1

TOWARDS UNIFYING DATABASE SCHEDULING AND CONCURRENCY CONTROL: A FRONTIER APPROACH

Gregory Jones*,
Lisa Cingiser DiPippo, Victor Fay Wolfe****

** Naval Undersea Warfare Center, Newport, RI*

*** Department of Computer Science, University of Rhode Island, Kingston, RI*

1 INTRODUCTION

Real-time databases have numerous applications, including commodities trading, military command and control, patient monitoring, air traffic control, and flexible manufacturing. All of these applications require that data be processed in such a way that the data and output of the database remains logically consistent . They also require that deadlines imposed on database operations are met (temporal consistency) . Thus, the ultimate goal of scheduling and concurrency control for real-time databases is to maintain both temporal and logical consistency of the transactions and of the data. In situations, such as overload, where both temporal consistency and logical consistency cannot be met, a transaction scheduler may have to sacrifice one requirement for the other and make a trade-off between the two. For example, suppose that a transaction t_{read} is reading some sensor data in a real-time database, and another transaction, t_{update}, needs to update that data in order to maintain the temporal consistency of the data and/or the transaction. If t_{update} were allowed to execute, the logical consistency of t_{read} would be violated, but if t_{update} were blocked, the temporal consistency of the sensor data or of t_{update} could be violated. Thus, one requirement must be traded-off for the other.

Most previous work in real-time database concurrency control indexconcurrency control and scheduling has focused on combining variations of traditional concurrency control techniques with real-time scheduling techniques [1]. These solutions typically attempt to meet temporal consistency requirements while maintaining serializability of transactions. Work described in [2], presents a semantic concurrency control technique for real-time object-oriented databases that allows the database designer to express the trade-off of logical consistency for temporal consistency for each object [3]. These results are limited to concurrency control that assumes simple real-time transaction scheduling. In all of this previous work, concurrency control and scheduling are handled independently. We believe that a unified approach to scheduling and concurrency control will better enforce the often conflicting requirements of temporal and logical consistency.

Commonly, when two criteria, such as temporal consistency and logical consistency, are to be considered for producing optimal results, a *secondary criteria* approach is used [4]. Such an approach attempts to optimize one criteria, say logical consistency; then within the set of optimal schedules for logical consistency, the schedule which maximizes temporal consistency is chosen. Most current real-time database concurrency control techniques do just that. A more flexible approach is *bi-criteria* scheduling [5], where an attempt is made to maximize both measures without preference toward either one. This approach leads to a set of *efficient* schedules , i.e., schedules such that no improvement can be made to one performance measure, without a reduction in the other [5]. Several bi-criteria scheduling techniques for manufacturing systems appear in the literature [6, 7]. However, these methods only consider temporal constraints, for example maximum tardiness and the number of deadlines missed, and not logical consistency.

In this paper, we develop a scheduling model for unified database scheduling and concurrency control that supports a bi-criteria scheduling approach. Techniques based on this model handle logical consistency, temporal consistency, and the trade-off between them. The database system designer expresses the trade-off of logical and temporal consistency based on semantics of the system, mapping both logical consistency and temporal consistency to a common metric called *value* . Temporal consistency is mapped to a *value function* that expresses the value of completing a transaction at a certain time. We extend the model of transaction scheduling based on value functions, which was first developed by Locke [8] and later enhanced by Clark [9] and Jensen [10]. In our model, logical consistency is mapped to a *value penalty table* that expresses the penalty value associated with the violation of desired orderings of database operations, such as serialization orderings. Our model also includes a *frontier* plot of all possible schedules of tasks with temporal value on one axis and value penalty on the other. The scheduler chooses the point on the frontier that best meets the system requirements for temporal and logical consistency.

Given our general model for unifying concurrency control and scheduling in a real-time database, our approach towards a solution is incremental. We begin with a very restricted version of the model, which lends itself to an elegant algorithmic solution. From there, we remove some restrictions and extend the solution accordingly. While the ultimate goal is value-based unified database concurrency control and scheduling, the solutions to these smaller problems have two benefits: (1) they may lead us toward our final solution, and (2) they provide solutions for a range of applications for which the restricted models are sufficient.

Section 2 of this paper outlines our general scheduling model. In Section 3 we present two restricted versions of the model and discuss algorithmic solutions. We also motivate the restrictions by describing applications for which each of the restricted models is sufficient. Section 4 concludes with a summary and a discussion of the applicability and scalability of our work.

