Lock-Based Concurrency Control in Distributed Real-Time Database Systems*

Özgür Ulusoy†
Bilkent University

A real-time database system (RTDBS) is designed to provide timely response to the transactions of data-intensive applications. The transactions processed in a RTDBS are associated with real-time constraints typically in the form of deadlines. With the current database technology it is extremely difficult to provide schedules guaranteeing transaction deadlines. This difficulty comes from the unpredictability of transaction response times. Efficient resource scheduling algorithms and concurrency control protocols are required to schedule RTDB transactions so as to maximize the number of satisfied deadlines. In this paper, we describe several distributed, lock-based, real-time concurrency control protocols and report on the relative performance of the protocols in a distributed database environment. The protocols are different in the way real-time constraints of transactions are involved in controlling concurrent accesses to shared data. A detailed performance model of a distributed RTDBS was employed in the evaluation of concurrency control protocols.

A real-time database system (RTDBS) can be defined as a database system where transactions are associated with real-time constraints typically in the form of deadlines. A RTDBS is designed to provide timely information to data-intensive applications such as stock market, computer-integrated manufacturing, telephone switching systems, network management, and command and control systems (Ramamritham, 1992). Traditional database systems are designed to provide functionally correct information. Maintaining data consistency is the primary consideration in transaction scheduling. On the other hand, the basic issue considered in traditional real-time systems is the satisfaction of timing constraints associated with transactions. The problem of maintaining the consistency of shared data is usually not addressed. Design of a RTDBS requires the integration of scheduling concepts from both real-time systems and database systems to handle the timing and consistency requirements together. With the current database technology it is extremely difficult, if not impossible, to provide schedules guaranteeing transaction deadlines. This difficulty comes from the unpredictability of transaction response times. Each transaction operation accessing to a data item takes a variable amount of time due to concurrency control and disk IO (Stankovic & Zhao, 1988). The general approach to the scheduling problem in RTDBS's is using existing techniques in CPU scheduling, buffer management, IO scheduling and concurrency control, and to apply time-critical scheduling methods to make a best effort to satisfy transaction deadlines. The performance goal in satisfying timing constraints can change depending on the application environment. If the only real-time parameter associated with each transaction is the assigned deadline, the goal is to minimize the number of transactions that miss their deadlines. A priority order is established among transactions based on their deadlines. There are RTDB applications where transactions may be assigned different values, where the value of a transaction reflects the return the application expects to receive if the transaction commits within its deadline (Biyabani et al., 1988; Haritsa et al., 1991; Huang et al., 1989). For such applications, the performance goal is to maximize the value realized by the in-time transactions. Transactions are assigned

Manuscript originally submitted November 10, 1992; Accepted March 3, 1993 for publication.

priorities which are functions of both their values and deadlines.

Most of the recent research in RTDB transaction scheduling has concentrated on development and evaluation of concurrency control protocols. Each locking protocol proposed for RTDBS's is based on either one of the following two schemes: *priority inheritance* and *priority abort*. The priority inheritance scheme allows a low priority transaction to execute at the highest priority of all the higher priority transactions it blocks (Sha et al., 1988). The priority abort scheme is based on aborting the lower priority transaction when priority inversion[1] occurs. Development and evaluation of various lock-based concurrency control protocols are reported in (Abbott & Garcia-Molina, 1988, 1989; Agrawal et al., 1992; Huang et al., 1989, 1991a; Lin & Son, 1990; Sha et al., 1991; Son & Chang, 1990; Son et al., 1992; Ulusoy, 1992). Another class of proposed protocols is based on the optimistic method of concurrency control. Haritsa et al. (1990a, 1990b) and Huang et al. (1991b) present a set of optimistic protocols and provide the comparison of those protocols with locking. These performance comparisons do not completely agree due to the different types of systems used and the assumptions made in evaluating the protocols. Timestamp-based concurrency control protocols, which involve real-time priorities in constructing a timestamp order among transactions, are studied in (Son & Lee, 1990; Ulusoy, 1992).

Other recent research addressing the scheduling problem in RTDBS's can be summarized as follows. Some new approaches to priority-based IO scheduling are discussed in (Abbott & Garcia-Molina, 1990; Carey et al., 1989; Chen et al., 1991; Kim & Srivastava, 1991). Huang et al. (1989) provides the development and evaluation of several real-time policies for handling CPU scheduling. Özsoyoglu et al. (1990) introduces new techniques to process database queries within fixed time quotas. Different degrees of accuracy of the responses to the queries can be achieved by using those techniques. Evaluation of priority-based buffer management policies is reported in (Carey et al., 1989; Huang & Stankovic, 1990).

In this paper, we focus on *lock-based* concurrency control protocols for distributed RTDBS's. We describe several distributed, real-time concurrency control protocols and study the relative performance of the protocols in a nonreplicated database environment. The protocols aim to maximize the satisfaction of real-time requirements while maintaining data consistency via enforcing serializability. Concurrency control protocols are different in the way real-time constraints of transactions are involved in controlling concurrent accesses to shared data.

A detailed performance model of a distributed RTDBS was employed in the evaluation of concurrency control protocols. The performance model captures the basic characteristics of a distributed database system that processes transactions, each associated with a real-time constraint in the form of a deadline. A unique priority is assigned to each transaction based on its deadline. The transaction scheduling decisions are basically affected by transaction priorities. Various simulation experiments were carried out to study the relative performance of the protocols under many possible real-time and database environments. The performance metric used in evaluation of the protocols is *success-ratio*, which gives the fraction of transactions that satisfy their deadlines.

Distributed Real-Time Concurrency Control Protocols

Lock-based concurrency control in a distributed system is either *centralized*, where the lock management is provided by one of the sites, or *distributed*, where the lock managers are distributed along with the database. Our system model includes distributed concurrency control in which each scheduler manages locks for the data items[2] stored at its site based on the two-phase locking rules (Eswaren & Gray, 1976).

The concurrency control protocols studied assume a distributed transaction model in the form of a master process that executes at the originating site of the transaction and a collection of cohorts[3] that execute at various sites where the required data items reside. The master process is responsible for coordinating the execution of the cohorts as to be detailed in the section, *A Distributed RTDBS Model*. The cohorts carry the real-time priority of their transaction.

Each cohort process executing at a data site has to obtain a shared lock on each data item it reads, and an exclusive lock on each data item it updates. Conflicting lock requests on the same data item are ordered based on the real-time strategy implemented. Local serializability is provided by enforcing the rules of two-phase locking. In order to provide global serializability, the locks held by the cohorts of a transaction are maintained until the transaction has been committed. This constraint is enforced for the application of two-phase locking rules globally so that a transaction cannot obtain a lock after releasing a lock at another site.

The concurrency control protocols described in the following sections are distinguished based on whether they make use of a prior knowledge of data access patterns of transactions or not. The first group of protocols assume that data requirements of a transaction are not known before the execution of the transaction, while the second group of protocols assume that a list of data items to be accessed is submitted by each arriving transaction.

Protocols with Unknown Data Requirements

The first protocol is the basic version of two-phase locking and does not take real-time priorities into account in processing data access requests[4]. Performance of basic two-phase locking provides a basis of comparison for studying the performance of the priority-based protocols.

Always Block Protocol (AB)

```
lock_request_handling(D,C) {
   /* Cohort C requests a lock on data item D */
  if (A cohort C' is holding a conflicting lock on D)
       C is blocked by C';
    otherwise
       Lock on D is granted to C;
}
```

Figure 1: Lock request handling in protocol AB.

As presented in Figure 1, when a lock request results in a conflict, the cohort requesting the lock is always blocked by the cohort holding the conflicting lock. The cohort remains blocked until the conflicting lock is released. The real-time priority of the cohorts is not considered in processing the lock requests; lock requests are processed in FIFO order.

