Scheduling In Real-Time Transaction Systems*

John A. Stankovic, Krithi Ramamritham, and Don Towsley

Dept. of Computer and Information Science
University of Massachusetts
Amherst, Mass. 01003

Abstract

In many application areas database management systems may have to operate under real-time constraints. We have taken an integrated approach to developing algorithms for cpu scheduling, concurrency control (based both on locking and on optimistic concurrency control), conflict resolution, transaction restart, transaction wakeup, deadlock, buffer management, and disk I/O scheduling. In all cases the algorithms directly address real-time constraints. We have developed new algorithms, implemented them on an experimental testbed called RT-CARAT, and evaluated their performance. We have paid particular note to how the algorithms interact with each other and to actual implementation costs and their impact on performance. The experimental results are numerous and constitute the first such results on an actual real-time database testbed. The main algorithms and conclusions reached are presented in this Chapter.

*This work was supported by ONR under contracts NOOO14-85-K-0389 and N00014-87-K-796, and NSF under grants IRI-8908693 and DCR-8500332.

INTRODUCTION

Real-time transaction systems are becoming increasingly important in a wide range of applications. One example of a real-time transaction system is a computer integrated manufacturing system where the system keeps track of the state of physical machines, manages various processes in the production line, and collects statistical data from manufacturing operations. Transactions executing on the database may have deadlines in order to reflect, in a timely manner, the state of manufacturing operations or to respond to the control messages from operators. For instance, the information describing the current state of an object may need to be updated before a team of robots can work on the object. The update transaction is considered successful only if the data (the information) is changed consistently (in the view of all the robots) and the update operation is done within the specified time period so that all the robots can begin working with a consistent view of the situation. Other applications of real-time database systems can be found in program trading in the stock market, radar tracking systems, command and control systems, and air traffic control systems.

Real-time transaction processing is complex because it requires an integrated set of protocols that must not only satisfy database consistency requirements but also operate under timing constraints. In our work we have developed, implemented, and evaluated integrated suites of algorithms that support real-time transaction processing. In total, the algorithms that we have developed deal with the following issues: cpu scheduling, concurrency control (based on locking and on optimistic concurrency control), conflict resolution, transaction restart, transaction wakeup, deadlock, buffer management, and disk I/O scheduling. In all cases the algorithms directly address real-time constraints. The implementation was performed on a single node testbed called Real-Time Concurrency And Recovery Algorithm Testbed (RT-CARAT). This testbed contains all the major features of a transaction processing system.

As an example of the evaluation of one suite of algorithms, based on two-phase locking, we have implemented and evaluated 4 cpu scheduling algorithms, 5 conflict resolution policies, 3 policies for transaction wakeup, 4 deadlock resolution policies, and 3 transaction restart policies, all tailored to real-time constraints. We compared various combinations of these algorithms to each other and to a baseline system where timing constraints are ignored. In addition, we have studied other suites of algorithms to investigate real-time buffering, using priority inheritance in a real-time database setting, real-time optimistic concurrency control, and real-time disk scheduling. We also studied (1) the relationship between transaction timing constraints and criticality, and their combined effects on system performance, (2) the behavior of a CPU bound system vs. an I/O bound system, and (3) the impact of deadline distributions on the conducted experiments. The major observations from these experiments are presented in this Chapter. For detailed performance data the reader is referred to the referenced material.

The Chapter is organized in the following manner. We first describe our database and transaction model. We then describe the suite of algorithms we have developed to study real-time transaction processing based on two-phase locking. Extensions to the basic work that include real-time buffer management, the impact of applying priority inheritance to real-time transaction systems, and a comparison of two-phase locking with optimistic concurrency control are then presented. The main performance results are presented throughout. All of the evaluations to this point in the Chapter were conducted on the RT-CARAT testbed. In the last part of the Chapter we present two new real-time disk scheduling algorithms and their performance results. The evaluation of the disk scheduling algorithms was performed via simulation since it was impossible for us to modify the physical disk controllers on our testbed. We conclude with a summary of the results and present several open questions.

A REAL-TIME DATABASE MODEL

In our work to date we have investigated a centralized, secondary storage real-time database. As is usually required in traditional database systems, we also require that all the real-time transaction operations maintain data consistency as defined by serializability. Serializability may be relaxed in some real-time database systems, depending on the application environment and data properties [27, 29, 23], but this is not considered here. Serializability is enforced either by using the two-phase locking protocol or via optimistic concurrency control.