2 GENERAL SCHEDULING MODEL

Our scheduling model consists of *transactions* that are decomposed into *tasks*. Associated with each task is a *value function* to capture temporal consistency (temporal value). In addition, there is a system-wide *value penalty table* to capture logical consistency (logical value). The model also incorporates a *frontier* which represents the synthesis of temporal value and logical value in a two-dimensional graphical form. Finally, the model includes a *scheduler* that chooses a schedule on the frontier that maximizes value as some combination of temporal value and logical value in the system.

Transactions and Tasks. We define a *transaction* as an executable database entity that manipulates the data of the database through a partially ordered set of tasks. A *task* is an atomic, non-preemptable database action or process. That is, a transaction is a collection of related tasks, as defined by the database user, to perform a complete database access, such as a query or an update.

Temporal Consistency. Temporal consistency requirements of tasks are expressed in our model through value functions that define the value of completing a task at any instant of time [8]. Figure 1 depicts several examples of value functions. Value function (a) depicted in the figure implies a *firm* deadline . That is, there is no value to the system for completion of the task after its deadline. Value function (b) implies a *soft* deadline . Here, the task has a deadline, but completion after the deadline is still acceptable, although with less value to the system. Value function (c) has no associated deadline,

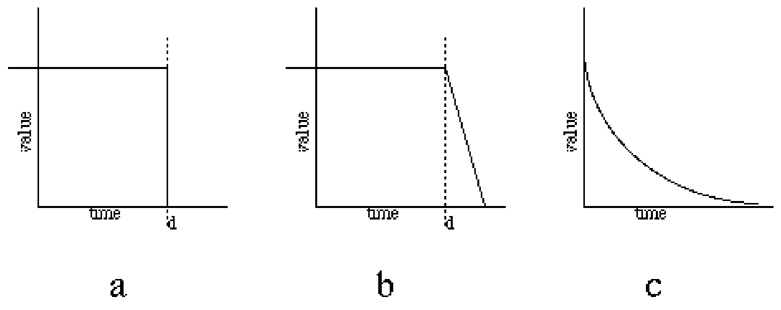

Figure 1 Temporal Value Functions

instead value to the system decreases in a non-linear fashion with later completion times. In general, value functions may have any arbitrary shape. However, issues of scheduler complexity and the requirement that the value function map to some practical requirement may restrict functions to several broad classes.

All tasks need not have an associated value function. If a value function is not explicitly defined for a task, then a constant value of zero is assumed as the default value function for that task. We assume the value function to be non-negative, bounded, and defined for all time within the *schedulability interval* . The schedulability interval is the interval from the earliest time the task could be scheduled to complete, to the latest time the task could be scheduled to complete. Given the value functions for a set of tasks, we define the *temporal value* of a schedule of those tasks to be the sum of all of the values accrued by each of the tasks based on the time at which it completed in the schedule.

Many applications lend themselves to value metrics. For instance, financial applications involving *time-valued* transactions [11] are naturally modeled with value functions that relate their profitability to the time at which they are executed.

We have chosen to use value functions for our model since they can express more information about timing requirements than a simple deadline. This additional information will be useful in relaxing timing constraints. For example, a soft deadline can be missed, but a firm deadline should not be. Value functions also can express the desirability of completing a task early, something that a deadline cannot. The fact that deadlines can be modeled indicates that the use of value functions need not be complicated. However, if more information than a deadline is known about the temporal requirements of the task, a value function can capture that information.

Logical Consistency. Logical consistency in our model is expressed through a system-wide partial ordering of tasks along with a value penalty assessed for violation of each ordering. The value penalty quantifies the loss of logical precision due to out-of-order execution of database operations. It is expressed by the designer in a *value penalty table* , which associates a value deduction with each ordering relation among tasks.

We express partial ordering using the "$\prec$" operator to order task pairs, where $A \prec B$ conveys that task A occurs before task B. Note that because partial ordering is transitive, the user is responsible for specifying value penalties for constraints that result transitively. This can be done by explicitly specifying the penalties, or by defining a rule for assigning transitive penalties.

The "$\prec$" notation that we have chosen to express logical consistency is simple, well understood and useful for a wide range of orderings. However, it is not sufficient to express such conventional concurrency control techniques for enforcing logical consistency such as two-phase locking. For such techniques, more complex notations are available [12] which extend the partial ordering notation to express any arbitrary ordering. Since any pessimistic concurrency control simply enforces some allowable set of task orderings, an extended partial ordering notation can be used to express any [pessimistic] concurrency control scheme, including orderings required by locks. However, in this paper, the simpler notation for partial ordering is sufficient to illustrate our concept.

