REAL-TIME TRANSACTION SCHEDULING: A COST-CONSCIOUS APPROACH

By

DONG-KWEON HONG

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

1992

ACKNOWLEDGEMENTS

First, I would like to thank my advisors, Dr. Sharma Chakravarthy and Dr. Theodore Johnson, for showing me the path of research, and for providing me with constant encouragement throughout my work. I would also like to thank the other member of my supervisory committee, Dr. Ravi Varadarajan, for willingly agreeing to serve on my committee.

Next I would like to thank Dr. Paul Fishwick for providing me SIMPACK simulaton package. Without this great package, my work would have been delayed.

Finally, I would like to thank all the student at the Data Base System Research and Development Center for their help and friendship.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis
Presented to the Graduate School of the University of Florida
in Partial Fulfillment of the Requirements for the
Degree of Master of Science

REAL-TIME TRANSACTION SCHEDULING: A COST-CONSCIOUS APPROACH

By

DONG-KWEON HONG

DECEMBER, 1992

Chairman: Dr. Sharma Chakravarthy
Cochairman: Dr. Theodore Johnson
Major Department: Computer and Information Sciences

Real-time databases are an increasingly 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 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 as proposed in this thesis. In this thesis, we propose a new cost-conscious pre-analysis based 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.

CHAPTER 1
INTRODUCTION

People can share much more information with the help of computers and database technology. The main purpose of a database system is to allow people to save, share and find necessary information easily in the computer. With the rapid changes of computer technologies, the expectations of people have greatly increased. For data-intensive real-time applications, the ability of database system to satisfy the timing constraint of a transaction (a basic unit of work that should be done completely or not at all in the context of database) becomes a key factor in determining the feasibility of the application.

Real-time systems which manipulate *real-time data* (any data that must be manipulated in some way subject to a timing constraint) are characterized by the fact that severe consequences will result if the timing as well as the logical correctness properties of the system are not satisfied.

Examples of real-time systems are command and control systems, process control systems, flight control systems, the space shuttle avionics systems, future systems such as the space station and the space-based defense system. Most of the real-time computer systems are special-purpose and complex, requiring a high degree of fault tolerance, and are typically embedded in a large system. Also, real-time systems have substantial amounts of knowledge concerning the characteristics of the application and the environment built into the system. Thus database systems which are embedded in real-time systems also have much knowledge of the application and the knowledge should be used as much as possible.

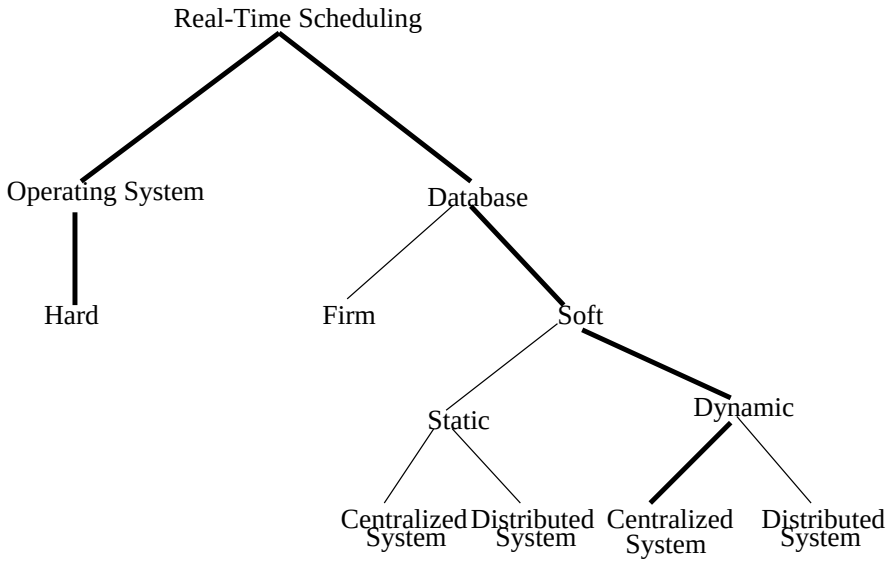

Figure 1.1. Taxonomy of real-time scheduling

One of the most important research area in the real-time system is in real-time scheduling theory and much research is being done by the operating system and database system groups. Taxonomy of the real-time scheduling is illustrated in Figure 1.1 where thick lines indicate active areas of research.

1.1 Real-Time Operating System

Real-time operating systems play a key role in most real-time systems. Scheduling theory in this area is usually based on task that have a time constraint. A task is a software module that can be invoked to perform a particular function and it is the scheduling entity in an operating system. Some important operating system research issues are [20]:

Time-driven resource management Traditionally, when many tasks are waiting for access to a shared resource, the allocation policy is to provide access in First In First Out (FIFO) order. However, this policy totally ignores task's timing constraints. Time-driven allocation policies must be developed that can meet the real-time scheduling requirements. Such management policies should be

applied not only for processor but also for memory, I/O and communication resources.

Problem specific OS facilities A real-time operating system's functions should be able to adapt to a variety of user and system needs. Thus, a user can choose or the system can select the time-driven resource management algorithm most suitable for a particular application or situation.

Integrated system-wide scheduling support Real-time scheduling principles must be applied to system resources, application tasks, and operating system's overall design. In particular, a focussed effort is necessary to integrate real-time communication with real-time CPU scheduling, and with real-time database support for a large, complex real-time computer system.

1.2 Real-Time Database System

Usually Real-time Database System (RTDBS) is part of a large, complex real-time systems. The basic unit of scheduling in RTDBS is a real-time transaction and most important research area is how to combine the priority assignment and concurrency control protocols. Generally, a real-time transaction can be classified as a hard-deadline, soft-deadline and firm-deadline, and most research in this area is for scheduling soft or firm transactions.

Hard real-time transaction If these transactions miss their deadlines, there are catastrophic consequences.

Soft real-time transaction These transactions have timing constraints, but there may still be some benefit for completing the transactions after its deadline, and catastrophic result do not occur if these transactions miss their deadlines.

Firm real-time transaction These transactions have timing constraints, and there is no value for completing the transactions after its deadline, and catastrophic result do not occur if these transactions miss their deadlines.

In this thesis, we view a RTDBS 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.

1.2.1 Priority Assignment

The performance goal of conventional Database Management System (DBMS) is usually expressed in terms of average response times rather than meeting the timing requirement of the transaction. Thus improving the response time of one transaction at the expense of another is not considered as improvement. In an RTDBS, the objective is to reduce the number of transactions that have missed their deadlines or total lateness. Let's consider two scenarios here: If the transactions share the system resources on an equal basis, the transactions that have tight deadlines miss their deadlines while the transactions that have loose deadlines are likely to meet their deadlines. Alternatively, if the transactions that have urgent deadlines execute at the expense of the transaction that have loose deadlines, they can complete before its deadline. After that the other transactions execute and still complete in time due to their loose deadlines. From these two scenarios, we can see that the service precedence which is decided by priority assignment policy affects the performance of RTDBS.

1.2.2 Concurrency Control

The usual correctness criteria of database transactions is serializability. A serial schedule has no concurrency, but it is of interest since it preserves database consistency. We are interested in the large class of schedules, which may exhibit

consistency and which are equivalent to some serial schedule. Such schedule are said to be *serializable*. Widely used serialization mechanism are locking, validation and timestamping. Each mechanism takes a different approach to achieve serializability. Whenever a data conflict occurs, concurrency control protocols use blocking, or transaction restarts or combination of methods. In RTDBS the decision of blocking or transaction restarts should include transaction priorities.

1.3 Problem Statement

The primary 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, 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 [4, 16, 21, 23, 22] used in current real-time systems assume *a priori* 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.

1.4 Thesis Organization

The organization of the thesis is as follows. Chapter 2 discusses previous related work on time constraint, real-time task scheduling, real-time transaction scheduling and the effect of lock types in RTDBS. In Chapter 3 we explain the motivation

and basic idea of a Cost Conscious (CC) approach. In Chapter 4, we analyze the characteristics of CC approach and shows the effect of CC approach compared with Earliest Deadline First (EDF) using illustrative examples. In Chapter 5, we present the performance evaluation of our approach on main memory resident database with simulation. In Chapter 6, we present the performance evaluation again on disk resident database. Chapter 7 discusses the effect of multiple exit point in the transaction program. Chapter 8 concludes the thesis with the future works and the contribution of this thesis.

CHAPTER 2
SURVEY OF RELATED WORKS

Satisfying the timing requirements of real-time systems demands the scheduling of system resources according to some well understood algorithms so that the timing behavior of the system is understandable, predictable, and maintainable. In real-time computing systems, there is no advantage to minimize the response times other than meeting the deadlines. The real-time system is often highly dynamic requiring on-line, adaptive scheduling algorithms. Such algorithms must be based on heuristics since the scheduling problems are usually NP-hard [7, 19]. The RTDBS which is an important component of embedded real-time system is also very dynamic in nature. Thus all dynamic real-time transaction scheduling algorithms are also based on heuristics.

Many real-time scheduling algorithms can be integrated with database concurrency control algorithms. In this chapter we will survey previous works on time constraint, real-time task scheduling, and real-time transaction scheduling.

2.1 Time Constraints

In order to represent time constraints for real-time tasks the value function model was developed [8, 14]. The completion of a task has a value to the application that can be expressed as a function of the task completion time. A task with a hard deadline is modeled by a step function. This is illustrated in Figure 2.1. The value functions are more expressive than deadlines that represent only a single instant of time.

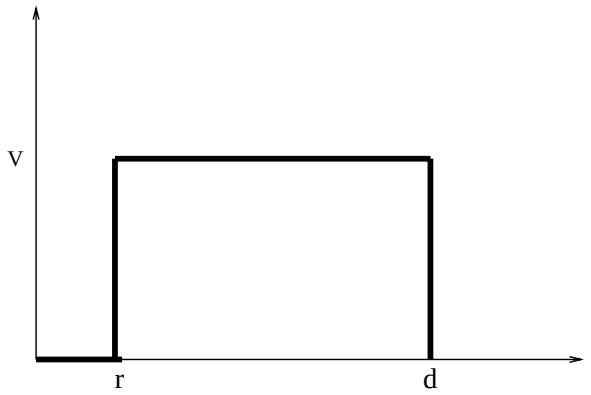

Figure 2.1. Value function of hard deadline

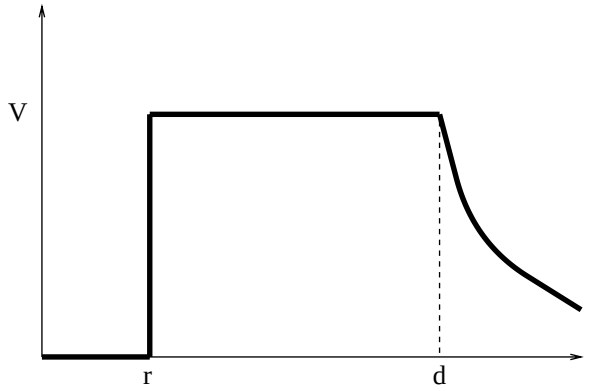

Figure 2.2. Value function of soft deadline

Abbot and Garcia-Molina [1] used the value function as a way to express the time requirement of a real-time transaction. Figure 2.2 shows how we can use value functions to model soft real-time transactions. Completion of the transaction before time d which is started at time r yield a value V. The value of completing the task after time d decreases.

2.2 Task Scheduling

The timing constraints of a task are specified in terms of one or more of the following parameters:

Arrival time The time at which a task is submitted to the system.

Ready time The earliest time at which a task can begin execution. The ready time of a task is equal to or greater than its arrival time.

Worst case execution time The execution time of a task is always less than this amount of time.

Deadline The time by which a task must finish.

There are two approaches to task scheduling with the above information: static and dynamic scheduling. All the static scheduling algorithm for real-time systems assume they know the arrival times of the tasks. But, except for a periodic tasks, most of the tasks are aperiodic. Furthermore, because run-time cost is an important factor for dynamic task scheduling, most static algorithms are not appropriate for dynamic scheduling. Because of these reasons, heuristic algorithms became important to dynamic scheduling problem.

There exists many heuristic real-time task scheduling algorithms that are not suitable for RTDBS. Zhao, Ramamritham, and Stankovic [23] developed a heuristic function and an efficient backtracking scheme for scheduling nonpreemptable tasks with resource constraints. They found that with a limited number of backtracking, the success ratio of their search algorithm for scheduling tasks can be as high as 99.5% of that of an exhaustive search algorithm. Their approach was extended to preemptive tasks [22]. The improved performance that results from the use of complex mechanisms, such as backtracking, may be offset by the computational overheads introduced by such mechanisms. But they said that such overhead may not be of concern if a separate specially designed coprocessor is used for scheduling.

Among many real-time task scheduling algorithms, just few of them are used on RTDBS and integrated with concurrency control algorithms. In the following section we will look at task scheduling algorithms that are used on RTDBS.

A scheduling algorithm is said to be *static* if priorities are assigned to tasks once and for all. A static scheduling algorithm is also called a fixed priority scheduling algorithm. A scheduling algorithm is said to be *dynamic* if priorities of tasks might change from request to request.

2.2.1 Static Approach

Liu and Layland [13] developed a rate-monotonic static priority assignment scheme to determine the schedulability of a set of periodic tasks. Their approach assigns higher priorities to tasks with higher request rate. They showed that the scheme is optimal among static priority assignment scheme for mutually independent tasks. The deadline of a request for periodic task is defined to be the time of the next request for same task.

