Pergamon

*Information Systems* Vol. 18, No. 8, pp. 559–580, 1993

0306-4379/94 $6.00 + 0.00

# REAL-TIME TRANSACTION SCHEDULING IN DATABASE SYSTEMS

ÖZGÜR ULUSOY[1] and GENEVA G. BELFORD[2]

[1]Department of Computer Engineering, Bilkent University, Bilkent, Ankara 06533, Turkey
[2]Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, U.S.A.

*(Received 10 February 1993; in revised form 15 July 1993)*

**Abstract**—A database system supporting a real-time application, which can be called "a real-time database system (RTDBS)", has to provide real-time information to the executing transactions. Each RTDB transaction is associated with a timing constraint, usually in the form of a deadline. Efficient resource scheduling algorithms and concurrency control protocols are required to schedule the transactions so as to satisfy both timing constraints and data consistency requirements. In this paper,† we concentrate on the concurrency control problem in RTDBSs. Our work has two basic goals: real-time performance evaluation of existing concurrency control approaches in RTDBSs, and proposing new concurrency control protocols with improved performance. One of the new protocols is locking-based, and it prevents the priority inversion problem‡ by scheduling the data lock requests based on prioritizing data items. The second new protocol extends the basic timestamp-ordering method by involving real-time priorities of transactions in the timestamp assignment procedure. Performance of the protocols is evaluated through simulations by using a detailed model of a single-site RTDBS. The relative performance of the protocols is examined as a function of transaction load, data contention (which is determined by a number of system parameters) and resource contention. The protocols are also tested under various real-time transaction processing environments. The performance of the proposed protocols appears to be good, especially under conditions of high transaction load and high data contention.

*Key words:* Real-time database systems, transaction scheduling, concurrency control, performance evaluation

## 1. INTRODUCTION

A real-time database system (RTDBS) can be defined as a database system which is designed to provide real-time response to the transactions of data-intensive applications, such as computer-integrated manufacturing, the stock market, banking and command and control systems. Similar to a conventional real-time system, the transactions processed in a RTDBS have timing constraints, usually in the form of deadlines. What makes a RTDBS different from a conventional real-time system is the requirement of preserving the consistency of data besides considering the real-time requirements of the transactions. This makes it difficult to schedule RTDB transactions so as to satisfy the timing constraints of all transactions processed in this system. Concurrency control protocols proposed so far to preserve data consistency in database systems are all based on transaction blocking and transaction restart, which makes it virtually impossible to predict computation times and hence to provide schedules that guarantee deadlines. We believe that one of the most important issues in RTDB transaction scheduling is the need for the development of efficient concurrency control protocols that process as many transactions as possible within their deadlines.

The goal of our work presented in this paper is twofold: real-time performance evaluation of existing concurrency control approaches in RTDBSs, and proposing new concurrency control protocols with improved performance. The first of the new protocols is locking-based, and it prevents the priority inversion problem by scheduling the data lock requests of the transactions based on prioritizing data items. The second protocol extends the basic timestamp-ordering method by involving real-time priorities of transactions in the timestamp assignment procedure.

†An earlier version of this paper was published in the *Proceedings of ACM Computer Science Conference '92.*
‡Priority inversion is uncontrolled blocking of high priority transactions by lower priority transactions [1].

A detailed performance model of a single-site RTDBS was employed in the evaluation of selected, representative concurrency control protocols.† Each protocol studied belongs to one of the three basic classes of concurrency control; i.e. locking, timestamp-ordering, or optimistic. Various simulation experiments were conducted to study the relative performance of the protocols under many possible real-time and database environments. The performance metrics used were *success-ratio*; i.e. the fraction of transactions that satisfy their deadlines, and *average-lateness*; i.e. the average lateness of the transactions that missed their deadlines.

Among the locking protocols chosen from previous work are the Priority Inheritance protocol (PI) which allows a low-priority transaction to execute at the highest priority of all the higher priority transactions it blocks [1]; the Priority Ceiling protocol (PC) which extends PI by eliminating deadlocks and bounding the blocking time of high priority transactions to no more than one transaction execution time [1, 3]; and the Priority-Based conflict resolution protocol (PB) which aborts a low priority transaction when one of its locks is requested by a higher priority transaction [4]. A real-time optimistic concurrency control protocol described in Ref. [5] is also included in the set of protocols selected for performance evaluation.