To illustrate our expression of logical consistency, consider a serialization ordering of tasks in a database transaction. If a read task, r, must occur before a write task, w in the serialization order, our model expresses this requirement as the precedence constraint $r \prec w$. Our model also associates a value penalty with violation of these constraints. One possible technique for establishing penalties is to use the difference between the data values written by w and the data value read by r as the penalty for violating $r \prec w$. If the data value was 20 and w is writing 25, then the penalty would be 5. Such a technique for accumulating penalties as *imprecision* in data is detailed in [2].

Given the value penalties associated with each ordering, we can determine an overall penalty to the system due to the logical inconsistency in a schedule. The sum of all value penalties accrued in a schedule is referred to as the *logical value* of the schedule. Work that has been done in Epsilon Serializability

[13] and semantic locking with bounded imprecision [2] provides techniques for determining how to accumulate the lost value due to violation of serializability when the data involved is in a metric space, such as in a financial application.

In general, temporal values and precedence penalties do not necessarily reflect any real quantities; hence, they do not have associated units. Rather, these values and penalties reflect the designer's quantification of the relative importance of each ordered task pair and timing constraint. In some applications, we may be able to make a direct mapping from the semantics of the data to the value penalty. For instance, in a financial application, a task might have a monetary value that it should yield when it is executed. Penalties for out-of-order execution of tasks might be expressed in monetary units which reflect increased risk or lower rate of return.

Frontier. Considering the problem of temporal constraints alone, Locke's Best Effort Heuristic [8] has been shown to provide good results in maximizing temporal value. However, it is not necessarily true that logical constraints will be met, indeed, such constraints, should they exist, are not accounted for by the Best Effort Heuristic. Alternatively, logical constraints can be met by finding a schedule in which no precedence orderings are violated. If each precedence relationship is associated with a penalty, then an optimal schedule in regard to logical consistency is a schedule with no penalties. Such a schedule exists provided there are no cycles in the precedence graph. However, finding such an optimal logically consistent schedule does not consider temporal consistency. When both temporal and logical constraints are to be considered with equal weight, a bi-criteria scheduling approach may be appropriate.

Given the performance measures of temporal value and logical value, an *efficient schedule* is one for which no other schedule exists that is at least as good in both measures, and strictly better in one. If, for a set of tasks, these two measures of performance (temporal value and logical value) are plotted on orthogonal axes for all possible schedules, then the resulting scatter plot defines a *frontier* at the maximum limits of performance. The frontier is simply the line drawn through all efficient schedules.

Figure 2 is an example of a frontier plot given some set of tasks with value functions and precedence orderings. The complete solution space can be found by an exhaustive enumeration of all schedules. Each schedule is a point on the graph represented by its temporal value on the x-axis and its logical value on the y-axis. Note that any point on the plot may be generated by more than

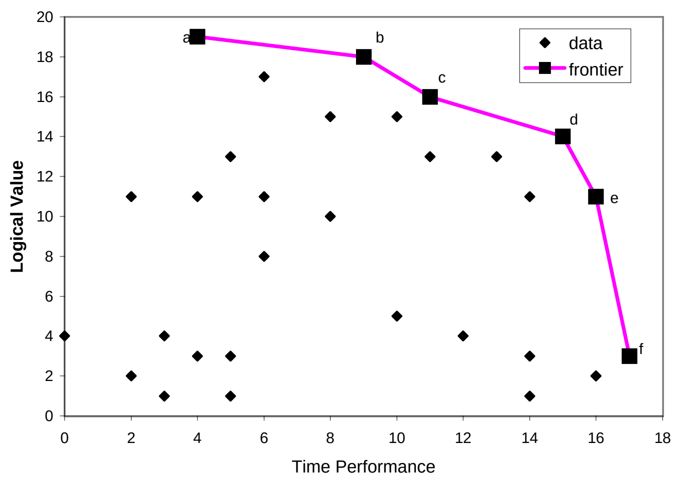

Figure 2 Typical Frontier Plot

one schedule because more than one schedule can result in the same logical value and temporal value. In Figure 2, the set of schedules represented by points $\{a, b, c, d, e, f\}$ define the set of efficient schedules and also the frontier. From the set of efficient schedules, a trade-off must be made to choose the best one. Schedule a in Figure 2 is a schedule for which the logical constraints are given greatest importance, while in schedule f temporal constraints are given greatest importance. The other schedules on the frontier each represent some compromise between the two. The decision of which schedule is best depends on the requirements of the specific application and the equivalence defined between the units of temporal and logical value. For example, in Figure 2, if the units for temporal performance and logical value are considered equivalent, then the best schedules are represented by points d and e, which are the points furthest from the origin. Each of these schedules yields the greatest overall value of 29 (by summing their respective x and y coordinates). However, if the unit for logical value is weighted to be twice as important as the unit for temporal value (double the values along the y axis of Figure 2), then point b is the best schedule with a total value of 45 (36 logical + 9 temporal).