Theorem 2.2.1 *A set of* n *periodic tasks scheduled by the rate-monotonic algorithm can always meet their deadlines if*

$$\frac{C_1}{T_1} + \cdots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1)$$

where C_i and T_i are the execution time and period of task τ_i respectively

Above theorem offers a sufficient (worst-case) condition that characterizes the rate-monotonic schedulability of a given periodic task set. Following theorem shows that a rate-monotonic scheme is optimal among static priority scheme.

Theorem 2.2.2 *If a feasible (all tasks are able to meet their deadlines) priority assignment exists for some task set, rate-monotonic priority assignment is feasible for that task set.*

2.2.2 Dynamic Approach

EDF (Earliest Deadline First) is a deadline driven dynamic scheduling algorithm. Using this algorithm, priorities are assigned to tasks according to the deadlines of

their current requests. A task will be assigned the highest priority if the deadline of its current request is the nearest and will be assigned the lowest priority if the deadline of its current request is the furthest. EDF is optimal for completely preemptable periodic task set with hard deadlines executing on a single processor. It is optimal in the sense that if a set of tasks can be successfully scheduled by some priority policy, then this task set is guaranteed to be successfully scheduled by EDF as well [13].

In Least Slack First(LSF) policy tasks with less slack time will have higher priorities. In LSF with fixed evaluation policy slack time S = d - (t + C), where d is deadline, t is current time and C is worst case execution time and in LSF with continuous evaluation policy slack time S = d - (t + C -P)) where P is effective service time. Recently Locke, Tokuda, and Jensen [8] compared a number of dynamic scheduling policies and they found that the LSF and EDF are two good heuristics for task scheduling.

2.3 Transaction scheduling

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 [1, 2, 3, 5, 6, 12, 11, 15, 17, 18, 19]. Priority scheduling without knowing the data access pattern is presented as a representative of algorithms with incomplete knowledge of resource requirements. The recent works [1, 2, 3, 12, 11, 18, 19] falls into this category. They are combined with 2-phase locking or optimistic concurrency control algorithms. In Figure 2.3 we can see the representative approaches for real-time transaction scheduling.

2-phase locking 2-phase locking algorithm execute transactions in two phases. Each transaction has a growing phase, where it obtains locks and accesses data items,

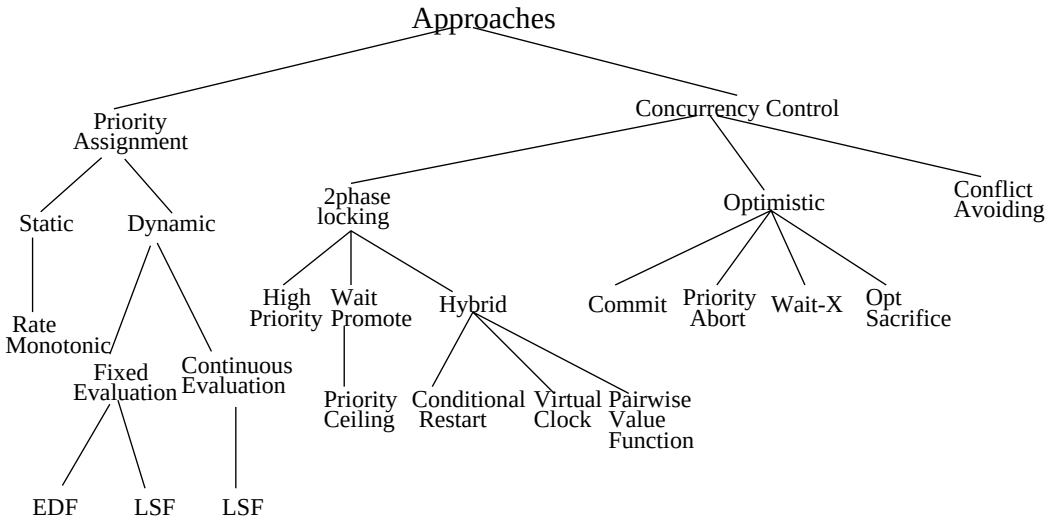

Figure 2.3. Taxonomy of real-time transaction scheduling

and a shrinking phase, during which it releases locks. No transaction should request a lock after it release one of its lock. It is well known that any concurrency control algorithm that obeys the 2-phase locking rule is serializable.

optimistic concurrency control Some concurrency control algorithms based on locking or time stamp are pessimistic in nature. They assume that the conflicts between transactions are quite frequent and do not permit a transaction to access a data item if there is a conflicting transaction that accesses that data item. Thus the execution of any operation of a transaction follows the sequence of phase: validation, read, computation, write. Optimistic algorithms, on the other hand, delay the validation phase until just before the write phase. Thus an operation submitted to an optimistic scheduler is never delayed. Each transaction initially makes its updates on local copies of data items. The validation phase consists of checking if these updates would maintain the consistency of the database. If the answer is affirmative, the changes are written into the actual database. Otherwise, the transaction is aborted and has to restart.

2.3.1 Approaches with 2-phase Locking

Abbot and Garcia-Molina [1, 2] proposed the following algorithms: EDF-HP (High Priority), LSF-HP, EDF-CR (Conditional Restart) and LSF-CR. These algorithms are based on 2-phase locking.

HP conflict resolution method is the same as priority-based wound wait conflict resolution method (In the priority-based wound-wait protocol, transaction T_i can wait for a conflicting transaction T_j if T_i has a lower priority. Otherwise, T_j is aborted (wounded)). The idea of this method is to resolve a conflict in favor of the transaction with the highest priority. The favored transaction gets the resources, both data lock and the processor, that it needs to proceed. The loser of the conflict relinquishes the control of any resources that are used by itself.

Sometimes HP may be too conservative. Let T_R be a transaction requesting a lock held by T_H. We would like to avoid aborting T_H because we lose all the service time that it has received. The idea of CR conflict resolution method is to estimate if T_H 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. If so then we let T_H proceed to completion, release its locks and then let T_R execute. 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 there exist deadlock problem in CR.

There are several problems with HP and CR if LSF is used to assign priorities, and priorities are assigned continuously [2].

Circular abort Rolling back a transaction to its beginning reduces its effective service time to 0 and raise its priority under the LSF policy. Let T_r be a transaction

requesting a lock held by T_h. A transaction T_h, which loses a conflict and is aborted to allow a higher priority transaction T_r to proceed, can have a higher priority than T_r immediately after the abort. The next time the scheduler is invoked, T_r will be preempted by T_h. T_h may again conflict with T_r initiating another abort and rollback [2].

Priority Reversal Under the LSF policy, a transaction's priority depends on the amount of service time that it has received. The slack time of a transaction which is not executing decreases. Hence the priority of that transaction increases. Even if we use HP conflict resolution method [2], a deadlock is possible. Let's consider following scenario: Let T_R be a transaction requesting a lock held by T_H. When data conflict occurred the priority of T_R is lower than that of T_H and T_R is blocked. After executing some time T_H is requesting a lock held by T_R. During this time the priorities of T_R and T_H are reversed. Because the priority of T_H is lower than that of T_R, T_H is also blocked and deadlock occurs. Priority reversal occurs if we use different priority assignment policy at the CPU conflict and data conflict.

With VCAP (Virtual Clock Access Protocol) [19], transactions are statically categorized into n classes according to their criticalness. The criticalness of a task is indicative of the *level of importance* that is attached to that task relative to the other task. Depending on the functionality of a task, meeting the deadline of one task may be considered more critical than another. The smaller the class number of a transaction, the less critical it is. Each transaction has a virtual clock associated with it. Let $VT(T_i)$ the virtual clock value for transaction T_i. When transaction T_i starts, $VT(T_i)$ is set to the current real time, t_{si}, i.e, the start time of transaction T_i. The virtual clock then starts to run at rate β_k, where k is transaction T_i's class

number. The more critical a transaction is, the faster its virtual clock runs. If t is the current time, then at any time

$$VT(T_i) = t_{si} + \beta_k(t - t_{si})$$

If the deadline of requesting transaction T_j is earlier than that of lock holding transaction T_i and VT(T_i) is less than T_i's deadline, T_j aborts T_i. This protocol synthesizes elapsed time of a transaction, relative deadline and criticalness of running transaction in the preemption decision.

In the same paper [19] Stankovic and Zhao suggested another protocol, PVF (Pairwise Value Function Access Control Protocol) that considers the criticalness of the requesting transaction also. Transactions are categorized into n classes according to their criticalness. Each transaction T_i has a value function associated with it. Let $VU(T_i)$ denote the value function for transaction T_i. Let γ_k denote the criticalness of a transaction where k is transaction T_i's class number. Then

$$VU(T_i) = \gamma_k(\omega_1(t - t_{si}) - \omega_2 d_i + \omega_3 C_i - \omega_4 l_i)$$

where the ω's are weighting factors and d_i is deadline, C_i is computation time consumed by the transaction and l_i is the laxity approximation (i.e, slack time) of transaction T_i.

While transaction T_i is accessing data item, D, another transaction T_j may request access to D. If the value of the function VU(T_j) is less than or equal to the value of the function VU(T_i), T_j waits, and otherwise T_j aborts T_i. The main problem of VCAP, PVF are deadlock due to priority reversal. In these algorithms they used different priorities on CPU and data conflict that easily make deadlock. Let's consider following scenario: Let T_R be a transaction requesting a lock held by T_H. When data conflict occurred the value of $VU(T_R)$ is smaller than that of $VU(T_H)$ and T_R is blocked. After executing some time T_H is requesting a lock held by T_R. During this

time the VU value of of T_R and T_H are reversed due to different criticalness. Because the value of $VU(T_H)$ is smaller than that of $VU(T_R)$, T_H is also blocked and deadlock occurs.

Another problem is they did not suggest any way that can adjust the weighting factor ω for better performance.

2.3.2 Approaches with Optimistic Algorithm

An OCC (Optimistic Concurrency Control scheme) with a deadline and transaction length based priority assignment scheme [12] and an OCC with Adaptive Earliest Deadline have also been proposed [11]. Ideally OCC has the properties of being non-blocking and of freedom of deadlock. Due to its potential for a high degree of parallelism, optimistic concurrency control is expected to perform better than two-phase locking when integrated with priority-driven CPU scheduling in real-time database systems [11]. With OCC, an algorithm is needed to resolve the access conflicts during the validation phase. Some resolution policies are:

Commit Always let the validating transaction commit and abort all the conflicting transactions. This strategy guarantees that if a transaction reaches its validation phase, it will always finish.

Priority abort Abort the validating transaction only if its priority is less than that of all the conflicting transactions. This strategy takes transaction priority into account, but still favors the validating transaction.

Priority Wait If the priority of the validating transaction is not the highest among the conflicting transactions, wait for the conflicting transactions with higher priority to complete. In some cases, the strategy of aborting conflicting transactions appears too conservative, causing unnecessary transaction abort.

Opt-Sacrifice If conflicts are detected and at least one of the transactions in the conflict set has a higher priority over validating transaction, then the validating transaction is restarted.

Among these resolution policies Priority Wait is most promising. In some cases, the strategy of aborting transactions appears too conservative because it causes unnecessary transaction aborts. Wait-50, the version of Priority Wait strategy where a validating transaction will wait if at least 50% of the conflicting transaction have a higher priority over the validating transaction, is commonly used as a real-time optimistic concurrency control resolution method [11, 12].

2.3.3 Approaches with Transaction Pre-analysis

Priority scheduling with transaction pre-analysis is an approach that makes more use of the available knowledge of the resource requirements [5, 15, 17]. A. Buchmann [5] proposed two transaction pre-analysis based algorithms: conflict avoiding non-preemptive method and Hybrid algorithms which use conflict avoiding scheme in the non-overload case and CR conflict resolution method in the overload (we say that the system is in *overload* situation if we cannot find a schedule in which every transaction can finish within its deadline for hard real-time transactions) case.

Priority Inheritance In a priority-inheritance scheme, a low priority transaction inherits the priority of the high priority transaction that it blocks.

Priority Inversion Priority inversion is said to occur when a high priority transaction is blocked by lower priority transaction. Priority inversion is inevitable phenomenon in non-abortive priority-driven system.

Priority Ceiling The priority ceiling of a data object is the priority of the highest priority task that may lock this object.

Priority-ceiling protocol A transaction J requesting to lock a data item S is granted the lock only if $p(J) > c(K)$, where K is the data item with the highest priority ceiling among all data items currently locked by transactions other than J, p(J) is the priority of transaction J and c(K) is the priority ceiling of K. If J cannot lock S, J is blocked and the transaction holding the lock on K inherits the priority p(J) until K is unlocked.

Lui Sha [15, 17] proposed static priority assignment based Priority Ceiling protocol using priority inheritance with exclusive lock and read/write Priority Ceiling protocol. These protocols are transaction pre-analysis based nonabortive methods using priority inheritance to prevent priority inversion and indefinite blocking. It is important to note that the concept of priority ceiling assumes that we know a lot about transactions that will access the database. This is a reasonable assumption for dedicated real-time application such as tracking.

Although the priority ceiling protocol introduces unnecessary blocking, the worst case blocking for any task is reduced to the duration of at most one low priority transaction to finish in one critical section, and no deadlock will ever occur. The critical problem of this protocol is that it is not appropriate for disk resident database because a lower priority transaction is unnecessarily blocked during IO wait time of a higher priority transaction.

2.3.4 Lock Type

Many previous works didn't consider the effect of shared locks in real-time database. With exclusive locks only [15], conflicts always involve a pair of transactions: the holder and the requester. With shared locks [3, 17, 12], the holder can be a set of concurrently reading transactions, each with a different deadline. If we change all shared locks in transaction into exclusive locks, the opportunities of data conflict will be increased greatly.

