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

Özgür Ulusoy, Geneva G. Belford
Department of Computer Science
University of Illinois at Urbana–Champaign
Urbana, IL 61801

Abstract

In a real-time database system, it is difficult to meet all timing constraints due to the consistency requirements of the underlying database. Real-time database transaction scheduling requires the development of efficient concurrency control protocols that try to maximize the number of transactions satisfying their real-time constraints. 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 nonreplicated database environment. The protocols take the real-time requirements of the transactions into account in ordering data accesses, while maintaining data consistency via enforcing serializability.

Keywords: *Real-time database systems, distributed concurrency control, performance evaluation.*

1 Introduction

A 'real-time database system (RTDBS)' is a database system designed to provide real-time information to data-intensive applications such as production control and manufacturing. Each real-time database transaction is associated with a timing constraint, typically in the form of a deadline. It is difficult to meet the timing constraints of all transactions due to the consistency requirements of the underlying database. Efficient concurrency control protocols are required in RTDBSs to maximize the number of transactions satisfying their real-time constraints while maintaining the consistency of data. Our earlier study [15] investigated the performance of various concurrency control approaches in a single-site RTDBS. In this paper, our work concentrates on 'distributed' concurrency control protocols. We describe several distributed, lock-based, real-time, concurrency control protocols and study the relative performance of the protocols in a nonreplicated database environment. The protocols attempt to maximize the satisfaction of real-time requirements while maintaining data consistency via enforcing serializability. Concurrency control protocols are different in the way data conflicts among the transactions are resolved once they are detected. The performance metric used in evaluation of the protocols is *success-ratio*, which gives the fraction of transactions that satisfy their deadlines.

The rest of the paper is organized as follows. In the next section, we summarize recent work that has addressed the scheduling problem in RTDBSs. Section 3 describes the distributed transaction structure and distributed execution model used in our system. The structure and characteristics of our distributed database system model are presented in Section 4. The lock-based concurrency control protocols studied in our system are described in Section 5. Section 6 presents the performance evaluation experiments and discusses the results obtained.

2 Related Work

The scheduling problem in RTDBSs has been addressed by a number of recent studies. The first performance evaluation work in RTDBSs was provided by Abbott and Garcia-Molina [1], [2]. They described a group of lock-based algorithms for scheduling real-time transactions, and evaluated the algorithms through simulation. Sha et al. [11], [12] presented a concurrency control protocol, called priority ceiling, which prevents blocking deadlocks and attempts to minimize the blocking time of high priority transactions. The performance of the protocol was examined in [13]. In [14], Son and Chang investigated methods to apply the priority-ceiling protocol as a basis for real-time locking protocol in a distributed environment. Huang et al. [8] developed and evaluated several algorithms for handling CPU scheduling, data conflict resolution, deadlock resolution, transaction wakeup, and transaction restart in RTDBSs. Later, their work was extended to the optimistic concurrency control method [9]. Haritsa et al. [6] studied the relative performance of two well known classes of concurrency control algorithms (locking protocols and optimistic techniques) in a RTDBS environment. They presented and evaluated a new real-time optimistic concurrency control protocol through simulations in [7].

3 Distributed Transaction and Execution Model

We model a distributed transaction as a master process that executes at the originating site of the transaction and a collection of cohorts that execute at various sites where the required data items reside. At the arrival of a transaction request at a site, the transaction manager of that site creates a master process and cohort processes[1] after determining the sites storing the required data items. The master process is responsible for the coordination of cohort processes; it does not itself perform any database operations. The execution model adopted in our system executes the operations of a transaction sequentially. There can be at most one cohort of a transaction at each site, and only one cohort can be active at a time. After the successful completion of each operation, the next operation in sequence is executed by the appropriate cohort. After the execution of the last operation, the transaction can be committed. Atomic commitment of each transaction is provided by the centralized two-phase commit protocol [3].

Another execution model which processes the cohorts of a transaction in parallel is discussed in Section 6.1.

4 Distributed Database System Model

This section provides the model of a distributed RTDBS used in this paper. In the distributed system model data is distributed over a number of data sites connected through a communication network. There exists exactly one copy of each data item[2] in the system. Each site contains a transaction manager, a scheduler, a resource manager, and a message server.

