SCHEDULING REAL-TIME TRANSACTIONS

Robert Abbott
Hector Garcia-Molina

CS-TR-129-87

December 1987

Scheduling Real-time Transactions

Robert Abbott
Hector Garcia-Molina

Department of Computer Science
Princeton University
Princeton, NJ 08544

December 1987

ABSTRACT

Scheduling transactions with real-time requirements presents many new problems. In this paper we discuss solutions for two of these problems: what is a reasonable method for modeling real-time constraints for database transactions? Traditional hard real-time constraints (e.g., deadlines) may be too limited. May transactions have soft deadlines and a more flexible model is needed to capture these soft time constraints. The second problem we address is scheduling. Time constraints add a new dimension to concurrency control. Not only must a schedule be serializable but it also should meet the time constraints of all the transactions in the schedule.

1. Introduction

Transactions in a database system can have real time constraints. Consider program trading, or the use of computer programs to initiate trades in a financial market with little or no human intervention [Voel87a]. A financial market (e.g., a stock market) is a complex process whose state is partially captured by variables such as current stock prices, changes in stock prices, volume of trading, trends, and composite indexes. These variables and others can be stored and organized in a database to model a financial market. (A sophisticated model could also include hard economic data such as current interest rates and rate of growth of the money supply, or even softer political information such as the state of trade bills or spending proposals in Congress.)

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

A second type of process is an analysis/output process. In general terms this process reads and possibly analyzes database information in order to respond to a user query or to initiate a trade in the stock market. An example of this is a query to discover the current bid and ask prices of a particular stock. This query has a real-time response requirement of 3-5 seconds. Another example is a program that searches the database for arbitrage opportunities. Arbitrage trading involves finding discrepancies in prices for objects, often on different markets. For example, an ounce of silver might sell for $10 in London and fetch $10.50 in Chicago. Price discrepancies are normally very short-lived and to exploit them one must trade large volumes on a moments notice.

Thus the detection of these arbitrage opportunities is certainly a real-time task.

Another kind of real-time database system involves threat analysis. This system consists of a radar to track objects and a computer to perform some image processing and control. A radar signature is collected and compared against a database of signatures of known objects. The data collection and signature lookup must be done in real-time.

Banking and airline reservations are two traditional examples of interactive database systems with real-time performance requirements. Unlike traditional hard real-time systems, the design of these systems typically does not emphasize deadline modeling. The performance goal is expressed in terms of acceptable response times rather than meeting specific deadlines for user transactions. However, if the proper mechanisms were provided, users could take advantage of them by specifying the time requirements of their jobs. For example, monthly statements would have one kind of deadline, teller transactions would have another.

There are many new problems to solve for real-time transaction processing: what is a correct model for a real-time database, what languages can we use to specify real-time constraints, what methods are needed for describing and evaluating triggers (a trigger is an event or a condition in the database that causes some action to occur) and so on. In this paper we discuss two of these problems. First: what is a reasonable method to model real-time constraints for database transactions? Traditional hard real-time constraints (e.g., deadlines) may be too limited. Many transactions have soft deadlines and a more flexible model is needed to capture these soft time constraints.

The second problem we address is scheduling. Not only must a schedule meet time constraints (like traditional real-time scheduling) but also the schedule must be serializable i.e., it preserves database consistency. Time constraints add a new dimension to concurrency control. For example, what happens when a transaction requests a lock that is already held by another transaction? Both of the conflicting transactions have time constraints yet only one can hold the lock. Furthermore, to preserve serializability, we cannot preempt the lock holding transaction without rolling it back. When restarted it must execute from the beginning.

The following sections discuss these two problems in more depth and present some solutions.