Tr(arriving transaction)	Th(holding transaction)
S lock	X lock
X lock	X lock
X lock	S(single) lock
X lock	S(multiple) lock
S lock	S lock(with waiting X)

Table 2.1. Condition of conflict

A reader can join a read group only if it has a higher priority then all waiting writers. Otherwise, the reader must wait [3]. The most interesting case occurs when a transaction T_R requests an exclusive lock on a data item when transactions T_{H1}, ..., T_{Hn} already hold shared locks on the data item. There are three possibilities:

1. Priorities of transactions T_{H1} through T_{Hn} are greater than that of T_R.
2. Priorities of T_R is greater than or equal to T_{Hj} and less than or equal to that of T_{Hj+1}.

   $Pr(T_{Hj}) \leq Pr(T_R) \leq Pr(T_{Hj+1}) \qquad 1 \leq j \leq n$
3. Priorities of transactions T_{H1} through T_{Hn} are less than that of T_R.

In case 1, it is intuitively simple, T_R waits. In case 2 and 3, however, only a few methods have been proposed and one of them is the percentage related decision methods with OCC [12]. Much research is needed to find proper method in case 2 and case 3. Abbot and Garcia-Molina [3] proposed that in case 3 T_{h1} through T_{Hn} are aborted. However, if all priority of transactions aborted are slightly less than that of priority of T_R, aborting T_{H1} through T_{Hn} is not the best choice.

2.3.5 IO Scheduling

In conventional systems the goal of IO scheduling is to maximize the throughput of the IO system. One way to do this is by using disk scheduling algorithm to order

the sequence of IO request so that the mean seek time is minimized. However the goal of the RTDBS is different from conventional system.

There have been suggested two ways to schedule the IO queue for RTDBS [2].

FIFO When FIFO is used to schedule the IO queue, requests are serviced in the order in which they are generated. This service order is somewhat related to transaction priorities because IO requests are generated by the CPU, which is selected by priority.

Priority Under this policy each IO request has a priority which is equal to the priority of the transaction which issued the request.

CHAPTER 3
COST-CONSCIOUS APPROACH

Static priority assignment in a real-time transaction processing system is not adequate because it cannot consider the urgency of deadline. The transaction pre-analysis method which has been considered until now is not adequate either because it is considered too pessimistic to use in real-time systems. EDF and LSF priority assignment policy are restrictive for real-time transaction scheduling because they ignore the transaction relationships and the rollback and restart effects. To the best of our knowledge, the effects of transaction rollback and restart overhead has not been used in real-time transaction scheduling.

3.1 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 [21, 23, 22]. However, these policies are not acceptable in the context of real-time transaction scheduling.

LSF is not appropriate for RTDBS because the worst case estimated execution time of a transaction is not easy to get due to the existence of disk IO and the branches in the transaction programs.

The weakness of EDF under high level of resource and data contention is that this policy causes most transactions to miss their deadlines since they receive high priority only when they are close to missing their deadline [11]. If EDF is combined with HP conflict resolution method, the transaction restart and rollback easily makes the system heavily loaded. Thus EDF-HP causes too many transaction aborts.

In order to solve the problem of too many transaction aborts of EDF-HP, EDF-WP has been proposed. However, EDF-WP has too much wait due to its nonabortive conflict resolution method and has deadlock problem.

Several hybrid methods that are using the combination of abortive and non-abortive method has been proposed [1, 19]. These methods make decisions about transaction blocking and rollback using additional information like slack time or estimated execution time. However, they still have deadlock problem.

In this thesis, we suggest a new cost conscious dynamic priority assignment policy with continuous priority evaluation that solves the problem of EDF-HP which causes too many transaction aborts and the deadlock problem of Hybrid methods.

3.2 Transaction Response Time

It is difficult to anticipate the finishing time of a real-time transaction due to transaction blocking, restarts and disk IO. The transaction response time consists of the time needed to execute a transaction in an isolated environment, T_{static}, and blocking and restart overhead, $T_{dynamic}$[1]. T_{static} is dependent on the semantics (data value and branches) in the transaction application program. $T_{dynamic}$, as it is computed continuously, take the current state of the transaction and the status of the system. The total execution time is

$$T_{total} = T_{static} + T_{dynamic}$$

In the real-time database context, $T_{dynamic}$ is very difficult to compute because we don't know the future events. Even if we consider the transaction in an isolated environment, static transaction response time, T_{static}, is also difficult to compute due to the existence of branches in the transaction program. Thus task scheduling algorithms that are usually based on worst case execution time, and the transaction

[1]Perhaps $T_{dynamic}$ can be broken into T_{IO}, $T_{blocking}$, $T_{restart}$. In this thesis, we group them into one component for the sake of simplicity

scheduling algorithms that use estimated execution times for wait and abort decision are very restrictive.

3.2.1 Dynamic Cost

The dynamic portion of transaction processing cost ($T_{dynamic}$) in the RTDBS depends on the current status of the system and future arrivals. 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, then: (i) If we use Earliest Deadline First priority assignment policy, several partially executed transactions that conflict with T_a might have to be rolled back. (ii) If we consider dynamic cost, we might find that we lose too much time for the execution of the highest priority transaction.

3.3 Assumptions

We assume that our system contains a single CPU that manages the data, which can be disk-resident. All transactions that the system executes are instances of one of a number of transaction types. We assume that we know the programs of these transactions and have analyzed them. We allow only write locks in our current analysis. If we allow shared locks, dynamic cost will be an even more important factor than that of exclusive lock only system due to the increased number of concurrently running transactions in real-time transaction scheduling decision. When a transaction arrives, we assume that we know its deadline.

3.4 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 3.1:

```
 T1                                  T2

  ...                               ....
access w ....
If(w > 100)                         .....
     access I1, I2, I3          access I1, I2, I3
else
     access I4, I5, I6              .....
 ....
```

Figure 3.1. Transaction programs

Suppose that Tr1 executes program T1 and Tr2 executes program T2. If Tr1 executes the If statement and finds $w > 100$, Tr1 and Tr2 conflict. Otherwise, Tr1 finds that $w <= 100$, and Tr1 and Tr2 don't conflict. Before Tr1 executes the If statement, Tr1 and Tr2 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

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

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 the programmer, or by a compiler. In Figure 3.2, we show the transaction trees of transaction programs T1 and T2. T1's decision point splits the transaction tree into T1a and T1b, which have different data sets. Since T2 contains no decision points, its transaction tree consists of a single vertex. When we analyze the transaction programs, we find that T1 conflicts with T2, T1a conflicts with T2, but that T1b doesn't conflict with T2.

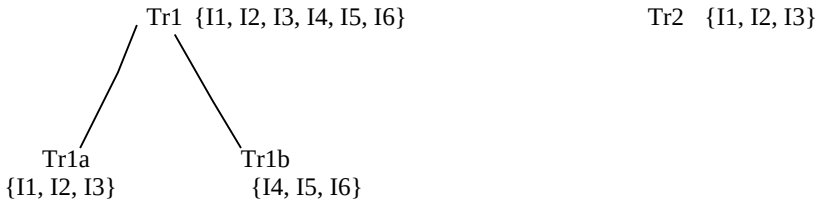

Figure 3.2. Transaction access tree

In the Figure 3.2, before Tr1 reaches the decision point, it might conflict with Tr2, since Tr1 might take the branch to Tr1a, or it might not conflict with Tr2, if Tr1 takes the other branch. Suppose Tr1 makes the branch to Tr1a. At this point we are certain that Tr1 conflicts with Tr2. 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.

Suppose that transaction Tr1 conflicts with transaction Tr2, and Tr1 is scheduled to execute. If Tr2 has not yet accessed any data items that Tr1 might access, then there is no need to roll back Tr2, we only need to block it. In this case, we say that Tr2 is *safe* with Tr1. If Tr2 has accessed a data item that Tr1 will access, then Tr2 is *unsafe* with Tr1 and need to be rolled back. Finally, Tr2 is *conditionally unsafe* with

Tr1 if Tr2 might be safe or unsafe with Tr1, depending on Tr1'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 transaction A transaction that will execute no further decision points.

accesses(T) Set of data items that a transaction of type T accesses before it reaches its next decision point.

hasaccessed(T) Set of data items that a transaction of type T has accessed up to this point.

mightaccess(T) Set of data items that a transaction of type T might access.

leaves(T) The leaves in the subtree rooted at T.

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) for every node T in the transaction tree. If P is the set of nodes on the path from the root to T, inclusive, then

$$\begin{aligned} \text{hasaccessed}(T) = & \quad \cup_{p \in P} \text{accesses}(p) & \\ \text{mightaccess}(T) = & \quad \text{hasaccessed}(T) & T \text{ a leaf transaction} \\ = & \quad \cup_{C \text{ a child of } T} \text{ mightaccess}(C) & T \text{ not a leaf transaction} \end{aligned}$$

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

- Leaf transactions P, Q *conflict* iff mightaccess(P) $\cap$ mightaccess(Q) $\neq \phi$

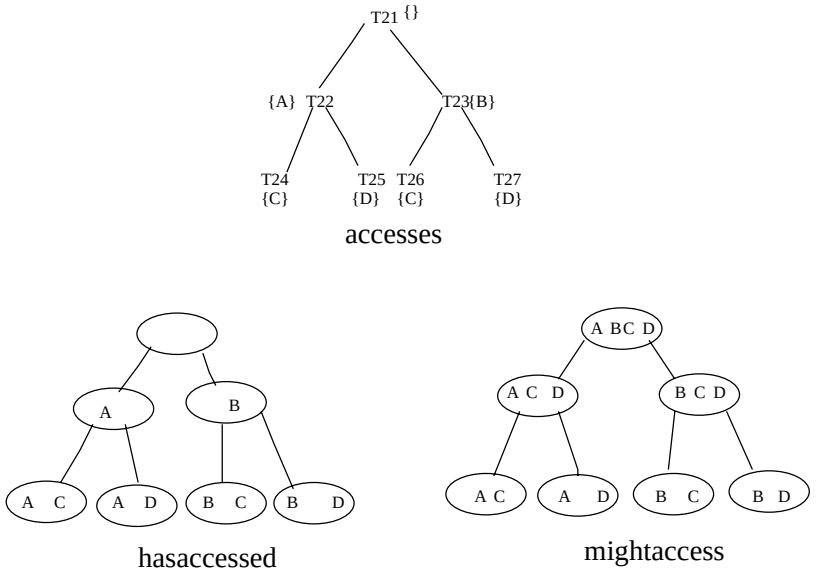

Figure 3.3. Auxiliary transaction access tree

- Transactions P, Q *conflict* iff $\forall_p \in$ leaves(P) $\forall_q \in$ leaves(Q), mightaccess(p) $\cap$ mightaccess(q) $\neq \phi$.
- Transactions P, Q *conditionally conflict* iff $\exists_{i,j} \in$ leaves(P), $\exists_{m,n} \in$ leaves(Q) such that mightaccess(i) $\cap$ mightaccess(m) $\neq \phi$ and mightaccess(j) $\cap$ mightaccess(n) $= \phi$.
- Transactions P, Q *don't conflict* iff they neither conflict nor conditionally conflict.
- Transaction P is *safe* wrt Q iff hasaccessed(P) $\cap$ mightaccess(Q) $= \phi$.
- Transaction P is *unsafe* wrt Q iff $\forall_q \in$ leaves(Q), hasaccessed(P) $\cap$ mightaccess(q) $\neq \phi$.
- Transaction P is *conditionally unsafe* wrt Q iff hasaccessed(P) $\cap$ mightaccess(Q) $\neq \phi$, and $\exists_q \in$ leaves(Q) such that hasaccessed(P) $\cap$ mightaccess(q) $= \phi$.

These transaction relationships will be used to calculate transaction priorities more accurately. Maintaining the access lists is not much overhead to our approach

because almost every transaction processing systems maintain access lists of each transaction for concurrency control and recovery.

3.5 Scheduling Algorithm

A real-time transaction scheduling algorithm consists of the priority assignment and concurrency control module.

3.5.1 Priority Assignment

A priority assignment policy is classified as being a static assignment policy if it is based on static information (e.g, estimated execution time) or as a dynamic assignment policy if it is based on dynamic information (e.g, relative deadline, effective service time). A dynamic priority assignment policy can use a static evaluation method which evaluates the priority only once (e.g. Earliest Deadline First), or a continuous evaluation method which evaluates the priority several times (e.g, Least Slack First) during the execution of a transaction. Even though several static priority assignment policies have been proposed [13], they do not capture all the dynamic features of database transactions. The dynamic, on-line assignment policies that have been proposed can be described with the following priority formula [19].

$$Pr(T_i) = \gamma_i(\omega_1 t - \omega_2 r_i - \omega_3 d_i + \omega_4 C_i + \omega_5 U_i - \omega_6 E_i)$$

$Pr(T_i)$: Priority of transaction T_i

γ_i: Criticalness of transaction T_i

t: Current time

r_i: release time of transaction T_i

d_i: deadline of transaction T_i

C_i: Effective service time of transaction T_i

U_i: Elapsed Service time of transaction T_i

E_i: Estimated execution time of transaction T_i

With the appropriate setting of the ω_j parameters, this priority formula produces the FCFS, EDF, LSF, or any combination of them. Our Cost Conscious priority assignment policy, however, cannot be produced from the above formula. If the transaction T_a which is selected to be run next conflicts with m transactions that are *unsafe* with T_a, we might lose

$T_{lost} = \Sigma_{t \in M} \ (rollback_t + exec_t)$ where

$M = \{transaction\ t \mid \text{t is unsafe with } T_a\}$

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 T_{lost} is large, it waste system resources. We characterize the time lost as the Penalty of Conflict.