The transaction manager is responsible for generating transaction identifiers and assigning real-time priorities to transactions. Each transaction submitted to the system is associated with a real-time constraint in the form of a deadline. The transaction is assigned a globally distinct real-time priority based on its deadline. For each transaction, the transaction manager creates a master process to be executed locally and specifies the appropriate sites for the execution of the cohort processes. If there exist any local data items in the access list of the transaction, one of the cohorts will be executed locally. The master process initiates the execution of each cohort process. Remote cohorts are initiated by sending a message to the appropriate sites. Each cohort is executed with its transaction's priority.

Each cohort performs one or more database operations on specified data items. 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.

Concurrent data access requests of the cohort processes at a site are controlled by the scheduler at that site. The scheduler orders the data accesses based on the concurrency control protocol executed. When a cohort completes all its accesses and processing requirements, it waits for the master process to initiate two-phase commit. Following the successful commitment of the distributed transaction, the cohort writes its updates, if any, into the local database.

Each site's resource manager is responsible for providing IO service for reading/updating data items, and CPU service 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 the cohorts' real-time priorities. Preemptive-resume priority scheduling is used by the CPUs at each site; a higher-priority process preempts a lower-priority process, and the lower-priority process can resume when there exists no higher-priority process waiting for the CPU. Communication messages are given higher priority at the CPU than other processing requests. Besides preemption, the CPU can be released by a cohort process as a result of lock conflict, for IO, or communication to other data sites.

The message server at each site is responsible for sending/receiving messages to/from other sites. It listens on a well-known port, waiting for remote messages.

Reliability and recovery issues are not addressed in our work. We assume a reliable system, in which no site failures or communication network failures occur.

5 Lock-Based Concurrency Control

Each local scheduler can process lock requests without requiring any information from other sites. 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 (2PL) [5]. In order to provide global serializability, the locks held by the cohorts of a transaction are maintained until the transaction has been committed.

Deadlock is a possibility with blocking-based concurrency control protocols. Local deadlocks can be detected by maintaining a local Wait-For Graph (WFG) at each site. WFGs contain the 'wait-for' relationships among the transactions. Local deadlock detection is performed 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 lock managers [3], [4]. A simple way to detect global deadlocks is to use a global WFG. The global WFG is constructed by unifying local WFGs. 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 and later restart the whole transaction.

5.1 Distributed Real-Time Concurrency Control Protocols

We distinguish the locking protocols based on whether they make use of the data access patterns of transactions or not. The first group of protocols assume that the data requirements of each transaction are not known before the execution of the transaction, while the second group of protocols assume that a list

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

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

```
lock_request_handling(D,C) {
  /* Cohort C requests a lock on data item D */
  if (D was locked by a cohort C') {
    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 1: Lock request handling in protocol PI.

```
lock_request_handling(D,C) {
  /* Cohort C requests a lock on data item D */
  if (D was locked by a cohort C') {
    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 2: Lock request handling in protocol PB.

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 involve priorities in processing the data access requests[3]. Performance of basic two-phase locking provides a basis of comparison for studying the performance of the priority-based protocols.

Always Block Protocol (AB)

This protocol resolves lock conflicts by blocking a cohort that requests a lock that is already held. 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)

An undesirable event in scheduling prioritized transactions is the so called 'priority inversion' problem; i.e., blocking of a high priority transaction by lower priority transactions. 'Priority inheritance' is one method proposed to overcome the problem of uncontrolled priority inversion [11]. This method makes sure 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.

Fig.1 shows how the lock requests are handled by the 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. 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.

Two or more sibling cohorts may inherit different priorities at about the same time, causing an inconsistency in the inherited priority assigned to the sibling cohorts. To prevent this inconsistency, when a cohort receives an inheritance message from its master it compares its current priority with the inherited priority. If the inherited priority has a smaller value, the message is ignored. Thus, all of the sibling cohorts will eventually carry the same priority, the maximum of all inherited priorities.

When a transaction is aborted due to a local or global deadlock, it is restarted with its original priority.

Priority-Based Locking Protocol (PB)

This protocol prevents priority inversion by aborting low priority transactions whenever necessary [1]. We have implemented a distributed version of the protocol in our model as summarized in Fig.2. 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. 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[4].

[3]However, as described earlier, the preemptive-resume strategy based on real-time priorities is used in scheduling the CPU.