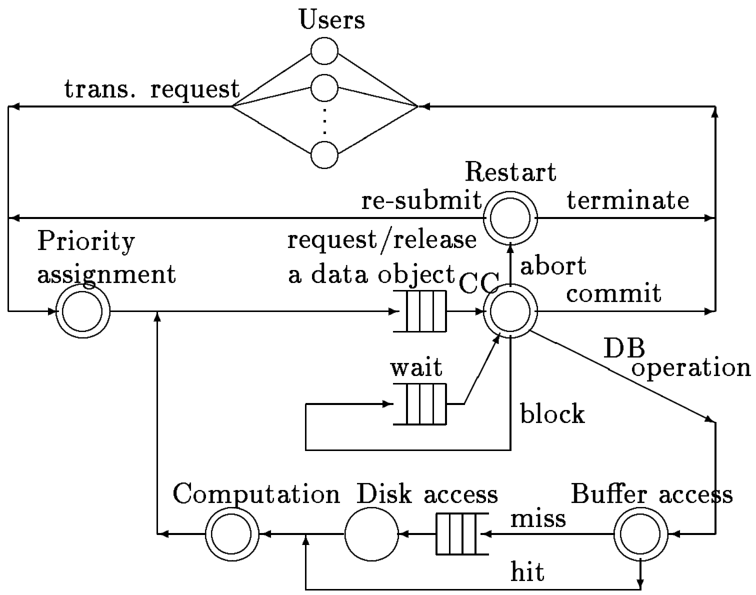

Figure 1: Real-Time Database Model

Figure 1 depicts our system model from the perspective of transaction flow. This model is an extended version of the model used in [4]. The system contains a fixed number of users that submit transaction requests separated by a think time. This model captures many applications in the real world, although certainly not all applications (e.g., an open system model is more appropriate for a process control system). For example, in an airline reservation system, there is a fixed number of computer terminals. The airline clerk at each terminal may check a flight, reserve a seat, or cancel a reservation for customers. After submitting a request to the system, the clerk waits for a result. He may submit another request after getting a response from the previous one.

In the system, any new or re-submitted transaction is assigned a priority that orders it relative to other concurrent transactions. Before a transaction performs an operation on a data object, it must go through the *concurrency control* component (CC), e.g., to obtain a lock on that object. If the request is denied, the transaction will be placed into a wait queue. The waiting transaction will be awakened when the requested lock is released. If the request is granted, the transaction will perform the operation which consists of *global buffer access*, *disk access* (if there is a buffer miss) and *computation*. A transaction may continue this "request-operation cycle" many times until it commits. At its commit stage, the transaction releases all the locks it has been holding. The concurrency control algorithm may abort a transaction for any number of reasons (to be discussed later). In that case, the *restart* component will decide, according to its current policy, whether the aborted transaction should be re-submitted or terminated.

Note that this model only reflects the logical operations involved in transaction processing and it shows neither the interaction of the processing components with physical resources nor the CPU scheduling algorithm. In practice, all of the processing components depicted by a double circle in Figure 1 compete for the CPU.

A real-time transaction is characterized by its length and a value function.[1] The transaction length is dependent on the number of data objects to be accessed and the amount of computation to be performed, which may not always be known. In this study, some of the protocols

[1]Note that there are no standard workloads for real-time transactions, but a value function has been used in other real-time system work [24, 1].

assume that the transaction length is known when the transaction is submitted to the system. This assumption is justified by the fact that in many application environments like banking and inventory management, the transaction length, i.e., the number of records to be accessed and the number of computation steps, is likely be known in advance.

In a real-time database, each transaction imparts a value to the system, which is related to its criticalness and to when it completes execution (relative to its deadline). In general, the selection of a value function depends on the application [24]. In this work, we model the value of a transaction as a function of its criticalness, start time, deadline, and the current system time. Here criticalness represents the importance of transactions, while deadlines constitute the time constraints of real-time transactions. Criticalness and deadline are two characteristics of real-time transactions and they are not necessarily related. A transaction which has a short deadline does not imply that it has high criticalness. Transactions with the same criticalness may have different deadlines and transactions with the same deadline may have different criticalness values. Basically, the higher the criticalness of a transaction, the larger its value to the system. On the other hand, the value of a transaction is time-variant. A transaction which has missed its deadline will not be as valuable to the system as it would be if it had completed before its deadline. We use the following formula to express the value of transaction T:

$$V_T(t) = \begin{cases} c_T, & s_T \leq t < d_T \\ c_T \times (z_T - t)/(z_T - d_T), & d_T \leq t < z_T \\ 0, & \text{otherwise} \end{cases} \qquad (1)$$

where t - current time;
s_T - start time of transaction T;
d_T - deadline of transaction T;
c_T - criticalness of transaction T,
$1 \leq c_T \leq c_{Tmax}$;
c_{Tmax} - the maximum value of criticalness.

In this model, a transaction has a constant value, i.e., its criticalness value, before its deadline. The value starts decaying when the transaction

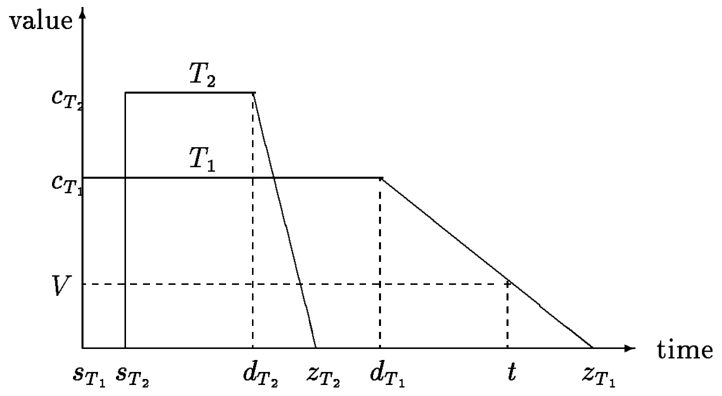

Figure 2: Value functions for transaction T_1 and T_2

passes its deadline and decreases to zero at time z_T. We call z_T the *zero-value point.* As an example, Figure 2 shows the value functions of two transactions T_1 and T_2. Note that when a transaction passes its zero-value point it is not immediately aborted because this may negatively affect the currently executing transaction. Rather, the transaction is aborted the next time the cpu scheduling algorithm attempts to execute it.