Penalty of Conflict is the value T_{lost} 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 T_a's commit point without interruption.

The notion of Penalty of Conflict, described above, can be introduced into the earlier dynamic priority computation formula as follows:

$Pr(T_i) = \gamma_i(\omega_1 t - \omega_2 r_i - \omega_3 d_i + \omega_4 C_i + \omega_5 U_i - \omega_6 N_i)$, where

N_i: $\Sigma_{t \in M} \ (rollback_t + exec_t)$,

M = $\{transaction\ t \mid \text{t is unsafe with } T_i\}$

In our approach we assigned the value 1 to ω_1, ω_3 and ω_6 and we did not use the other terms. The value of ω_6 will be readjusted accordingly to get the best performance.

3.5.2 High Priority Preference 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 because of their different restrictions [11].

Data conflict If there is a data conflict between two transactions, a priority-based wound-wait strategy [5] is the simplest to implement. The Conditional Restart algorithm with the estimated execution time [1] has been proposed to avoid needless aborts and rollback. The algorithm, however, has unpredictable blocking of high priority transactions due to deadlock, arrival of intermediate priority transactions, and chain blocking. The problem of unpredictable blocking due to arrival of intermediate priority transaction and chain blocking can be solved with priority inheritance [15]. But if a set of transactions are deadlocked (they access the same data items in different order), the CR method can degrade performance. The idea of HP [2, 3], which is the same as priority-based wound-wait strategy [5], is to resolve a conflict in favor of the transaction with the higher priority. In our approach, 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 (or a transaction that doesn't have the highest priority but has a zero Penalty of Conflict) 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. If the scheduler cannot be invoked immediately for several reasons (e.g Real-time UNIX [10]), the highest priority transaction can be selected from among transactions that are in the

ready queue or are currently running. When a running transaction finishes, all transactions blocked by the resources that currently running transaction hold wake up and move to ready queue. Then, the highest priority transaction is chosen as the next one for execution.

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 [3, 6] in order to reduce I/O wait. There is an I/O wait related CPU scheduling problem in real-time transaction scheduling. Consider the following scenario: Transaction T_1 is blocked and is waiting for an I/O completion. The next highest priority transaction in the ready queue, T_2 gets the CPU and starts executing so as not to waste the CPU during the I/O waiting time of T_1. 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.

The following is the pseudo code for the scheduling algorithm proposed in this thesis and is based on the notion of cost incurred due to conflicts. The procedure "IOwait-schedule" 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 relations. The procedure "tr-arrival-schedule" is called whenever a new transactions arrives and the procedure "tr-finish-schedule" is invoked whenever the running transaction finishes. These two functions use the dynamic cost of transactions in order to minimize number of missed deadlines. In this algorithm the ready queue is assumed to be sorted by Earliest Deadline First policy. The sleep

queue holds transactions that are blocked and the partially executed transaction list (P list) links all transactions that are executed partially.

```
penaltyofconflict(Tx)
Tx is a candidate for execution;
{
     sum = 0;
     for all Ti in P list
     {
          if (checkunsafe(Ti, Tx))
               sum = sum + effective service time
                           and rollback time of Ti;
     }
     return(sum);
}

checkunsafe(t1, t2)
{
   if t1 is unsafe with t2
        return(true);
   else
        return(false);
}

IOwait-schedule()
{
       if ready queue is empty  return(NIL);
       else
```

```
        {
             best = the highest priority transaction
                    from the ready queue
                    which doesn't conflict with
                    partially executed transactions
                    or null if none exists;
             return(best);
        }
}


prd(t)
t is a transaction;
{ /* We assume the value of deadline as the absolute value */
    return (negative value of deadline of transaction t);
}
```

We introduce a parameter penalty-weight (i.e, ω_6 in our priority formula) that can be used to weight the contribution of Penalty of Conflict on the value of the priority value computed. By assigning different values to penalty-weight, different scheduling policies can be obtained. For instance, if the parameter penalty-weight is assigned 0, it produces the EDF-HP. If penalty-weight is ∞ (i.e a value large enough that all decisions are based on the Penalty of Conflict), it produces the EDF-Wait. We use a value of penalty-weight between 0 and ∞ for our algorithm.

```
pr(t)
t is a transaction;
{
```

```
    return( prd(t)-penalty_weight * penaltyofconflict(t));
}

tr-arrival-schedule(Th)
Th is running transaction;
{
    Ta = choose first one in the ready queue;
    if prd(Ta) <= pr(Th)
       return(Th);        /* If prd(Ta) is less than pr(Th)  */
    else                  /* pr(Ta) also less than pr(Th)     */
    {                     /* and no one in the ready queue    */
                          /* can be higher than Th            */
        put Th in the ready queue;
        best = choose first one in the ready queue;
        Ti = choose second one in the ready queue;
        while(pr(best) < prd(Ti))/* If pr(best) >= prd(Ti) then */
        {                        /* no one in the ready Q after  */
          if (pr(best) < pr(Ti)) /* Ti can be higher than best   */
              best = Ti;
          Ti = Next(Ti); /* next one in the ready queue */
        }
        return(best);
    }
}/* end of schedule */

tr-finish-schedule()
{
    best = choose first one in the ready queue;
```

```
    Ti = choose second one in the ready queue;
    while(pr(best) < prd(Ti))
    {
       if (pr(best) < pr(Ti))
           best = Ti;
       Ti = Next(Ti); /* next one in the ready queue */
    }
    return(best);
}/* end of schedule */
```

CHAPTER 4
ANALYSIS

In order to reduce the scheduling overhead incurred from the calculation of Penalty of Conflict, we propose that priority calculation be done only whenever a transaction arrives or when a running transaction finishes. This is significantly different from previously proposed dynamic priority assignment policies with continuous evaluation, in which new priorities are calculated whenever a transaction arrives or when a running transaction finishes, or when data conflict occurs. Also note that our approach does not have any deadlock detection overhead, which is severe problem in real-time systems.

We summarize the characteristics of the proposed Cost Conscious approach in Table 4.1 which was made by using the framework that had been proposed to compare several real-time transaction scheduling in [5].

Timing Information	Arrival time Deadline
Resource requirement	Transaction access pattern
Performance metric(time)	minimize number of missed deadlines
Performance metric(consistency)	strict serializability
On-line/Off-line scheduling	Off-line transaction access pattern On-line penalty of Conflict
conflict management policy	high priority resolution
conflict resolution(active resource)	preemption
conflict resolution(data resource)	rollback
overload management	-

Table 4.1. Characteristics of new scheduling method

In order to analyze the effects of Penalty of Conflict on transaction scheduling, we did schedulability test based on strict 2-phase locking. Real-time transaction scheduling requires the transaction be preemptable, so rollback and restart are needed to keep the database consistent. We modified the following terminology from [21] for a real-time transaction context.

- s[i]: execution start time of a transaction i.
- r[i]: release time (arrival time) of a transaction i.
- e[i]: completion time of a transaction i.
- d[i]: deadline of a transaction i.
- $dataset_t^i$: data set which has been accessed by transaction i up to time t.

Valid schedule A *valid schedule* of a set of transactions T is a schedule of T satisfying the following properties:

$\forall_i$ where i $\in$ T,

1. $s[i] \geq r[i]$. This condition states that each transaction can only start execution after its release time.
2. $\forall_{j\ CONFLICT\ i\ and\ j\ UNSAFE\ with\ i}\ dataset_t^i \cap dataset_t^j = \phi$. This condition states that transactions which conflict and are not safe with can not access shared data simultaneously.

Lateness The *lateness* of a schedule of T is defined by $\Sigma(e[i] - d[i])$ where d[i] is deadline of transaction i and $i \in T$.

Feasible schedule A *feasible schedule* of a set of transactions T is a valid schedule of T such that its lateness is less than or equal to zero.

The schedulability test of real-time transactions is based on the assumption that we already know the worst case execution time, the deadline, the release time (arrival time), and the conflict relationship which is calculated off-line with a transaction pre-analysis. Schedulability assumptions of real-time transactions can be so pessimistic (especially, the execution time and the conflict relation assumptions) that it may be unrealistic. However, it can give us predictable, general behavior of real-time transaction scheduling and can be a good guideline for periodic real-time transaction systems. In Figure 4.1, we assumed that data conflict between transactions that have CONFLICT relation occurred at the beginning of a transaction.

Three of the possible valid schedules (including one feasible schedule) in Figure 4.1 are instances of scheduling according to EDF-HP, Cost Conscious (EDF with Penalty of Conflict) with HP, EDF-Wait [3]. To show the performance of our approach, we illustrate the effect of Penalty of Conflict in a few examples. If we set the value of penalty-weight in the "pr" function of the scheduling algorithm to 0, 1 and ∞, we get 3 different algorithms respectively: EDF-HP, CC and EDF-Wait. In the following examples we assumed all data were memory resident and data conflict occurred at the beginning of transactions.

Let's explain the Example 1 in Figure 4. At time 0 transaction D arrives and it is the only transaction that can be executed. At time 20 transaction D finishes its execution and no schedule happened during its execution. At time 40 transaction A arrives and it is the only candidate. At time 50 transaction C arrives and C has earlier deadline than currently executing transaction A. According to previous priority calculation of CC approach the priority of A is -110 and the priority of C is -101 (-91-10) because A is partially executed transaction and A CONFLICT with B. Thus higher priority transaction C is executed and A is aborted. Until this point, The CC scheduling is the same as the EDF scheduling.

```
With the following data we can derive several valid schedules.
r[A]=40  r[B]=60 r[C]=50 r[D]=0   A CONFLICT B
c[A]=20  c[B]=20 c[C]=20 c[D]=20  A CONFLICT C
d[A]=110 d[B]=90 d[C]=91 d[D]=120 B CONFLICT C
r[i]: release time
c[i]: worst case execution time
d[i]: deadline
CONFLICT relation is symmetric.
```

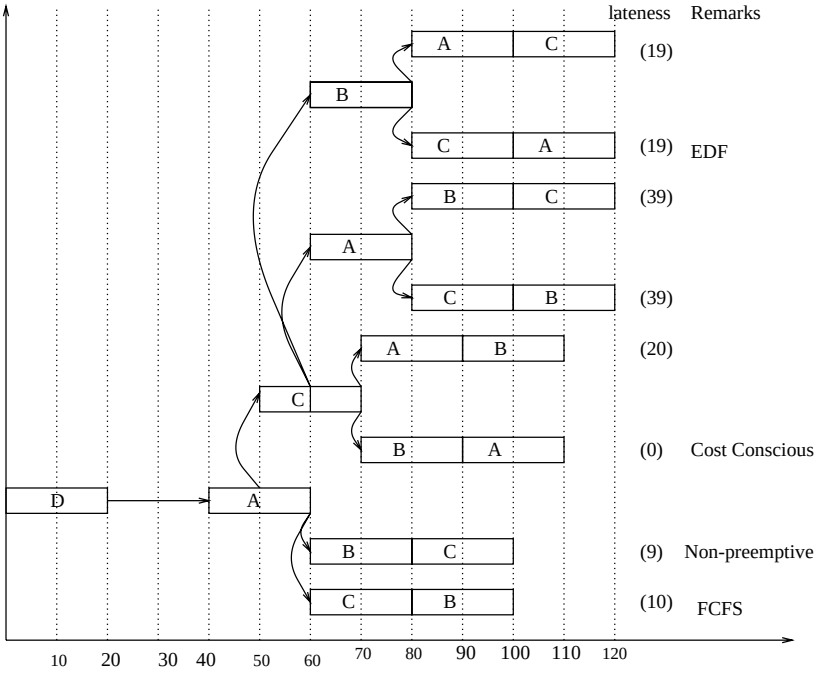

Figure 4.1. Valid schedules

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. The priority of B is -100 (-90-10), the priority of C is -91 and the priority of A is -120 (-110-10). Thus the highest priority transaction C is executed in the CC scheduling. However transaction C is aborted at time 60, as B has the earliest deadline. This is what our approach differs from EDF-HP.

At time 70 transaction C finishes and B,A is in the ready queue. The priority of B is -90 and the priority of A is -110 here. Thus B is executed. After B finishes A is executed in the CC scheduling.

From the following 2 examples in Figure 4.3 and Figure 4, at time 10 transaction A and B is possible candidates. The priority of A is -110 and the priority of B is -111 (-101-10) because A is partially executed transaction. Thus A is not aborted by transaction A which has earlier deadline.

In Example 3 we changed the CONFLICT relationship to make high data contention. In this case the load of the system is increased by transaction abort and restart not by transaction arrival rate. Thus the total lateness of EDF approach is increased to 30.

The data in Example 1 is borrowed from [21]. Here we can see that CC approach performs better than the others.

Why the CC scheduling is better than EDF-HP? In this example the number of transaction restart in EDF-HP is 2 and the CC scheduling is 1. The CC scheduling reduced the number of transaction restart using Penalty of Conflict. Thus the CC scheduling tried to solve the problem of EDF-HP which has too many transaction aborts.

Why the CC scheduling is better than EDF-Wait? The problem of EDF-Wait is too many transaction blocking. The CC scheduling tried to solve this problem with well decided transaction restart.

In Example 3 we can see also the same result. The CC scheduling is better than EDF-HP and EDF-Wait. The reason for better performance is that the CC scheduling did more clever decision on blocking and restart.