2. Modeling Time Constraints

Many discussions of real-time scheduling use two numbers: a release time r and a deadline d to model a task's real-time requirements. Although this may be suitable for applications where all tasks have absolute hard deadlines, we believe that a more flexible model is needed to capture the time constraints of tasks with softer deadlines. From [Jens86a] we have adopted value functions as a way to express the time requirements of real-time transactions. The key idea is that the completion of a task or set of tasks has a value to the system which can be expressed as a function of time. These time varying functions are usually derived from the physical process with which the system interacts. Evaluating the function at time t yields the value to the system of completing the task at time t. We interpret a positive value to be good for the system. A zero value is neither good nor bad and a negative value is bad. A greater positive value is better than a smaller and similarly for negative values.

A task with a hard deadline is well modeled by a step function. This is illustrated in figure 1.

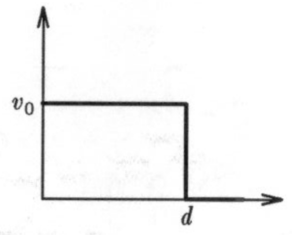


Figure 1.

Completing the task before or at time d yields some positive value v_0. The value for completing the task after time d is zero. Clearly d is a critical time for this task; it represents a hard deadline. Tasks which have identical hard deadlines can be prioritized by assigning a greater value function to one of the tasks.

But a value function is more than just a deadline. The value function models a task's real-time requirements over some window of time whereas a deadline represents only one instant in time. This is particularly important for modeling the soft deadline of a task which can still be completed profitably after its deadline although it is less valuable when it is tardy. The value function also includes the idea of a release time. This is accomplished by defining $v(t) \leq 0$ for all $t < r$. The reason why we use ≤ 0 instead of just $=0$ is to allow the case where it may bad, not just neutral to execute a task before its release time. Consider a database that maintains a company's inventory and a transaction that produces a summary report on the day's inventory activity. The report is used to plan the next day's production schedule and thus the transaction which produces it has a deadline. The transaction should not run too early i.e., before inventory activity has ceased, because it might produce an incorrect report which could adversely affect the next day's production planning.

Figure 2 shows how we can use value functions to model tasks with soft deadlines.

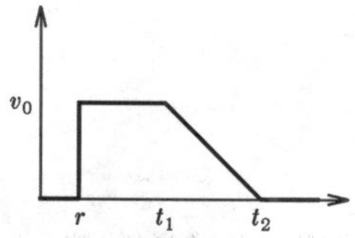


Figure 2.

Completion of the task before time t_1 yields a value v_0. The value of completing the task after time t_1 decreases gradually until time t_2 where it becomes zero. $v(t)=0$ for all $t > t_2$. Also $v(t)=0$ for all $t < r$, meaning that there is no value in doing the task before it is ready to be done. The value function of figure 2 really has two critical times: t_1 and t_2. However we would like to speak of a unique deadline for every task. Thus we define the deadline d for task T with value function $v(t)$ to be the greatest t such that $v(t)$ is maximum. In our example t_1 is the deadline for the task.

If, as we have argued, it can make sense for $v(t)$ to be negative before the release time r then it is also possible for $v(t)$ to be negative after the deadline d. This is true for a task which must be executed regardless of whether it meets its deadline and which effects a penalty when it is completed after its deadline. Consider a value function to represent a strategy for filing your income tax form. (A task which must be done.) If you expect to owe taxes then you would like to file at the latest possible date in order to earn more interest on your money. However you do not want to file after April 15 because then you will be assessed penalty charges. A value function for this strategy is illustrated by figure 3 where $r=$ January 1 and $d=$ April 15. $v(t)=0$ for $t<r$ because it is not possible to file before January 1. $v(t)$ for $r \leq t \leq d$ is strictly increasing, reflecting the fact that we earn more interest on our money by delaying payment. $v(t)$ is maximum at $t=d$, the deadline for filing, and is negative for $t>d$ because this will cause a penalty.