Priority Inheritance Protocol (PI)

Priority inheritance is one method proposed to overcome the problem of uncontrolled priority inversion (Sha et al., 1988, 1990). This method ensures that when a transaction blocks higher priority transactions, it is executed at the highest priority of the blocked transactions; in other words, it inherits the highest priority. Due to the inherited priority, the transaction can be executed faster resulting in reduced blocking times for high priority transactions.

```
lock_request_handling(D,C) {
   /* Cohort C requests a lock on data item D */
   if (A cohort C' is holding a conflicting lock on
D) {
       C is blocked by C';
       if (priority(C) > priority(C')) {
          C' inherits priority(C);
          A priority-inheritance message is sent to
the master of C';
       }
    }
    otherwise
       Lock on D is granted to C;
 }
```

Figure 2: Lock request handling in protocol PI.

Figure 2 shows how the lock requests are handled by this protocol in our distributed system model. When a cohort is blocked by a lower priority cohort, the latter inherits the priority of the former. Whenever a cohort of a transaction inherits a priority, the scheduler at the cohort's site notifies the transaction master process by sending a priority inheritance message, which contains the inherited priority. The master process then propagates this message to the sites of other cohorts belonging to the same transaction, so that the priority of the cohorts can be adjusted.

Some other details related to the implementation of protocol PI in simulations are as follows. When a transaction, which has inherited a priority, is aborted due to a deadlock, it is restarted with its original priority. If the holder of a data lock is a group of cohorts sharing the lock, and if a high priority cohort *C* is blocked due to a conflict on that item, the cohorts which are in the shared lock group and have lower priority than *C* inherit the priority of *C*.

When a cohort in the blocked state inherits a priority, that priority is also inherited by the blocking cohort (and its siblings) if it is higher than that of the blocking cohort.

Priority Abort Protocol (PA)

```
lock_request_handling(D,C) {
   /* Cohort C requests a lock on data item D */
  if (A cohort C' is holding a conflicting lock on D)
{
       if (priority(C) > priority(C')) {
          C' is aborted;
          An abort message is sent to the master of
C';
          Lock on D is granted to C;
       }
       otherwise
          C is blocked by C';
    }
    otherwise
       Lock on D is granted to C;
}
```

Figure 3: Lock request handling in protocol PA.

This protocol prevents priority inversion by aborting low priority transactions whenever necessary (Abbott & Garcia-Molina, 1988). We have implemented a distributed version of the protocol in our model as summarized in Figure 3. In the case of a data lock conflict, if the lock-holding cohort has higher priority than the priority of the cohort that is requesting the lock, the latter cohort is blocked. Otherwise, the lock-holding cohort is aborted and the lock is granted to the high priority lock-requesting cohort.

If the lock on a data item is shared by a group of cohorts, a cohort *C* requesting an exclusive lock on the data item is blocked if any cohort sharing the lock has higher priority than the priority of *C*. Otherwise (if the priority of *C* is higher than the priorities of all lock sharing cohorts), the transactions of all the cohorts in the lock share group are aborted.

Upon the abort of a cohort, a message is sent to the master process of the aborted cohort to restart the whole transaction. The master process notifies the schedulers at all relevant sites

to cause the cohorts belonging to that transaction to abort. Then it waits for the abort confirmation message from each of these sites. When all the abort confirmation messages are received, the master can restart the transaction.

Since a high priority transaction is never blocked by a lower priority transaction, this protocol is deadlock-free[5].

Protocols with Known Data Requirements

Some RTDBS's process certain kinds of transactions that can be characterized with well defined data requirements (Haritsa et al., 1990a). The concurrency control protocols executed in such systems can make use of this knowledge in data access scheduling. The following two concurrency control protocols assume that the data requirements of each transaction are known before the execution of the transaction. A list of data items that are going to be read or written is submitted to the scheduler by an arriving transaction. The protocols involve this information in scheduling the access requests of transactions.

Priority Ceiling Protocol (PC). This protocol, proposed by Sha et al. (1988) provides an extension to the priority inheritance protocol (PI). It eliminates transaction deadlocks and chained blockings from protocol PI. The *priority ceiling* of a data item is defined as the priority of the highest priority transaction that may have a lock on that item. To obtain a lock on a data item, the protocol requires that a transaction T must have a priority strictly higher than the highest priority ceiling of data items locked by the transactions other than T. Otherwise transaction T is blocked by the transaction which holds the lock on the data item of the highest priority ceiling.

```
lock_request_handling(D,C) {
   /* Cohort C requests a lock on data item D */
   if (priority(C) > MAX_PC_S)
       Lock on D is granted to C;
    otherwise {
       C is blocked by transaction TR_S;
       if (priority(C) > priority(TR_S)) {
           TR_S inherits priority(C);
             A priority-inheritance message is sent
to
             the master of TR_S;
       }
    }
}
```

Figure 4: Lock request handling in protocol PC.

In our model, a transaction list is constructed for each individual data item D in the system. The list contains the id and priority of the transactions in the system that include item D in their data access patterns. The list is sorted based on transaction priorities; the highest priority transaction determines the priority ceiling of D. At each data site S, the scheduler is responsible for keeping track of the current highest priority ceiling value of the locked data items at site S and the id of the transaction holding the lock on the item with the highest priority ceiling. Denoting these two variables by MAX_PC_S and TR_S respectively, Figure 4 shows how a lock request of a cohort process is handled by the scheduler. To determine the current values of MAX_PC_S and TR_S, the scheduler maintains a sorted list of (priority ceiling, lock-holding transaction) pairs of all the locked data items at that site.

In order to include read/write semantics in the protocol, the read/write priority ceiling rule introduced in (Sha et al., 1991) is used. The read/write priority ceiling of a read-locked data item is set to the priority of the highest priority cohort that will write into that item, and the read/write priority ceiling of a write-locked data item is set to the priority of the highest priority cohort which will read from or write into that item. In order to obtain a read or write lock on any data item, the protocol requires that a cohort C must have a priority strictly higher than the highest read/write priority ceiling of data items locked by the cohorts other than C.

Data-Priority-Based Locking Protocol (DP). This section presents the distributed version of the real-time concurrency control protocol we introduced in (Ulusoy, 1992). Like protocol PC, protocol DP is based on prioritizing data items; however, in ordering the access requests of transactions on a data item D, it considers only the priority of D without requiring a knowledge of the priorities of all locked items.

Each data item carries a priority which is equal to the highest priority of all transactions in the system that include the data item in their access lists. When a new transaction arrives at the system, the priority of each data item to be accessed is updated if the item has a priority lower than that of the transaction. When a transaction commits and leaves the system, each data item that carries the priority of that transaction has its priority adjusted to that of the highest priority active transaction that is going to access that data item. The DP protocol assumes that there is a unique priority for each transaction.

```
lock_request_handling(D,C) {
   /* Cohort C requests a lock on data item D */
   if (priority(C) = priority(D)) {
       if (D was locked by a cohort C') {
           C' is aborted;
           An abort message is sent to the master of
C';
       }
       Lock on D is granted to C;
    }
    otherwise
      C is blocked by the cohort that is responsible
        for the current priority of D;
}
```

Figure 5: Lock request handling in protocol DP.

A transaction list is maintained for each individual data item like in protocol PC. The list, located on the site of the data item, contains the id and priority of the transactions currently in the system that will access (or have accessed) the item. The list is sorted based on transaction priorities and the highest priority transaction determines the priority of the data item. The list is updated by the scheduler during the initialization and commit of relevant transactions.

