses the above two, and iv) simulation of the algorithm for several operating conditions and workloads.

Our dynamic priority assignment policy (termed the *cost conscious approach or CCA*) adapts well to fluctuations in the system load without causing excessive numbers of transaction restarts. Our simulations indicate that i) CCA performs better than the EDF-HP algorithm for both soft and firm deadlines, ii) CCA is more fair than EDF-HP, iii) CCA is better than EDF-CR for soft deadline, even though CCA requires less information, and iv) CCA is especially good for disk-resident data.

Index Terms: Deadlines, Real-time transactions, Scheduling, Static and dynamic costs, Time constraints, Transaction pre-analysis, soft, hard, and firm deadlines

1 Introduction

The main focus of research in the real-time systems area has been the problem of scheduling tasks to meet the time constraints associated with each task, while the focus in database area has been concurrency control to guarantee database consistency and recovery in the presence of various kinds of failures (i.e., ACID properties). Design of a scheduling policy for a real-time database system

(RTDBS) entails synergistically combining techniques from both areas and fine-tuning them to obtain a policy that meets the requirements of scheduling transactions in real-time databases. This dual requirement makes real-time transaction scheduling more difficult than task scheduling in real-time systems or transaction scheduling in database systems.

Typically, applications in real-time systems do not share disk-resident data. Even when they share data, the consistency of shared data is not managed by the system but by the application program. For the assumptions used in real-time systems, it is possible to predict some of the characteristics of tasks needed for the scheduling algorithms. As a result, scheduling algorithms [ZRS87b, ZRS87a] used in current real-time systems assume *a priori* knowledge of tasks, such as arrival time, deadline, resource requirement, and worst case (cpu) execution time.

For database applications, on the other hand, the following sources of unpredictability exist [Ram93] which makes it difficult to predict some of the resource requirements for transactions that need to meet time constraints:

1. Resource conflicts (e.g., wait for disk I/O)
2. Data dependence (e.g., execution path based on the database state)
3. Dynamic paging and I/O (e.g., page faults, caching, and buffer allocation)
4. Data interference (e.g., aborts, rollbacks, and restarts)
5. Algorithmic variations for disk-resident data access (e.g., clustered scan vs. use of index)

Note that most of the sources of unpredictability are related to the database characteristics[1] (i.e., interference or secondary storage access). Although it is possible to make conservative (or worst case) estimates for some of the above (e.g., read and write sets gleaned from a transaction), it is, in general, not possible to predict *a priori* the interference among transactions. Although serial execution avoids interference, in the presence of deadlines, completion of transactions without violating timing constraints is completely determined by the arrival order. Knowledge of transaction semantics, such as *write-only transactions*, *update transactions* and *read-only transactions* can also affect performance if they are taken into account in the development of a scheduling policy. Hence, a synthesis of pre-analyzed information and the use of dynamic costs obtained from the actual execution seems to be a viable approach for obtaining scheduling policies to meet the requirements of transaction executions in real-time databases.

Transactions that have deadlines have been categorized into *hard deadline*, *soft deadline*, and *firm deadline* transactions. Transactions that have hard deadlines have to meet their deadlines; otherwise, the system does not meet the specification. Typically, transactions that are in this category have catastrophic consequences if their deadlines are not met. Sometimes contingency measures may be included as an alternative. Soft real-time transactions have time constraints, but there may be still some residual benefit for completing the transaction after its deadline. Conventional transactions with response time requirements can be considered soft real-time transactions. In contrast to the above two, firm transactions are those which need not be considered any more if their deadlines are not met, as there is no value to completing the transaction after its deadline. Typically applications that have a definite window (e.g., banking and stock market applications)

[1] If the entire database is assumed to be in memory (i.e., main memory resident database), some of the above unpredictability sources will disappear.

within which transactions need to be executed come under this category. In this paper, we view a real-time database system as either memory resident or disk-resident 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 consider only soft and firm deadline transactions.

We propose a new real-time transaction scheduling algorithm that includes a novel transaction pre-analysis scheme and a cost conscious dynamic priority assignment policy in order to minimize some of the objective functions commonly used (e.g., the number of transactions that miss their deadlines, mean lateness). Our approach overcomes the problems inherent to pessimistic transaction analysis methods and non-adaptive EDF algorithms. In fact, our approach can be viewed as an adaptive optimistic/pessimistic algorithm which can cover the spectrum ranging from optimistic to pessimistic scheduling algorithms. Indeed, this property is responsible for its ability to adapt to a variety of transaction mix and workload.

The rest of the paper is structured as follows. Section 2 summarizes previous approaches to real-time scheduling. Section 3 provides motivation for our approach and the scope of the work presented in this paper. Section 4 presents details of transaction pre-analysis and its relevance to dynamic cost computation. Section 5 describes the dynamic cost used in the scheduling policy and describes the scheduling algorithm that uses the dynamic priority assignment policy. Sections 6, through simulation, compares our approach to EDF-HP (EDF priority assignment policy with High Priority conflict resolution method [AGM88b]) and EDF-CR (EDF priority assignment policy with conditional restart) for main memory and disk resident database for both soft and firm cases. Section 7 contains conclusion and future research.

2 Previous work

Figure 1 succinctly illustrates the taxonomy of real-time transaction scheduling approaches that have been proposed in the literature. Broadly, the approaches can be categorized into priority assignment (for real-time systems) and concurrency control (for real-time databases) based approaches.

Concurrency control based real-time database (time-critical database) scheduling algorithms combine various properties of time-critical schedulers with properties of concurrency control algorithms [AGM88a, AGM92, BMH89, C$^+$89, HLC91, 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 scheduling policies presented in [AGM92, HSRT91, HLC91, SZ88] fall into this category. These algorithms combine priority scheduling either with 2 phase locking or optimistic concurrency control (OCC) algorithms. EDF-HP (Earliest Deadline First with High Priority), LSF-HP (Least Slack First with HP), EDF-WP (EDF with Wait Promote), EDF-CR (EDF with Conditional Restart), AEDF-HP (Adaptive EDF with HP) [HLC91], Virtual Clock and Pairwise Value Function [SZ88] are combined with 2 phase locking. As a variation of single version 2 phase locking real-time multiversion CC (Concurrency Control) [KS91] has been introduced to increase concurrency by adjusting serialization order dynamically.

An OCC scheme with a deadline and transaction length based priority assignment scheme is presented in [HSRT91]. An OCC with adaptive EDF has also been proposed in [HLC91]. With OCC approach, a policy is needed to resolve the access conflicts during the validation phase. Some of the policies proposed are *commit* (always let the transaction being validated commit), *priority*

abort (abort the validating transaction only if its priority is less than that of each conflicting transaction), *priority wait* (wait for higher priority transactions to complete), and *opt-sacrifice* (restart the validating transaction if at least one of the transactions in the conflict set has a higher priority). OCC schemes display better performance for firm real-time transactions [HCL90a]. Lin and Son [LS90] have proposed a new concurrency control algorithm which is based on mixed integrated CC [BHG87] to adjust the serialization order dynamically.

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 (PCP) using priority inheritance with exclusive lock and read/write PCP have been proposed in [Sha88, SRSC91].

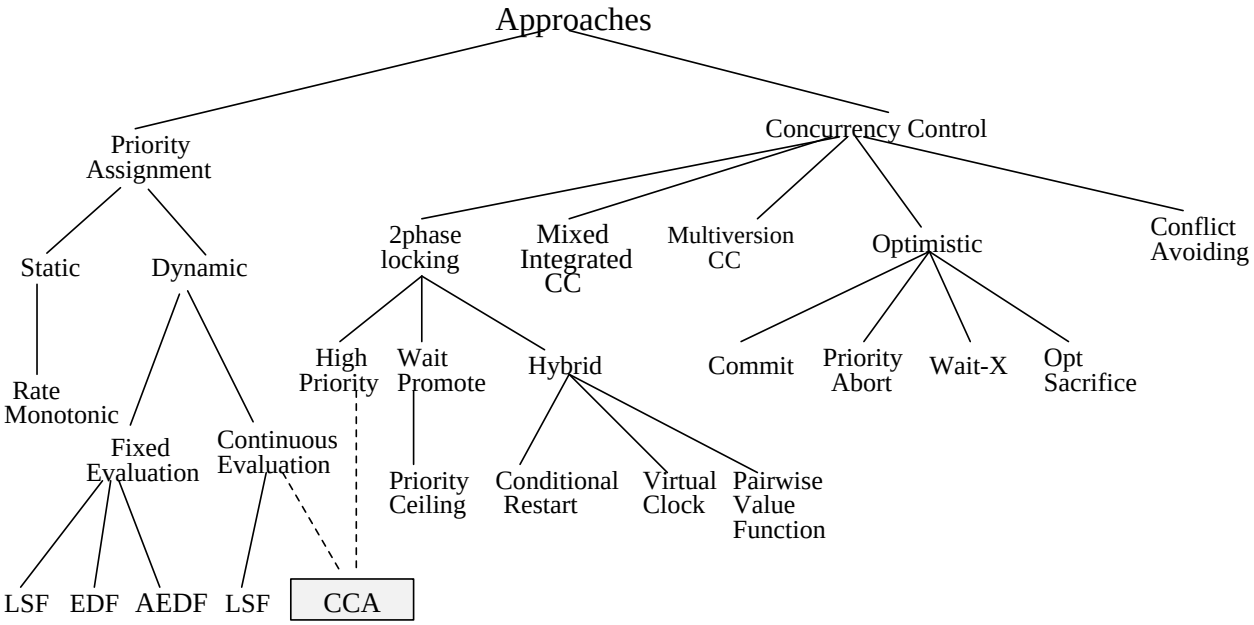


Figure 1: Taxonomy of real-time transaction scheduling approaches

2.1 Discussion

Some of the approaches such as EDF-HP and LSF-HP are straightforward combinations of earlier approaches. As all transactions are considered equal in traditional databases, priorities were assigned based on deadline for the purpose of abort in case of conflict.

EDF-HP scheme is overly conservative as only the priority information (based on deadline) is used for conflict resolution and for aborting a transaction. EDF-CR performs better than EDF-HP as it uses additional information in the form of an estimate of execution time and further allows transactions to complete if they do not force conflicting transactions to miss their deadline. Sometimes we would like to avoid aborting a transaction as we lose all the service time that it has received. The idea behind CR (conditional restart) conflict resolution method [AGM88b] is to estimate whether a transaction T_H which is holding the lock requested by the transaction T_R can be finished within the amount of time that T_R 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, where E_H and P_H are estimated execution time and the amount of service time of T_H respectively. If $S_R \geq E_H - P_H$ then we estimate that T_H can finish within the slack of T_R and we let T_H proceed to completion, release its locks and then execute T_R. This policy saves us from aborting and restarting T_H. If T_H cannot be finished in the slack time of T_R then we abort and restart T_H and run T_R. However, EDF-CR requires a good estimated execution time which is usually difficult to get due to database characteristics. In addition CR can be applied only when write-write conflicts occur.

OCC outperforms the lock-based counterparts mainly because their discarded transactions never restart other transactions for firm real-time transaction systems. Although variations of OCC algorithms perform better for firm deadline transactions, their superiority is derived, in essence, from the cost of restarts (both wasted and mutual) that do not occur on account of the semantics of firm deadline transactions. It is likely that the same behavior cannot be obtained for non-firm deadline transactions with the OCC and the simulation results in [HCL90b, HSRT91] show superiority of lock-based algorithms for soft deadlines.

The approach presented in this paper (CCA) can be viewed as a combination (as shown in Figure 1) of dynamic priority assignment with continuous evaluation that uses conflict as well as other runtime factors for determining the priority of transactions and HP conflict resolution.