Scheduler. The scheduler in our model determines a schedule for all tasks. For now, we assume that all tasks are known a priori and thus scheduling is static.

The goal of the scheduler is to choose a point on the frontier (i.e. a schedule) that provides the desired temporal and logical value to the system. That is, the scheduler will make the trade-off between logical and temporal consistency.

General Formal Model. The above description lays out the parts of the model in a somewhat informal way in order to relate it to real-time database concurrency control and scheduling. Here we define a formal description of the general model.

The model consists of a set of partially ordered tasks, $1, 2, ..., N$. For every task $i = 1, 2, ..., N$ there is a value function $f_i(\tau)$ where τ represents the completion time of the task. Each task i also has an associated set S_i of successor tasks . For every task $j \in S_i$, there is a Q_{ij} that represents the penalty incurred if j is scheduled before i. Also for every $j \in S_i$, there is an associated indicator variable, I_{ij} that takes the value 0 if i is scheduled before j and 1 if j is scheduled before i. Given this model, the scheduling goal is to maximize total temporal value and minimize total penalty. That is:

$$max(\sum_{i=1}^{N} f_i(\tau_i))$$

and

$$min_{i=1}^{N} \sum_{j=1}^{N} I_{ij} Q_{ij}$$

where τ_i is the actual completion time of task i. Clearly, a schedule that is optimal for one performance measure (minimizes penalties, for example) may not be optimal for the other performance measure (termporal value). However, if a frontier of efficient schedules can be constructed, then a trade-off can be made to choose a best schedule.

3 AN INCREMENTAL APPROACH

Given the complexity of the general problem that we have proposed to solve, our approach is to investigate solutions to smaller, more restrictive problems and take steps towards making these solutions more general. Algorithms exist which provide optimal solutions for these simpler models, and in spite of their simplicity, they do cover a broad range of applications, though not as broad as our general model.

This section describes two restricted versions of our general model along with algorithms for solving the scheduling problems given the restrictions. While each of these models does not apply to as wide a range of applications as our full model, we describe simplified database applications for which each of the models is sufficient.

3.1 Problem 1: Makespan vs. Penalty

We have developed a formulation of the bi-criteria scheduling problem in which individual tasks do not have deadlines. We describe the model for this problem in terms of the restrictions that are placed on the general model described in Section 2. We then define the formal model for this problem, followed by an algorithmic solution.

Restrictions There are three basic restrictions that we have made on our general model for the purposes of this solution:

1. **Individual tasks do not have deadlines.** The temporal value function for each task is a constant (zero). The temporal measure of performance is *makespan*, that is, the total time required to complete all scheduled tasks. Thus, there is no temporal penalty for changing the order of tasks in a schedule.
2. **An overall deadline is specified for the entire schedule.** This deadline is used to define the point on the frontier that should be chosen.
3. **If a task is scheduled after all of its successor tasks , it is not executed.** A successor task of a task A is one that is expressed on the right-hand-side of a precedence constraint, where A is on the left-hand-side. The reason for this restriction is that we assume that there is no value in executing a task if all of its successor tasks have already executed. Eliminating tasks from the schedule has the benefit of decreasing overall makespan, which is the measure of temporal value in this restricted model.

As an example, consider a set of real-time database tasks (reads and updates). Each task has an associated processing time, and some of them are involved in precedence constraints. Table 1 lists each task along with its processing time.

Task	Description	Time
1	$Read_1$	4
2	$Read_2$	2
3	$Update_1$	3
4	$Read_3$	5
5	$Read_4$	4
6	$Update_2$	3
7	$Update_3$	2

Table 1 Example Task Definitions

Precedence	Penalty
$1 \prec 3$	10
$2 \prec 3$	8
$3 \prec 4$	4
$4 \prec 6$	4
$4 \prec 7$	5
$5 \prec 7$	5

Table 2 Example Precedence and Penalties

Table 2 displays the precedence orderings and associated penalties. To visually depict the dependencies defined by the precedence constraints, Figure 3 shows a network representation of these tasks and precedence constraints. Recall that transitivity of the precedence operator is not implied.