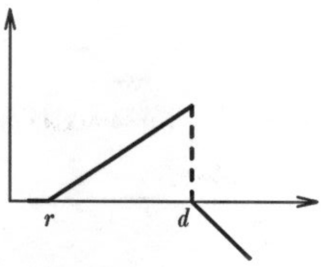


Figure 3.

The value functions of figure 1-3 are all piecewise linear. There is no requirement that this be so. In general a value function can have any shape as long as it is a correct and meaningful model of the real-time requirements of the corresponding real-time task.

There is an important difference between value functions which do not become negative after the deadline and those that do. The distinction arises when we make the requirement that a task must be executed regardless of whether or not it has missed its deadline. Let T be an unfinished transaction of the first type i.e., its value function does not become negative. Suppose that T has missed its deadline and its value function has declined to zero. Suppose further that T is not of the "must execute" type. Since there is no longer any value to completing T we abort it and return to the system any resources that it holds. This is correct because T is a "worthless" transaction and there is no reason to allow it to consume more resources. If however, T must be executed then it can be delayed without losing any value but we must schedule it for execution sometime. If our scheduler only executes transactions which have some value (as determined by the value function) then T can suffer from indefinite postponement.

A somewhat different situation is presented by "must execute" transactions with value functions that assume negative values after the deadline. Normally we would want our scheduler to avoid executing a task that will yield a negative value (i.e., a penalty). Instead, we want to execute T immediately, before its value function declines further and the penalty increases.

One possible solution to this problem is to adjust the priority of a "must execute" task after it has missed its deadline. This adjustment would ensure that T gets the proper service from the scheduler. Another approach is to abort T and reinsert it into the arriving job stream. The new task could be identical to the old but with a different value function or it could be a special task that is started to compensate for the fact that T has missed its deadline. The effects of value functions with negative values and must execute tasks on scheduling decisions is a subject of continuing research.

A goal of a real-time database system is to produce an execution schedule that is timely. In the context of our deadline model we can interpret timely in different ways. Timely can mean that each transaction is completed before its deadline. Or timely can mean that each transaction has a positive value at completion as determined by its value function. For a task with a step function this is equivalent to meeting a deadline. But for tasks with soft deadlines it means that a task can miss its primary deadline, but not by too much, and still be timely.

In addition to meeting time constraints, a schedule must preserve database consistency. The most common way to achieve this is to produce serializable schedules [Date86a]. Next we discuss scheduling algorithms that yield serializable schedules while trying to execute transactions in a timely fashion.

3. Some Scheduling Algorithms

In this section we present a few algorithms to schedule real-time transactions. We assume that transactions are scheduled dynamically on a single processor. A transaction is characterized by its timing constraints and its data and computation requirements. The timing constraints are modeled by a value function $v(t)$ from which we can derive a release time r and a deadline d. A computation requirement is represented by a runtime estimate C which approximates the amount of computation remaining to complete the transaction. These two characteristics, value function and runtime estimate, are known to the scheduler when a task enters the system. The third characteristic, data requirements, is not known beforehand but is discovered dynamically as the transaction executes. Our decision to assume knowledge of computation requirements but no knowledge of data requirements is justified by the existence of real-time database systems e.g., banking, airline reservations, where the execution time of some transactions is approximately the same regardless of the data that are accessed.

A scheduling algorithm has two components: a policy for assigning priorities to tasks and a concurrency control mechanism. The concurrency control mechanism can be thought of as a policy for resolving conflicts between two (or more) transactions that want to lock the same data object. Some concurrency control mechanisms permit deadlocks to occur. For these a deadlock detection and resolution mechanism is needed.

A priority assignment policy or concurrency control mechanism may use only some of the available information about a transaction. In particular we distinguish between policies which do not make use of C, the runtime estimate, and those that do. The former we call ignorant, the latter cognizant. A goal of our research is to understand how the accuracy of the runtime estimate affects the algorithms that use it.