3 Motivation For our Approach

A static priority assignment is not adequate in a real-time transaction processing system because it cannot consider the urgency of deadline. The conventional transaction pre-analysis (in terms of read- write-sets) is also inadequate because it is too pessimistic to use in real-time systems. The EDF-HP and LSF-HP are restrictive for some real-time applications because they ignore the blocking among transactions, the rollback, and restart effects. The effects of transaction rollback and restart overhead need to be used in conjunction with finer analysis of conflicts among transactions, which is the main contribution of this paper.

Under high level of resource and data contention, EDF-HP causes more transactions to miss their deadlines since they receive high priority only when they are close to missing their deadline [HLC91]. Also, EDF-HP causes many transaction aborts. If a higher priority transaction always aborts lower priority transactions, the performance is primarily sensitive to data contention. Furthermore, 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 Wait Promote conflict resolution method [AGM89]) has been proposed. However, EDF-WP causes too much waiting due to its nonabortive conflict resolution method. Several hybrid methods that use combinations of abortive and nonabortive methods [AGM88a, SZ88] make decisions about transaction blocking and rollback using additional information, such as effective service time, slack time based on an estimated execution time. However, it is difficult to estimate transaction execution time in real-time applications on account of the sources of unpredictability and its dependency on system load and transaction mix.

In this paper, we introduce the *cost conscious* approach (CCA) to real-time transaction processing that uses data requirement information rather than estimated execution time of transactions. The CCA includes the cost of aborted transactions in its priority calculation to solve EDF-HP's problem of excessive aborts and uses dynamic priority assignment with continuous evaluation method to solve nonadaptive behavior of EDF. Consider the following scenario. 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 subsequent 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 of the partially executed transactions, we have to consider several choices. If we use EDF-HP several partially executed transactions that conflict with T_a might have to be rolled back. If we consider the dynamic cost (time lost incurred by aborting conflicting transactions), we might realize that we lose too much time that has already been spent on the execution of the transaction that has the earliest deadline.

Several types of information are useful for designing real-time scheduling algorithms. Intuitively, we can do better if we have more knowledge and can use them to generate different scheduling policies. In order to get better results one has to use the available knowledge appropriately. Below, we broadly classify the knowledge and the corresponding 2-phase locking based approaches that have been proposed:

Type 0 No a priori knowledge. Only available timing information is deadline (EDF-HP [AGM89]).

Type 1 Deadline and data access pattern are available (CCA [HJC93]).

Type 2 Deadline and estimated execution time are available (EDF-CR, LSF-CR [AGM89]).

Type 3 Data access pattern and static transaction priorities are available (Priority Ceiling [SRL90]).

EDF priority assignment policy minimizes the number of late transactions when the system is lightly loaded. The performance, however, quickly degrades in overloaded systems. There have been several approaches to overcome this shortcoming and we can group them into two general approaches.

1. Use overload detection and management [HLC91].
2. Delay the build up of overload [AGM89, HJC93].

Overload detection mechanisms for real-time tasks are quite easy because we assume that we know all required information such as arrival time, execution time, resource requirement and deadline [DLT85]. For database applications knowledge about transactions are usually not available or not correct due to database characteristics. AED (Adaptive Earliest Deadline) [HLC91] priority assignment for firm deadline uses feedback mechanism that detects overload conditions and modifies transaction priority assignment policy accordingly. AED uses past history (that has been gathered dynamically) rather than a priori knowledge to detect overload.

Another group of approaches [AGM89, HJC93] uses additional information to improve EDF-HP further. Even though these approaches do not have a specific overload management mechanism their methods improve the performance by delaying overload condition. The idea here is to save valuable system resources by not aborting partly executed conflicting transactions blindly.

We believe that well estimated execution time is most important information for RTDBS. However, it should be combined with system load appropriately to get a good estimation for soft deadline. Estimated execution time of a transaction can be roughly calculated with estimated resource time of a transaction, and its error is bounded by its deadline for firm real-time system as tardy transactions are removed from the system. However, the estimation error is unbounded for

soft real-time system because tardy transactions are not removed from the system. This indicates that type 2 knowledge for soft real-time system can only be obtained by combining type 1 knowledge and a proper load characterization and detection mechanism. In this paper we only use type 1 knowledge but we are investigating a method that can combine type 1 and type 2 together with proper load detection mechanisms.

In order to motivate the applicability of CCA, we apply it to an example drawn from [XP90] (Figure 2). For this example, we assume that all data are memory resident, transactions have soft deadlines, data conflict occurs at the beginning of transactions, and all possible *valid schedules* were made based on strict 2-phase locking. Some of the schedules can be explained using EDF-HP, LSF-HP, EDF-CR, FCFS, non-preemptive and CCA while the others randomly generated.

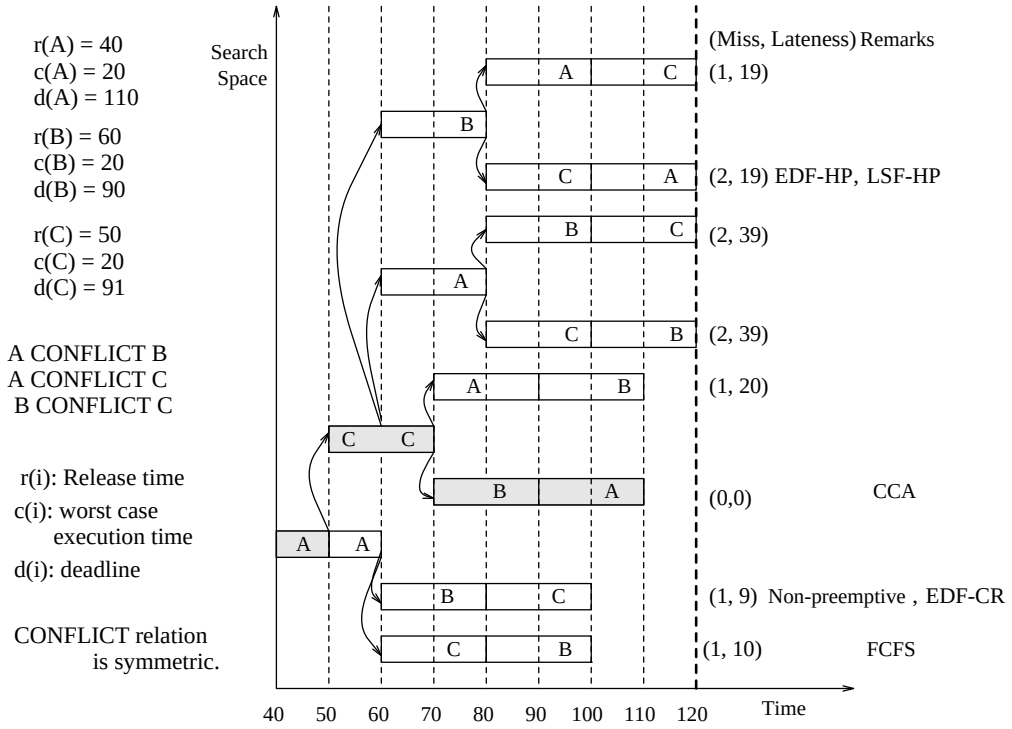


Figure 2: Valid Schedules for soft deadlines

In Figure 2, CCA works as follows: At time 40 transaction A arrives and it is the only candidate. At time 50 transaction C arrives and C has earlier deadline than that of currently executing transaction A. According to CCA the urgency of transaction C over transaction A (19 time unit) dominates C's dynamic cost (time lost incurred by aborting A, 10 time unit). Thus higher priority transaction C is executed and A is aborted. At time 60 transaction B arrives and B, C, and A are in the ready queue in the increasing order of deadline, and transaction C is partially executed. Even though transaction B has earlier deadline than that of C, transaction B does not have higher priority than transaction C because B's dynamic cost (10 time unit) dominates the urgency of its deadline (1 time unit) over C in CCA. Thus the highest priority transaction C is executed in the CCA scheduling.

EDF-HP: At time 50 transaction C arrives and C has a earlier deadline than that of currently executing transaction A. EDF-HP aborts transaction A and executes C. At time 60 B arrives and EDF-HP abort C and execute B because B has the earliest deadline. After the completion of B

transaction C and A are executed serially according to their deadline orders.

If we assume that estimation is the same as worst case execution, EDF-CR works as follows: At time 50 transaction C arrives and C has an earlier deadline than that of currently executing transaction A. EDF-CR keep executing A because A's remaining execution time (10 time unit) is less than the slack time of C (21 time unit). At time 60 A finishes its execution and B arrives at the same time. EDF-CR execute B and C according to their deadline orders.

Our approach is predicated upon folding a number of realistic dynamic information, in addition to traditionally used factors, into the formula used for computing the priority of transactions as well as for performing conflict resolution where necessary. As we know, priority assignment governs CPU scheduling whereas conflict resolution determines which transactions from among a set will be given access to data. Hence, dynamic information can be used in assigning priority or for resolving data conflict or for both. Furthermore, some of the dynamic information (e.g., effective service time) can only be computed at run time whereas some others (e.g., access pattern, conflict information) can be obtained by pre-analyzing transactions or transaction groups.

For the remainder of the paper, 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 group of transactions. Further, we assume that we have pre-analyzed these groups as described below. 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. In order to calculate the approximate dynamic cost which will be used in our priority assignment policy, we analyze transaction programs.

4 Transaction Pre-analysis

In real-time applications, it is unlikely that queries/transactions are *ad hoc* in nature. It is more likely that a transaction is an instance of a pre-defined (or canned) set of transactions with different set of parameters. In other words, the structure of transactions as well as the types of data accessed are likely to be known if not the actual data instances. Under these assumptions, it is reasonable to perform transaction pre-analysis to obtain as much information about the transaction structure and the data type accessed by a transactions instance as possible.

The read- and write-sets (termed data set) typically used for transactions are the result of a pre-analysis that assumes, conservatively, that all elements in the read- and write-sets might be accessed by a transaction. The set of data items that a transaction of some type *might* access is called its *data set*. This assumption is indeed true if the transaction were to be a sequential piece of code without any branch points as part of the transaction. The presence of control structures within a transaction reduces the set of data actually accessed and is not taken into consideration. A particular execution of a transaction is likely to actually access only a 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 3:

Suppose that T_{A1} executes program A (transaction type) and T_{B1} executes program B. If T_{A1} executes the `If` statement and $W > 100$, T_{A1} and T_{B1} conflict. Otherwise, T_{A1} and T_{B1} do not

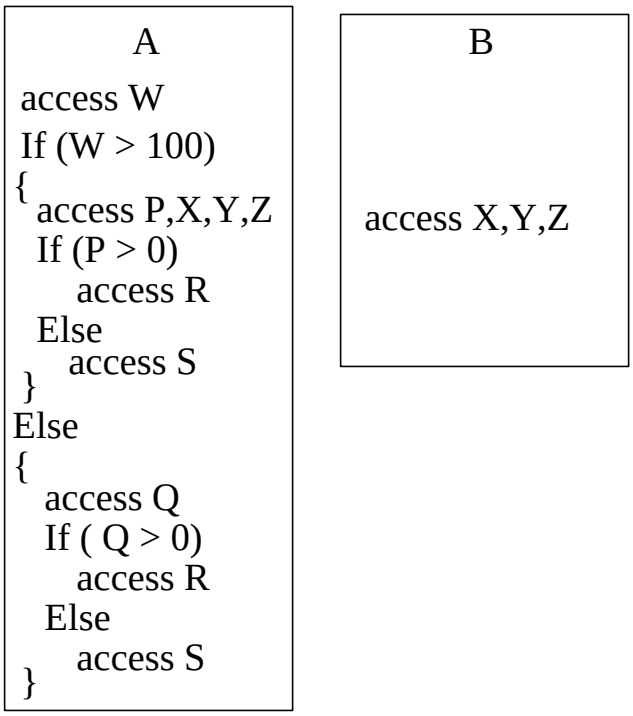


Figure 3: Sample Transaction programs

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[2], (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 4 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 denotes the instance of a transaction and superscript represents the label of a decision point) conflicts with T^B_{B1}, T^{Aa}_{A1} conflicts with T^B_{B1}, but that T^{Ab}_{A1} does not conflict with T^B_{B1}.