The decay rate, i.e., the rate at which the value of a transaction drops after its deadline, is dependent on the characteristics of the real-time transaction. To simplify the performance study, we model the decay rate as a linear function of deadline and criticalness. We have studied two models with z_T expressed by the following two formulas.

$$z_T = d_T + (d_T - s_T)/c_T \tag{2}$$

$$z_T = d_T + (d_T - s_T)/(c_{Tmax} - c_T + 1). \tag{3}$$

For a given c_{Tmax}, when c_T increases, under Eq. (2), z_T decreases, whereas under Eq. (3), z_T increases. With Eq. (2), if a transaction is extremely critical ($c_T \to \infty$), its value drops to zero immediately after its deadline. This is typical of many hard real-time systems. In this work, we use Eq. (1) and Eq. (2) as the base model, and we consider Eq. (3) as an alternative to Eq. (2).

The transactions considered here are solely soft real-time. Given the value function, real-time transactions should be processed in such a way that the total value of completed transactions is maximized. In particular, a transaction should abort if it does not complete before time z_T (see Figure 2), since its execution after z_T does not contribute any value to the system at all. On the other hand, a transaction aborted because of deadlock or data conflict may be restarted if it may still impart some value to the system.

Finally, at times, the estimated execution time of a transaction, r_T, may be known. This information might be helpful in making more informed decisions regarding which transactions are to wait, abort, or restart. This hypothesis is tested in our experiments by using certain algorithms that make use of r_T.

REAL-TIME TRANSACTION PROCESSING

Given the above system model and the characteristics of real-time transactions, one objective of our work is to develop and evaluate policies that provide the necessary support for real-time transactions. In this section, we explicitly address the problems of CPU scheduling, conflict resolution, and deadlock resolution. The algorithms for transaction wakeup and transaction restart are not presented here due to space limitations and due to the fact that these algorithms do not significantly impact performance. See [15] for a full description of all these algorithms and their performance evaluation.

CPU Scheduling

There is a wide variety of algorithms for scheduling the CPU in traditional database systems. Such algorithms usually emphasize fairness and attempt to balance CPU and I/O bound transactions. These scheduling algorithms are not adequate for real-time transactions. In real-time environments, transactions should get access to the CPU based on criticalness and deadline, not fairness. If the complete data access requirements and timing constraints are known in advance, then scheduling can be done through transaction preanalysis [5]. On the other hand, in many cases complete knowledge may not be available. Then a priority based scheduling algorithm may be used, where the priority is set based on deadline, criticalness, length of the transaction, or some combination of these factors.

We consider three simple CPU scheduling algorithms. The first two algorithms are commonly found in real-time systems, and the third is an attempt to combine the first two so as to achieve the benefits of both.

- **Scheduling the most critical transaction first** (MCF)
- **Scheduling by transaction with the earliest deadline first** (EDF)
- **Scheduling by criticalness and deadline** (CDF): In this algorithm, when a transaction arrives, it is assigned a priority based on the formula $(d_T - s_T)/c_T$. The smaller the calculated value, the higher the priority.

Under all of these cpu scheduling algorithms, when a transaction begins its commit phase, its priority is raised to the highest value among all the active transactions. This enables a transaction in its final stage of processing to complete as quickly as possible so that it will not be blocked by other transactions. This policy also reduces the chance for the committing transaction to block other transactions. Under all three algorithms, the transactions are preemptable, i.e., an executing transaction (not in its commit phase) can be preempted by a transaction with higher priority.

Conflict Resolution Protocols (CRP)

Two or more transactions have a data conflict when they require the same data in incompatible lock modes (i.e. *write-write* and *write-read*). The conflict should be resolved according to the characteristics of the conflicting transactions. Here we present five protocols for conflict resolution.

In the following descriptions, T_R denotes the transaction which is requesting a data item D, and T_H is another transaction that is holding a lock on D. The five protocols have the following common algorithmic structure:

```
T_R requests a lock on the data item D
if no conflict with T_H
  then T_R accesses D
  else  call CRPi    (i = 1,2,3,4,5)
end if
```

We start with the simple protocols in terms of complexity and the amount of information required.

Protocol 1 (CRP1): Based on criticalness only.

This simple protocol only takes criticalness into account.

if $c_{T_R} < c_{T_H}$ for all T_H
  thenT_R waits
  else
    if $c_{T_R} > c_{T_H}$ for all T_H
      then T_R aborts all T_H
      else T_R aborts itself
    end if
end if

Note that protocol 1 is a deadlock-free protocol, since waiting transactions are always considered in order of criticalness. In addition, this protocol implements an *always-abort* policy in a system where all the transactions have the same criticalness.

Protocol 2 (CRP2): Based on deadline-first-then-criticalness.

We anticipate that criticalness and deadlines are the most important factors for real-time transactions. Protocol 2 only takes these two factors into account. Here we separate deadline and criticalness by checking the two parameters sequentially. The algorithm for this protocol is:

if $d_{T_R} > d_{T_H}$ for any T_H
  then T_R waits
  else
    if $c_{T_R} \leq c_{T_H}$ for any T_H

then T_R waits
else T_R aborts all T_H
end if
end if

Protocol 3 (CRP3): Based on deadline, criticalness and estimation of remaining execution time.

CRP3 is an extension of CRP2. This protocol takes the remaining execution time of the transaction into account in addition to deadline and criticalness. Here we assume that the computation time and I/O operations of a transaction are known and that they are proportional to each other. Then the remaining execution time of transaction T can be estimated by the following formula:

$$time\_needed_T(t) = (t - s_T) \times (R\_total_T - R\_accessed_T(t))/R\_accessed_T(t)$$

where R_total_T is the total number of records to be accessed by T; $R\_accessed_T(t)$ is the number of records that have been accessed as of time t. The protocol is as follows:

```
if d_{T_R} > d_{T_H} for any T_H
  then T_R waits
  else
       if c_{T_R} < c_{T_H} for any T_H
         then T_R waits
         else
              if c_{T_R} = c_{T_H} for any T_H
                then
                     if (time_needed_{T_R}(t) + t) > d_{T_R}
                       then T_R waits
                       else T_R aborts all T_H
                     end if
                else  T_R aborts all T_H
              end if
       end if
end if
```

Protocol 4 (CRP4): Based on a virtual clock.