The precedence constraints and associated penalties defined for this example are based on the semantics of the application. For instance, $Read_1$ and $Read_2$ are both specified to occur before $Update_1$. This is due to the fact that $Update_1$ involves writing data read by $Read_1$ and $Read_2$. Because for both of these read tasks, $Update_1$ is the only successor task, if the read is scheduled after the update, it will not be executed. The penalties associated with the two precedence constraints are different to reflect the fact that $Read_1$ is more important to execute (i.e. less likely to be removed from the schedule) than $Read_2$. Along with the precedence constraints and penalties, the user specifies an overall deadline by which the schedule must be completed.

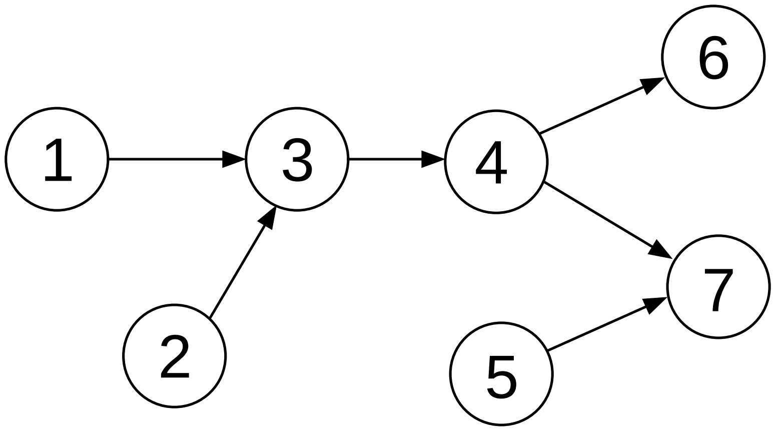

Figure 3 Example Task Network Diagram

Restricted Formal Model 1

We use our general formal model of Section 2 to formally model the first restricted problem. Consider a partially ordered set of tasks, $1, 2, \ldots, N$. For each task $i = 1, 2, \ldots N$, there is a set of successor tasks S_i such that task i precedes task j for all $j \in S_i$. Furthermore, for each $j \in S_i$ there is a penalty $Q_{ij} \geq 0$ if task i is scheduled after task j. Each task has a known processing time, $t_i > 0$. An indicator variable, Z_i takes the value 1 if all successor tasks of task i are scheduled before it and 0 otherwise. Thus, the makespan for the entire schedule of tasks is $\sum_{i=1}^{i=N} t_i(1 - Z_i)$. The maximum makespan is $\sum_{i=1}^{i=N} t_i$, and the minimum makespan is zero. Also, the maximum penalty possible is $\sum_{i=1}^{i=N} \sum_{j \in S_i} Q_{ij}$ and the minimum penalty is zero.

The problem can then be stated as follows: Given a common deadline T by which the entire schedule must be completed, it is necessary to determine the set of precedences that will have to be violated to meet this timing constraint. This can be formulated as

$$\min \sum_{i=1}^{i=N} \sum_{j \in S_i} I_{ij} P_{ij} \tag{1.1}$$

subject to:

$$\sum_{i=1}^{i=N} (t_i(1 - Z_i)) \quad \leq \quad T \tag{1.2}$$

$$I_{ij} \quad \in \quad \{0, 1\} \tag{1.3}$$

Solution. By the assumptions of the model, any tasks that are scheduled to execute after all successor tasks will be processed in zero time, i.e. those tasks will be deleted. If we consider all possible schedules, we can calculate the sum of penalties as a result of tasks executing out of order. We can also calculate the makespan of the schedule, which is affected by out of order tasks that may not be executed.

While exhaustive enumeration provides a solution, we observe that this problem is similar to the well known *knapsack problem* [14]. In the knapsack problem, we seek to fill a container (knapsack) of limited weight capacity (or volume) with a series of objects. Each object has a weight and a value. The goal is to fill the knapsack such that total value is maximized without exceeding the capacity. If we begin with a schedule that requires all tasks to be deleted (i.e. an empty knapsack), then this schedule has the shortest makespan, but also the highest penalty. If we then accept a slightly longer makespan, the question arises as to just what task or tasks of the deleted set should be executed (by scheduling it prior to all successor tasks), such that the makespan is less than or equal to the longer makespan, and the reduction in penalty is maximized. We then again increase the allowable makespan, and again choose tasks to fill the time with the maximum reduction in penalty. Through this process we are solving successive knapsacks, each of a slightly larger size. While at first, this appears to be unacceptably tedious, dynamic programming [14] provides a solution to a knapsack of given size by using the solutions of smaller knapsacks. Thus, by using dynamic programming to solve the problem for the maximum makespan (all tasks included), we will solve the problem for all shorter makespans in the process, and hence define the frontier.