In the Figure 4, before a transaction of type A reaches the decision point A, it might conflict with another transaction of type B (if at node A, it branches to node Ab), or it might not conflict with a transaction of type B (if node A takes the other branch). Suppose the transaction of type A makes the branch to node Ab. At this point we are certain that it conflicts with the other transaction. Based on this we define different flavors of data conflict that can be obtained from a pre-analysis. 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 data sets. If two transactions might or

[2]Although, a loop-free program is a directed acyclic graph, we use a tree representation for the sake of simplicity. It is always possible to lump a loop into a node sacrificing some granularity.

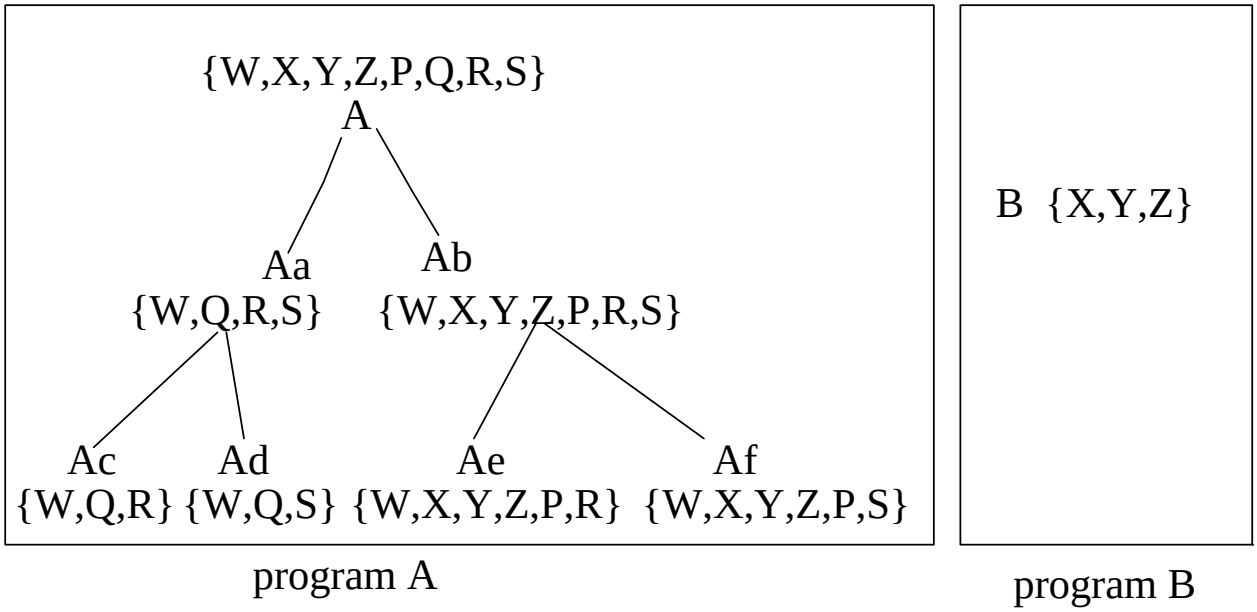


Figure 4: Transaction Tree of sample programs

might not conflict based on their future execution, then they *conditionally conflict*.

Suppose that transaction T_N^P conflicts with transaction T_M^Q, and T_N^P is scheduled to execute. If T_M^Q has not yet accessed any data items that T_N^P might access, then there is no need to roll back T_M^Q, we only need to block it. In this case, we say that T_M^Q is *safe* with T_N^P. If T_M^Q has accessed a data item that T_N^P will access, then T_M^Q is *unsafe* with T_N^P and needs to be rolled back (strategy for choosing a transaction for rollback is discussed in a later section) when T_P^N accesses the conflicting item. Finally, T_M^Q is *conditionally unsafe* with T_N^P if T_M^Q might be safe or unsafe with T_N^P, depending on T_N^P's execution beyond the current point. Before, we define these concepts rigorously, we illustrate how they are used.

4.1 Use of Pre-analysis Information

Concurrent execution of transactions gives rise to partially executed transactions in the system waiting for resources (including cpu time) for completion. The resources consumed up to a given point of execution (termed the cost or the actual cost) is known to the system. However, the resources required to complete the transaction need to be estimated (termed the heuristic) to make decisions that optimizes the metric used by the system. This problem is not different from state space search algorithms (e.g., A*) used to find an optimal (or a suboptimal solution within certain bounds). A weighted combination of cost and heuristic is used to determine the next step of the search algorithm. The admissibility properties of state space search stipulate that the heuristic used be a lower bound on the actual value. However, the deviation from the optimal strategy is dependent on how close the heuristic is compared the actual value.

Several types of information can be gathered from the pre-analysis. The decision points are useful as points where dynamic information is transformed into priority information. The availability of data sets for decision points (or even an estimate of the heuristic) will help in making a decision on the abort of a transaction. Furthermore, the decision points can also be used as checkpoints for

partial rollback in case of a conflict instead of always doing a complete rollback. The partial rollback reduces the resources wasted for dealing with data conflicts. Pre-analysis information can also be used for scheduling I/O operations to avoid or minimize conflicts and to reduce non-contributing resource usage.

4.2 Conflict Determination

Once the data items a transaction accesses between decision points is known, the conflict and safety relationships can be inferred in a straightforward manner.

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

accesses(T_N^P) Set of data items that a transaction N at node P accesses between P and its next decision point or the end of transaction.

hasaccessed(T_N^P) Set of data items that a transaction N at node P has accessed up to this point from the beginning of the transaction.

mightaccess(T_N^P) Set of data items that a transaction N at node P might access from P till its completion.

leaves(T_N^P) Set of leaves of the subtree rooted at node P of a transaction N.

We now give precise definitions of the conflict and safety relationships, which also provide a method for computing these relationships. 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

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

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

- Leaf transactions T_N^P and T_M^Q *conflict* if and only if mightaccess(T_N^P) $\cap$ mightaccess(T_M^Q) $\neq \phi$
- Transactions T_N^P and T_M^Q *conflict* iff $\forall_{p \in leaves(T_N^P)}$ $\forall_{q \in leaves(T_M^Q)}$ mightaccess(T_N^p) $\cap$ mightaccess(T_M^q) $\neq \phi$.
- Transactions T_N^P and T_M^Q *conditionally conflict* if and only if $\exists_{i,j \in leaves(T_N^P)}$, $\exists_{m,n \in leaves(T_M^Q)}$ such that mightaccess(T_N^i) $\cap$ mightaccess(T_M^m) $\neq \phi$ and mightaccess(T_N^j) $\cap$ mightaccess(T_M^n) $= \phi$.
- 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) $= \phi$.

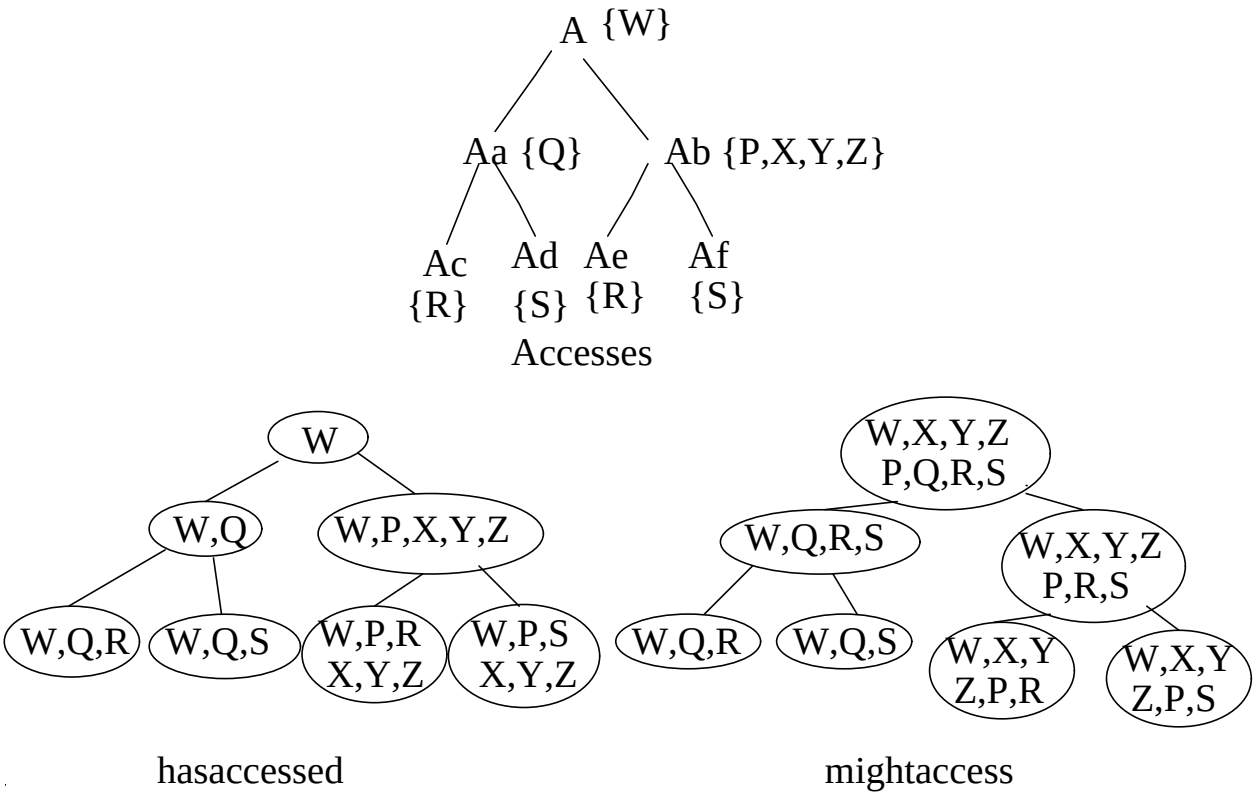


Figure 5: Values of accesses, hasaccessed, and mightaccess for the program A

- 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 real-time database systems to trade space for improved performance.

Although the mightaccess includes the hasaccessed set at each node, this is done to facilitate checking between transactions at various stages of execution. Conceptually, it is easier to interpret in terms of cost and heuristic values as a conflict or a safety check is done for transactions that have not conflicted so far. It is evident from the above that the accuracy of the heuristic is important for computing various relations defined above.

Another approach proposed for gathering the necessary information is by using the two-phase (or pre-fetch) execution suggested by [Ram93]. A transaction is run primarily for determining the computational demands of that transaction. This approach can also yield pertinent hasaccessed and mightaccess data sets described in this paper.

5 Cost Formulation

In a real-time database, irrespective of whether it is memory resident or disk resident, the (wall clock) response time has two distinct components: T_{static}, the time needed to execute a transaction in an isolated environment and $T_{dynamic}$, the time spent in waiting (both I/O and concurrency related) as well as abort/restart overhead. T_{static} is dependent on the semantics of the transaction (e.g., data values accessed and branch points) and is relatively straightforward to estimate. $T_{dynamic}$, on the other hand, is dependent on the current state 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 database context, $T_{dynamic}$ is extremely difficult to compute or even estimate as it is not only dependent on the resources consumed so far but also on the resources required for its completion which may be affected by future events. Furthermore, $T_{dynamic}$ is sensitive to the transaction mix and can vary considerably when the transaction mix changes. Nevertheless, the inclusion of an approximate dynamic cost as part of the strategy for meeting timing requirements is likely to perform better than those where the dynamic information is not included at all.