Figure 5 summarizes how the lock requests are handled by protocol DP. In order to obtain a lock on a data item *D*, the priority of a cohort *C* must be equal to the priority of *D*. Otherwise (if the priority of *C* is less than that of *D*), cohort *C* is blocked by the cohort that determines the current priority of *D*. Suppose that *C* has the same priority as *D*, but *D* has already been locked by a lower priority cohort *C'* before *C* has adjusted the priority of *D*. *C'* is aborted at the time *C* needs to lock *D*. When a cohort is aborted due to data conflict, the aborted cohort's master is notified to restart the whole transaction.

The DP protocol can be augmented with read/write lock semantics. For this augmentation, each data item is associated with two priority values, one for read accesses and one for write accesses. To obtain a read lock on a data item *D*, the priority of a cohort *C* must be larger than or equal to the write priority of *D*. A write lock request of cohort *C* on data item *D* is honored if *C* has a priority equal to the write priority of *D*, and larger than or equal to the read priority of *D*.

The protocol is deadlock-free since a high priority transaction is never blocked by lower priority transactions and no two transactions have the same priority.

A Distributed RTDBS Model

This section provides the model of a distributed RTDBS that we used in evaluating the concurrency control protocols described in the preceding section. In the distributed system model, a number of data sites are interconnected by a local communication network. There exists exactly one copy of each data item in the system. Each site contains a transaction generator, a transaction manager, a resource manager, a message server, a scheduler, and a buffer manager.

The transaction generator is responsible for generating the workload for each data site. The arrivals at a data site are assumed to be independent of the arrivals at the other sites. Each transaction in the system is distinguished by a globally unique transaction id. The id of a transaction is made up of two parts: a transaction number which is unique at the originating site of the transaction and the id of the originating site which is unique in the system.

Each transaction is characterized by a real-time constraint in the form of a deadline. The transaction deadlines are *soft*; i.e., each transaction is executed to completion even if it misses its deadline. The transaction manager at the originating site of a transaction assigns a real-time priority to the transaction based on the *earliest deadline first* priority assignment policy; i.e., a transaction with an earlier deadline has higher priority than a transaction with a later deadline. If any two transactions originated from the same site carry the same deadline, a scheduling decision between those two transactions prefers the one that has arrived earlier. To guarantee the global uniqueness of the priorities, the id of the originating site is appended to the priority of each transaction.

A distributed transaction is modeled as a master process that executes at the originating site of the transaction and a collection of cohort processes that execute at various sites where the required data items reside. The transaction manager is responsible for the creation of the master process for each new transaction. The master process coordinates the execution of cohorts through communicating with the transaction manager of each cohort's site. There can be at most one cohort of a transaction at each data site. The operations of a transaction are executed in a sequential manner, one at a time. For each operation executed, the transaction manager refers to a global data dictionary to find out which data site stores the data item referenced by the operation. A cohort process is initiated at that site (if it does not exist already) by the master process by sending an 'initiate cohort' message to the transaction manager of that site. The initialization message contains all the information required for the execution of the cohort. The transaction manager refers to this information to initiate the cohort. The initialization information includes the id of the cohort's transaction, its priority, and, if required by the concurrency control protocol, the list of data items to be accessed by the cohort. The activation of each operation of a cohort is provided by the transaction manager of the cohort's site upon receiving an 'activate operation' message from the master process of the cohort. After the successful completion of an operation, the next operation in sequence is executed by the appropriate cohort. When the last operation is completed, the transaction can be committed. Another execution model in which the cohorts of a transaction act in parallel is discussed in the section, *Introducing Parallel Execution*.

The effects of a distributed transaction on the data must be made visible at all sites in an *all or nothing* fashion. The so called *atomic commitment* property can be provided by a commit protocol which coordinates the cohorts such that either all of them or none of them commit. In our model, atomic commitment of the distributed transactions is provided by the centralized two-phase commit protocol (Bernstein et al., 1987). For the commitment of a transaction *T*, the master process of *T* is designated as *coordinator*, and each cohort process executing *T*'s operations acts as a *participant* at its site. Following the execution of the last operation of transaction *T*, the coordinator (i.e., the master process of *T*) initiates Phase 1 of the commit protocol by sending a 'vote-request' message to all participants (i.e., cohorts of *T*) and waiting for a reply from each of them. If a participant is ready to commit,

it votes for commitment, otherwise it votes for abort. An abort decision terminates the commit protocol for the participant. After collecting the votes of all participants, the coordinator initiates Phase 2 of the commit protocol. If all participants vote for commit, the coordinator broadcasts a 'commit' message to them; otherwise, if any participant's decision is abort, it broadcasts an 'abort' message to the participants that voted for commit. If a participant, waiting for a message from the coordinator, receives a 'commit' message, the execution of the cohort of *T* at that site finishes successfully. Following the successful commit of *T*, each cohort can write its updates (if any) into the local database of its site. An 'abort' message from the coordinator causes the cohort to be aborted. In that case the data updates performed by the cohort are simply ignored.

There is no globally shared memory in the system, and all sites communicate via message exchanges over the communication network. A message server at each site is responsible for sending/receiving messages to/from other sites.

Access requests for data items are ordered by the scheduler on the basis of the concurrency control protocol executed. An access request of a cohort may result in blocking or abort of the cohort due to a data conflict with other cohorts executed concurrently. The scheduler at each site is also responsible for effecting aborts, when necessary, of the cohorts executing at its site. The schedulers at all sites together constitute a distributed scheduler.

If the access request of a cohort is granted, but the data item does not reside in main memory, the cohort waits until the buffer manager transfers the item from the disk into main memory. The FIFO page replacement strategy is used if no free memory space is available. Following the access, the data item is processed. When a cohort completes its data access and processing requirements, it waits for the master process to initiate two-phase commit. The master process commits a transaction only if all the cohort processes of the transaction run to completion successfully, otherwise it aborts and later restarts the transaction. A restarted transaction accesses the same data items as before (i.e., the items in its access list).

Deadlock is a possibility with blocking-based concurrency control protocols. Local deadlocks are detected by maintaining a local Wait-For Graph (WFG) at each site. WFG's contain the *wait-for* relationships among transactions. Local deadlock detection is performed by the scheduler each time an edge is added to the graph (i.e., when a cohort is blocked). Global deadlock is also a possibility in distributed systems. The detection of a global deadlock is a distributed task, requiring the exchange of information between local schedulers (Bernstein et al., 1987). A global WFG is used in our distributed system model to detect global deadlocks. The global WFG is constructed by unifying local WFG's. One of the sites is employed for periodic detection/recovery of global deadlocks. A deadlock is recovered from by selecting the lowest priority cohort in the deadlock cycle as a victim to be aborted. The master process of the victim cohort is notified to abort the whole transaction.

IO and CPU services at each site are provided by the resource manager. IO service is required for reading or updating data items, while CPU service is necessary for processing data items, performing various concurrency control operations (e.g. conflict check, locking, etc.) and processing communication messages. Both CPU and IO queues are organized on the basis of real-time priorities, and preemptive-resume priority scheduling is used by the CPU's at each site. The CPU can be released by a cohort process either due to a preemption, or when the process commits or it is blocked/aborted due to a data conflict, or when it needs an IO or communication service. Communication messages are given higher priority at the CPU than other processing requests.

Reliability and recovery issues were not addressed in this paper. We assumed a reliable system, in which no site failures or communication network failures occur. It was also assumed that the network has enough capacity to carry any number of messages at a given time, and each message is delivered within a finite amount of time.