Each transaction, T, has a virtual clock associated with it. The virtual clock value, $VT_T(t)$, for transaction T is calculated by the following formula.

$$VT_T(t) = s_T + \beta_T * (t - s_T), \quad t \geq s_T$$

where β_T is the clock running rate which is proportional to transaction T's criticalness. The higher the c_T, the larger the value β_T. The protocol controls the setting and running of the virtual clocks. When transaction T starts, $VT_T(t)$ is set to the current real time s_T. Then, the virtual clock runs at rate β_T . That is, the more critical a transaction is, the faster its virtual clock runs. In this work, $\beta_T = c_T$. The protocol is given by the following pseudo code.

if $d_{T_R} > d_{T_H}$ for any T_H
  then T_R waits
  else
    if any $VT_{T_H}(t) \geq d_{T_H}$
      then T_R waits
      else T_R aborts all T_H
    end if
end if

In this protocol, transaction T_R may abort T_H based on their relative deadlines, and on the criticalness and elapsed time of transaction T_H. When the virtual clock of an executing transaction has surpassed its deadline, it cannot be aborted. Intuitively, this means that for the transaction T_H to make its deadline, we are predicting that it should not be aborted. For further details about this protocol, the reader is referred to [29].

Protocol 5 (CRP5): Based on combining transaction parameters.

This protocol takes into account a variety of different information about the involved transactions. It uses a function $CP_T(t)$ to make decisions.

$$CP_T(t) = c_T * (w_1 * (t - s_T) - w_2 * d_T + w_3 * p_T(t) + w_4 * io_T(t) - w_5 * l_T(t))$$

where $p_T(t)$ and $io_T(t)$ are the CPU time and I/O time consumed by the transaction, $l_T(t)$ is the approximate laxity[2] (if known), and the w_k's are non-negative weights. The protocol is described by the following pseudo code.

> **if** $CP_{T_R}(t) \leq CP_{T_H}(t)$ for any T_H
>   **then** T_R waits
>   **else** T_R aborts all T_H
> **end if**

By appropriately setting weights to zero it is easy to create various outcomes, e.g., where a smaller deadline transaction always aborts a larger deadline transaction. Again, the reader is referred to [29] for further discussion of this protocol.

In a disk resident database system, it is difficult to determine the computation time and I/O time of a transaction. In our experiments, we simplify the above formula for CP calculation as follows:

$$CP_T(t) = c_T * [w1 * (t - s_T) - w2 * d_T + w3 * (R\_accessed_T(t)/R\_total_T)]$$

where R_total_T and $R\_accessed_T(t)$ are the same as defined in CRP3.

In summary, the five protocols resolve data conflict by either forcing the lock-requesting transaction to wait or aborting the lock holder(s), depending on various parameters of the conflicting transactions.

Deadlock Resolution

The use of a locking scheme may cause deadlock. This problem can be resolved by using deadlock detection, deadlock prevention, or deadlock avoidance. For example, CRP1 presented in the previous section prevents deadlock. In this study, we focus on the problem of deadlock

[2] Laxity is the maximum amount of time that a transaction can afford to wait but still make its deadline.

detection as it is required by the remaining concurrency control algorithms.

Under the deadlock detection approach, a deadlock detection routine is invoked every time a transaction is queued for a locked data object. If a deadlock cycle is detected, one of the transactions involved in the cycle must be aborted in order to break the cycle. Choosing a transaction to abort is a policy decision. For real-time transactions, we want to choose a victim so that the timing constraints of the remaining transactions can be met as much as possible, and at the same time the abort operation will incur the minimum cost. Here we present five deadlock resolution policies which take into account the timing properties of the transactions, the cost of abort operations, and the complexity of the protocols.

Deadlock resolution policy 1 (DRP1): *Always abort the transaction which invokes deadlock detection.* This policy is simple and efficient since it does not need any information from the transactions in the deadlock cycle.

Deadlock resolution policy 2 (DRP2): *Trace the deadlock cycle. Abort the first transaction T with $t > z_T$; otherwise abort the transaction with the longest deadline.*

Recall that a transaction which has passed its zero-value point, z_T, may not have been aborted yet because it may not have executed since passing z_T, and because preempting another transaction execution to perform the abort may not be advantageous. Consequently, in this and the following deadlock protocols we first abort any waiting transaction that has passed its zero-value point.