5.1 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] different priority assignment policies are used for CPU and data conflicts. However, the use of different priority assignment policies for different resource types might lead to more instances of priority reversal leading to deadlocks [BMH89]. In CCA, we use a single priority assignment policy for CPU and data conflicts.

5.1.1 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 to capture all the dynamic features as the transaction progresses.

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

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

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

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 aborted and rolled back to execute T_a to its commit point without interruption.

The notion of the penalty of conflict, described above, is introduced into the our CCA dynamic priority computation formula as follows. 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 (termed penalty-weight), the weight given to the dynamic cost, can be changed to vary the emphasis between deadline and penalty of conflict. Note that if the transactions are executed serially, then the penalty of conflict does not exist and hence the priority is determined only by the deadline value.

5.1.2 Conflict Resolution

There are three types of resources in the system: CPU, disk and data. The active resources in real-time database systems are the CPU and the disk, whereas data is the passive resource. We apply different scheduling disciplines to different resources as there effect on transaction execution is different.

Data conflict If there is a data conflict between two transactions, a priority-based wound-wait strategy [BMH89] is the simplest to implement. The Conditional Restart algorithm with an estimated execution time [AGM88a] has been proposed to avoid needless aborts and rollback. The idea of HP [AGM88b, AGM89], which is the same as the priority-based wound wait strategy [BMH89], is to resolve a conflict in favor of the transaction with the higher priority. In our approach, we apply HP conflict resolution method for data conflicts.

CPU conflict Even in 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. If the scheduler cannot be invoked for any reason (e.g., Real-time UNIX [FF91]), the highest priority transaction can be selected from among transactions that are in the ready queue or are currently running. When an executing transaction finishes, all transactions blocked by the resources that is held by the currently running transaction wake up and move to the ready queue. Then, the highest priority transaction is chosen as the next one for execution. In our approach we assume that whenever a new transaction arrives, or a running transaction finishes, or an I/O wait occurs, the scheduler is invoked immediately.

I/O conflict If the real-time database contains disk resident data, a transaction might perform many I/O waits during its execution. Several real-time I/O scheduling methods have been proposed [AGM89, C$^+$89] in order to reduce I/O wait. In our approach we use FCFS I/O scheduling method.

Disk I/O introduces new problems in real-time transaction scheduling. There are several choices when I/O wait occurs and we have considered the following 3 choices:

1. Pick the highest priority blindly.
2. Pick the highest among transactions that does not conflict or conditionally conflict with partially executed higher priority transaction.
3. Pick the highest among transactions that does not conflict or conditionally conflict with partially executed transactions.

Among the above, we found that the second one comes out as the best for soft real-time transactions. Consider the following scenario: Transaction T_1 is blocked and is waiting for an I/O 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* 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 I/O 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 I/O wait using the pre-analyzed information.

A **noncontributing execution** is defined as a lower priority transaction's execution during the I/O wait of higher priority transaction that has to be rolled back when the higher priority transaction finishes its I/O.

5.1.3 Algorithm

Below, we describe the components of the scheduling algorithm (using pseudo code) proposed in this paper which is based on the notion of cost incurred due to conflicts. The function "I/Owait-sched" is invoked whenever a transaction blocks waiting for I/O completion. This function reduces the noncontributing execution and hence avoids rollback by using transaction conflict relationships.

```
Function I/Owait-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 higher priority transactions
        then
           return the one with
           the highest priority among them;
        else
           return NIL;
end
```

The procedure "tr-arrival-sched" is invoked whenever a new transaction 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. The penalty-weight (ω) introduce in the priority formula is used to weigh the contribution of penalty of conflict on the value of the priority value computed. A penalty-weight value between 0 and a large integer can be used.

```
Function Pr
begin
```

```
    calculate penalty of conflict
    return( - (deadline + penalty-weight * penalty of conflict));
end
```

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

```
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
```

The HP conflict resolution scheme is a deadlock prevention mechanism if it is combined with a fixed or dynamic priority assignment, which is statically evaluated. If the HP conflict resolution is combined with dynamic priority assignment with a continuous evaluation method (e.g., LSF) it can cause deadlock due to *priority reversal*. CCA uses a dynamic priority assignment with continuous evaluation method in order to adapt to the changes of systems load effectively. Thus it is prone to deadlocks and a deadlock detection mechanism is used by maintaining the wait-for graph.

5.1.4 Firm Deadline

Firm real-time transactions are those which need not be considered any more if their deadlines are not met, as there is no value in completing the transaction after its deadline. We can drop the transaction that already missed its deadline after its deadline (*observant* approach) or the transaction that will miss its deadline before its deadline (*predictive* approach) from the firm real-time systems [AGM92]. In this paper we only consider a observant approach that drops a transaction immediately when its deadline is reached. Although we delimit our discussion to *observant* approach in this paper, we can readily extend CCA to use *predictive* approach if we can use best case (as opposed to worst case) execution time to assign intermediate deadlines to branch points. Figure 6 shows that transaction T_1 should finish its first branch point by T_d - (b+c), because the best case execution time between first branch point to the end of the transaction is the sum of b and

c. If transaction T_1 misses any of its intermediate deadlines, we can drop the transaction without waiting for the expiration of final deadline. Dropping transactions that cannot finish within their deadlines as early as possible improves the performance of firm real-time transaction systems, not only by not wasting system resources [AGM92] but also by reducing wasted restarts.

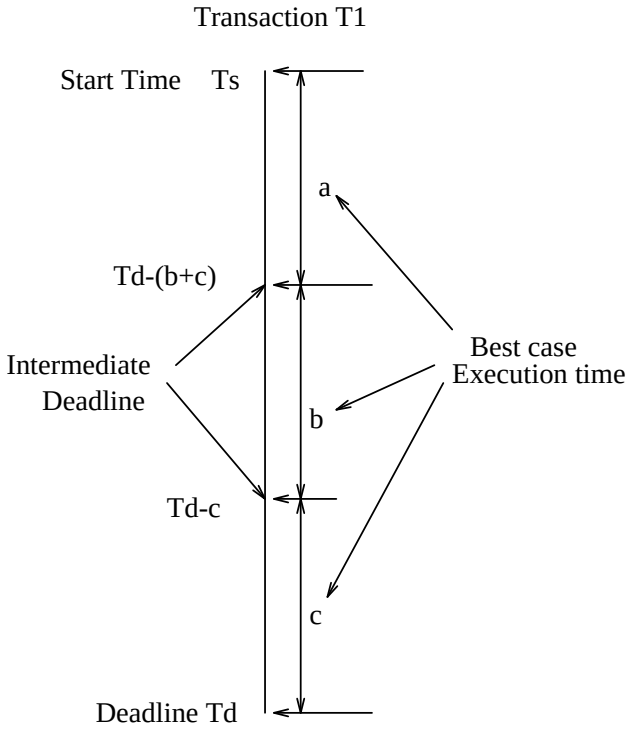


Figure 6: Transaction deadline and intermediate deadline

5.1.5 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 (statically evaluated) 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. Thus it might have deadlock on disk resident databases. However, unlike LSF, CCA does not have priority reversal for a single CPU main memory databases.

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

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

A secondary transaction is defined as the transaction T_S that is not chosen as T_H in the system.

Proof: 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.[3] Also, the priority of any transaction in the ready queue that does not 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 secondary 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 1, there is no priority reversal and an abort occurs between conflicting transactions only.

6 Performance Evaluation

In order to evaluate the performance of the CCA algorithm described in this paper, two simulations of a real-time transaction scheduler were implemented using C language and SIMPACK simulation package [Fis92] for main memory resident databases and disk resident databases. The parameters used in the simulations are shown in Table 1.

Parameter	Meaning
db_size	Number of objects in database
max_size	Size of largest transaction
min_size	Size of smallest transaction
i/o_time	I/O time for accessing an object (read/write)
cpu_time	CPU computation per object accessed
disk_prob	Probability that an object is accessed from disk
update_prob	Probability that an object accessed is updated
min_slack	Minimum slack
max_slack	Maximum slack
restart_time	Time needed to rollback and restart
penalty_weight	Weight of penalty of conflict

Table 1: Parameters and their meanings

In these simulations, 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). The number of

[3] 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.

objects updated by a transaction is chosen uniformly from the range of *min_size* and *max_size* and the actual database items are chosen uniformly from the range of *db_size*.

After accessing an object a transaction spends *cpu_time* in order to do some work with or on that object and then it accesses the next object.

The assignment of a deadline is controlled by the resource time of a transaction and two parameters *min_slack* and *max_slack* which set, respectively, a lower and upper bound of percentage of slack time relative to the resource time. A deadline 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*.

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

Disk accesses for disk resident database are controlled by *disk_prob* when a transaction reads an object. The use of disk_prob to some extent models data maintained in the buffer. At commit time, objects that have been updated are flushed. The parameter *update_prob* controls the number of data that should be written at the commit time. We use *restart_time* for modeling the rollback of a transaction and its restart. The restarted transaction will access the same data objects. The weight associated with the *penalty of conflict* is controlled by *penalty_weight*.

In our performance evaluation we measure three performance metrics (defined below) commonly used in the literature for RTDBS: i) miss percent, ii) restart rate, and iii) mean lateness.

$$Miss\ Percent\ = \frac{Total\ number\ of\ transactions\ that\ missed\ the\ deadline}{Total\ number\ of\ transactions\ that\ entered\ the\ system} \times 100$$

$$Restart\ Rate\ = \frac{Total\ number\ of\ restart}{Total\ number\ of\ transactions\ that\ entered\ to\ the\ system}$$

$$Mean\ Lateness\ = \frac{\sum_{Ti \in tardy\ transactions}(completion\_time(Ti) - deadline(Ti))}{Total\ number\ of\ transactions\ that\ entered\ to\ the\ system}$$

6.1 Simulation of main memory database

Figure 7 shows the open network model of RTDBS for main memory database. In this model the processor is taken into account implicitly. In this simulation we have a single processor and a memory resident database. We do not consider durability property of a transaction here in order to see the effects of transaction scheduling and concurrency control methods more clearly. Thus the resource time of a transaction only depends on *cpu_time* and the number of objects a transaction accesses.

$$resource\ time = number\ of\ objects \times cpu\_time$$

The values of parameters used in this simulation are shown in Table 2. The value of *db_size* has been chosen to increase data conflict among transactions. 10,000 transactions were executed for each experiment.

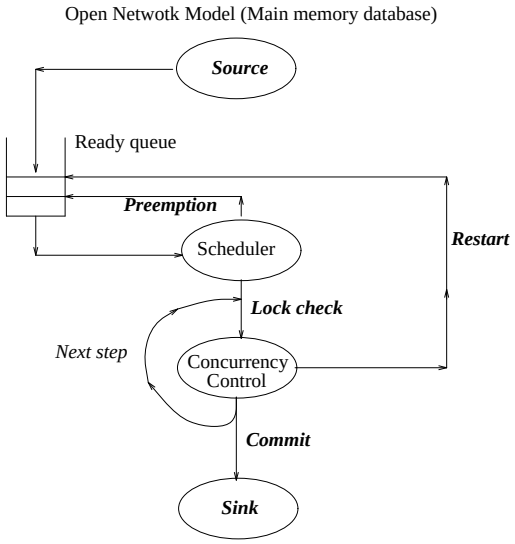


Figure 7: Open Network Model for main memory database

Parameter	Value
db_size	250
max_size	24
min_size	8
cpu_time	10 ms
min_slack	50 (%)
max_slack	550 (%)
restart_time	5 ms
penalty_weight	1

Table 2: Base parameters for main memory database

6.1.1 Effect of Arrival Rate (soft deadline)

In this experiment, we varied arrival rate from 1 tr/sec to 7 trs/sec with the base parameters shown in Table 2 and measured the miss percent, the number of restarts per transaction, and mean lateness for both EDF-HP and CCA. With the base parameters the maximum capacity of the system (assuming no blockings and aborts) is:

$$\frac{10\ ms}{object} \times \frac{16\ objects}{transaction} = \frac{160\ ms}{transaction} = 6.25\ transactions/second$$

If we consider the effects of blocking and aborting (dynamic factors) the capacity of the system will be much less than the maximum capacity of the system. Figure 8 (a) shows the effect of arrival rate on the percentage of transactions that miss their deadline. The system gets heavily loaded beyond an arrival rate of 4.5 trs/sec. Figure 8 (b) shows the effect of arrival rate on the restart rate of transactions.