Distributed RTDBS Model Parameters

The following parameters were used to specify the system configuration and workload. The parameter *nr_sites* represents the number of data sites in the distributed system. *db_size* specifies the number of data items stored in the database of a site, and *mem_size* specifies the number of data items that can be held in the main memory of a site. The mean interarrival time of transactions to each of the sites is determined by the parameter *iat*. The times between the arrival of transactions are exponentially distributed. The transaction workload consists of both query and update transactions. The transaction generator determines the type (i.e. query or update) of a transaction randomly using the parameter *tr_type_prob* which specifies the update type probability. *access_mean* corresponds to the mean number of data items to be accessed by a transaction. Accesses are uniformly distributed among data sites. For each data access of an update transaction, the probability that the accessed data item will be updated is specified by the parameter *data_update_prob*. The CPU time for processing a data item is determined by the parameter *cpu_time*, while the time to access a data item on the disk is specified by *io_time*. It is assumed that each site has one CPU and one disk. For each new transaction, there exists an initial CPU cost of assigning a unique real-time priority to the transaction. This processing overhead is simulated by the parameter *pri_assign_cost*. The parameter *mes_proc_time* corresponds to the CPU time required to process a communication message prior to sending or after receiving the message. The communication delay of messages between the sites is assumed to be constant and specified by the parameter *comm_delay*.

To evaluate the protocols fairly, the overhead of performing various concurrency control operations should be

taken into account. Our model considers the processing costs of the following operations:

- Conflict check: checking for a possible data conflict at each data access request of a transaction.
- Lock: obtaining a lock on a data item.
- Unlock: releasing the lock on a data item.
- Deadlock detection: checking for a deadlock cycle in a wait-for graph.
- Deadlock resolution: clearing the deadlock problem in case of detection of a deadlock.
- List manipulate: insert and delete operations on various kind of graphs/lists used for concurrency control purposes, which include the WFG's maintained for deadlock detection, the priority ceiling list of locked data items in protocol PC, and the transaction lists kept for data items in protocols PC and DP.

Each of these concurrency control operations is performed by executing a number of *elementary operations*. The set of elementary operations includes table lookup, table element value setting, comparison of any two values, and setting the pointer of a list element. It is assumed that all concurrency control information is stored in main memory. The conflict check operation performs a table lookup to determine if the requested data item has already been locked. Depending on the conflict detection policy of the protocol, the table lookup operation can be performed to get the priority of the lock-requesting transaction and the data item; and the table lookup is followed by a priority comparison operation. The lock and unlock operations use the table element value-setting operation to set/reset the lock flag of a data item. The deadlock detection requires a number of table lookup operations to get the wait-for information from the WFG and comparison operations to compare transaction ids in checking for a deadlock cycle. To recover from a deadlock, the minimum priority transaction in the cycle is chosen to abort; finding the minimum uses a number of priority comparison operations. Inserting an element into a sorted list requires performing some comparison operations to determine the insertion point, and a pointer-setting operation to insert the element into that place. Binary search is used to find the proper place in the list for insertion. Deleting from a sorted list requires the comparison operations for the search and a pointer-setting operation to provide the deletion. Insert/delete operations also perform table lookup to get the id or priority of the list elements to be used in comparisons. The number of comparison operations performed on any transaction list/graph is dependent on the size of the list/graph, and thus the transaction load in the system.

Assuming that the costs of executing each of the elementary operations are roughly comparable to each other, the processing cost for an elementary operation is simulated by using a single parameter: *basic_op_cost*.

Deadline Calculation

slack_rate is the parameter used in assigning deadlines to new transactions. The slack time of a transaction is chosen randomly from an exponential distribution with a mean of *slack_rate* times the estimated processing time of the transaction. The deadline of a transaction T is determined by the following formula:

$$deadline(T) = start\text{-}time(T) + processing\text{-}time\text{-}estimate(T) + slack\text{-}time(T)$$

where

$$slack\text{-}time(T) = expon(slack\_rate * processing\text{-}time\text{-}estimate(T))$$

$$processing\text{-}time\text{-}estimate(T) = t_1 + t_2 + t_3 + t_4 + t_5 + t_6 + t_7$$

Each component of the formula is specified as follows.

t_1: Priority assignment delay.

$t_1 = pri\_assign\_cost$

t_2: Delay to locate the execution site(s) for the operations.

$t_2 = items(T) * basic\_op\_cost$

items(T) refers to the actual number of data items accessed by T. *basic_op_cost* corresponds to the processing cost of locating a single data item.

t_3: Delay due to the cohort initialization messages.

$t_3 = nr\_coh\_sites(T) * mes\_proc\_time$

nr_coh_sites(T) is the actual number of remote data sites on which T has cohorts to perform its operations. A message is sent to each remote site to initialize the cohort of the transaction at that site. Each message is processed before being sent, resulting in a total delay of *nr_coh_sites(T) * mes_proc_time* units at its source.

t_4: Delay due to 'activate operation' and 'operation complete' messages for the remote operations.

$t_4 = 2 * rem\_items(T) * (mes\_proc\_time + comm\_delay + mes\_proc\_time)$

rem_items(T) is the actual number of remote data items to be accessed by T. Each 'activate operation' and 'operation complete' message has a communication overhead of (*mes_proc_time + comm_delay + mes_proc_time*) time units.

t_5: Processing delay of the operations of transaction T.

$t_5 = items(T) * cpu\_time$

t_6: IO delay of the operations of the transaction.

For a read-only transaction T,

$t_6 = items(T) * (1 - mem\_size/db\_size) * io\_time$

For an update transaction T,

t_6 = *items(T) * (1 - mem_size/db_size) * io_time + w_items(T) * io_time*

w_items(T) refers to the actual number of data items updated by *T*.

t_7: Commit protocol overhead.

t_7=[*num_coh_sites(T) * mes_proc_time + comm_delay + 2 * mes_proc_time + comm_delay + num_coh_sites(T) * mes_proc_time*]

+ [*num_coh_sites(T) * mes_proc_time*]

The terms contained within the first and the second square brackets correspond to overheads of Phase 1 and Phase 2 of the two-phase commit protocol, respectively. For Phase 1 of the protocol, *num_coh_sites(T) * mes_proc_time* is the CPU time spent at the source of transaction to process the 'vote-request' messages before sending them to each of the remote cohorts; *comm_delay* is the communication delay of the messages before arriving at their destinations; *2 * mes_proc_time + comm_delay* is the delay due to processing the 'vote-request' message and processing the reply message before sending it and the communication delay of the replies sent to the master; and *num_coh_sites(T) * mes_proc_time* is the time to read the reply messages from the remote cohorts. In determining the overhead of Phase 2, *num_coh_sites(T) * mes_proc_time* is the processing time for the final decision messages before they are sent to remote cohorts.

Performance Evaluation

The performance of the concurrency control protocols were evaluated by simulating them on the distributed RTDBS model described in the preceding section. The system parameter values used in simulation experiments are presented in Table 1. All data sites in the system are assumed identical and operate under the same parameter values. These values were selected to provide a transaction load and data contention high enough to bring out the differences between the real-time performances of concurrency control protocols. Since the protocols are different in handling data access conflicts among transactions, the best way to compare the performance characteristics of the protocols can be to conduct experiments under high data conflict conditions. The small *db_size* value is to create a data contention environment which produces the desired high level of data conflicts among transactions. This small database can be considered as the most frequently accessed fraction of a larger database. The values of *cpu_time* and *io_time* were chosen to yield a system with almost identical CPU and IO utilizations. Neither a CPU-bound nor an IO-bound system is intended to prevent the isolation of the effects of CPU contention or IO contention on the performance of the system. The method used in calculating expected CPU and IO utilizations in terms of system parameters is provided in (Ulusoy, 1992). The parameter *basic_op_cost* was assigned a value large enough to cover any elementary operation used by the various concurrency control operations; thus the cost of concurrency control operations was not ignored in evaluating the protocols.[6]

nr_sites	10
db_size	200
mem_size	50
iat	260 msec (exponential)
tr_type_prob	.5
access_mean	6 (exponential)
data_updata_prob	.5
cpu_time	8 msec (constant)
io_time	28 msec (constant)
comm_delay	5 msec (constant)
mes_proc_time	2 msec (constant)
pri_assign_cost	1 msec (constant)
slack_rate	5 (exponential)
basic_op_cost	0.1 msec (constant)

Table 1: Distributed RTDBS Model Parameter Values

The performance metric used in the evaluation of the protocols is *success-ratio*; i.e., the fraction of transactions that satisfy their deadlines. The other important performance metrics that helped us to understand the behavior of the protocols were *conflict-ratio* (i.e., the total number of conflicts observed over the total number of transactions processed), and *restart-ratio* (i.e., the total number of restarts observed over the total number of transactions processed).