From the Example 2 and Example 3, we can see that the change of transaction relationship easily make the system heavily loaded and EDF-HP could not adapt to

the change. But our CC scheduling shows very stable performance and ability of adaptation.

These examples show us great possibility of CC approach. In the following chapters we do performance comparison and present the effect of penalty-weight using the RTDBS simulation program.

```
Example 1:
r[A]=40  r[B]=60 r[C]=50 r[D]=0   A CONFLICT B
c[A]=20  c[B]=20 c[C]=20 c[D]=20  A CONFLICT C
d[A]=110 d[B]=90 d[C]=91 d[D]=120 B CONFLICT C
```

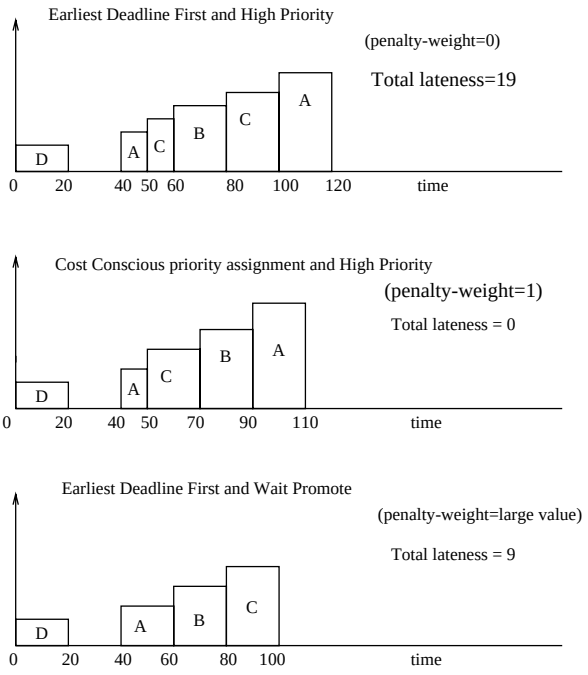

Figure 4.2. Output of Example 1

```
Example 2:
r[A]=0   r[B]=10  r[C]=60 A CONFLICT B
c[A]=50  c[B]=20  c[C]=30 B CONFLICT C
d[A]=110 d[B]=101 d[C]=90
```

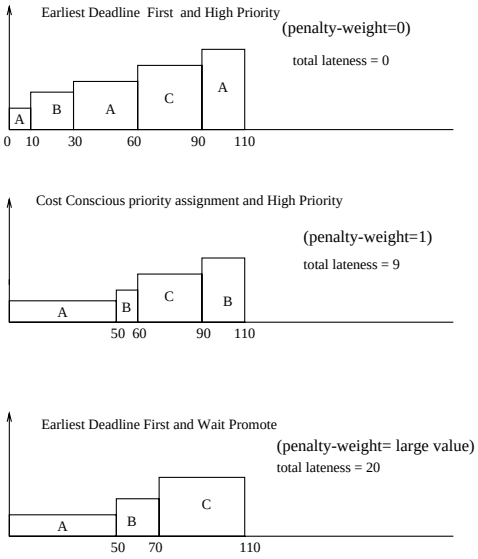

Figure 4.3. Output of Example 2

```
Example 3: The same as Example 2 except transaction relationship
r[A]=0   r[B]=10  r[C]=60 A CONFLICT B
c[A]=50  c[B]=20  c[C]=30 B CONFLICT C
d[A]=110 d[B]=101 d[C]=90 A CONFLICT C
```

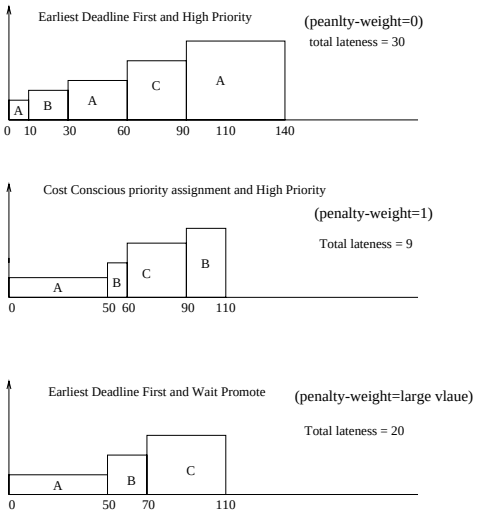

Figure 4.4. Output of Example 3

CHAPTER 5
COST CONSCIOUS FOR MAIN MEMORY DATABASE

5.1 Simulation Result

In order to evaluate the performance of the CC algorithm we wrote a simulation of real-time transaction scheduler using C language and SIMPACK simulation package [9]. The discrete events and actions of the simulation program are illustrated in the Figure 5.1.

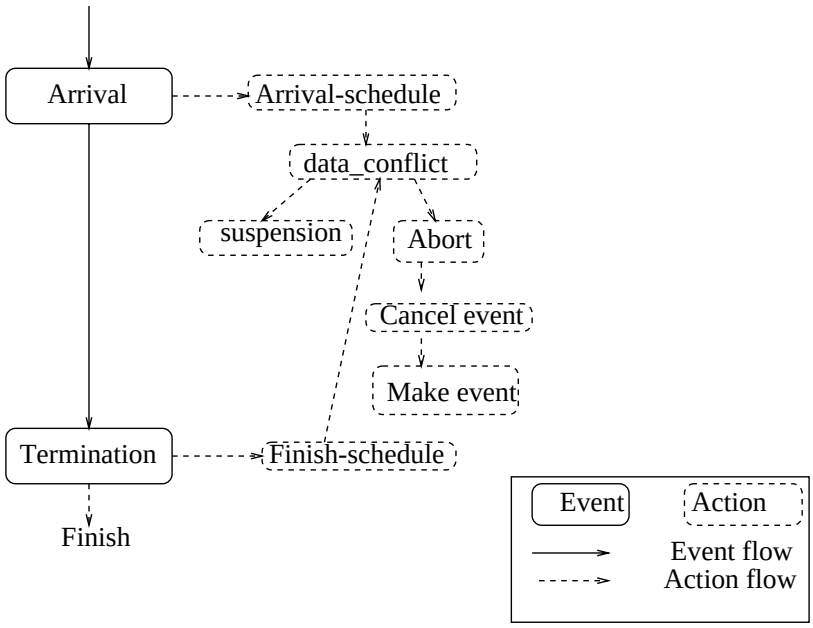

Figure 5.1. Flow of simulation program

In this simulation we have single processor, memory resident database and single entry and exit points in the transaction program. Thus there is no I/O wait and no condition in transaction programs.

Transactions enter the system with exponentially distributed inter-arrival times and they are ready to execute when they enter the system (i.e., release time equals

arrival time). 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 execution 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 is calculated by summing execution time and slack time which is calculated by multiplying slack percent and execution time. Slack percent is chosen uniformly from the range of Min-slack and Max-slack.

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

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 following parameters in Table 5.1 in order to see the behavior of CC method on various environment.

Arrival-rate The average arrival rate of new transactions entering the system.

Update-time The variation of time needed to update one item.

DBsize The number of objects in the database.

Penalty-weight The portion of penalty of conflict compared to that of deadline in priority calculation.

The simulation results shows that CC performs better than EDF in the wide range of arrival rate and in the situation where many transaction aborts are occurred

due to heavy data contention among transactions. Thus we can see that CC can adapt to changes on the system load caused not by transaction arrival rate but by data contention among transactions.

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 5.1. Base parameters

5.1.1 Effect of Arrival Rate

In this experiment, we varied arrival rate from 1 trs/sec to 20 trs/sec with the base parameters shown in Table 5.1. With this base parameters the capacity of the system is:

$$\frac{4\ ms}{update} \times \frac{20\ update}{transaction} = \frac{80\ ms}{transaction} = 12.5\ transactions/second$$

Thus we are interested in the range from 1 trs/sec to 13 trs/sec because past this we are on overload (it's different from overload in hard real-time system) and we need a special mechanism that works well on this situation. This calculation is very optimistic because it doesn't include abort cost nor the cost of re-executing transactions. Table 5.2 shows the behavior of CC and EDF on the range from 1 to 20 of arrival rate and we can see that more than 90 % of the transactions are missed from the arrival rate 14. From now on we are focusing the simulation on the range from 1 to 15 and Figure 5.2 plotted Table 5.2 on the range 1 to 15.

Table 5.3 and Figure 5.3 show the improvement of CC over EDF in term of lateness and the number of transactions that have missed their deadlines. The improvement of CC over EDF is calculated like below:

$$n = \frac{EDF \ - \ CC}{EDF} \times 100$$

It means that the miss rate or total lateness of CC is n% of the miss rate or total lateness of the EDF.

We observe that with the base parameters in Table 5.1 the number of restarts climbs steeply up to arrival rate 8 and then declines sharply from the peak point in Figure 5.4. 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 after peak point. After peak point, it is usually the case that the currently running transaction arrived a long time ago, but could not get system services due to heavy load of the system. Thus fewer transactions are preempted and there are fewer opportunities for restarts [1]. The improvement graph of CC over EDF is almost the same shape as the graph of transaction abort. Before this peak point, better decisions about transaction aborts and restarts help to improve performance.

After this peak point, however,an abortive scheduling algorithm cannot contribute much to the performance of the system. Instead nonabortive method like CC with large penalty-weight shows great performance in Table 5.4.

The number of partially executed transactions in the experiment with base parameters is 1 to 2 with the changes of arrival rate from 1 trs/sec to 15 trs/sec. Thus scheduling overhead of the CC scheduling will not make problem.

5.1.2 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 is:

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

The variation of update time create a lot of variance in the transaction execution time. The execution time of transaction varies from 4 ms to 1200 ms. So there will be more chances for transaction preemption. Table 5.5 and Figure 5.5 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 more transactions are preempted and there are more opportunities for restart. In Figure 5.6 we can see that there are more restarts than in the previous experiment. In addition the improvement of CC over EDF is a little higher in term of miss rate and total lateness. In table 5.5 the improvement is calculated in terms of transaction miss rate.

5.1.3 Effect of Database Size

In this experiment, we fixed every parameter except database size. When the database size increases, system load decreases. We selected 2 values of arrival rate one before and one after the peak point to see the effect of database size on the different load situation. Table 5.6 and Figure 5.7 show the effect of DB size on arrival rate 5. Table 5.7 and Figure 5.8 show the effect of DB size on arrival rate 10. On these 2 different load situation CC shows better performance and flatter curve than EDF when database size is smaller.

5.1.4 Effect of Penalty-Weight

With the environment of base parameters the peak point of the system is around value 8 of arrival rate. This value is largely dependent on the lengths of transactions that are run on the system. After the peak point, Table 5.4 shows that CC performs better with large penalty-weight.

We can see that the best value of penalty-weight slightly increase with the changes of arrival rate up to the peak point and the rate of increase is not much dependent on the data contention in Table 5.8 and Table 5.9. With the information in Table 5.8 and Table 5.9 the value 3 is the best for penalty-weight because CC shows best performance in the range of 1 trs/sec to 8 trs/sec with this value of penalty-weight.

Figure 5.10 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.

Arrival	*EDF*			*Cost Conscious*		
Rate(n/sec)	Miss	Restart	lateness	Miss	Restart	lateness
1	11.4	39.4	420.6	11.1	36.0	401.0
2	20.9	81.7	1445.1	20.2	74.5	1419.9
3	12.9	116.5	527.1	11.1	104.9	402.0
4	22.6	160.2	1143.9	18.7	143.7	970.8
5	34.8	190.7	2170.6	27.3	168.4	1601.7
6	47.4	224.4	4456.9	39.2	202.1	3500.5
7	60.5	235.8	4876.5	46.6	212.7	3308.2
8	113.3	246.5	10834.2	86.7	224.0	7248.0
9	192.3	235.7	29078.7	157.0	214.4	22834.9
10	407.2	170.1	126201.4	350.9	159.3	106718.6
11	579.2	119.1	293405.2	536.9	112.3	272081.5
12	661.9	96.2	675215.1	624.4	90.1	649990.5
13	887.7	30.7	2279295.6	865.5	28.9	2213611.4
14	953.8	13.9	4882713.6	942.5	14.4	4813463.6
15	949.7	12.5	5906608.0	940.2	13.0	5875306.4
16	970.5	7.9	7319176.0	964.6	7.9	7275576.0
17	988.5	5.0	9830349.6	986.2	4.5	9763058.4
18	988.8	3.7	12152108.8	988.1	3.2	12136897.6
19	988.1	4.4	11439751.2	986.7	3.1	11384251.2
20	987.8	3.8	13984155.2	987.3	3.5	13970342.4