CCA shows better performance as compared to EDF-HP especially when the arrival rate is between 3 and 5.5 trs/sec. Within this arrival range CCA also shows much less number of transaction restarts. Generally, less number of transaction restarts does not guarantee better performance but CCA reduces *expensive restarts* to achieve better performance. This phenomenon can be seen clearly in the multiclass experiment presented later. Observe that for the base parameters shown in Table 2, the number of restarts climbs steeply up to the arrival rate of 4 and then declines sharply (Figure 8 (b)). The reason for sharp decline is that beyond a specific arrival rate, it is less likely that an arriving transaction will have an earlier deadline than the currently running transaction. After the peak point in Figure 8 (b), it is usually the case that the currently running transaction arrived a long time ago, but could not get system services due to the heavy load on the system (most of the dynamic factors in heavily loaded situation are *arrival blockings* rather than *preemption blockings* and aborts [Tay92]). Thus, fewer transactions are preempted and there are fewer opportunities for restarts [AGM88a].

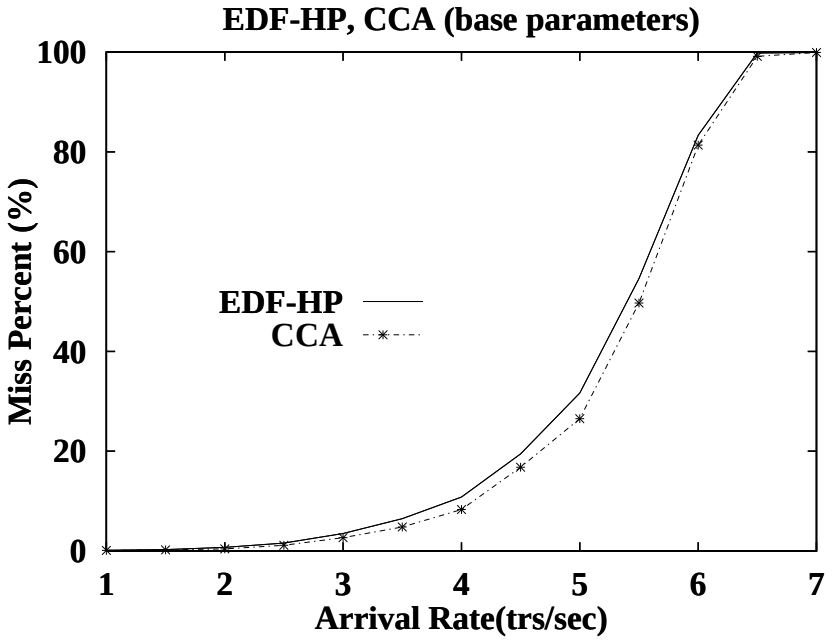
In order to observe the correlation between the maximum capacity, arrival rate, and the behavior of the performance metrics, we performed an experiment by doubling the capacity. We expected that the arrival rate at which the system gets heavily loaded will also shift with the maximum capacity of the system. In order to double the maximum capacity of the system we assigned 5 ms and 2.5 ms for *cpu_time* and *restart_time* respectively and assigned the same values for remaining parameters. With these parameters the maximum capacity of the system is:

$$\frac{5\ ms}{object} \times \frac{16\ objects}{transaction} = \frac{80\ ms}{transaction} = 12.5\ transactions/second$$

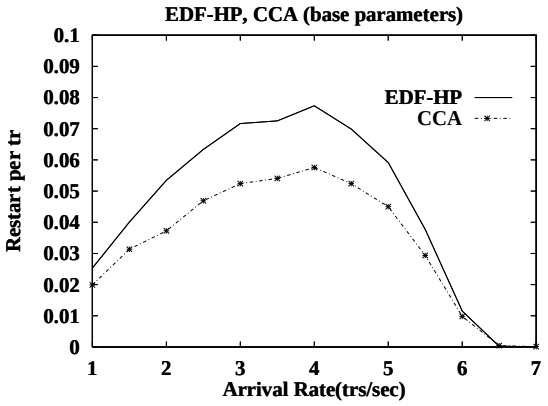
In this experiment, we varied arrival rate from 1 tr/sec to 14 trs/sec and the results are shown in Figure 9 (a) and Figure 9 (b). As expected, the mean lateness shown in Figure 8 (c) on a logarithmic scale indicates improvement over that of EDF-HP.

6.1.2 Effect of multiclass (transaction mix)

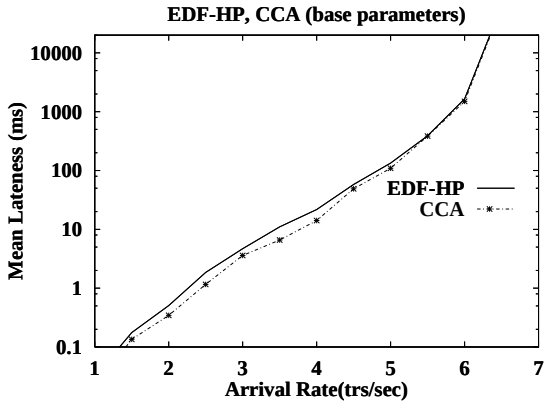
In this experiment, the arriving transactions are divided into three classes (class 0, 1, and 2) and assigned different values of *cpu_time* – 1 for class 0, 10 for class 1, and 100 for class 2. We assigned 1 ms as *restart_time* for all classes because the resource times of class 0 transactions are between



(a) Miss percent of EDF-HP,CCA

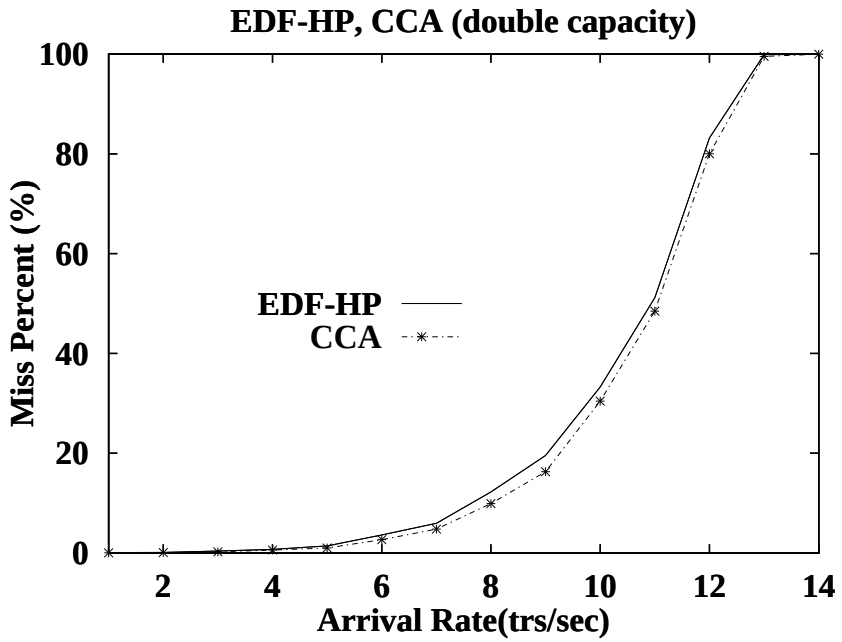


(b) Restart rate

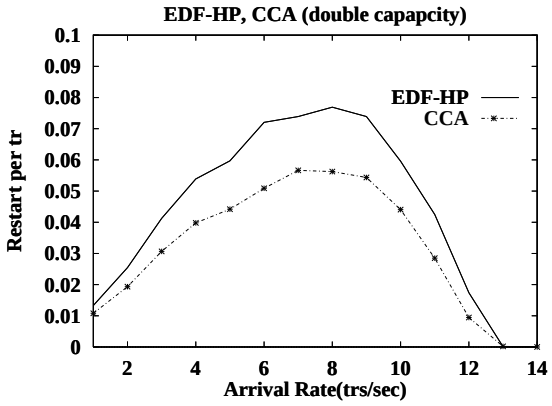


(c) Mean Lateness

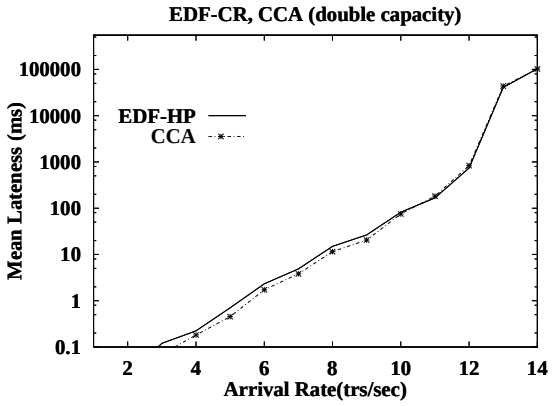
Figure 8: Result of experiment on main memory database



(a) Miss percent of EDF-HP,CCA



(b) Restart rate



(c) mean lateness

Figure 9: Result of experiment for double the capacity

8 ms and 24 ms. The other parameters are the same as that of the previous experiment. Thus data contention remains the same but the amount of resource time for each class is different. With these assignments a lower class (the lowest is class 0) transaction has a shorter resource time. As a result it has a shorter slack time. The maximum capacity of the system (disregarding blockings and aborts) is:

$$\frac{\frac{1+10+100}{3}}{object} \times \frac{16\ objects}{transaction} = \frac{592\ ms}{transaction} = 1.7\ transaction/second$$

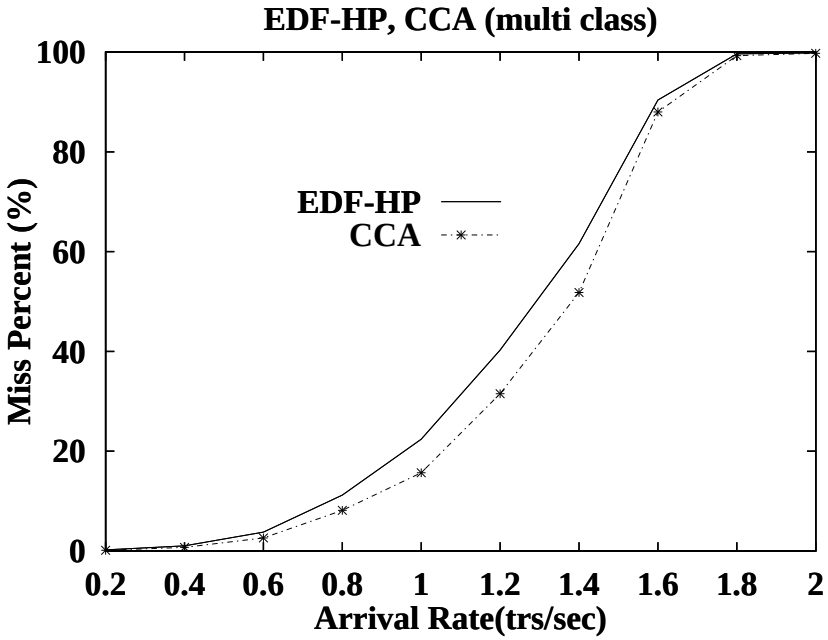
Different assignments of *cpu_time* for each transaction class creates a lot of variance in the transaction execution time (the execution time of transaction varies from 8 ms to 2400 ms). Therefore, there will be more chances for transaction preemption. Figures 10 (a) and 10 (b) show the results of this experiment. With the variation of *cpu_time* there is higher possibility that an arriving transaction will have an earlier deadline than the currently executing transaction. Thus restart rate per transaction of this experiment is increased for both approaches as can be observed from Figures 8 (b) and 10 (b). CCA shows better performance especially when the arrival rate is between 0.6 and 1.4 trs/sec. Within this arrival range CCA shows much less number of transaction restarts as well. CCA reduces *very expensive restarts* to achieve better performance in the multiclass situation. This experiment also indicates the adaptive nature of the CCA approach in which the dynamic cost changes as the transaction mix changes and reduces the effect of deadline accordingly.