Deadlock resolution policy 3 (DRP3): *Trace the deadlock cycle. Abort the first transaction T with $t > z_T$; otherwise abort the transaction with the earliest deadline.*

Deadlock resolution policy 4 (DRP4): *Trace the deadlock cycle. Abort the first transaction T with $t > z_T$; otherwise abort the transaction with the least criticalness.*

Deadlock resolution policy 5 (DRP5): Here we use $time\_needed_T(t)$ as defined in CRP3. A transaction T is *feasible* if $(time\_needed_T(t)+t) < d_T$ and *tardy* otherwise. This policy aborts a tardy transaction with the least criticalness if one exists, otherwise it aborts a feasible transaction with the least criticalness. The following algorithm describes this policy.

```
Step 1: set tardy_set to empty
        set feasible_set to empty

Step 2: trace deadlock cycle
        for each T in the cycle do
        if t > z_T
          then abort T
               return
          else
               if T is tardy
                 then add T to tardy_set
                 else add T to feasible_set
               end if
        end if

Step 3: if tardy_set is not empty
          then search tardy_set for T with the least criticalness
          else search feasible_set for T with the least criticalness
        end if
        abort T
        return
```

In general, the experimental results from the testbed indicate the following:

- In a CPU-bound system, the CPU scheduling algorithm has a significant impact on the performance of real-time transactions, and dominates all of the other types of protocols. In order to obtain good performance, both criticalness and deadline of a transaction should be used for CPU scheduling;
- Various conflict resolution protocols which directly address deadlines and criticalness produce better performance than protocols that ignore such information. In terms of transaction's criticalness, regardless of whether the system bottleneck is the CPU or the I/O, criticalness-based conflict resolution protocols always improve performance; performance improvement due to cpu scheduling predominates that due to conflict resolution;
- Both criticalness and deadline distributions strongly affect transaction performance. Under our value weighting scheme, criticalness

is a more important factor than the deadline with respect to the performance goal of maximizing the deadline guarantee ratio for high critical transactions and maximizing the value imparted by real-time transactions;

- Overheads such as locking and message communication are shown to be non-negligible and cannot be ignored in real-time transaction analysis.

Real-Time Buffer Management

Data buffering plays an important role in database systems where part of the database is retained in a main memory space so as to reduce disk I/O and, in turn, to reduce the transaction response time. The principle of buffer management is based on transaction reference behaviors [20]. In terms of *locality*, there are basically three kinds of reference strings in database systems:

1. *intra-transaction locality*, where each transaction has its own reference locality, i.e., the probability of reference for recently referenced pages is higher than the average reference probability.

2. *inter-transaction locality*, where concurrent transactions access a set of shared pages.

3. *restart-transaction locality*, where restarted transactions repeat their previous reference behavior.

Buffer management policies should capitalize on one or more of these three types of locality.

Buffer allocation and buffer replacement are considered to be two basic components of database buffer management [11]. Buffer allocation strategies attempt to distribute the available buffer frames among concurrent database transactions, while buffer replacement strategies attempt to minimize the buffer fault rate for a given buffer size and allocation. The two schemes are closely related to each other and are usually integrated as a buffer management component in database systems.

In this work, we consider buffer management in real-time database systems where transactions have timing constraints, such as deadlines.

In a real-time environment, the goal of data buffering is not merely to reduce transaction response time, but more importantly, to increase the number of transactions satisfying their timing constraints. To achieve this goal, buffer management should consider not only transaction reference behaviors, but also the timing requirements of the referencing transactions.

We investigated several real-time buffer organizations based on the system structure of RT-CARAT which includes a workspace buffer for each transaction [16]. On RT-CARAT, we then implemented a global buffer in connection with a transaction recovery scheme using after-image journaling. Based on the overall system structure, we studied both real-time buffer allocation and real-time buffer replacement for the management of this global buffer which captures inter-transaction locality and restart-transaction locality.

Our basic idea for real-time buffer allocation is to distribute the available (global) buffer frames to the transactions with shorter deadlines. Let $T_i (i = 1, 2, ...n)$ be the total of n concurrent transactions in the system. The allocation scheme is described by the following algorithm.

1. sort T_i by $T_i.dl$ in *ascending* order, for $i = 1, 2, ...n$;
2. allocate the global buffer to the first m T_j's such that the following condition holds.

$$\sum_{j=1}^{m} T_j.ws \leq G\_buffer\_size < \sum_{j=1}^{m+1} T_j.ws \tag{4}$$

The replacement policy comes into play when there are no free global buffer frames for newly fetched pages. In a real-time database environment, the replacement scheme should aim not only at minimizing the buffer fault rate, but also at maximizing the number of transactions in meeting their timing constraints. The replacement scheme considered in this study is a modification of the LRU policy. Under the real-time replacement scheme a deadline and a count of active transactions using the page, is associated with each page. The deadline represents the largest deadline value of the transactions that have accessed that page. We also define a search window which is the maximum distance from the bottom of the LRU stack which the new algorithm will traverse. The algorithm then searches the LRU stack backwards checking either if no active transaction is using the page or if all transactions using the page have now passed their deadlines. If so the page is removed. If we do not find any such page in the window, then simply remove the last page in the window.

The experimental results obtained from the testbed indicate that under two-phase locking, the real-time oriented buffer management schemes do not significantly improve system performance. With regard to global buffer allocation, we have shown that data contention is a constraint on the performance improvement of buffer management. Under data contention, conflict resolution becomes a key factor in real-time transaction processing. In addition, CPU scheduling is more important than buffer allocation, even if the system is not CPU bound. Concerning buffer replacement, we have seen that the modified LRU algorithm which deals with dealines performs no better than the simple LRU policy. Again, under data contention, it is the conflict resolution that significantly improves transaction performance. This study suggests that, given an architecture with both local and global buffers, rather than developing sophisticated real-time buffer management schemes, it is more important to improve the performance of other processing components, such as conflict resolution, CPU scheduling and I/O scheduling.

Priority Inheritance Applied to Real-Time Transactions

Priority Inheritance is a technique for dealing with soft real-time tasks that access shared resources. In this approach, a task blocked by a lower priority task imparts its priority value to the task holding its needed resource. The idea is to allow the lower priority task to finish and release its resources quickly so that the higher priority task can continue. It has been shown that this approach is effective in real-time operating systems.