Table 5.2. EDF, CC with base parameters

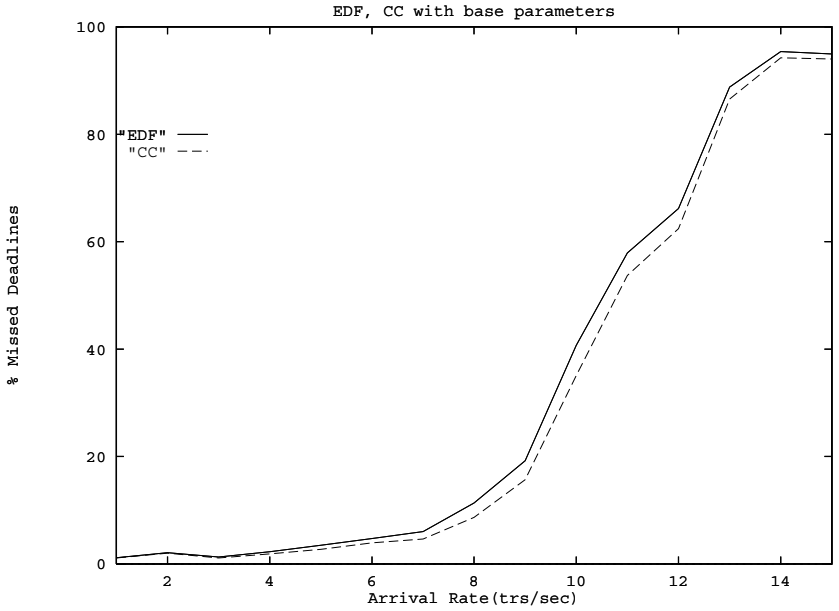

Figure 5.2. The plot of EDF, CC with base parameters

Arrival Rate (n/sec)	*Improvement(%)*	
	Miss	Lateness
1.0	2.6	4.7
2.0	3.4	1.7
3.0	14.0	23.7
4.0	17.3	15.1
5.0	21.6	26.2
6.0	17.3	21.5
7.0	23.0	32.2
8.0	23.5	33.1
9.0	18.4	21.5
10.0	13.8	15.4
11.0	7.3	7.3
12.0	5.7	3.7
13.0	2.5	2.9
14.0	1.2	1.4
15.0	1.0	0.5

Table 5.3. Improvement of CC over EDF

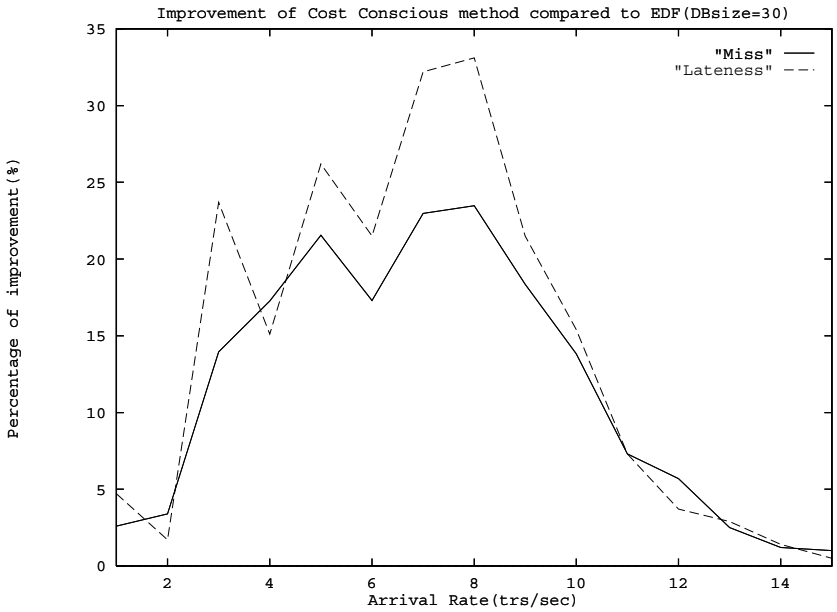

Figure 5.3. The plot of improvement

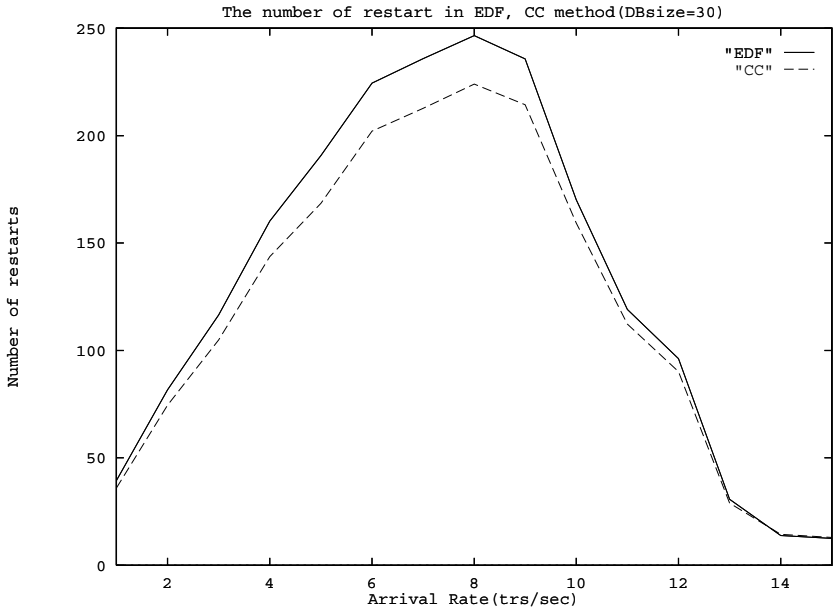

Figure 5.4. The plot of restart

Arrival Rate	*CC(penalty-weight=150)*		
(n/sec)	Miss	Restart	Lateness
1.0	16.9	2.9	576.8
2.0	30.6	6.1	1774.1
3.0	25.7	9.9	904.8
4.0	36.2	12.0	1615.5
5.0	47.8	14.2	2341.3
6.0	57.8	16.9	3631.5
7.0	57.8	18.6	2939.1
8.0	86.3	21.0	4196.6
9.0	130.1	23.1	11728.5
10.0	270.3	18.8	63055.4
11.0	429.2	15.8	206092.3
12.0	515.7	15.8	556632.7
13.0	798.2	6.6	2124728.8
14.0	918.0	2.4	4648724.8
15.0	922.7	3.6	5756830.8

Table 5.4. CC with penalty-weight=150, Dbsize=30

Arrival	*EDF*			*Cost Conscious*			%
Rate	Miss	Restart	lateness	Miss	Restart	lateness	Improvement
0.1	6.8	19.7	1964.1	6.6	19.1	1896.8	3.0
0.2	21.0	45.7	2539.7	20.3	43.1	2160.4	3.3
0.3	10.8	67.8	1793.4	10.3	65.3	1367.8	4.6
0.4	13.0	98.2	5812.8	11.1	92.8	4799.3	14.6
0.5	20.2	126.2	2804.1	18.8	119.9	1725.4	6.9
0.6	28.0	174.6	11073.1	26.2	164.1	7711.3	6.4
0.7	24.2	174.1	5645.8	21.4	165.2	3284.8	11.6
0.8	35.2	197.9	11083.9	30.5	187.5	6916.6	13.4
0.9	45.6	245.3	11724.4	42.4	231.6	7625.6	7.0
1.0	51.9	310.1	28814.0	42.5	292.1	16033.8	18.1
1.1	58.8	317.6	36333.7	44.7	301.1	20636.8	24.0
1.2	78.6	337.6	48822.1	61.9	319.4	27246.1	21.2
1.3	94.0	376.5	54429.3	73.1	357.3	25645.9	22.2
1.4	115.4	395.5	84087.4	87.5	371.0	49457.8	24.2
1.5	160.9	404.2	136465.1	128.4	383.7	83283.7	20.2
1.6	153.0	411.7	113285.9	120.1	389.2	65795.6	21.5
1.7	186.9	421.4	177602.0	154.8	404.1	111429.5	17.2
1.8	284.0	433.2	287929.5	228.3	424.5	187944.2	19.6
1.9	264.7	408.9	258739.9	227.7	393.9	179705.5	14.0
2.0	326.1	400.0	364927.7	255.5	397.6	232006.3	21.6
2.1	394.2	386.1	523226.4	337.9	382.4	361329.8	14.3
2.2	443.6	368.2	582882.0	364.5	371.8	385641.7	17.8
2.3	504.2	316.8	935464.7	434.2	323.9	725028.9	13.9
2.4	507.7	333.2	836843.5	439.6	346.7	633239.6	13.4
2.5	519.0	306.3	1046373.2	461.7	306.6	865621.1	11.0
2.6	648.8	234.3	1822628.6	592.1	243.5	1550638.4	8.7
2.7	675.0	222.4	2224897.2	614.7	232.8	1985902.8	8.9
2.8	609.2	253.5	2053094.2	549.2	260.1	1834712.8	9.8
2.9	704.8	197.3	2486331.2	654.4	207.2	2155261.6	7.1
3.0	790.0	146.3	4737714.0	742.8	160.5	4320996.8	6.0
3.1	809.8	131.0	4954517.6	768.6	140.8	4605974.4	5.1
3.2	773.9	154.7	3146292.0	742.0	157.7	2954192.0	4.1
3.3	836.0	108.7	5283436.0	804.2	116.1	4925655.2	3.8
3.4	880.1	83.4	12420894.4	857.5	88.0	12167400.0	2.6
3.5	898.6	68.4	11563330.4	877.5	72.4	11169936.8	2.3
3.6	885.8	79.2	12664900.0	867.7	82.1	12426168.8	2.0
3.7	902.1	71.1	11365877.6	876.5	77.4	10935632.8	2.8
3.8	935.2	45.8	16432209.6	923.9	46.3	15940080.0	1.2
3.9	942.4	41.2	18107593.6	934.3	41.7	17596182.4	0.8
4.0	960.7	28.3	23524508.8	953.0	29.2	23225604.8	0.1

Table 5.5. EDF,CC with variation of update time

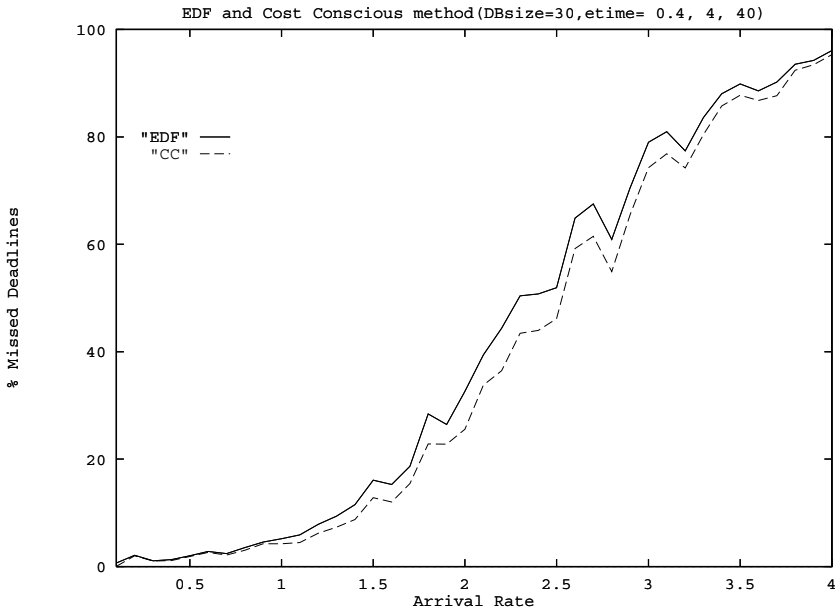

Figure 5.5. Effect of variation of update time

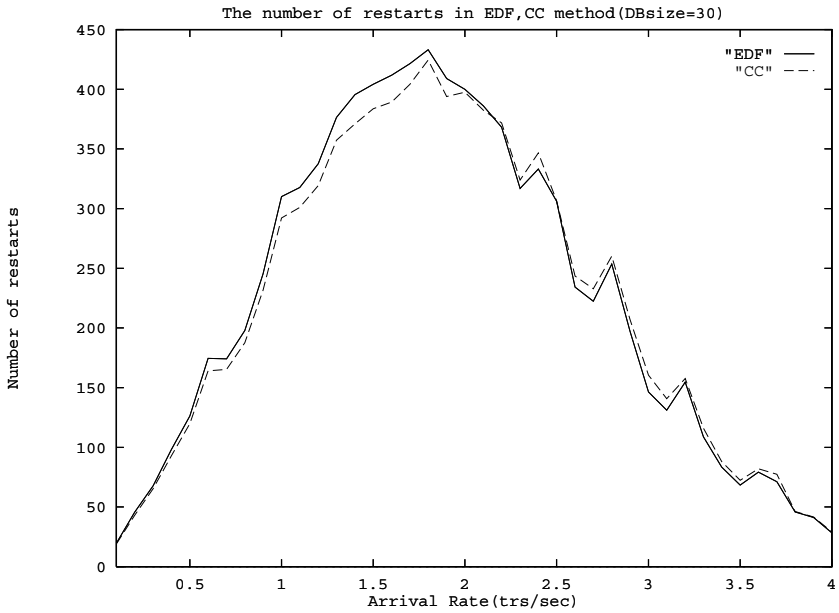

Figure 5.6. Number of restart with variation of update time

DBsize	*Number of miss*		% of improvement
	E D F	Cost Conscious	(Miss)
30.0	34.8	27.3	21.6
50.0	34.8	26.8	23.0
100.0	33.0	26.6	19.4
200.0	30.1	25.1	16.6
300.0	28.5	24.7	13.3
400.0	27.1	24.5	9.6
500.0	27.2	24.0	11.8
800.0	25.1	22.9	8.8
1000.0	24.4	22.7	6.9

Table 5.6. Effect of DB size(Arrival Rate=5)

DBsize	*Number of miss*		% of improvement
	E D F	Cost Conscious	(Miss)
30.0	407.2	350.9	13.8
50.0	397.0	343.7	13.4
100.0	371.2	329.6	11.2
200.0	343.8	315.3	8.3
300.0	326.6	303.0	7.2
400.0	318.6	296.3	7.0
500.0	305.3	288.0	5.7
800.0	289.9	279.1	3.7
1000.0	285.5	274.0	4.0

Table 5.7. Effect of DB size(Arrival rate=10)