The simulation maintains all the data structures (lists, graphs) specific to each protocol. Since it is assumed that the data structures used for concurrency control purposes are stored in main memory, no IO delay is involved in simulating the accesses to them. However, the processing cost of various operations on the lists/graphs is taken into account as detailed in the preceding section. The simulation program explicitly simulates data conflicts, wait queues for locks, CPU and IO queues, processing delay at the CPU, delay for disk IO, abort/restart of a transaction and all the concurrency control overheads considered in the model.

The simulation program was written in CSIM (Schwetman, 1986), which is a process-oriented simulation language based on the C programming language. For each configuration of each experiment, the final results were evaluated as averages over 25 independent runs. Each configuration was executed for 500 transactions originated at each site. 90% confidence intervals were obtained for the performance results. The width of the confidence interval of each data point is within 4% of the point estimate.

Varying Interarrival Time

The first experiment was conducted to examine the performance of protocols under varying transaction loads in the system. Mean time between successive transaction arrivals at a site was varied from 180 to 340 mseconds in steps of 40. These values of *iat* correspond to an average IO utilization

of about .93 to .50, and CPU utilization of about .92 to .49 at each of the sites (Ulusoy, 1992). We are basically interested in the evaluation of the protocols under high levels of transaction load.

The first group of protocols evaluated are the ones that do not assume any prior knowledge of data access requirements of transactions (i.e., AB, PI, and PA). The results are shown in Figure 6 in terms of *success-ratio*. Protocols PI and PA both provide improved performance, compared to protocol AB, by making use of real-time priorities in resolving data conflicts. Comparing the performance of these two protocols, it can be seen that PA provides considerably better performance than PI throughout the *iat* range. Resolving data conflicts by aborting low priority transactions seems to be more desirable in nonreplicated distributed RTDBS's than allowing the low-priority transactions to execute with inherited high priorities. Although we obtained the same result for the comparative performance of these protocols with a single-site RTDBS (Ulusoy, 1992), that result for the distributed system is rather interesting since the overhead of transaction restarts in distributed systems is much more compared to that in single-site systems. Aborting a distributed transaction leads to a waste of resources at each data site it has been executing on.

These results are similar to what Huang et al. obtained in (Huang et al., 1991a), where it was shown that protocol PA works better than protocol PI in single-site RTDBS's. They also found in (Huang et al., 1989) that the performance obtained by employing real-time policies based on the PA concurrency control protocol was better, in general, than that obtained in a nonreal-time transaction processing system. Abbott and Garcia-Molina (1989) also found that both protocols PI and PA perform better than protocol AB. Their results indicated that no protocol is the best under all conditions; the comparative performance of the protocols PI and PA depends on some other factors they considered, such as the type of load, and the priority policy. Under continuous and steady load, the performance of protocol PI was observed to be better than that of protocol PA. This result is different from what we have obtained in our experiment. The difference is probably due to the different assumptions made. Their experimental work ignored the CPU cost of deadlock detection and recovery. Our simulations captured the effects of deadlocks in protocol PI.

We next evaluated the performance of the concurrency control protocols that make use of a prior knowledge of data access patterns of transactions. Each transaction is assumed to submit its data access list as well as its deadline at the beginning of its execution. It can be seen from Figure 7 that the deadline satisfaction rate of transactions for protocol PC is quite low compared to that of protocol DP. One basic assumption made by protocol PC is that when an executing transaction *T* releases the CPU for a reason other than preemption (e.g., for IO), other transactions in the CPU queue are not allowed to get the CPU.[7] The CPU is idle until transaction *T* is reexecuted or a transaction with higher priority arrives at the CPU queue. CPU time is simply wasted when the CPU is not assigned to any of the ready transactions. Another factor leading to the unsatisfactory performance for protocol PC is the restrictive nature of the priority ceiling blocking rule, in the sense that, even if there exists no data conflict, the transactions can be blocked to satisfy the data access constraints of the protocol. A high conflict ratio is observed for protocol PC due to priority ceiling conflicts (average number of times each transaction is blocked due to the priority ceiling condition) rather than data conflicts. All these factors result in low concurrency and resource utilization in the system. Especially at high transaction loads, many transactions miss their deadlines.

If we compare the performance results of all the protocols from both groups, we can see that protocol DP shows a substantial improvement in real-time performance over all other protocols. The difference between the protocols' perfor-

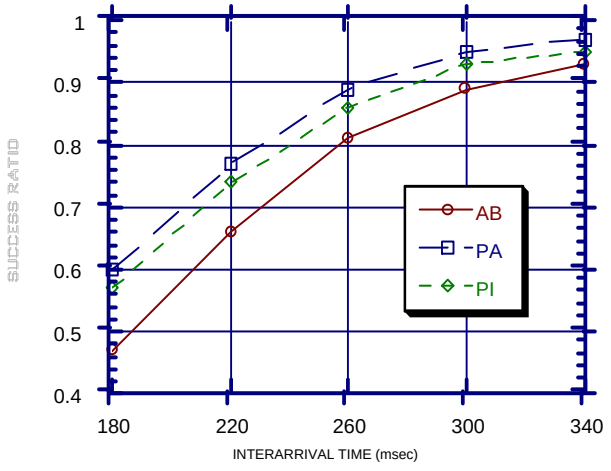

Figure 6: Real-time performance of the distributed concurrency control protocols AB, PI, and PA as a function of *iat* (mean transaction interarrival time).

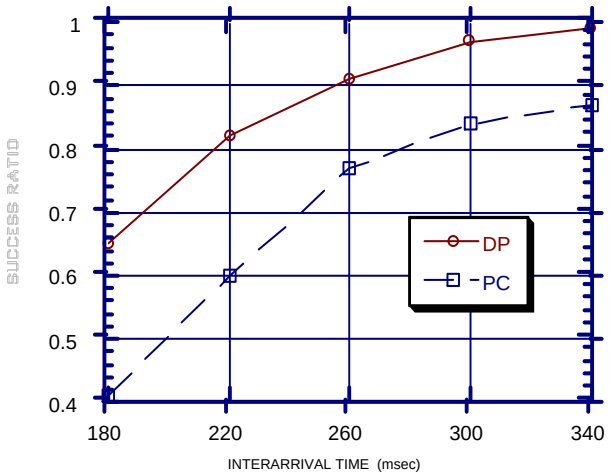

Figure 7: Real-time performance of the distributed concurrency control protocols PC and DP as a function of *iat* (mean transaction interarrival time).

mances increases as the load level increases with decreasing *iat* value. Similar to PA, protocol DP does not allow the situation in which a high priority transaction can be blocked by a lower priority transaction. This feature makes the protocol deadlock-free. The transaction restart rate is much lower than that of protocol PA since protocol DP restricts the possibility of transaction abort only to the following case. Between any two conflicting transactions, if the lower priority transaction accesses the data item before the higher priority transaction is submitted to the system (before the priority of the data item is adjusted by the entry of the higher priority transaction) the lower priority transaction is aborted when and if the higher priority transaction accesses the data item before the commitment of the lower priority transaction. One interesting observation is that the performance of protocol PC is worse than that of all the protocols placed in the first group. Even protocol AB, which does not use real-time priorities in scheduling data accesses, performs better than PC.