3.1. Assigning Priorities

There are many ways to assign priorities to real-time tasks. Here we list just a few of those that researchers have suggested or that are known to be used in real-time systems. We can think of a priority assignment policy as a function that can take two different kinds of arguments: a single task or a set of tasks. When applied to a single task the result of the function is the priority of the task as determined by the corresponding priority assignment policy. When applied to a set of tasks, the result of the function is an ordered list of the tasks (those that are in the set) with the highest priority task ranked first. We assume that we only consider tasks which are ready to execute, i.e., they are not suspended waiting for I/O or blocked waiting for a lock on a data item.

1. First come first serve (FCFS), ignorant.

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

The primary strength of FCFS is simplicity. The primary weakness of FCFS is that it does not make use of deadline information. FCFS will discriminate against a newly arrived task with an urgent deadline in favor of an older task which may not have such an urgent deadline. This is not desirable for real-time systems. One case where FCFS could perform well is when most transactions have the same value functions and deadlines are always a fixed amount of time from the arrival time. In this case ordering transactions by arrival time is essentially equivalent to ordering by deadline. This is a strategy adopted by transaction processing systems like banking and airline reservations.

2. Earliest deadline (ED), ignorant.

The transaction with the earliest deadline has the highest priority. A major weakness of this policy is that it can assign the highest priority to a task that has already missed or is about to miss its deadline. The next policy tries to correct this flaw.

3. Earliest feasible deadline (EFD), cognizant.

The task with the earliest feasible deadline is assigned the highest priority. A deadline is feasible if we think we can meet it. More concretely, a deadline d is feasible at the current time t if $t+C \leq d$. If all tasks have infeasible deadlines then we assign priority according to earliest deadline. Thus tasks with infeasible deadlines are not aborted immediately because they can receive service when the system is otherwise idle.

4. Least slack (LS), cognizant.

For a transaction T we define a slack time $s=d-(t+C)$. (Recall that C estimates the remaining runtime.) The slack time is an estimate of how long we can delay the execution of T and still meet its deadline. If $s \geq 0$ then we expect that if T is executed without interruption then it will finish at or before its deadline. A negative slack time is an estimate that it is impossible to make the deadline.

Least slack is similar to earliest deadline in that it can assign high priorities to jobs about to miss their deadlines. Again it is possible to modify this policy so that we only consider tasks with feasible deadlines. This is done by rejecting all tasks with negative slack values.

5. Greatest value density (VD), cognizant [Jens86a].

We define a value density function $VD = \frac{v(t+C)}{C}$. In other words the value density VD is the expected value of T at completion divided by the amount of computation needed to realize that value. The reason for dividing by C is to give higher priority to jobs that are shorter, if the expected values are equal. These jobs return more value per time unit consumed. Tasks are ordered by value density such that the task having the greatest value density receives the highest priority.

3.2. Concurrency Control

If transactions are executed concurrently then we need a concurrency control mechanism to order the updates to the database so that the final schedule is a serializable one. The concurrency control mechanism we use is a form of two-phase locking in which locks are acquired gradually and held until the transaction commits or aborts [Date86a]. When lock conflicts occur we need a method to decide which transaction will get the lock and which will wait. We have chosen two-phase locking over other types of concurrency control such as timestamp, or optimistic protocols for two reasons. First, locking is the most widely implemented concurrency control protocol. Second, locking detects and resolves conflicts early in the execution of a transaction. In an optimistic protocol each transaction is executed to completion and then it is verified whether the transaction conflicted in an unserializable way with another transaction. If so it is rolled back and restarted. If not then it is committed. If conflicts occur frequently it can be terribly wasteful to execute transactions to completion only to roll them back and restart them. Locking checks for conflicts every time a lock request is made. Conflicts can be resolved immediately before additional service is granted to transactions which are going to be restarted anyway.

We now discuss some possible solutions. Once again we distinguish between policies which make use of the runtime estimate C and those that do not.