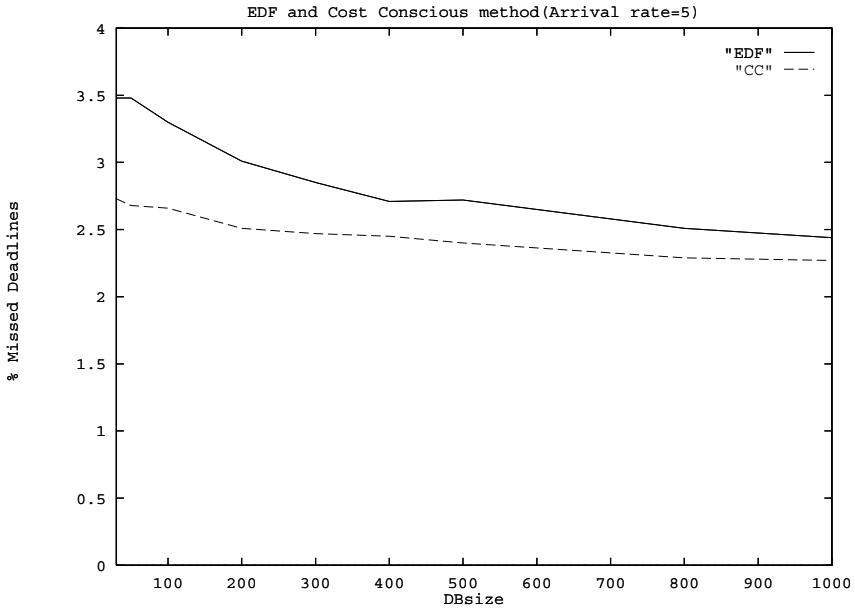

Figure 5.7. Effect of DB size (Arrival Rate=5)

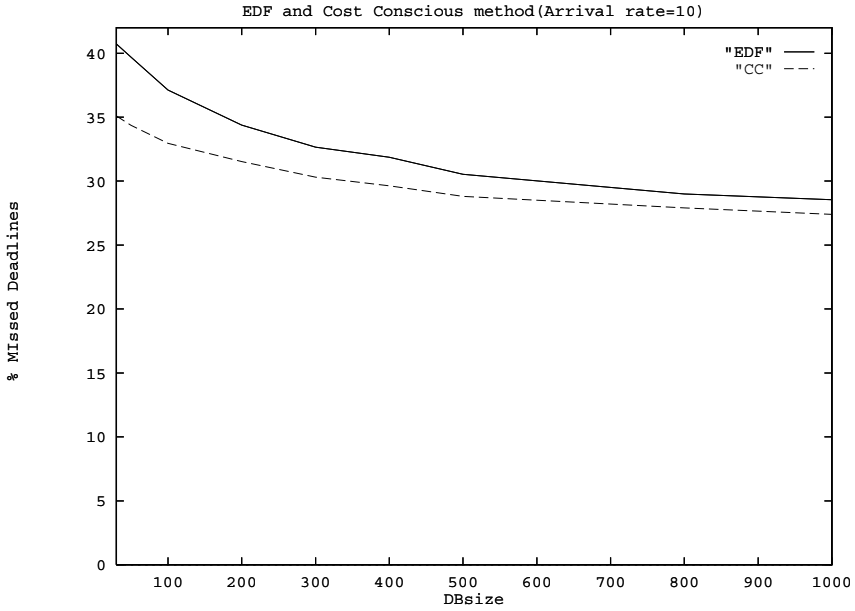

Figure 5.8. Effect of DB size(Arrival Rate=10)

Arrival rate	*Penalty-weight*					
	0	1	2	3	4	5
1	11.4	11.1	11.4	11.5	12.1	12.2
2	20.9	20.2	20.1	20.6	21.1	21.5
3	12.9	11.1	11.7	12.4	12.6	13.1
4	22.6	18.7	18.0	17.8	18.4	19.2
5	34.8	27.3	26.7	27.1	27.5	28.5
6	47.4	39.2	35.6	32.4	33.0	34.2
7	60.5	46.6	39.2	38.0	38.2	38.2
8	111.3	86.7	76.5	68.8	64.6	65.4

Table 5.8. Number of miss with the changes of penalty-weight(DBsize=30)

Arrival rate	*Penalty-weight*					
	0	1	2	3	4	5
1	10.5	10.5	10.5	10.5	10.6	10.6
2	19.9	20.0	20.1	20.3	20.6	20.6
3	10.0	9.8	10.0	10.0	10.0	10.2
4	16.0	15.6	15.8	16.1	16.2	16.8
5	24.4	22.7	23.0	23.1	23.5	23.6
6	27.7	27.2	27.8	27.6	28.3	28.4
7	31.7	30.5	30.4	29.7	30.1	30.8
8	58.6	52.9	51.3	50.3	50.5	50.1

Table 5.9. Number of miss with the changes of penalty-weight(DBsize=1000)

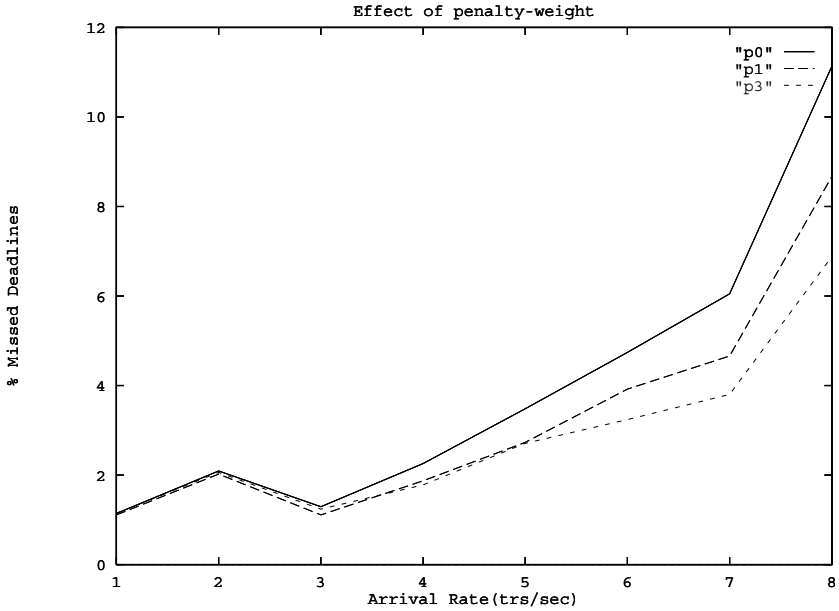

Figure 5.9. Effect of penalty-weight

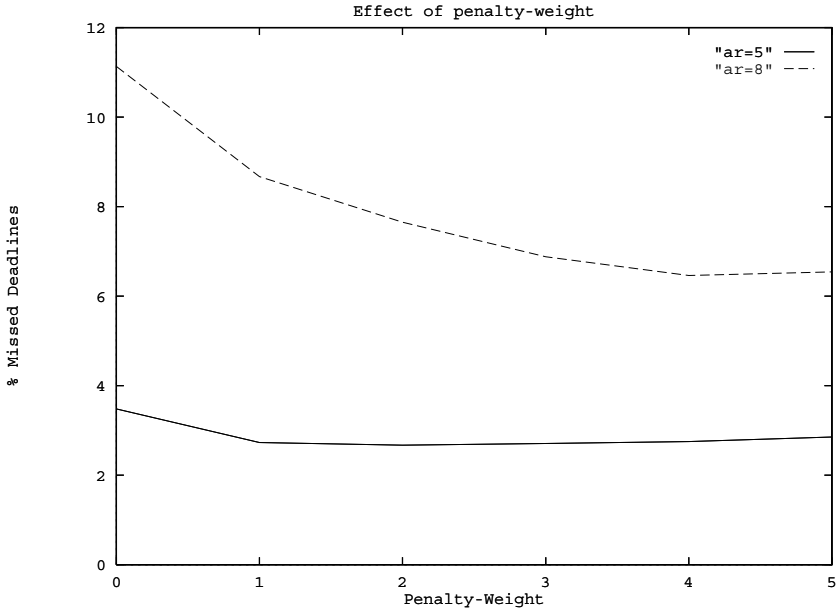

Figure 5.10. Stability of penalty-weight

CHAPTER 6
COST CONSCIOUS FOR DISK RESIDENT DATABASE

6.1 Simulation Result

In order to see the performance of our algorithm on disk resident database we extended the simulation program of real-time database system. The event and action flow of the simulation program are illustrated in Figure 6.1. In this simulation we assumed that we have single processor, single disk, FCFS I/O scheduling and 1 enter and 1 exit point in transaction programs.

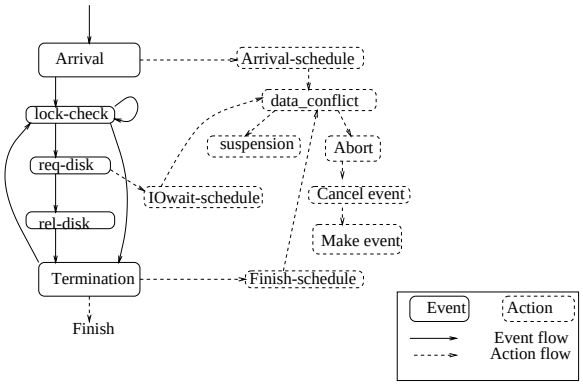

Figure 6.1. Flow of simulation program

The environment of this experiment is the same as previous simulation on main memory database. We ran the simulation with the same parameter for 10 different random number seeds and 500 transactions are executed at each run. For each algorithm the result were collected and averaged over the 10 runs.

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 release the disk.

Parameter	Value
Transaction type	50
Update per transaction(mean, std)	(20, 10)
Database size	100
Min slack as fraction of total runtime	20(%)
Max slack as fraction of total runtime	800(%)
abort cost(ms)	5
weight of penalty of conflict	1
Computation/Update time(ms)	4
Disk access time(ms)	25
Disk access probability	1/10

Table 6.1. Base parameters

The base parameters in this simulation program was selected to avoid the bottleneck at the disk access. With the base parameters in the Table 6.1 the capacity of the system is:

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

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:

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

The number of partially executed transactions is from 1.2 to 2.7 with the changes of arrival rate from 1 tr/sec to 6 trs/sec. This value is just a little larger than that of previous simulation on main memory database. Thus the transaction scheduling overhead will not make problem either.

6.1.1 Effect of Arrival Rate

In this experiment, we varied arrival rate from 1 trs/sec to 6 trs/sec with the base parameters shown in Table 6.1. Table 6.2 shows the improvement of CC over EDF.

The improvement is better than that of main memory resident case. The reason for better result is that CC not only do better decision about transaction blocking and restart but also prevent noncontributing execution.

With the increase of arrival rate the number of transaction restarts are also increased. This phenomenon is different from previous simulation on main memory database.

The reason for continuous increase in our CC approach is that when the arrival rate is high the possibility of restart of the partially executed transactions which are picked by "IOwait-schedule" is very high while the possibility of restart of partially executed transactions which are selected by "tr-arrival-schedule" or "tr-finish-schedule" is very low.

The improvement over EDF in terms of the number of transactions that have missed their deadlines increase to certain point and then decrease sharply. However the total lateness improvement do not decrease sharply in Table 6.2,Table 6.3. The reason for this phenomenon is that when the arrival rate is high "IOwait-schedule" function in CC approach prevent noncontributing execution effectively.

6.1.2 Effect of Disk Access

In this experiment we changed computation time per update as 1 and probability of disk access as 1/3. With this parameters the capacity of the system is dependent on the disk capacity. The capacity of the system is:

$$\frac{20/3\ items}{transaction} \times \frac{25}{item} = \frac{166.7\ ms}{transaction} = 6.0\ trs/second$$

When the load of the system reaches its disk capacity the utilization of the CPU is:

$$\frac{6.0 \times 20 \times 2/3 \times 1}{1000\ ms} \times 100 = 12.5\%$$

In Table 6.3 we can see that the performance is dominated by the disk IO and noncontributing execution. In this situation, the CC scheduling shows great performance.

6.1.3 Effect of Penalty-Weight

In this experiment with base parameters in Table 6.1 the best value of penalty-weight is 8. Figure 6.2 shows the stability of penalty-weight and Figure 6.3 represent the effect of penalty-weight with the changes of arrival rate. The performance of the system is not sensitive to the selection of penalty-weight within some range also in this experiment.