Son and Chang (1990) evaluated two different versions of the priority ceiling protocol (PC) in a distributed system environment, and observed that the performance of both protocols was better than that of the basic two-phase locking protocol (AB). In our work, protocol AB outperforms protocol PC. One basic assumption made by Son and Chang, which can give rise to this different result, is that the transaction deadlines are firm; i.e., transactions that miss their deadlines are aborted, and disappear from the system. This assumption can help to reduce blocking delays, because late transactions may be determining the priority ceiling of the data items, and when they are aborted, the blocked transactions continue their executions. As stated earlier, the transaction deadlines in our system are soft; i.e., all the transactions are processed to completion, regardless of whether they are late or not.

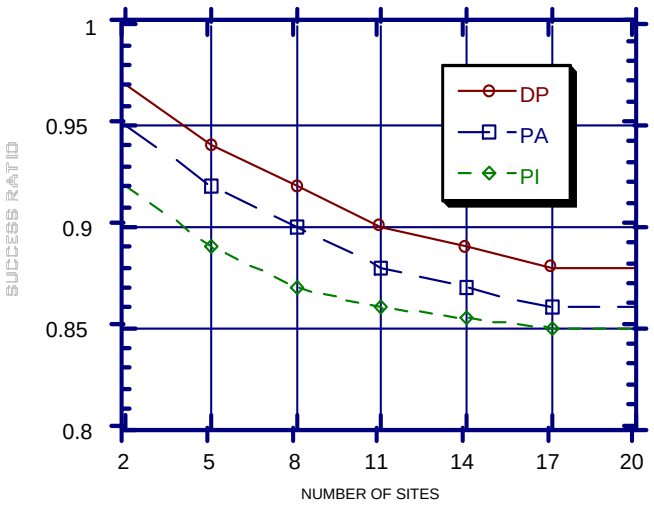

Figure 8: Real-time performance of protocols PI, PA, and DP as a function of *nr_sites* (number of data sites connected to the system).

Sha et al. (1991) also found that PC provides an improvement over basic two-phase locking. However, their experiments were performed in a restricted execution environment that required that a transaction could not start executing until the system prefetches data items that are in the access list of the transaction. The purpose of this restriction was to prevent the waste of the CPU resource during the IO activity of an executing transaction. They also assumed transactions with firm deadlines.

Impact of Number of Data Sites

In this experiment we studied the performance of the protocols for different numbers of data sites. For each configuration executed, the mean interarrival time of transactions at each site was fixed at *iat* = 260 msec. Since the data items accessed by each transaction are uniformly distributed over all data sites, increasing the number of sites leads to an increase in the average number of remote data accesses issued by each transaction. For instance, when the number of data sites in the system (i.e., *nr_sites*) is 2, about 50% of the items in a transaction's data access list resides at the remote site; however when 20 data sites exist in the system, about 95% of the data accesses will be remote. The number of remote data accesses directly affects the number of communication messages exchanged among data sites. The extra overhead of more communication messages is the increased average lifetime of a transaction due to the communication delay experienced, and increased CPU contention due to the processing delay of communication messages at each site. The effect of changing the size of the distributed system on the performance of concurrency control protocols is exhibited in Figure 8. For large-sized systems, almost all data accesses are remote; thus, the real-time performance is not much affected by further enlargement of the system by connecting new data sites. The figure presents the performance results in terms of *success-ratio* for three of the protocols (PI, PA, and DP). The performances of the other protocols were not displayed since all the protocols responded similarly to varying the number of data sites. Comparative performances of the protocols are similar to the results obtained in the experiment of the preceding section.

Effects of Communication Delay

In this experiment the effect of communication delay of messages on the real-time performance of the system was investigated. Again, the mean interarrival time of transactions at each site was fixed at 260 msec. The protocols were executed with *comm_delay* values of 1, 3, 5, 10, and 20 msec. There is no contention among the messages for the communication network due to our unlimited network capacity assumption. Thus, the messages do not experience any queuing delay for the communication resource, regardless of how long each message requires to use that resource. The length of commu-

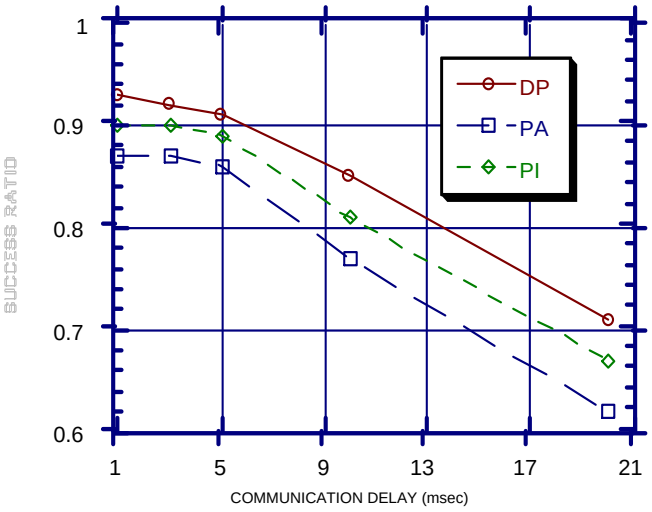

Figure 9: Real-time performance of protocols PI, PA, and DP as a function of *comm_delay* (communication delay).

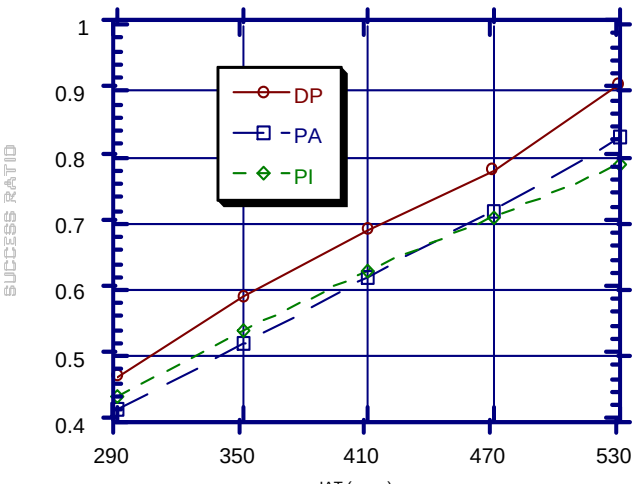

Figure 10: Real-time performance of protocols PI, PA, and DP as a function of mean interarrival time with *access_mean* (average number of data items accessed by a transaction) value of 10.

nication message delay can only affect the average lifetime of transactions. The performance results of the concurrency control protocols PI, PA, and DP are displayed in Figure 9. If the communication network is fast, corresponding to small values of *comm_delay* (less than 5 msec), communication network speed has not much effect on the protocols' performance. However, when the communication network becomes slower, the performance becomes more sensitive to varying the value of *comm_delay*, since the overall message delay experienced by a transaction becomes a more dominant factor in determining its lifetime. On the other hand, the relative performance of the protocols are not affected by the speed of the communication network.

Impact of Transaction Length

The simulation experiments of preceding sections were all carried out for the *access_mean* (average number of data items accessed by each transaction) value of 6. This experiment evaluated the concurrency control protocols under different execution environments each with a different average transaction length value. When the transactions are short-lived (with *access_mean* value less than 6), all protocols perform well due to small number of conflicts among concurrent transactions. The number of conflicts increases as the average lifetime of transactions becomes longer. This leads to an increase in blocking time or in the number of restarts (depending on the type of the concurrency control protocol executed), and thus to a poor performance for each protocol. The relative performance of protocols PA and PI is also affected by the average length of transactions. When the processed transactions are characterized by large number of data access requests, the performance of protocol PI becomes comparable to that of protocol PA. The relative degradation in the performance of PA can be explained by the increasing negative impact of transaction restarts with longer transactions. Restarting a longer transaction wastes more processing time and other system resources already used by that transaction. Figure 10 presents the results obtained with *access_mean* value of 10 for these two protocols (PA and PI), together with the results of protocol DP which again appeared to be the best. The *iat* values of 290 through 530 msec used with this particular *access_mean* value corresponds to average IO utilization of .95 to .54 (Ulusoy, 1992). Executions with larger average transaction lengths (i.e., *access_mean* > 10) produce similar comparative performance results for the protocols.