1. No conflicts (NC), ignorant.

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

If transactions are executed concurrently then we can expect conflicts to occur. In the following discussions let T_H denote a transaction which holds a lock on data object X. Let T_R be a transaction which is requesting a lock on X. T_R has a higher priority than T_H. This is why T_R is the currently executing transaction, one that is actively requesting resources. We now present some methods to resolve conflicting transactions.

2. Unconditional abort (UA), ignorant.

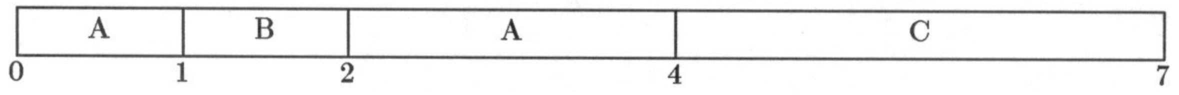
This policy resolves conflicts by always aborting T_H the lock holder. The reason for this strategy is to free up the resources for the higher priority transaction T_R. The lower priority transaction T_H is scheduled for restart.

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

Transaction	r	C	d	updates
A	0	2	3	X
B	1	1	3	X
C	1	3	6	Y

Example 1.

Note that transactions A and B both update item X. Therefore these transactions must be serialized. If we use earliest deadline to assign priority and unconditional abort to resolve conflicts then the following schedule is produced.



In this schedule, A runs in the first time unit during which it acquires a lock on item X. B gains the processor at time 1 (it has an earlier deadline) and immediately requests a lock on item X. Thus a conflict is created which is resolved by aborting A thereby freeing the lock on X. B regains the processor and completes before its deadline. A must be restarted after B completes at time 2. A now misses its deadline and so does C.

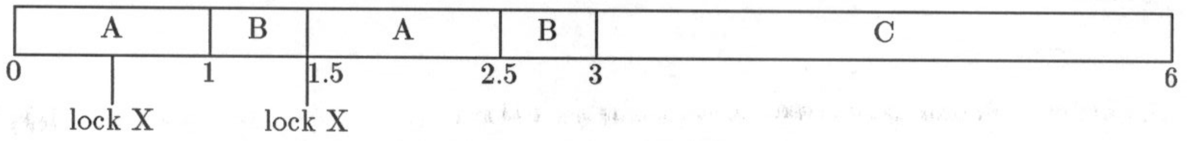
Sometimes unconditional aborts may be too conservative i.e., it is not always necessary to abort T_H. This is true if T_R can afford to wait some time to get its lock.

3. Conditional abort (CA), cognizant.

We can be a little cleverer by using a conditional abort policy to resolve conflicts. The idea here is to estimate if T_H can be finished within the amount of time that T_R can afford to wait. Let s be the slack of T_R and let C be the runtime estimate of T_H. If $s \geq C$ then we estimate that T_H can finish within the slack of T_R. If so then we let T_H proceed to completion, release its locks and then let T_R execute. This saves us from restarting T_H. If T_H cannot be finished in the slack time of T_R then we abort T_H and run T_R (as in the previous algorithm). Note that conditional abort allows transactions to wait for locks, thus deadlock is a possibility. Deadlock detection can be done using one of the standard algorithms [Islo80a]. Victim selection, however, should be done with consideration of the time constraints of the tasks involved in the deadlock.

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

The following schedule illustrates what happens when conditional abort is used with the earliest deadline priority assignment on the set of tasks from example 1.



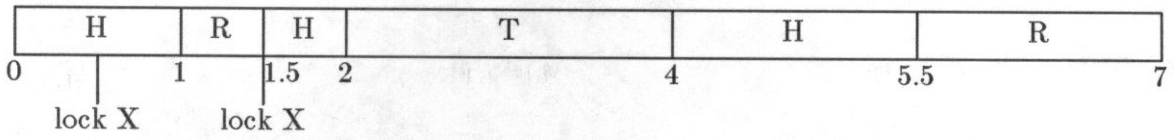
A conflict occurs when B requests a lock on X at time 1.5. The algorithm calculates the slack time for B as $s = 3 - 1.5 - .5 = 1$. This equals exactly the remaining runtime for A. Therefore B waits for A to finish and release its locks. A finishes at time 2.5 and B, with .5 time units left to compute, regains the processor and completes at time 3. All transactions meet their deadlines.