Another metric of comparison for this experiment is to observe miss percent for each class. In this experiment data contention is the same for all classes but their active resource requirements are different because the transactions belonging to classes 1 and 2 require more *cpu_time* to process their data objects. The relative difference of miss percent of each class is reduced after arrival rate 1 for both approaches (Figure 11). The reason being that after this point preemption of transactions is reduced and execution behavior is more serialized.

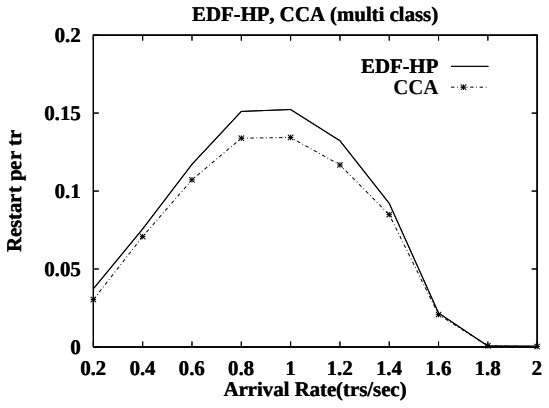
We plot miss percent for each class from arrival rate of 0.6 trs/sec to 1.4 trs/sec in Figure 11 (miss percent is too small to plot when the arrival rate is less than 0.6 trs/sec). Their relative difference is reducing when the arrival rate is greater than 1.4 trs/sec. From Figure 11, we can see that EDF-HP blindly favors lower class transactions. Thus EDF-HP causes very expensive restarts by aborting transactions that consumed a lot of resources. CCA also favors lower classes but CCA avoids expensive restarts by not aborting transactions that consumed a lot of resources. In Figure 11 miss percent of class 0 is higher than class 1 in our experiment. The reason is that class 0 transactions are very vulnerable due to their relatively small absolute slack time.

We expected that there would be less discrimination against long running transactions in CCA than EDF-HP because CCA implicitly considers the effective service time of a transaction as we can see it in Figure 11. Discrimination against long running transactions in RTDBS is discussed in [PLJ92]. In their experiment each class requires different ranges of object number. Thus each class has different level of data contention and resource time. In our experiment, however, each class only has different level of resource contention. That is the reason why their experiment shows more discrimination against long running transactions. Also, the formula used for priority computation currently does not distinguish between transaction classes. This can be easily included in the formula that computes penalty of conflict.

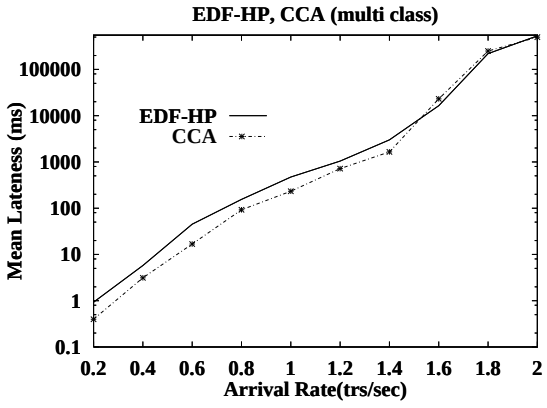
According to our previous experiments [HJC93], CCA show much better performance especially when there are high variances of execution time among transactions by not aborting transactions



(a) Miss percent of EDF-HP,CCA

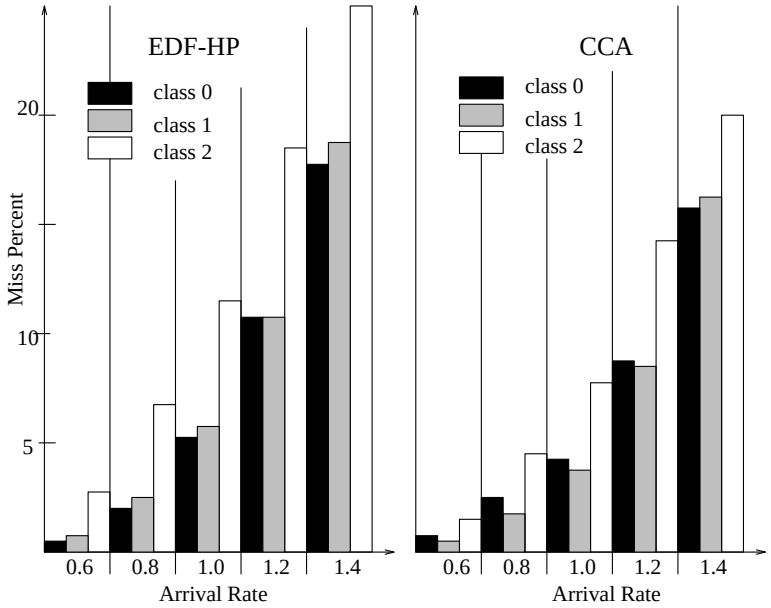


(b) Restart rate



(c) Mean Lateness

Figure 10: Result of multiclass experiment



	EDF-HP					*CCA*				
Arrival Rate(trs/sec)	0.6	0.8	1.0	1.2	1.4	0.6	0.8	1.0	1.2	1.4
Miss Percent(Class 2)	2.6	6.84	11.52	18.37	24.91	1.32	4.28	7.72	14.22	20.09
Miss Percent(Class 0)	0.49	2.03	5.14	10.97	17.76	0.74	2.28	4.16	8.70	15.44
Class 2 / Class 0	5.3	3.36	2.24	1.67	1.40	1.78	1.87	1.85	1.63	1.3

Figure 11: Miss percent for each class and Proportion of class 2 to class 0

that already consumed a lot of resource time.

6.1.3 Comparison with EDF-CR (soft deadline)

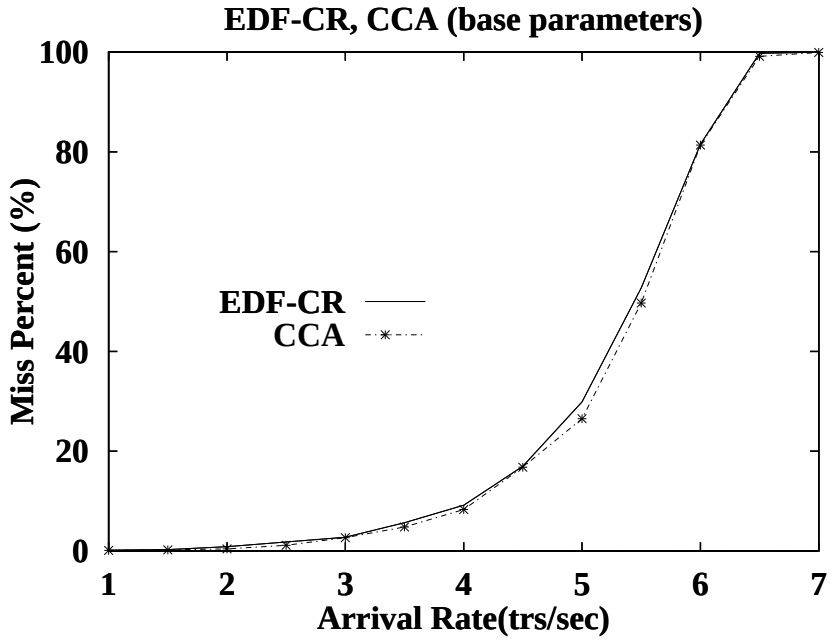
EDF-CR [AGM92] uses *estimated execution time* as an additional information to improve EDF-HP. The most difficult part of EDF-CR is in computing good estimates of execution time of transactions as it is largely dependent on the system load. In this experiment we used *resource time* as the *estimated execution time*, which only includes static information. By not including dynamic information, the estimated execution time in this experiment is relatively underestimated. The comparisons between EDF-CR and CCA are shown in Figure 12. In this experiment EDF-CR shows fewer restarts as compared to its performance with respect to CCA. EDF-CR uses the slack time of a higher priority transaction when a RTDBS make a decision whether it should abort the lower priority transaction or block the higher priority transaction. If we underestimate the execution time of transactions the system tends to block higher priority transactions rather than abort conflicting lower priority transactions. If the execution time of transactions are longer than the estimations EDF-CR blocks more urgent higher priority transactions for less urgent lower priority transactions by making a wrong judgment. The system assumes that higher priority transactions have enough slack time even when it is not the case. In the extreme case, EDF-CR will reduce to a nonabortive approach. That explains the reason why EDF-CR shows less number of restarts in spite of showing a slightly poorer performance than CCA in this experiment for the miss percent and mean lateness. Note that mean lateness is plotted using the logarithmic scale.

6.1.4 Effect of firm deadline

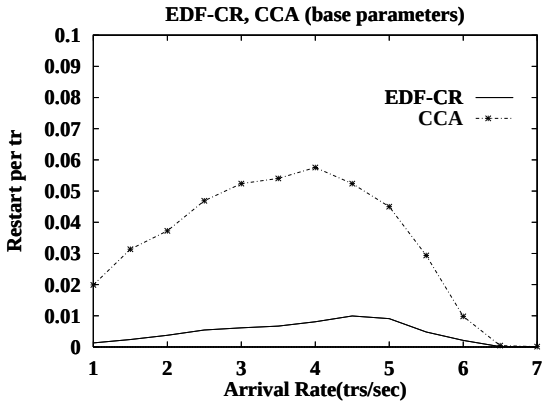
In this experiment we dropped a transaction immediately when it missed its deadline. Generally, the firm deadline case shows better performance than soft case in terms of transaction miss percent. The results of this experiment are shown in Figures 8 (a) and 13 (a). Removing tardy transactions from the system as soon as possible will help remaining transactions, not only by avoiding waste of system resources but also by reducing wasted restarts.

CCA shows marginal performance improvement over EDF-HP for firm real-time systems (as expected, shows less improvement than soft real-time systems in Figure 13). The reason why CCA works better for soft deadline transactions than firm deadline transactions is obvious. Actually CCA improves performance by reducing *expensive restarts*. In RTDBS the effect of reduced restarts do not disappear until the system has idle period. Even though there is little idle period in soft real-time systems when arrival rate is high but there might be some idle period in firm real-time systems by removing missed transactions. The effects of reduced *expensive restarts* in soft deadline case lasts longer than firm case.