The dynamic programming algorithm provides an optimal solution to the knapsack problem by starting with a knapsack of size 0. For this knapsack size, the solution is simply the null set of tasks. For a knapsack of size 1, choose from the set of deleted tasks with processing time of 1 (or less), the single task associated with the greatest penalty. If no task can be found, then the solution for 1 is the same as for 0.

For a knapsack of size 2, we consider a solution using either of the two previous solutions (knapsacks of size 0 and 1) and use the best one. In either case, add a single task such that a) the task hasn't been used in the previous solution, b) the total makespan is equal or less than the specified deadline T (knapsack size), and c) the task has the greatest penalty if there is a choice.

In general, for knapsack of size K, we try to add a single task which meets conditions a, b and c above, for each previous solution for knapsacks of size K-1, K-2, ...0. Of these K possibilities, choose the best one (that is, the one with the greatest total penalty reduction). If no solution can be found for K, then the solution for K is the same as the solution for K-1.

The graph of Figure 4 shows the solution points for all possible schedules of the tasks in our real-time database example of Tables 1 and 2. Each point represents a makespan and penalty for one or more schedules. Of specific interest are the points connected by a series of lines. These are the points established by dynamic programming that define the frontier. In this restricted model, "good" logical performance involves minimizing penalty, and "good" temporal penalty involves minimizing makespan. Thus, each point on the frontier is one for which there are no other points with a lower penalty and a shorter makespan. It is among these frontier points that the best schedule exists. The point that is ultimately chosen is determined by the user's requirements in the particular application.

In our example, if the specified deadline is 10, then the subset of frontier points that can be chosen are (10,24), (9,26), (7,28), and (5,36). While these points meet the deadline, they suffer in database precision as evidenced by the high penalties. If the database user requires exact precision in the database, then he must be willing to wait for the entire schedule to complete, and miss the specified deadline. Table 3 shows the frontier points of our example, and lists a schedule of tasks which results in the shown performance measures. Recall that each frontier point can be generated by several different schedules For purposes of illustration, we show only one schedule for each point.

While the restrictions described by this version of the model may seem to be rather limiting, there are real applications for which this model is sufficient. For example, consider a contact classification application on a submarine. The goal of such an application is to detect unknown contacts in the vicinity of the submarine and attempt to classify them based on sensor and other data collected about the contact. The actual classification involves reading data from a real-time database, calculating a best guess at the type of contact, and displaying a result to the user. So, there are precedence constraints between each of the database reads and the calculate task. It may be the case that some data to be read is more valuable than other data and so this would be reflected in the associated penalties. Clearly in this example, the sooner the classification is made, the more time the commander has to react in case the contact turns out to be unfriendly. On the other hand, the more data that

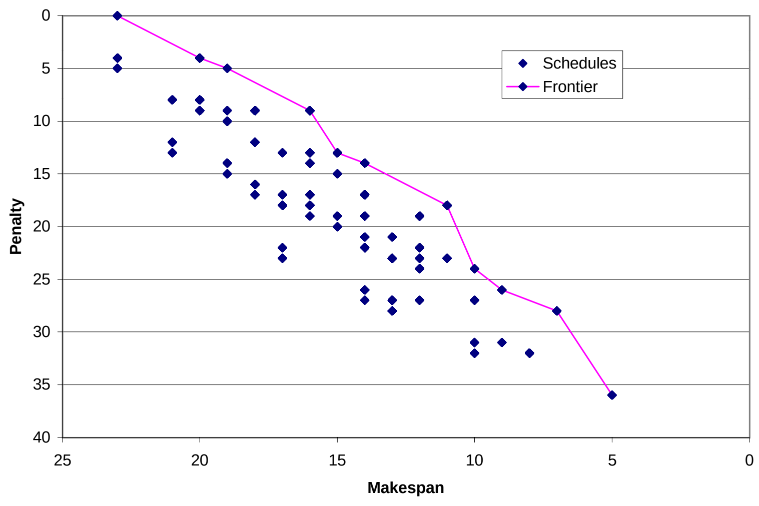

Figure 4 Example Frontier

Schedule	Makespan	Penalty
(1 2 3 4 5 6 7)	23	0
(1 2 4 3 5 6 7)	20	4
(1 2 3 4 6 7 5)	19	5
(1 2 4 3 6 7 5)	16	9
(1 2 5 6 7 4 3)	15	13
(1 2 3 6 7 4 5)	14	14
(1 2 6 7 4 3 5)	11	18
(2 3 1 6 7 4 5)	10	24
(1 6 7 4 3 2 5)	9	26
(2 6 7 4 3 1 5)	7	28
(6 7 4 3 1 2 5)	5	36

Table 3 Frontier Points For Sample Problem

is read, the more accurate will be the estimate of classification. We can also see that there is no reason to perform a read after the calculations have been performed. This example indicates the trade-off that must be made between logical consistency and temporal consistency in the context of this restricted model.