In this example the runtime estimate was an excellent approximation of the actual computation time of A. If the actual computation time for A was a little longer than the runtime estimate then B would miss its deadline. Thus the ability of this algorithm to successfully exploit slack time information in order to avoid aborting and restarting transactions is very dependent on the accuracy of the runtime estimates. Another goal of our research is to discover the nature and degree of this dependency.

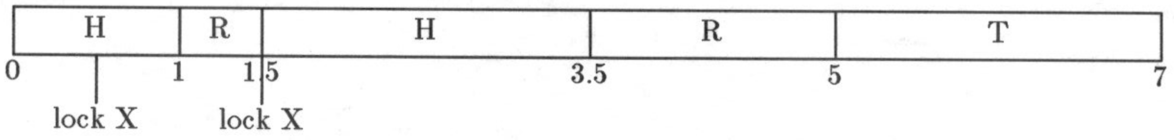
The second problem with conditional abort is illustrated by the following set of tasks and the schedule produced by using earliest deadline and conditional abort.

Transaction	r	C	d	updates
H	0	3	12	X
R	1	2	6	X
T	2	2	7	Y

Example 2.



At time 1.5 the scheduler decides to run *H* because it can be completed within the slack time of *R*. *H* only runs for a short while before it is preempted by an arriving transaction *T* with an earlier deadline and therefore a higher priority than *H*. (*R* is not considered in this priority assignment because it is not a ready task.) Scheduling *T* and *H* before *R* causes *R* to miss its deadline. The problem lies in the assumption that *H* would not be preempted while it was executing during the slack time of *R*. One way to correct this problem is to temporarily adjust the priority of *H* to be as high as that of *R*. The observation is that *H* and *R* are executing as a pair both of which must be finished by t=6, or the deadline of *R*. Now an arriving transaction with a deadline greater than *R* will not preempt *H*. This revised algorithm is illustrated by the following schedule.



In the more general case we need to adjust the priority of all the tasks in the chain D which were going to run before T_0. Furthermore the adjustment must be done according to the priority assignment policy that is being used. In this way we consider the priority of the chain tasks as a group when comparing it to the priority of a task which is attempting to preempt one of the tasks in the group.

4. Conclusions

Finding an optimal schedule for real-time database transactions is very hard. This is particularly true for our model because we do not know a transaction's data requirements in advance. Thus the presented algorithms are actually heuristics and the best way to evaluate them is via experimentation. Currently, we are conducting simulation experiments to study the performance and behavior of a variety of scheduling algorithms. We are particularly interested in learning how cognizant algorithms depend on the accuracy of the runtime estimates. Finally, we foresee that the results of our simulation studies will enable us to develop better heuristics for scheduling real-time transactions.

Acknowledgements

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

References

Date86a.
Date, C. J., *An Introduction to Database Systems,* 1, Addison Wesley, 1986.

Islo80a.
Isloor, Sreekaanth S. and T. Anthony Marsland, "The Deadlock Problem: An Overview," *IEEE Computer*, pp. 58-78, IEEE, September, 1980.

Jens86a.
Jensen, E. Douglas, C. Douglass Locke, and Hideyuki Tokuda, "A time-driven scheduler for real-time operating systems," *Proceedings IEEE Real-time Systems Symposium*, pp. 112-122, IEEE, 1986.

Voel87a.
Voelcker, John, "How Computers Helped Stampede the Stock Market," *IEEE Spectrum*, vol. 24, pp. 30-33, IEEE, December 1987.