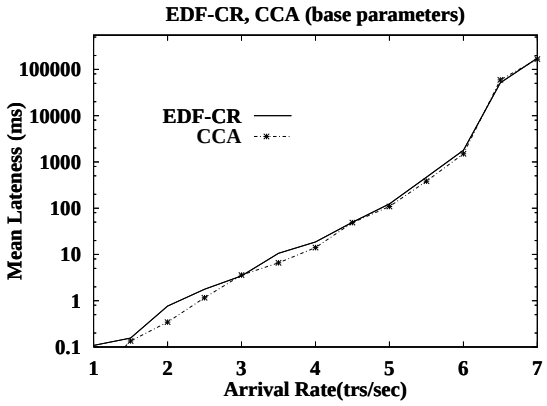
Intuitively, if we drop transactions that missed their deadline from reconsideration, we expect that the restart for the firm case decreases more rapidly than that of soft case as dropped transactions never cause transaction aborts in a firm real-time system. However, the number of restarts in a firm real-time system decreases more slowly as can be seen from Figures 8 (b) and 13 (b). The reason for this is that tardy transactions in soft real-time systems usually have higher priority than arriving transactions in a heavily loaded situation. Thus tardy transactions in the system block the incoming transactions rather than be preempted by arriving transactions. This arrival blocking is the main cause of decline of restart rate. Firm real-time system reduces arrival blocking by removing tardy transactions from the system.



(a) Miss percent

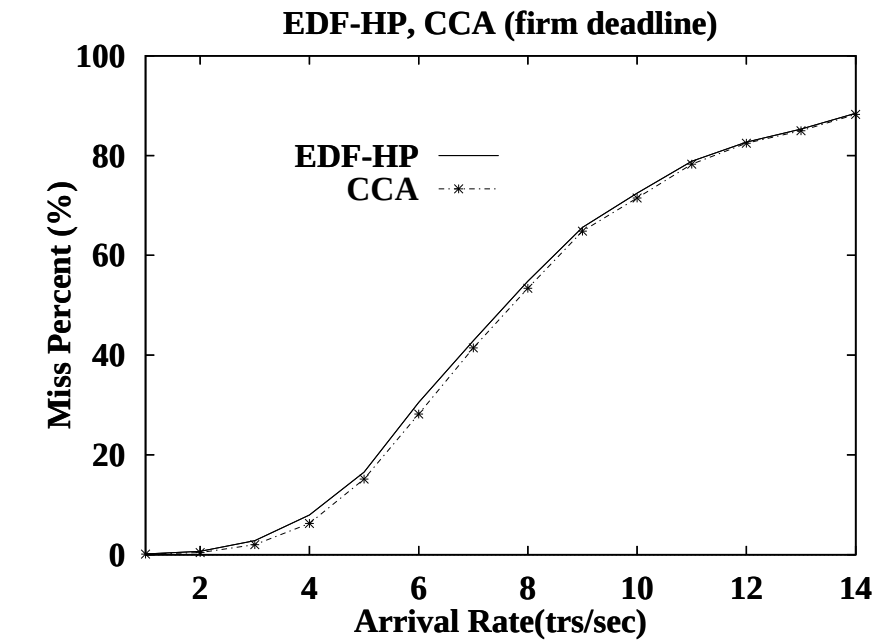


(b) Restart rate

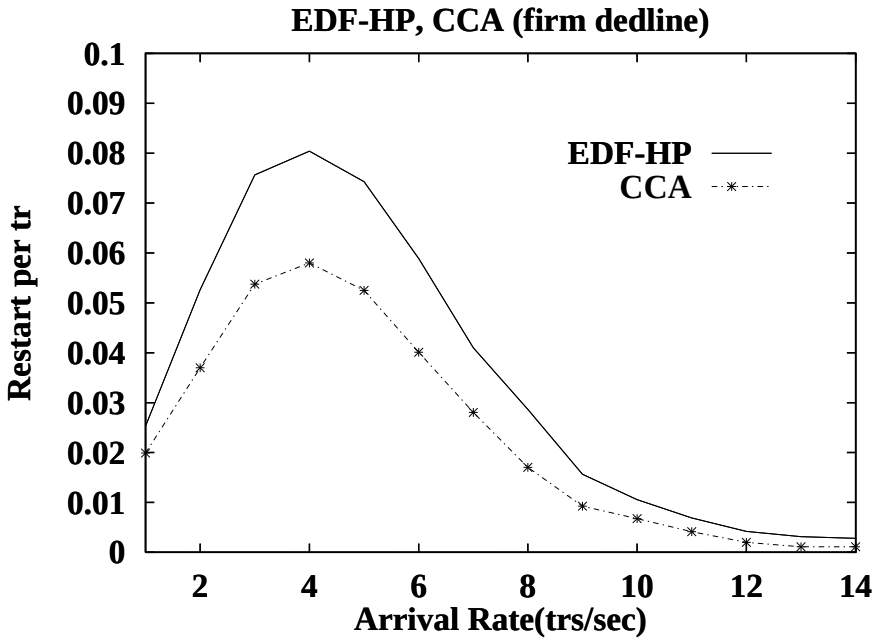


(c) mean lateness

Figure 12: Comparison with EDF-CR



(a) Miss percent of EDF-HP,CCA



(b) Restart rate

Figure 13: Experiment for firm real-time system

6.2 Simulation of Disk resident database

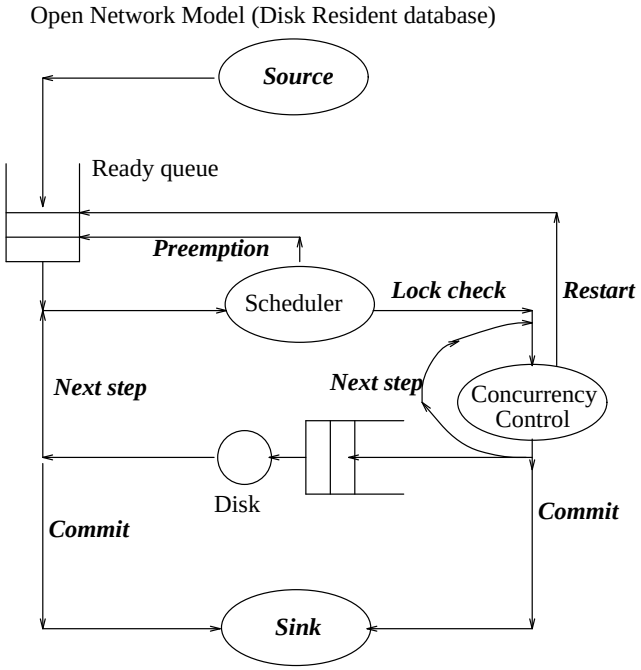


Figure 14: Open Network Model for disk resident database

In order to measure the performance of our algorithm on disk resident database, we extended the simulation program to perform experiments for this case. In this simulation we assumed that we have a single processor, single disk and FCFS I/O 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 I/O access it is not deleted until it releases the disk. We used deferred update rather than immediate update for fast rollback [AD85]. Thus we assume that transaction rollback and restart do not require any disk access. After completing deferred updates a transaction releases all its locks.

The values of parameters used for this experiment are shown in Table 3. The values of *cpu_time* and *i/o_time* are chosen to balance the utilization of CPU and disk [ACL87, AGM92, TSG85]. With this parameter assignments the system is slightly I/O bound. Resource time in this experiment depends on the *cpu_time*, the number of objects, the number of disk access, and *i/o_time*.

$$resource\ time = number\ of\ objects \times cpu\_time + number\ of\ disk\ access \times update\_prob \times io\_time$$

The deadline is assigned based on pre-commit time and timing requirement is inspected when a transaction pre-commits. As we have 2 system resources in this experiment and the *disk_prob* is 0.5, we assigned 0.5 as the value of penalty_weight. The rationale is to distribute the penalty of conflict over system resources.

With the base parameters in the Table 3 the maximum capacity of the system is:

$$\frac{16\ objects}{transaction} \times \frac{15\ ms}{object} = \frac{240\ ms}{transaction} = 4.2\ trs/second$$

Parameter	Value
db_size	250
max_size	24
min_size	8
i/o_time	25 ms
cpu_time	15 ms
disk_prob	0.5
update_prob	0.5
min_slack	100 (%)
max_slack	650 (%)
restart_time	5 ms
penalty_weight	0.5

Table 3: Base parameters for disk resident database

This calculation is very optimistic because it does not include abort cost nor does it include the blocking cost of transactions.

6.2.1 Effect of Arrival Rate

In this experiment, we varied arrival rate from 0.2 tr/sec to 4 trs/sec with the base parameters shown in Table 3 and compared EDF-HP, EDF-CR and CCA schemes. Here also EDF-CR uses the expected resource time as an estimated execution time. Increasing the arrival rate increases deadline as well as data contention thus increasing transaction miss percent for all three approaches. However, CCA shows a much larger improvement over EDF-HP and EDF-CR for the disk resident database (as expected) as compared to the main memory case in Figures 15 (a) and 15 (b).

The reason for rapid increase of restart rate in EDF-HP and EDF-CR is that when the arrival rate is high, the number of available transactions increases which in turn causes high data contention. High data contention makes many transactions to block which eventually increases the number of active transactions, transactions that have begun execution but have not finished yet, in the system. Thus the priority-based restarts of the active transactions that are blocked waiting for resources increases very rapidly. The increase in the restart ratio means that a longer fraction of disk time is spent doing work that will be redone later [ACL87]. Wasted resource time due to priority-based restart causes high resource utilization and easily makes bottleneck resource saturation that induces longer I/O wait time. With the longer I/O wait time more transactions are scheduled and that makes the I/O wait time longer and longer. Thus the possibility of restarting the active transactions increases further.

After the peak point, restart rates decline very slowly as shown in Figure 16 (b). This is because the number of restarts due to higher priority transaction's I/O wake up remains the same but the restarts by higher priority transaction's arrivals are gradually reduced. The number of restarts will flatten out eventually as the arrival rate increases.

Even though the available transactions increase as the arrival rate increases, the number of useful transactions for CCA increases very slowly. Thus the number of active transactions are relatively small as shown in Figure 16 (a). As a result, the number of priority-based restarts for CCA increases

slowly as can be seen in Figure 16 (a) and time contention is increased by not scheduling seemingly useless transactions. Relatively low resource utilization due to low wasted restart time slow down resource saturation that makes longer I/O wait time. Slowing down the resource saturation delays high time contention that makes most of transactions miss their deadlines.

As expected, EDF-CR is worse than EDF-HP in this experiment because we used a resource time as an estimated execution time and it was extremely underestimated. When the system is heavily loaded the behavior of EDF-CR is almost the same as that of EDF-HP because almost all transactions in the system do not have enough slack time to apply conditional restart mechanism any more.

7 Conclusions

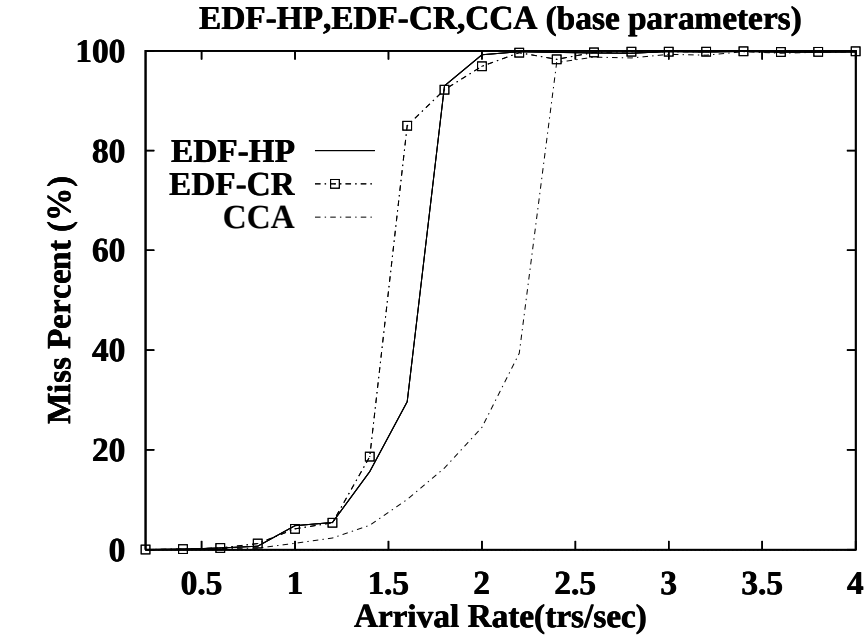
To the best of our knowledge, real-time transaction scheduling approaches have not considered the dynamic cost, the cost of rollback and restart of transactions. This may not be a key consideration in real-time task scheduling that only considers timing correctness. In real-time transaction scheduling, the cost incurred at run time to keep the database consistent can be a key factor.