The goal of this work is to investigate Priority Inheritance in real-time transaction systems. By implementing and evaluating Priority Inheritance in our testbed [17], we found that for short transactions the performance of the system using Priority Inheritance is better than using simple two-phase locking. However, when compared to a priority abort scheme (where priority inversions are avoided by simply aborting the lower priority transaction), and to a combined abort and priority inheritance scheme which we call conditional priority inheritance, the basic Priority Inheritance protocol performed poorly. The reasons that the abort policy works well are that the higher priority transaction is never blocked, and an abort occurs as early as possible, not wasting system resources. Conditional priority inheritance works well because it aborts a low priority transaction when that transaction has not executed for very long (thereby wasting few resources), but raises the priority of low priority transactions when they are near completing (again, wasting few resources and also permitting more low priority transactions to complete). Further, we found that the performance for basic Priority Inheritance is even worse than simple two-phase locking for long transactions. This occurs because Priority Inheritance increases the deadlock rate and the transactions which get an increased priority execute for too long a time for the strategy to be effective (i.e., they significantly increase the blocking time for higher priority transactions). The main conclusion is that basic Priority Inheritance is inappropriate for conflict resolution under two-phase locking, and that both simple priority abortion (best under low data contention or loose deadlines) and conditional priority inheritance (best under high data and resource contention) work very well.

Optimistic Concurrency Control

While two-phase locking is widely used for concurrency control in non real-time database systems, this approach has some inherent disadvantages for real-time systems, such as the possibility of deadlocks and long and unpredictable blocking times. In seeking alternatives of two-phase locking, we investigate the optimistic approach [22] which ideally has the properties of non-blocking and deadlock freedom. Owing 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.

In this study [18, 19], we examine the overall effects and the impact of the overheads involved in implementing real-time optimistic concurrency control. Using a locking mechanism to ensure the correctness of the OCC implementation, we develop a set of optimistic concurrency control protocols. The protocols possess the property of deadlock freedom and have the potential for a high degree of parallelism. Integrated with priority-driven preemptive scheduling, the blocking time under the proposed protocol is limited and is predictable compared with 2PL.

Our performance studies conducted on RT-CARAT show that the blocking effect caused by the locking mechanism adopted in the implementation scheme has a major impact on the performance of the optimistic concurrency control protocol. In particular, the protocols are sensitive to priority inversion, but not to resource contention (as measured by I/O utilization). Furthermore, in contrast to the simulation results from [12, 13], our experimental results show that OCC may not always outperform a 2PL protocol which aborts the lower priority transaction when conflict occurs. The optimistic scheme performs better than the two-phase locking scheme when data contention is low, and vice versa when data contention is high. The "degraded" performance of the optimistic approach becomes apparent only because we considered the implementation details and since ours is a testbed, the overheads of the implementation manifest themselves in the performance figures. The experimental results indicate that the physical implementation schemes have a significant impact on the performance of real-time optimistic concurrency control.

We also investigate optimistic concurrency control in the context of the starvation problem. Because of their higher probability to conflict with other transactions, long transactions are likely to be repeatedly restarted and thus have less chance to meet their deadline than short transactions. Instead of limiting the number of transaction restarts, as is often proposed for traditional database systems, we use length and deadline sensitive priority assignment to address the problem. We show that integrated with the proposed weighted priority scheduling policy the optimistic concurrency control approach is more flexible in coping with the starvation problem than the two-phase locking scheme.

REAL-TIME I/O (DISK) SCHEDULING

In this section, we present two new disk scheduling algorithms for real-time systems and discuss their performance. The two algorithms, called SSEDO(for *Shortest Seek and Earliest Deadline by Ordering*) and

SSEDV(for *Shortest Seek and Earliest Deadline by Value*), combine *deadline* information and *disk service time* information in different ways. While the algorithms were evaluated as part of an integrated collection of protocols for real-time transaction processing, we believe that the results can be applied to any soft real-time system that requires real-time disk scheduling.

Before describing the algorithms we make some preliminary remarks and define a few symbols. Both algorithms maintain a queue of I/O requests sorted according to the (absolute) deadline of each request. A window of size m is defined as the first m requests in the queue. Hence, we may also refer to these two algorithms, SSEDO and SSEDV, as window algorithms. Let

r_i : be the I/O request with the i-th smallest deadline at a scheduling instance;

$dist_i$: be the distance between the current arm position and request r_i's position;

L_i : be the absolute deadline of r_i.

The SSEDO Algorithm

At each scheduling instance, the I/O scheduler selects one of the disk I/O requests from the window of size m for service. The scheduling rule is to assign each request r_i, a weight, say w_i where $w_1 = 1 \leq w_2 \leq \ldots \leq w_m$ and m is the window size, and to choose the one with the minimum value of $w_i * dist_i$. We shall refer to this quantity $w_i * dist_i$ as p_i, the *priority value* associated with request r_i. If there is more than one request with the same priority value, the one with the earliest deadline is selected. It should be clear that for any specific request, its priority value varies at each scheduling instance, since $dist_i$, r_i's position with respect to the disk arm position, is changing as the disk arm moves.