[4]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.

```
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 3: Lock request handling in protocol PC.

Protocols with Known Data Requirements

Some real-time systems process certain kinds of transactions that can be characterized with well defined data requirements [6]. 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 real-time transactions.

Priority Ceiling Protocol (PC)

This protocol, proposed by Sha et al. [11] 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.

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 the transaction priorities; the highest priority transaction determines the priority ceiling of D. At each data site S, a priority ceiling manager 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, Fig.3 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 priority ceiling manager maintains a sorted list of (priority ceiling, lock-holding transaction) pairs of all the locked data items at that site.

Data-Priority-Based Locking Protocol (DP)

This section presents the lock-based concurrency control protocol we introduced in [15]. Like protocol PC, protocol DP is based on prioritizing data items; however, in ordering the access requests of the 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.

The implementation details of the protocol in our system are as follows. A transaction list is maintained for each individual data item. 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 the 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 the commit of relevant transactions.

Fig.4 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[5] 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. An abort is handled by the master process in a way similar to that used in protocol PB.

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

6 Performance Evaluation

The performance of the concurrency control protocols were evaluated by simulating them on the distributed system model described in the previous sections. The set of parameters presented in Table 1 is used to specify the system configuration and workload. *nr-sites* corresponds to the number of data sites in the distributed system. The parameter *db-size* represents the number of data items stored in the

[5]Remember that, a cohort process carries the priority of its 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 4: Lock request handling in protocol DP.

nr-sites	5	*slack-rate*	5
db-size	200	*mes-proc-time* (msec)	2
mem-size	50	*comm-delay* (msec)	5
iat (msec)	260	*list-update-oh* (msec)	1
tr-type-prob	.5	*conf-check-oh* (msec)	1
access-mean	6	*lock-oh* (msec)	1
data-update-prob	.5	*unlock-oh* (msec)	1
cpu-time (msec)	8	*deadl-det-oh* (msec)	1
io-time (msec)	28	*deadl-res-oh* (msec)	1

Table 1: System model parameter values

database of a site, and *mem-size* represents the number of data items that can be held in the main memory of a site. Transaction arrivals at a data site are simulated by a transaction generator belonging to that site. The mean interarrival time of transactions to each of the sites is specified 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. The access pattern (i.e. the list of data items that are going to be accessed) of each transaction is also determined by the transaction generator. *access-mean* specifies the mean number of data items to be accessed by a transaction. Accesses are uniformly distributed among the data sites. For each data access of an update transaction, the probability that the accessed data item will be updated is determined by the parameter *data-update-prob*. For each data item accessed, IO and CPU resources are utilized for a certain amount of time, specified by *io-time* and *cpu-time*, respectively. *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 parameter *mes-proc-time* corresponds to the CPU time required to process a communication message prior to sending or after receiving the message. It is assumed that each site has one CPU and one disk. The communication delay of messages between the sites is assumed to be constant and specified by the parameter *comm-delay*. The parameter *list-update-oh* specifies the processing cost of add and delete operations on various kind of graphs/lists used in the model, which include the WFGs maintained for deadlock detection, the priority ceiling list of locked data items in protocol PC, and the transaction lists kept for each data item in protocols PC and DP.