3.2 Problem 2: Independent Transactions with Deadlines

To move closer to the general scheduling problem that we want to solve, consider a model that allows deadlines on transactions, and some precedence constraints within transaction tasks, but does not allow for any precedence constraints between transactions. The restrictions that this model places on the general model are listed below.

Restrictions:

1. **Temporal value functions express deadlines only.** The measure of temporal consistency for this model is maximum tardiness of tasks. This measure was chosen because an earliest deadline first priority assignment scheme is known to be optimal for minimizing maximum tardiness. Accordingly, value functions are restricted to a unit step function: value 1 prior to and at the deadline and value 0 after the deadline.

2. **Only the final task in a transaction has a deadline.** While transactions are made up of several tasks, only the final task of the transaction has a deadline. This equates to a total deadline on the entire transaction, which is quite often the case in many real-time database applications.

3. **Precedence orderings can only be expressed between non-deadline tasks and the deadline task within a transaction.** Furthermore, a non-deadline task may only be expressed as a predecessor to a single deadline task. This restriction enforces independence among transactions. Thus, there are no precedence constraints between tasks from different transactions.

This model brings us closer to the more general model than the previous restricted model. Consider several transactions, each with an independent deadline. Each transaction has one or more tasks, and these tasks can interleave in such a way that maximum tardiness is minimized. When there is not enough time to execute all tasks, low value predecessor tasks are moved to the end of the schedule so that transactions can meet deadlines, but at the cost of executing the predecessor task out of order.

Restricted Formal Model 2. Formally, the model for the second restricted problem consists of a set of partially ordered tasks, 1,2,..,N where each task $i = 1, 2, ..., N$ has an associated processing time t_i, and an associated completion time c_i. There is a subset of these tasks, D, where the number of tasks in D is $M \leq N$. Each task i in D has an associated deadline, d_i. Another set of tasks, P_i represents the set of predecessor tasks to task $i \in D$. For each task $j \in P_i$, there is a penalty Q_{ji} for scheduling task i before task j. Also associated with each task $j \in P_i$ is an indicator variable I_{ji} which takes the value 0 if j is scheduled before i and 1 if i is scheduled before j.

The set of deadline tasks D and the predecessor task sets P_i are disjoint. That is we have:

$$\forall_{i \in D} D \cap P_i = \emptyset$$

Also, no task is a predecessor of more than one deadline task:

$$\forall_{i \neq j} P_i \cap P_j = \emptyset$$

Finally, we assume that the tasks are ordered by their deadlines. So we have $d_i \prec d_j$ if $i \prec j$.

Given this model, the scheduling goal is to minimize both maximum tardiness $(max_{i \in D}(c_i - d_i))$ and penalty incurred $(\sum_{j \in D} \sum_{i \in P_j} I_{ij} Q_{ij})$.

Solution. This problem is a multiple knapsack problem where the tasks of each distinct transaction must fit into a time frame delimited by the deadline of the transaction. A solution to this model is very similar to the solution for the first restricted model, and we are currently formalizing it.

Again, note that while this is not the full general model of real-time database scheduling and concurrency control, applications exist for which this restricted model is sufficient. Recall the contact classification example described in Section 3.1, but now with multiple contacts to be classified. Each of these classifications can be considered an independent transaction with its own deadline that depends on the time at which the contact was detected. Again, the database read tasks that should be performed before the associated classification calculation can be sacrificed in order to meet the transaction deadline.

4 CONCLUSION

In this paper we have presented a general model for unifying real-time database scheduling and concurrency control. We have also presented two restricted versions of the general scheduling model. The first model reduces to the knapsack problem, for which dynamic programming provides schedules that comprise the frontier. A similar approach may be possible for our second problem. While we have not solved the general problem, the solutions that we have developed can apply to a large subset of real-time database applications. Furthermore, these solutions are stepping stones towards a more general solution. We anticipate that as the model becomes more general, algorithmic approaches like dynamic programming may not be practical. We expect that a heuristic approach will be needed once we remove most of the restrictions and approach the general problem of unifying scheduling and concurrency control in real-time databases.