The idea behind the above algorithm is that we want to give requests with smaller deadlines higher priorities so that they can be serviced earlier. This can be accomplished by assigning smaller values to their weights. On the other hand, when a request with large deadline is "very" close to the current arm position (which means less service time), it should get higher priority. This is especially true when a request is to access the cylinder where the arm is currently positioned. Since there is no seek time in this case and we are assuming the seek time dominates the service time, the service time can be ignored. Therefore these requests should be given the highest priority. There are various ways to assign these weights w_i. In our experiments, the weights are simply set to

$$w_i = \beta^{i-1} \quad (\beta \geq 1) \qquad\qquad i = 1, 2, ..., n.$$

where β is an adjustable scheduling parameter. Note that w_i assigns priority only on the basis of the *ordering* of deadlines, not on their absolute or relative *values*. In addition, when all weights are equal ($\beta = 1$), we obtain an approximate Shortest Seek Time First (SSTF) algorithm which converges to pure SSTF as the window size becomes large. When the window size is equal to one, the algorithm is the same as the ED algorithm. Experimentally, we have shown that the performance of the system improves dramatically over ED when a window size of three or four is chosen even when the average queue length is as high as 15.

The SSEDV Algorithm

In the SSEDO algorithm, the scheduler uses only the ordering information of requests' deadline and does not use the differences between deadlines of successive requests. For example, suppose there are two requests in the window, and r_1's deadline is very close but r_2's deadline is far away. If r_2's position is "very" close to the current arm position, then the SSEDO algorithm might schedule r_2 first, which may result in the loss of r_1. However, if r_1 is scheduled first, then both r_1 and r_2 might get served. In the other extreme, if r_2's deadline is almost the same as r_1's, and the distance $dist_2$ is less than $dist_1$, but greater than $dist_1/\beta$, then SSEDO will schedule r_1 for service and r_2 will be lost. In this case, since there could be a loss anyway, it seems reasonable to serve the closer one (r_2) for its service time is smaller. Based on these considerations, we expect that a more intelligent scheduler might use not only the deadline *ordering* information, but also the deadline *value* information for decision making. This leads to the following algorithm: associate a priority value of $\alpha dist_i + (1 - \alpha)l_i$ to request r_i and choose the request with the minimum value for service, where l_i is the *remaining life time* of request

r_i, defined as the length of time between the current time and r_i's deadline L_i and $\alpha (0 \leq \alpha \leq 1)$ is a scheduling parameter. Again when $\alpha = 1$, this approximates the SSTF algorithm, and when $\alpha = 0$, we obtain the ED algorithm.

The performance of SSEDO and SSEDV algorithms is compared with three real-time disk scheduling algorithms proposed in the literature, ED, P-SCAN, and FD-SCAN, as well as four conventional algorithms SSTF, SCAN, C-SCAN, and FCFS. See [7] for a full description of these algorithms and their performance evaluation. An important aspect of the performance study is that the evaluation is not done in isolation with respect to the disk, but as part of an integrated collection of protocols necessary to support a real-time transaction system. The transaction system model was validated on RT-CARAT.

The main performance results are as follows:

- In a real-time system, I/O scheduling is an important issue with respect to the system performance. In order to minimize transaction loss probability, a good disk scheduling algorithm should take into account not only the *time constraint* of a transaction, but also the *disk service time*.

- The *earliest deadline* discipline ignores the characteristics of disk service time, and, therefore, does not perform well except when the I/O load is low.

- The window algorithms SSEDV and SSEDO consider two factors: *earliest deadline* and *shortest seek time*. The performance results show that SSEDV consistently outperforms SSEDO; that SSEDV can improve performance by 38% over previously-known real-time disk scheduling algorithms; and that all of these real-time scheduling algorithms are significantly better than non-real-time algorithms in the sense of minimizing the transaction loss ratio. We also showed that SSEDV algorithm performs better than SSEDO, since SSEDV uses more knowledge concerning the time constraint.

- For the SSEDV and the SSEDO algorithms, increasing the window size and the proper adjustment of parameters α and β can improve system performance, but increasing the window size beyond a particular value results in only marginal performance improvement.

- For a transaction system, if the number of operational steps for each transaction is known to the system as soon as a transaction is submitted to the system, we can define step deadlines according to the transaction's deadline and its step number. Scheduling by step deadlines is shown to be better than scheduling by transaction deadlines. This result may also have implications for cpu scheduling of tasks with precedence constraints, but with a single deadline.

- When transactions' read probability or sequential access probability are high, this improves system performance. In all cases, SSEDV and SSEDO algorithms are shown to be significantly better than the other disk scheduling algorithms considered. This conclusion also holds over a wide range of transaction deadline settings. In addition, by properly arranging the layout of the database on the disk, the SSEDV, SSEDO, and ED algorithms can improve performance to a proportionally greater degree than the other algorithms.

- The average transaction response time under SSEDV and SSEDO algorithms is higher than the SSTF and all the SCAN based algorithms, but lower than FCFS and ED.

Finally, with today's technology, the disk controller can be implemented to monitor the I/O load dynamically, and select a proper scheduling algorithm accordingly. This technique can be used with our SSEDV and SSEDO algorithms in a soft real-time environment. For example, when the I/O queue length is less than a threshold, the ED algorithm (window size 1 in SSEDV or SSEDO) might be used for scheduling, otherwise the window size would be set to 3 or 4. Alternatively, we might dynamically update the scheduling parameter α or β according to a queue length threshold. Finally, almost the entire execution time cost of executing the new algorithms can be done in parallel with disk seeks, thereby not adversely impacting disk service time.

CONCLUSIONS

In our work we have taken an integrated approach to developing algorithms for real-time transaction systems. We have developed new algorithms, implemented them on an experimental testbed called RT-CARAT, and evaluated their performance. Our main experimental results are that: (1) real-time cpu scheduling, conflict resoultion, and disk I/O scheduling are the three main factors in achieving good performance, (2) various conflict resolution protocols which directly address deadlines and criticalness can have a important impact on performance over protocols that ignore such information, (3) deadlock resolution and transaction restart policies tailored to real-time constraints seem to have negligible impact on overall performance, (4) optimistic concurrency control outperforms locking except when data contention is high, (5) basic priority inheritance should not be used in a locking-based real-time database setting, (6) real-time buffer management does not provide significant gain over typical buffer management techniques when the database is supported by local and global buffers, and (7) our new disk I/O scheduling algorithms are much more effective than others currently available.