The following parameters are included in our model to take into account the overhead of various concurrency control operations. *conf-check-oh* specifies the CPU time of conflict checking at each data access request of a transaction. The parameter *lock-oh* is used to simulate the CPU time required to request and get a lock on a data item, while *unlock-oh* is used to simulate the processing time to release the lock on a data item. *deadl-det-oh* corresponds to the processing time spent checking for a deadlock cycle in a wait-for graph, and *deadl-res-oh* specifies the cost of clearing the deadlock problem in the case of the detection of a deadlock.

6.1 Simulation Results

The system parameter values for the experiments described in the following sections are presented in Table 1. All sites of 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 observe the differences between the real-time performances of the protocols. Since the concurrency control protocols are different in handling data access conflicts among the transactions, the best way to compare the performance characteristics of the protocols can be to conduct the performance 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 the 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. The method used in calculating expected CPU and IO utilizations in terms of system parameters is provided in [16]. The performance metric used in the evaluation of the protocols is *success-ratio*; i.e., the fraction of transactions that satisfy their deadlines.

The experiments employ 'Earliest Deadline First' as the priority assignment policy; i.e., a transaction with earlier deadline obtains higher priority. If any two of the transactions have the same deadline, the one that has arrived at the system earlier is assigned a higher priority. The transaction deadlines are 'soft'; i.e., each transaction is executed to completion even if it misses its deadline.

A program to simulate our distributed system model and concurrency control protocols was written in CSIM [10], which is a process-oriented simulation language based on the C programming language. For each configuration of each experiment, the final results

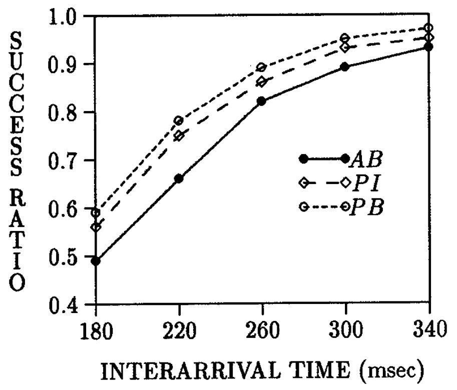

Figure 5: Real-time performance of the distributed concurrency control protocols AB, PI, and PB as a function of mean transaction interarrival time.

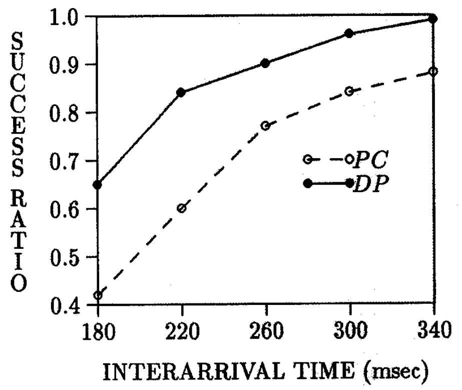

Figure 6: Real-time performance of the distributed concurrency control protocols PC and DP as a function of mean transaction interarrival time.

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 a CPU utilization of .96 to .51 respectively, at each of the sites [16]. IO utilization is about the same as CPU utilization. We are interested in the performance of the protocols under high transaction loads.

The first group of protocols evaluated are the ones with unknown data requirements (AB, PI, and PB). The results are shown in Fig.5 in terms of *success-ratio*. Protocols PI and PB 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, we can say that PB provides considerably better performance than PI throughout the *iat* range. Remember that, protocol PB resolves data conflicts by aborting low priority transactions whenever necessary. Although transaction aborts lead to a waste of resources in the system, this experiment shows that aborting a low priority transaction is preferable to blocking a high priority one in distributed RTDBSs. Protocol PB also prevents the possibility and cost of transaction deadlocks.

We next evaluated the performance of the concurrency control protocols with known data requirements. 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 Fig.6 that the deadline satisfaction rate of the 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.[6] 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' performances increases as the load level increases with decreasing *iat* value. Similar to PB, 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 PB 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

[6]The implementation of protocol PC in our model followed this assumption unlike the other protocols whose implementation included the CPU scheduling method described in Section 4.

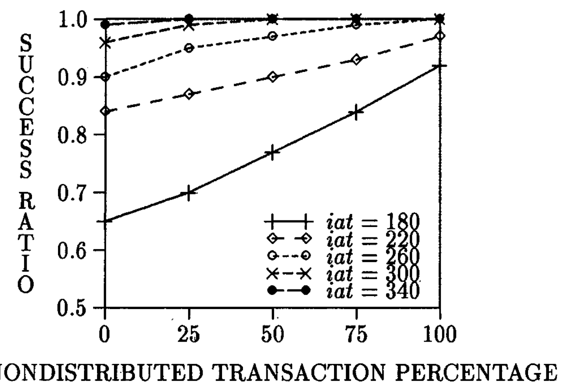

Figure 7: Real-time performance of protocol DP (for different values of mean transaction interarrival time) as a function of the percentage of nondistributed transactions submitted to the system.

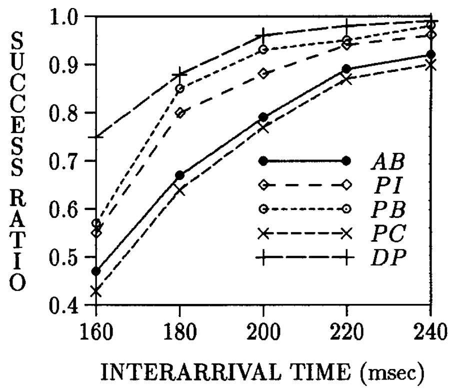

Figure 8: Real-time performance of the concurrency control protocols under the parallel transaction execution model.

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.