Our scheduling model is general by design. That is, we have purposely made it as general as possible so that it can apply to a wide variety of scheduling problems. For instance, presently most real-time database scheduling and concurrency control constraints are expressed in terms of deadlines and locking. Our model can express these constraints. Recall that Figure 1 depicts the value functions for tasks with soft and firm deadlines. Two-phase read/write locking can be expressed using the partial ordering notation provided by our model and extended by [12]. The strength of our model is that while it can express most currently well-known temporal and logical constraints like deadlines and locking, it can also apply to more general temporal and logical constraints.

The frontier approach that we represent in our model allows a user to make the trade-off between temporal and logical consistency that is appropriate to the problem domain. Traditional approaches to scheduling force the user to choose between logical performance and temporal performance in an all-or-nothing way. Most schedulers for real-time systems use a secondary criteria approach, or a simple variation. The secondary approach is limited in that it seeks to maximize a secondary criteria subject to optimizing the primary criteria. The frontier approach to bi-criteria scheduling considers each criterion equally and therefore the trade-off can be made according to the application requirements.

One interesting characteristic of our general model is that the concepts can scale from the micro level to the macro level. Temporal and logical constraints can be applied to the low level of transaction reads and writes, as we have illustrated here. The same trade-off between logical and temporal consistency can be made at the level of major functions. For example, in a mission planning system for an autonomous robotic device, high level goals can be decomposed into lower level goals, perhaps with intermediate deadlines and precedence constraints between them. Using similar techniques to those developed here for database reads and writes, we can define a frontier of trade-offs between meeting mission deadlines and completing all requested tasks correctly. Similar applications in flexible manufacturing systems, financial systems and military applications abound.

The scheduling technique discussed here still presents further challenges for implementation in a real-time system. We recognize that, given the solution to restricted problems such as those presented here, the solution to the general problem is still a long way off. With sufficient restrictions on the specification of logical and temporal constraints, however, we are confident a practical scheduling technique can be found. In this paper, we have presented a sound theoretical foundation for this future work.

Acknowledgements

This work has been supported in part by The Office of Naval Research under grant #535398.

REFERENCES

[1] Philip S. Yu, Kun-Lung Wu, Kwei-Jay Lin, and Sang H. Son. On real-time databases: Concurrency control and scheduling. *Proceedings of the IEEE*, 82(1):140–157, January 1994.

[2] Lisa Cingiser DiPippo and Victor Fay Wolfe. Object-based semantic real-time concurrency control with bounded imprecision. *IEEE Transactions on Knowledge and Data Engineering*, 9(1), January 1997.

[3] Lisa B. Cingiser DiPippo and Victor Fay Wolfe. Object-based semantic real-time concurrency control. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1993.

[4] C. Chuen-Lung and R. Bulfin. Scheduling a single machine to minimize two criteria: Maximum tardiness and number of tardy jobs. *IEE Transactions*, 26(5):76–84, Sep 1994.

[5] S. French. *Sequencing and Scheduling, An Introduction to the Mathematics of the Job-Shop.* Ellis Horwood Limited, 1982.

[6] A. Hariris and C. Potts. Single machine scheduling with deadlines to minimize the weighted number of tardy jobs. *Management Science*, 40(12), 1994.

[7] G. Vairaktarakis and L. Chung-Yee. The single-machine scheduling problem to minimize total tardiness subject to minimimum number of tardy jobs. *IEE Transactions*, 27:250–256, 1995.

[8] Douglas Locke. Best-effort decision making for real-time scheduling. PhD Dissertation, Department of Computer Science, Carnegie-Mellon University, 1986.

[9] Raymond Clark. Scheduling dependent real-time activities. PhD Dissertation, Department of Computer Science, Carnegie-Mellon University, 1990.

[10] D. Jensen. Real-time manifesto. Published on the Internet: http//www.realtime-os.com/rtmanifesto/, 1996.

[11] S. Westin V. Wolfe, K. Lau. Real-time object-oriented database support for program stock trading. *Journal of Database Management*, 5(2):3–17, 1994.

[12] Greg Jones and Manbir Sodhi. A method for describing operation sequences in flexible manufacturing systems. In *Proceedings of the Third International Conference on Computer Integrated Manufacturing*, July 1995.

[13] Krithi Ramamritham and Calton Pu. A formal characterization of epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering*, 7(6), December 1995.

[14] S. Martello and P. Toth. *Knapsack Problems; Algorithms and Computer Implementations.* John Wiley and Sons, 1990.