Many important open questions remain including:

- how can soft real-time transaction systems be interfaced to hard real-time components?
- how can real-time transactions themselves be guaranteed to meet hard deadlines?
- how will real-time buffering algorithms impact real-time optimistic concurrency control?
- how will semantics-based concurrency control techniques impact real-time performance?
- how will the algorithms and performance results be impacted when extended to a distributed real-time system?
- how can correctness criteria other than serializability be exploited in real-time transaction systems?

ACKNOWLEDGMENTS

We wish to thank S. Chen, J. Huang, and W. Zhao for their work on the RT-CARAT project.

References

[1] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions," *ACM SIGMOD Record*, March 1988.

[2] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *Proceedings of the 14th VLDB Conference*, 1988.

[3] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data," *Proceedings of the 15th VLDB Conference*, 1989.

[4] R. Agrawal, M.J. Carey and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Transaction on Database Systems*, Vol.12, No.4, December 1987.

[5] A.P. Buchmann, et. al., "Time-Critical Database Scheduling: A Framework For Integerating Real-Time Scheduling and Concurrency Control," *Data Engineering Conference*, February 1989.

[6] M. J. Carey, R. Jauhari and M. Livny, "Priority in DBMS Resource Scheduling," *Proceedings of the 15th VLDB Conference*, 1989.

[7] S. Chen, J. Stankovic, J. Kurose, and D. Towsley, "Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems," *submitted for publication*, August, 1990.

[8] S. Chen, and D. Towsley, "Performance of a Mirrored Disk in a Real-Time Transaction System," to appear *Proc. 1991 ACM SIGMETRICS*, May 1991.

[9] U. Dayal, et. al., "The HiPAC Project: Combining Active Database and Timing Constraints," *ACM SIGMOD Record*, March 1988.

[10] U. Dayal, "Active Database Management Systems," *Proceedings of the 3rd International Conference on Data and Knowledge Management*, June 1988.

[11] W. Effelsberg and T. Haerder, "Principles of Database Buffer Management," *ACM Transactions on Database Systems*, Vol.9, No.4, December 1984.

[12] J. R. Haritsa, M.J. Carey and M. Livny, "On Being Optimistic about Real-Time Constraints," PODS, 1990.

[13] J. R. Haritsa, M.J. Carey and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control," *Proceedings of the 11th Real-Time Systems Symposium*, Dec. 1990.

[14] M. Hsu, R. Ladin and D.R. McCarthy, "An Execution Model for Active Database Management Systems," *Proceedings of the 3rd International Conference on Data and Knowledge Management*, June 1988.

[15] J. Huang, J. Stankovic, D. Towsley, and K. Ramamritham, "Experimental Evaluation of Real-Time Transaction Processing," *Proc. Real-Time System Symposium*, Dec. 1989.

[16] J. Huang and J. Stankovic, "Real-Time Buffer Management," COINS TR 90-65, August 1990.

[17] J. Huang, J. Stankovic, D. Towsley, and K. Ramamritham, "Priority Inheritance Under Two-Phase Locking," submitted for publication, Dec. 1990.

[18] J. Huang and J.A. Stankovic, "Concurrency Control in Real-Time Database Systems: Optimistic Scheme vs. Two-Phase Locking," *A Technical Report, COINS 90-66*, University of Massachusetts, July 1990.

[19] J. Huang, J.A. Stankovic, K. Ramamritham and D. Towsley, "Performance Evaluation of Real-Time Optimistic Concurrency Control Schemes," submitted for publication *VLDB*, also appears as *A Technical Report, COINS 91-16*, University of Massachusetts, Feb. 1991.

[20] J. P. Kearns and S. DeFazio, "Diversity in Database Reference Behavior," *Performance Evaluation Review*, Vol.17, No.1, May 1989.

[21] W. Kohler and B.P. Jenq, "CARAT: A Testbed for the Performance Evaluation of Distributed Database Systems," *Proc. of the Fall Joint Computer Conference*, IEEE Computer Society and ACM, Dallas Texas, November 1986.

[22] H. T. Kung and J.T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems, Vol.6, No.2*, June 1981.

[23] K. J. Lin, "Consistency Issues in Real-Time Database Systems," *Proceedings of the 22nd Hawaii International Conference on System Sciences*, January 1989.

[24] C. D. Locke, "Best-Effort Decision Making for Real-Time Scheduling," *Ph.D. Dissertation*, Canegie-Mellon University, 1986.

[25] G. M. Sacco and M. Schkolnick, "Buffer Management in Relational Database Systems," *ACM Transaction on Database Systems*, Vol.11, No.4, December 1986.

[26] L. Sha, R. Rajkumar and J.P. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record*, March 1988.

[27] S. H. Son, "Using Replication for High Performance Database Support in Distributed Real-Time Systems," *Proceedings of the 8th Real-Time Systems Symposium*, December 1987.

[28] S. H. Son and C.H. Chang, "Priority-Based Scheduling in Real-Time Database Systems," *Proceedings of the 15th VLDB Conference*, 1989.

[29] J. A. Stankovic and W. Zhao, "On Real-Time Transactions," *ACM SIGMOD Record*, March 1988.