Arrival	*EDF*			*Cost Conscious*			*% improve*	
Rate	Miss	Restart	lateness	Miss	Restart	lateness	Miss	Late
1.0	81.3	99.4	11432.5	73.1	31.5	6412.4	10.0	43.9
2.0	175.1	396.4	117192.0	128.0	59.0	24578.8	26.9	79.0
3.0	392.5	8826.9	28565517.6	245.9	56.0	126220.7	37.4	99.6
4.0	486.8	15429.0	70993739.8	396.5	54.4	611048.4	18.5	99.1
5.0	494.5	23187.2	113379641.2	477.4	120.0	5002427.6	3.5	95.6
6.0	494.9	20348.1	115349316.2	488.8	153.7	8510464.8	1.2	92.6

Table 6.2. EDF and CC with base parameters

Arrival	*EDF*			*Cost Conscious*			*% improve*	
Rate	Miss	Restart	lateness	Miss	Restart	lateness	Miss	Late
1.0	12.3	81.2	2518.2	6.8	32.5	971.8	44.7	61.4
2.0	52.8	324.1	33680.4	13.4	68.6	3294.6	74.6	90.2
3.0	340.3	7273.8	23623866.3	29.3	96.0	4201.4	91.4	99.9
4.0	474.8	14306.9	65226303.9	83.4	87.8	37055.6	82.4	99.9
5.0	485.9	14910.3	75612101.0	261.6	85.8	420432.2	46.2	99.4
6.0	488.7	18702.4	101851721.8	427.5	108.9	2333854.8	12.5	97.7

Table 6.3. EDF,CC with high probability disk IO

Arrival rate	*Penalty-weight*						
	0	1	3	5	8	10	10000
1	81.3	73.1	70.3	69.6	68.5	70.0	74.2
2	175.1	128.0	116.7	114.3	115.2	119.0	126.3
3	392.5	245.9	237.0	211.6	197.3	195.8	201.1
4	486.8	396.5	417.0	360.0	405.5	369.8	356.7
5	494.5	477.4	470.9	482.8	462.7	465.2	458.4
6	494.9	488.8	481.6	482.1	482.4	480.6	481.1

Table 6.4. Number of miss with the changes of penalty-weight

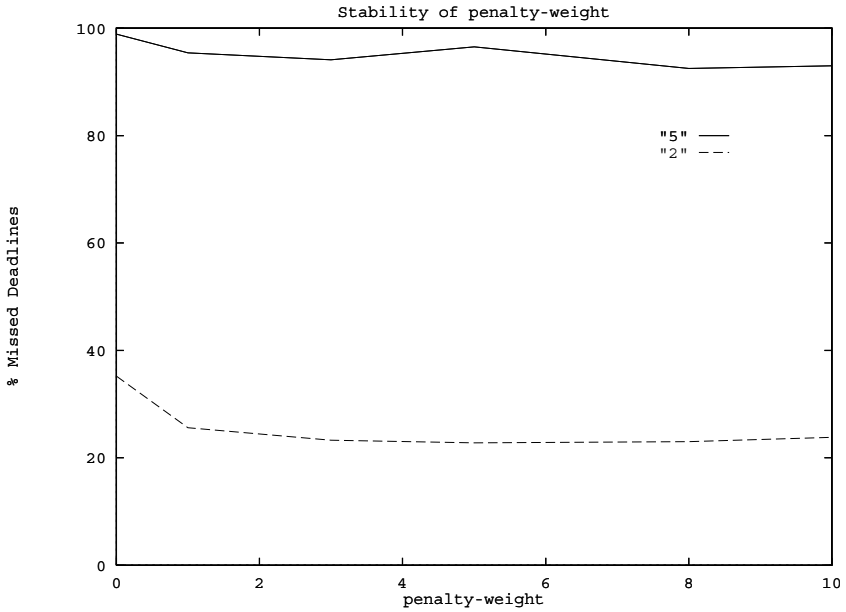

Figure 6.2. Stability of penalty-weight

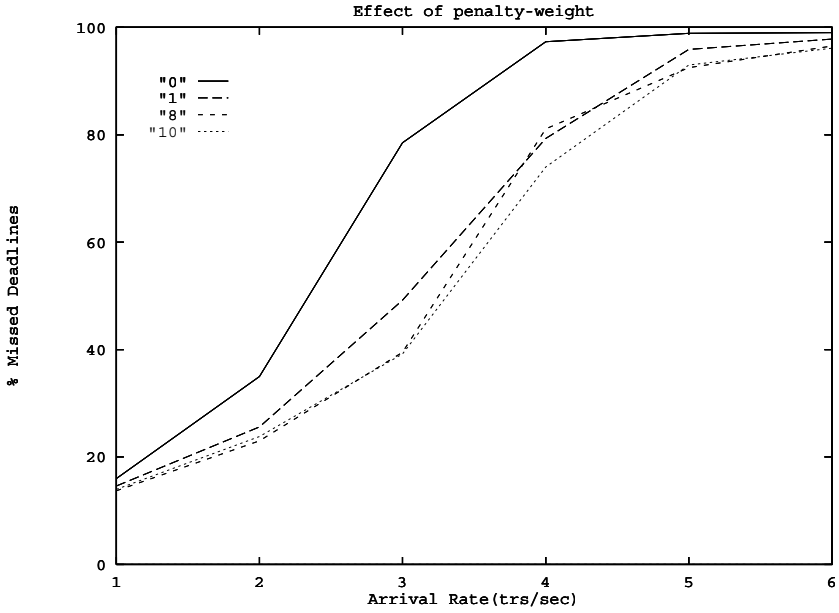

Figure 6.3. Effect of penalty-weight

CHAPTER 7
COST CONSCIOUS FOR MULTIPLE EXIT TRANSACTION

In the previous 2 chapters, all transactions have only single exit point in their programs. Thus, there is no decision point in the program. These transaction programs can be represented by transaction trees that have only one vertex.

Previous studies which are based on transaction pre-analysis [15],[5] made very simple assumption about the transaction access pattern. The weakness of transaction pre-analysis based approaches is that they have severe performance degradation for multiple exit point transactions. They might be worse than EDF-HP when dealing with complex multiple exit point transactions.

Abbot and Garcia-Molina [2] showed that EDF-CR which uses an estimated execution time for wait and abort decision might be worse than EDF-HP if the estimated execution time of a transaction is more than 120 % of real execution time of the transaction. If we consider a multiple exit point program which can be represented as a transaction tree that has variable length of paths, statically estimated execution time of a transaction is not appropriate.

A Priority Ceiling Protocol [15] that assigns a priority ceiling to each data item statically has severe problems handling multiple exit point transaction. A worst case static assignment of priority ceilings to data items makes the Priority Ceiling Protocol resemble to nonpreemptive method. It's an undesirable situation.

In the following experiment, we will look at the effect of multiple exit points in our approach, using the base parameters in chapter 5. In this case often we cannot decide whether two transactions are safe or unsafe which means we have a *conditionally unsafe*. We can deal with this in 3 ways:

1. Consider conditionally unsafe as safe.

2. Consider conditionally unsafe as unsafe.

3. Consider conditionally unsafe as unsafe and use only half of Penalty of Conflict.

In this experiment we selected the first method because the worst case bound on its performance is the same as that of EDF-HP. Whenever we decide the relationship to calculate the Penalty of Conflict the relationship is whether it is safe or not safe. Not safe can be unsafe or conditionally unsafe due to the existence of branches in a transaction program. The total number of decision minus the number of safe case is the number of not safe case. If we assume that m % out of the number of not safe case are unsafe and the others are conditionally unsafe and we consider conditionally unsafe as safe according to first method whenever we make a decision and our guess is completely wrong. It means all conditionally unsafe turn into unsafe. This situation is very unrealistic but it could happen. If m have value 100 it is the same as 1 exit point case.

In Table 7.1 and Figure 7.1 we can see that even if we assume worst situation the bound on performance of CC in terms of miss rate is the same as that of EDF-HP when we ignore the scheduling overhead. Additionally dynamically changing *mightaccess* will help to improve % of unsafe out of not safe. Thus CC approach is more promising than previous transaction pre-analysis based approaches that only use statically estimated information.

Arrival	*% of unsafe*					
Rate	0	25	50	75	90	100
5	0	0	1.4	4.5	19.0	21.6
8	0	0	0	5.9	12.9	23.5
12	0	0	0.3	1.2	3.7	5.7

Table 7.1. Effect of unsafe out of not safe

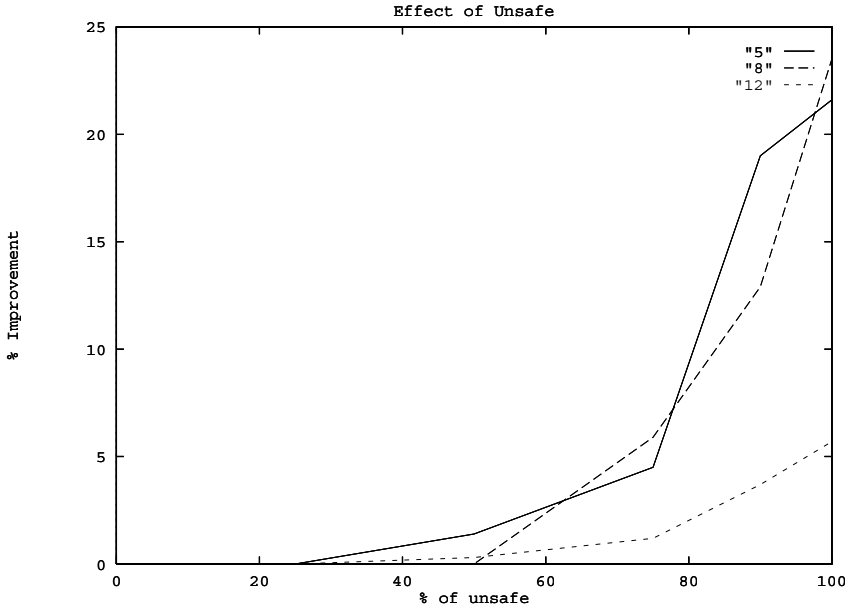

Figure 7.1. Effect of % of unsafe

CHAPTER 8
CONCLUSION AND FUTURE WORK

To the best of our knowledge, all previous abortive methods of real-time transaction scheduling have not considered the dynamic cost, i.e, the cost of rolling back and restarting transactions. This perhaps is not a key consideration in real-time task scheduling that only consider timing correctness. But in real-time transaction scheduling, the cost incurred at run time to keep the database consistent should be considered as a key factor.

EDF-HP and Priority Ceiling Protocol are the extreme methods of abortive and nonabortive approach respectively. Even though CR method and Stankovic's two protocols have been suggested to compromise abortive and nonabortive method, they all have deadlock problem.

In this thesis, we have proposed a new real-time transaction scheduling algorithm that has a cost conscious dynamic priority assignment policy. Our approach uses dynamic priority assignment with continuous evaluation method to adapt system to the changes of load effectively and resolves the problem of EDF-HP encounters in heavily loaded situations.

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 implicitly taken into account as it is a part of the Penalty of Conflict computed for conflicting transactions.

Second, our approach is deadlock free. Whenever data conflict occurs, running transaction gets the CPU and abort conflicting transaction. It is the same as the well known priority-based wound-wait scheme which is a deadlock avoidance scheme.

Third, to the best of our knowledge no research has mentioned the fact that the arrival of conflicting transactions easily make the systems heavily loaded. Our priority assignment policy easily adapts to the changes of data contention using Penalty of Conflict and works well in an high data contention.

Fourth, no starvation in our approach. If we consider the deadline of a transaction when calculating the transaction priority (i.e, $penalty - weight < \infty$), then we avoid starvation.

When all transactions access disjoint data sets or when system is lightly loaded, there is no starvation problem because the priority assignment policy proposed here is the same as EDF.

When the system is heavily loaded, transactions that have large penalty of conflict usually are delayed because of their low priority. But the effect of deadline prevent those transactions from going into starvation when the urgency of deadlines of the transactions compensate the effect of their penalties, those transactions get high priority.

Fifth, no priority reversal in our approach. The only thing that changes relative priorities is the partially executed transactions in the system and only the running transaction can abort partially executed transactions. Thus the transactions in the ready queue which are already compared the priority with the running transaction cannot have higher priorities than that of running transaction.

In this thesis we assumed that we only have exclusive lock and same criticalness in the system. The effect of shared lock in transactions and multiple criticalness will

affect the performance of whole system. Much more work is needed to include these factors.

Our approach is more computationally expensive than EDF-HP. However current trend of real-time system is tightly or loosely coupled multiprocessor which have more computational power and reliability and parallelism. Extending our approach to multiprocessor environment is more promising than simple EDF approach because our approach shows better performance than EDF-HP when data contention is high.

REFERENCES

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

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

[3] 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.

[4] T.P. Baker. A stack-based resource allocation policy for real-time process. In *Proceedings of Real-Time Systems Symposium*, pages 191–200, Dec 1990.

[5] 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.

[6] S. Chakravarthy, B. Blaustein, A. Buchmann, M. Carey, U. Dayal, D. Goldhirsch, M. Hsu, R. Jauhari, R. Ladin, M. Livny, D. McCarthy, R. McKee, and A. Rosenthal. HiPAC: A research project in active, time-constrained database management. Technical report XAIT-89-02, XEROX, July 1989.

[7] E.G. Coffman. *Computer and Job-Shop Scheduling theory.* Wiley, New York, 1976.

[8] 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.

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

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

[11] 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.

[12] Jiandong Huang, 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.

[13] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of ACM*, 20:46–61, 1973.

[14] C. Douglass Locke. Best-effort decision making for real-time scheduling. Technical Report CMU-CS-86-134, Carnegie-Mellon University, 1986.

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

[16] 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.

[17] 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.

[18] Sang H. Son, Seog Park, and Yi Lin. An integrated real-time locking protocols. In *Proceedings of the 8th Conference on Data Engineering*, pages 527–534, Feb 1992.

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

[20] John A. Stankovic and Wei Zhao. Real-time computing systems: The next generation. *Tutorial of Hard Real-time systems*, pages 14–37, 1988.

[21] 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.

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

[23] 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.

BIOGRAPHICAL SKETCH

Dong-kweon Hong was born on June 11, 1960 in Taegu, South Korea. He received his Bachelor of Engineering degree in computer sciences from the Kyung-Pook National University, South Korea. After finishing his undergraduate degree in 1985, he worked as an Research Engineer at Electronics and Telecommunications Research Institute at Taejeon, South Korea. In the Fall of '90, he started his graduate studies with a major in computer and information sciences at the University of Florida. He will receive his Master of Science degree in computer and information sciences from the University of Florida, Gainesville in August, 1992.