Other experiments were conducted to evaluate the effect of various other system parameters on the protocols' performance. These parameters included *mes_proc_time* (CPU time to process a communication message), *slack_rate* (ratio of the slack time of a transaction to its execution time), *tr_type_prob* (ratio of update type transactions), *mem_size* (size of main memory at each site), and *db_size* (size of database at each site). The results of each of these experiments can be found in (Ulusoy, 1992). The comparative performance of the protocols was not sensitive to varying the values of these parameters.

Sensitivity to Number of Nondistributed Transactions

In the experiments of preceding sections, all the transactions were distributed; i.e., data items to be accessed by each transaction were randomly distributed among all the sites. In this experiment some of the transactions are nondistributed, meaning that they only access local data items. A nondistributed transaction is executed at its site only and does

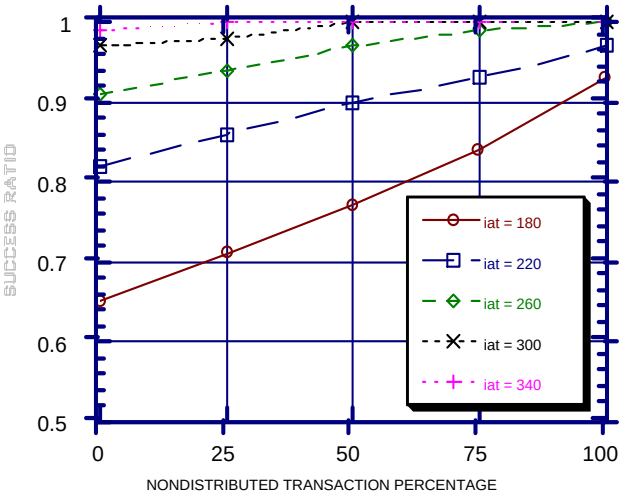

Figure 11: Real-time performance of protocol DP (for different values of mean transaction interarrival time) as a function of the percentage of nondistributed transactions submitted to the system.

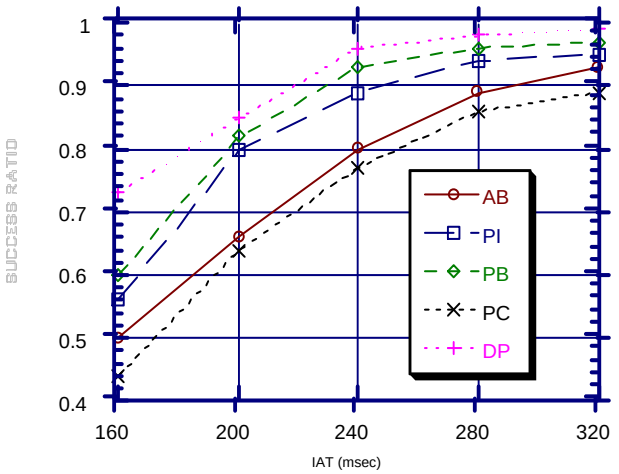

Figure 12: Real-time performance of the concurrency control protocols under the parallel transaction execution model.

not submit any cohorts to remote sites.

To evaluate the sensitivity of the real-time performance of the system to the number of nondistributed transactions, we varied the fraction of nondistributed transactions from 0.0 to 1.0 in steps of 0.25. For a nondistributed transaction we don't have the overhead of communication messages transferred between the master and cohort processes of a transaction. We also don't need a distributed commit protocol; a nondistributed transaction can commit after completing its last operation. Increasing the proportion of nondistributed transactions in the system results in better real-time performance for all concurrency control protocols. The *success-ratio* results are displayed for protocol DP in Figure 11, which shows the effect of increasing the fraction of nondistributed transaction for different values of mean interarrival time. We see that, as the transaction load level increases with decreasing mean interarrival times, the ratio of nondistributed transactions becomes more important in determining real-time performance.

Introducing Parallel Execution

In this experiment the system model was modified to include parallel execution. In the modified model, the master process of a transaction spawns cohorts all together, and the cohorts are executed in parallel. The master process sends to each remote site a message containing an (implicit) request to spawn a cohort, and the list of all operations of the transaction to be executed at that site. The assumption here is that the operations performed by one cohort are independent of the results of the operations performed at the other sites. The sibling cohorts do not have to transfer information to each other. A cohort is said to be completed at a site when it has performed all its operations. A completed cohort informs the master process by sending a 'cohort complete' message. The master process can start the two-phase commit protocol when it has received 'cohort complete' messages from all the cohorts. The lock management policy is similar to that of the sequential execution model. Each cohort holds its locks until the commitment of the whole transaction.

Various experiments were performed for the parallel execution model. Here we present the results of varying interarrival time from 160 through 320 mseconds in steps of 40. This *iat* range corresponds to the IO utilization range of [.94,.47], and CPU utilization range of [.93,.47]. Resource utilizations in the sequential and parallel execution models are different because the overhead is different (Ulusoy, 1992).

Figure 12 shows the performance results obtained for each of the protocols under the parallel execution model. The performance of each protocol is better, in general, than it was for the sequential execution model. Since the cohorts of a transaction do not have to wait for information from each other, they can execute faster, thus experiencing fewer data conflicts with the cohorts of other transactions. The comparative performance of the protocols looks similar to that for the sequential execution case. The best performing protocol is DP while PC performs worse than all the other protocols. Note that the largest amount of performance improvement provided by protocol DP over the other protocols is when the system is highly loaded.

Conclusions

Unlike traditional database systems, the transactions processed in a real-time database system (RTDBS) have timing constraints, typically in the form of deadlines. Crucial to the concurrency control problem in RTDBS's is processing

the transactions within their deadlines, as well as maintaining the consistency of data accessed by the transactions. In this paper, we have described a number of lock-based, distributed, real-time concurrency control protocols and studied their performance in a distributed database system environment. The protocols are different in the way real-time constraints of transactions are involved in controlling concurrent accesses to shared data. Various simulation experiments were conducted by using a detailed performance model to evaluate the effects of different system parameters on the performance of the protocols. The performance was measured in terms of the fraction of transactions that satisfy their deadlines.

The first group of protocols studied in this paper detect the data access requirements of transactions dynamically. The priority abort protocol (PA) aborts a low priority transaction when one of its locks is requested by a higher priority transaction (i.e., a transaction with an earlier deadline). The priority inheritance protocol (PI), on the other hand, allows a low priority transaction to execute at the highest priority of all the higher priority transactions it blocks. Both protocols appeared to perform better than the always block protocol (AB), which does not involve real-time priorities in access scheduling decisions. Protocol PA provided better performance than protocol PI under various execution environments except when the transactions processed were characterized by long execution times. The performance of protocol PI became comparable to that of protocol PA in a long-length-transaction environment as a result of more resource waste experienced by restarting transactions with protocol PA.

Other concurrency control protocols studied assume that data access requirements of transactions are known prior to their execution. Introducing a data-priority-based protocol (DP), we have shown that some of the transaction aborts, and the resulting resource waste experienced by protocol PA can be prevented by employing prior knowledge of data requirements in access scheduling. Protocol DP outperformed all other protocols under different levels of transaction load.