The approach described in this paper uses dynamic priority assignment with continuous evaluation method to adapt to load changes effectively and to reduce the excessive restart problem encountered by EDF-HP in high data contention situations. Our simulation results show that i) CCA performs better than the EDF-HP algorithm for both soft and firm deadlines, ii) CCA is more fair than EDF-HP, iii) CCA is better than EDF-CR for soft deadline, even though CCA requires less information, and iv) CCA is especially good for disk-resident data.

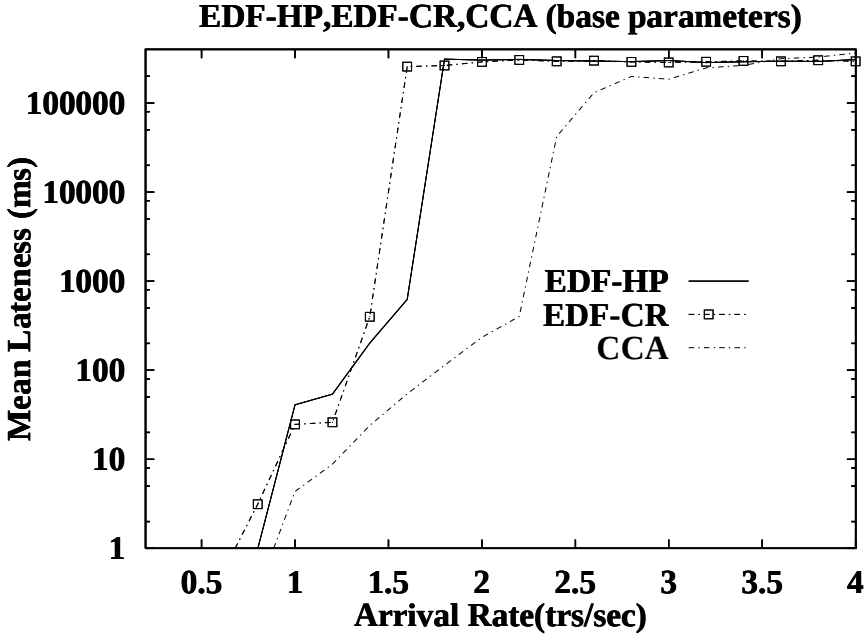
The distinctive features of our approach are: i) Our dynamic priority assignment policy synthesizes deadline and *penalty of conflict* together. The amount of effective service time of a transaction is implicitly taken into account as it is a part of the penalty of conflict computed for conflicting transactions, ii) 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.

In this paper we assumed that we only have exclusive locks and same criticalness for all transactions in the system. The effect of shared locks in transactions and multiple criticalness will affect the performance of RTDBS. The effect of recovery cost is included in a very simple way in our simulations. We assumed that we could recover a transaction very quickly within a fixed amount of time regardless of its execution time. If the recovery cost is proportional to the execution of a transaction or several disk I/O operations are required in transaction recovery then our approach is very attractive because CCA shows fewer number of transaction restarts over EDF-HP.

Although we discussed only lock-based protocols, the approach presented in this paper can be meaningful for optimistic [HCL90a] or even conflict avoiding approaches proposed in the literature [BMH89]. For optimistic approaches, decision to abort can be based on not only the priority information but also on the dynamic costs, such as how many restarts will occur and their effective service times. Also, transaction pre-analysis may provide information about rollback points that can be used for each transaction, thus avoiding complete rollback. It is also possible to develop a hybrid scheduling policy, using the approach presented in this paper, which will combine some features of lock-based protocols and some features of OCC protocols. For example, instead of postponing validation until the commit, partial validation can be performed at intermediate points using dynamic costs.

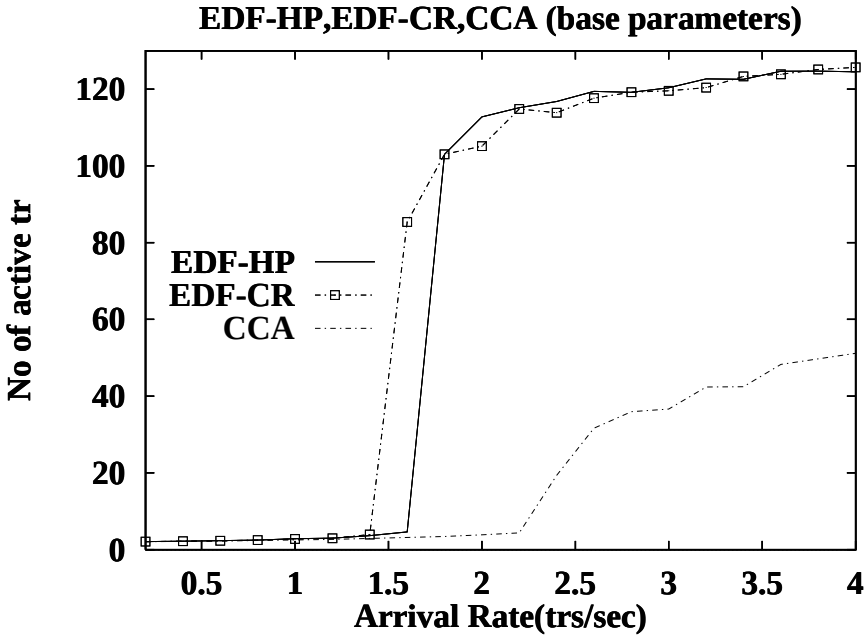


(a) Miss percent

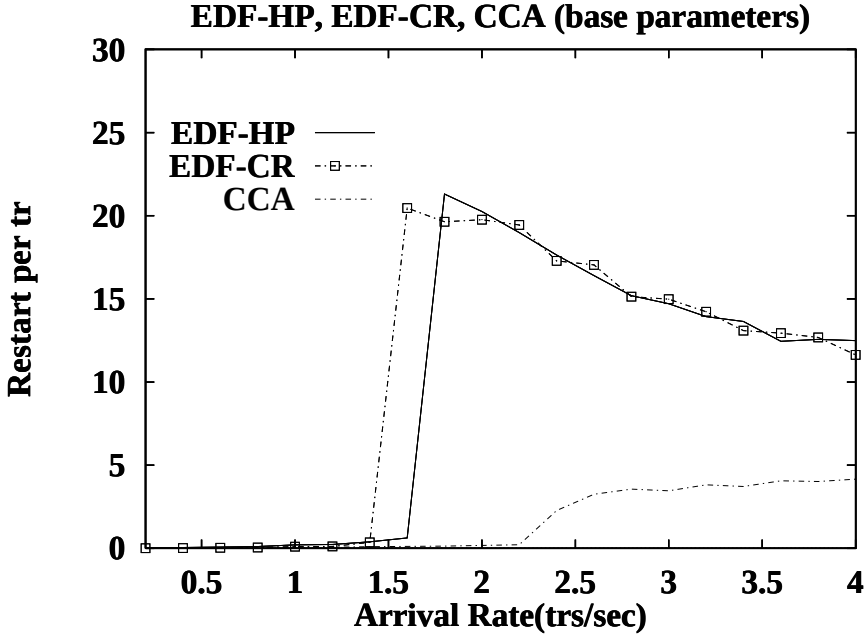


(b) Mean Lateness

Figure 15: Result of experiment on disk resident database(I)



(a) Number of active Tr



(b) Restart Rate

Figure 16: Result of experiment on disk resident database(II)

Current trend of real-time systems is to use multiprocessor (shared as well as distributed memory/disk) on account of additional resources, computational power, reliability as well as parallelism. Extending our approach to multiprocessor environment is more promising than simple EDF-HP approach because our approach shows better performance than EDF-HP when data contention is high. In addition the data requirement information that can be obtained from transaction pre-analysis can also be used to the distribution of data to several systems for more concurrency. Currently we are investigating a combination of CCA and EDF-HP for shared memory multiprocessors and shared nothing multiprocessors systems.

Acknowledgment

This work was supported in part by the National Science Foundation Grant IRI-9011216 and in part by the Florida High Technology and Industry Council. We would like to thank Jayant Haritsa for several discussions about the firm deadline case and the choice of parameters for the simulation presented in this paper.

References

[ACL87] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609–654, 1987.

[AD85] Rakesh Agrawal and D. DeWitt. Integrated concurrency control and recovery mechanism: Design and performance evaluation. *ACM Transactions on Database Systems*, 10(4):529–564, 1985.

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

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

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

[AGM92] Robert Abbott and Hector Garcia-Molina. Scheduling real-time transaction: Performance evaluation. *ACM Transactions on Database Systems*, 17(3):513–560, 1992.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[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.

[DLT85] Jensen E. Douglas, C. Douglass Locke, and Hideyuki Tokuda. A time-driven scheduler for real-time operating systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 112–122. IEEE, 1985.

[FF91] Borko Furht and Borivoje Furht. *Real-time UNIX systems: design and application guide*. Kluwer Academic, Boston, 1991.

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

[HCL90a] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. Dynamic real-time optimistic concurrency control. In *Proceedings of Real-Time System Symposium*, pages 94–103. IEEE, 1990.

[HCL90b] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. On being optimistic about real-time constraints. *ACM PODS*, 1990.

[HJC93] D. Hong, T. Johnson, and S. Chakravarthy. Real-time transaction scheduling: A cost-conscious approach. In *Proceedings of the 1993 ACM SIGMOD Int'l Conference on Management of Data*, pages 197–206. ACM, 1993.

[HLC91] Jayant R. Haritsa, Miron Livny, and Michael J. Carey. Earliest deadline scheduling for real-time database systems. In *Proceedings of Real-Time System Symposium*, pages 232–242. IEEE, 1991.

[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.

[KS91] Woosaeng Kim and Jaideep Srivastava. Enhancing real-time dbms performance with multiversion data and priority based disk scheduling. In *Proceedings of Real-Time Systems Symposium*, pages 222–231. IEEE, 1991.

[LS90] Yi Lin and Sang H. Son. Concurrency control in real-time databases by dynamic adjustment of serialization order. In *Proceedings of Real-Time Systems Symposium*, pages 104–112. IEEE, 1990.

[PLJ92] Hweeehwa Pang, Miron Livny, and Michael J.Carey. Transaction scheduling in multiclass real-time database systems. In *Proceedings of Real-Time System Symposium*, pages 23–34. IEEE, 1992.

[Ram93] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1993.

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

[SRL90] Lui Sha, Ragunathan Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39:1175–1185, 1990.

[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.

[Tay92] Y.C. Tay. A behavioral analysis of scheduling by earliest deadline. Technical Report No. 532, Department of Mathematics, National University of Singapore, 1992.

[TSG85] Y.C. Tay, R. Suri, and N. Goodman. Locking performance in centralized databases. *ACM Transactions on Database Systems*, 10(4):415–462, 1985.

[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.