Other experiments were conducted to evaluate the effect of various other system parameters on the protocols' performance. These parameters included *nr-sites* (number of data sites), *mes-proc-time* (communication message processing time), *comm-delay* (message transfer delay), *slack-rate* (ratio of the slack time of a transaction to its execution time), *access-mean* (average number of data items accessed by a transaction), *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 [16]. 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 the preceding section, 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 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 Fig.7, 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 240 mseconds in steps of 20. This *iat* range [160,240] corresponds to the IO/CPU utilization range of about [.94, .63]. Expected resource utilization calculations in the sequential and parallel execution models are different because the overhead is different [16].

Fig.8 shows the performance results obtained for each of the protocols under the parallel execution model. 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.

The performance of the protocols is better, in general, than it was for the sequential execution model. This result is due to fewer conflicts. The decrease in the conflict rate results from shorter lifetimes of the transactions.

7 Conclusions

In this paper, we have presented a number of lock-based, distributed, real-time concurrency control protocols and studied their performance in a distributed database system environment. Various experiments were conducted to evaluate the effects of different system parameters on the performance of the protocols.

Among the concurrency control protocols which detect the data access requirements of transactions dynamically, the best-performing protocol was PB, which always prefers high priority transactions to execute in the case of data conflicts while blocking or aborting lower priority transactions. Introducing a data-priority-based protocol (DP), we have shown that further improvement in real-time performance is possible if the data access requirements of transactions are known prior to their execution. Another observation obtained in the evaluations was the poor performance exhibited by priority-ceiling protocol PC. 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 RTDBSs.

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 fraction of transactions which submit operation to remote sites. All the protocols performed worse as the ratio of distributed/locally-processed transactions increased.

References

[1] Abbott R., Garcia-Molina H. 'Scheduling Real-Time Transactions: A Performance Evaluation', *14th Int. Conf. on Very Large Data Bases*, 1988, pp.1-12.

[2] Abbott R., Garcia-Molina H. 'Scheduling Real-Time Transactions with Disk Resident Data', *15th Int. Conf. on Very Large Data Bases*, 1989, pp.385-396.

[3] Bernstein P.A., Hadzilacos V., Goodman N. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[4] Ceri S., Pelagatti G. *Distributed Databases: Principles and Systems*, McGraw-Hill, 1984.

[5] Eswaren K.P., Gray J.N. 'The Notion of Consistency and Recovery in a Database System', *Comm. of ACM*, 19:11, 1976, pp.624-633.

[6] Haritsa J.R., Carey M.J., Livny M. 'On Being Optimistic About Real-Time Constraints', *ACM SIGACT-SIGMOD-SIGART*, 1990, pp.331-343.

[7] Haritsa J.R., Carey M.J., Livny M. 'Dynamic Real-Time Optimistic Concurrency Control', *11th Real-Time Systems Symposium*, 1990, pp.94-103.

[8] Huang J., Stankovic J.A., Towsley D., Ramamritham K. 'Experimental Evaluation of Real-Time Transaction Processing', *10th Real-Time Systems Symposium*, 1989, pp.144-153.

[9] Huang J., Stankovic J.A., Towsley D., Ramamritham K. 'Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes', *17th Int. Conf. on Very Large Data Bases*, 1991, pp.35-46.

[10] Schwetman H. 'CSIM: A C-Based, Process-Oriented Simulation Language', *Winter Simulation Conference*, 1986, pp.387-396.

[11] Sha L., Rajkumar R., Lehoczky J. 'Concurrency Control for Distributed Real-Time Databases', *ACM SIGMOD Record*, March 1988, pp.82-98.

[12] Sha L., Rajkumar R., Lehoczky J. 'Priority Inheritance Protocols: An Approach to Real-Time Synchronization', *IEEE Transactions on Computers*, 39:9, 1990, pp.1175-1185.

[13] Sha L., Rajkumar R., Son S.H., Chang C.H. 'A Real-Time Locking Protocol', *IEEE Transactions on Computers*, 40:7, 1991, pp.793-800.

[14] Son S.H., Chang C.H. 'Performance Evaluation of Real-Time Locking Protocols Using a Distributed Software Prototyping Environment', *10th Int. Conf. on Distributed Computing Systems*, 1990, pp.124-131.

[15] Ulusoy Ö., Belford G.G. 'Concurrency Control in Real-Time Database Systems', *ACM 20th Annual Computer Science Conf.*, March 1992, pp.181-188.

[16] Ulusoy Ö. *Scheduling Real-Time Database Transactions*, Ph.D. Thesis in preparation, Department of Computer Science, University of Illinois at Urbana-Champaign.