An interesting result obtained in the evaluations was the poor performance exhibited by the priority ceiling protocol (PC), which is another protocol making use of the prior knowledge of data access requirements of transactions. Although the protocol was intended to be an improvement over protocol PI, its performance was even worse than the basic two-phase locking protocol. The priority ceiling rule that might block the lock requesting transactions even without an existence of data conflicts, is too restrictive to be implemented in RTDBS's.

The comparative performances of the protocols were similar under two different models of transaction execution: sequential and parallel. A system parameter critical to the performance of the protocols was the volume of communication messages exchanged among the sites. All the protocols performed worse as the message traffic in the system increased as a result of increasing the size of the distributed system (number of sites) or the ratio of distributed/locally-processed transactions. It was observed that, under high levels of transaction load the ratio of distributed transactions became more important in determining real-time performance.

Acknowledgement

I would like to thank Prof. Geneva G. Belford of University of Illinois for her helpful comments on earlier versions of this paper.

Endnotes

* An earlier version of this paper was published in the Proceeding of the 12th International Conference on Distributed Computing Systems, 1992.

† This work was done while the author was at the Computer Science Department, University of Illinois at Urbana-Champaign.

1 Priority inversion is blocking of a high priority transaction by lower priority transactions.

2 The basic unit of access is referred to as a data item.

3 The cohorts of a transaction will be referred to as 'sibling cohorts' of each other throughout the paper.

4 However, as to be described in the section *A Distributed RTDBS Model.*, a preemptive-resume strategy based on real-time priorities is used in scheduling the CPU.

5 The assumption here is that the real-time priority of a transaction does not change during its lifetime and that no two transactions have the same priority.

6 The results and discussions for some other settings of the value of this parameter can be found in (Ulusoy, 1992).

7 The implementation of protocol PC in our model followed this assumption unlike the other protocols whose implementation included the CPU scheduling method described in the section, *A Distributed RTDBS Model.*

References

Abbott R. & H. Garcia-Molina (1988). Scheduling Real-Time Transactions: A Performance Evaluation. *In Proceedings of 14th International Conference on Very Large Data Bases,* pp.1-12.

Abbott R. & H. Garcia-Molina (1989). Scheduling Real-Time Transactions with Disk Resident Data. In *Proceedings of 15th International Conference on Very Large Data Bases,* pp.385-396.

Abbott R. & H. Garcia-Molina (1990). Scheduling I/O Requests with Deadlines: A Performance Evaluation. In *Proceedings of 11th Real-Time Systems Symposium,* pp.113-124.

Agrawal D., A. El Abbadi & R. Jeffers (1992). Using Delayed Commitment in Locking Protocols for Real-Time Databases. In *Proceedings of ACM SIGMOD Conference,* pp.104-113.

Bernstein P.A., V. Hadzilacos & N. Goodman (1987). *Concurrency Control and Recovery in Database Systems,* Addison-Wesley.

Biyabani S.R., J.A. Stankovic & K. Ramamritham (1988). The Integration of Deadline and Criticalness in Hard Real-Time Scheduling. In *Proceedings of 9th Real-Time Systems Symposium,* pp.152-160.

Carey M.J., R. Jauhari & M. Livny (1989). Priority in DBMS Resource Scheduling. In *Proceedings of 15th International Conference on Very Large Data Bases,* pp.397-410.

Ceri S. & G. Pelagatti (1984). *Distributed Databases: Principles and Systems*, McGraw-Hill.

Chen S., J.A. Stankovic & J. Kurose, (1991). Townley D. Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems. *Real-Time Systems Journal, 3*(3):307-336.

Eswaren K.P. & J.N. Gray (1976). The Notion of Consistency and Recovery in a Database System. *Communications of ACM, 19*(11):624-633.

Haritsa J.R., M.J. Carey & M. Livny (1990a). On Being Optimistic About Real-Time Constraints. In *Proceedings of ACM SIGACT-SIGMOD-SIGART Conference*, pp.331-343.

Haritsa J.R., M.J. Carey & M. Livny (1990b). Dynamic Real-Time Optimistic Concurrency Control. In *Proceedings of 11th Real-Time Systems Symposium*, pp.94-103.

Haritsa J.R., M.J. Carey & M. Livny (1991). *Value-Based Scheduling in Real-Time Database Systems*, Technical Report, No.1204, Dept. of Computer Science, University of Wisconsin-Madison.

Huang J., J.A. Stankovic, D. Towsley & Ramamritham K. (1989). Experimental Evaluation of Real-Time Transaction Processing. In *Proceedings of 10th Real-Time Systems Symposium*, pp.144-153.

Huang J. & J.A. Stankovic (1990). *Real-Time Buffer Management*, COINS TR 90-65, Dept. of Computer and Information Science, University of Massachusetts, Amherst.

Huang J. & J.A. Stankovic, K. Ramamritham, D. Towsley (1991a). On Using Priority Inheritance In Real-Time Databases. In *Proceedings of 12th Real-Time Systems Symposium*, pp.210-221.

Huang J., J.A. Stankovic, D. Towsley & K. Ramamritham (1991b). Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes. In *Proceedings of 17th International Conference on Very Large Data Bases*, pp.35-46.

Kim W. & J. Srivastava (1991). Enhancing Real-Time DBMS Performance with Multiversion Data and Priority Based Disk Scheduling. In *Proceedings of 12th Real-Time Systems Symposium*, pp.222-231.

Lin Y. & S.H. Son (1990). Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order. In *Proceedings of 11th Real-Time Systems Symposium*, pp.104-112.

Özsoyoglu G., Z.M. Özsoyoglu & W.C. Hou (1990). Research in Time and Error-Constrained Database Query Processing. In *Proceedings of 7th IEEE Workshop on Real-Time Operating systems and Software*, pp.32-38.

Ramamritham K. (1992). Real-Time Databases' to appear in *International Journal of Distributed and Parallel Databases.*

Schwetman H. (1986). CSIM: A C-Based, Process-Oriented Simulation Language. In *Proceedings of Winter Simulation Conference*, pp.387-396.

Sha L., R. Rajkumar & J. Lehoczky (1988). Concurrency Control for Distributed Real-Time Databases. *ACM SIGMOD Record, 17*(1):82-98.

Sha L., R. Rajkumar & J. Lehoczky (1990). Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers, 39*(9):1175-1185.

Sha L., Rajkumar R., S.H. Son & C.H. Chang (1991). A Real-Time Locking Protocol. *IEEE Transactions on Computers, 40*(7):793-800.

Son S.H. & C.H. Chang (1990). Performance Evaluation of Real-Time Locking Protocols Using a Distributed Software Prototyping Environment. In *Proceedings of 10th International Conference on Distributed Computing Systems*, pp.124-131.

Son S.H. & J. Lee (1990). Scheduling Real-Time Transactions in Distributed Database Systems. In *Proceedings of IEEE Workshop on Real-Time Operating Systems and Software*, pp.39-43.

Son S.H., S. Park & Y. Lin (1992). An Integrated Real-Time Locking Protocol. In *Proceedings of 8th International Conference on Data Engineering*, pp.527-534.

Stankovic J.A. & W. Zhao (1988). On Real-Time Transactions. *ACM SIGMOD Record, 17*(1):4-18.

Ulusoy Ö. (1992). *Concurrency Control in Real-Time Database Systems*, Technical Report, UIUCDCS-R-92-1762, University of Illinois at Urbana-Champaign.

Ozgur Ulusoy received the B.S. degree from Middle East Technical University, Ankara, Turkey, the M.S. degree from Bilkent University, Ankara, Turkey, and the Ph.D. degree from the University of Illinois at Urbana-Champaign in 1986, 1988, and 1992 respectively, all in Computer Science. He is currently an Assistant Professor at Bilkent University. His research interests include Computer Networks and Distributed Systems, Database Systems, Real-Time Systems, and Performance Evaluation.