The results of our experiments indicate that locking protocols PI and PB both provide an improvement in the performance obtained by the basic two-phase locking protocol. Between these two protocols, PB performs better, in general, than PI does under various conditions tested in simulations. Our basic conclusion is that aborting a low priority transaction can be preferable in RTDBSs to blocking a high-priority one, although aborts lead to a waste of resources in the system. One interesting result obtained is the relatively poor performance exhibited by protocol PC. Although the protocol was intended to be an improvement over protocol PI, it performs even worse than the basic two-phase locking protocol, which does not involve real-time priorities of the transactions in data access scheduling decisions. The pessimistic priority-ceiling condition, which blocks transactions even if no data conflict exists among them, is the major drawback of this protocol. The performance evaluation section of the paper will provide a detailed interpretation of all the results obtained.

Experiments with our new Data-Priority-based locking protocol (DP) show that the real-time performance provided by protocol PB, which appeared to be the best locking protocol among the ones tested, can be further improved if the data access requirements of transactions are known in advance. The protocol is based on prioritizing data items; each data item carries a priority equal to the highest priority of all transactions currently in the system that include the data item in their access lists. In order to obtain a lock on a data item $D$, the priority of a transaction $T$ must be equal to the priority of $D$. Otherwise (if the priority of $T$ is less than that of $D$), transaction $T$ is blocked by the transaction that is responsible for the priority of $D$. Suppose that $T$ has the same priority as $D$, but $D$ has already been locked by a lower priority transaction $T'$ before $T$ arrives at the system and adjusts the priority of $D$. $T'$ is aborted at the time $T$ needs to lock $D$.

Some of the transaction aborts, and the resulting resource waste, experienced in protocol PB can be prevented by employing the new protocol DP. Consider the following scenario: suppose that two transactions $T_x$ and $T_y$, both in the system, have conflicting accesses on item $D$, and transaction $T_x$ has higher priority. Under protocol DP, if $T_y$ tries to lock data item $D$ before $T_x$ does, the lock request of $T_y$ is not accepted. Under protocol PB, $T_y$ would get the lock on $D$, but would be aborted when the higher priority transaction $T_x$ requests $D$. As a result, the processing time spent by $T_y$ would simply be wasted. This wasted time might have been used to help another transaction meet its deadline.

Our expectations about the performance of proposed protocol DP are confirmed by the performance experiment results. It appears to perform better than other locking protocols.

The priority inversion problem that was defined for locking protocols can also exist in a RTDBS that provides data consistency through a timestamp-ordering concurrency control protocol. It is possible that a high priority transaction $T$ is aborted at its access to a data item, since a lower priority transaction $T'$ carrying a timestamp higher than the timestamp of $T$, has accessed that data item previously. This problem can be called *priority inversion for timestamp-ordering protocols.*

†The performance evaluation of various locking-based concurrency control protocols in a "distributed RTDBS" is provided in Ref. [2].

We propose a new timestamp-ordering protocol that attempts to control this priority inversion. Our Priority-based Timestamp-Ordering protocol (PTO) provides a performance improvement over the basic timestamp-ordering protocol by making use of transaction priorities in assigning timestamps to transactions. It categorizes the transactions into timestamp groups and the transactions of the same timestamp group are scheduled based on their real-time priorities. The performance level achieved by protocol PTO is not as high as that obtained with locking protocol DP. The optimistic protocol performs very well, compared to other protocols, when the system is lightly loaded and the data contention level among the transactions is low.

The remainder of the paper is organized as follows. The next section summarizes recent work on the transaction scheduling problem in RTDBSs. Section 3 describes the real-time concurrency control protocols studied. Section 4 provides the structure and characteristics of our simulation model. The real-time performance experiments and results are presented in Section 5. Finally, Section 6 summarizes the basic conclusions of the performance evaluation work.

## 2. RELATED WORK

The transaction scheduling problem in RTDBSs has been addressed by a number of recent studies. Abbott and Garcia-Molina described and evaluated through simulation a group of real-time scheduling policies based on enforcing data consistency by using a two-phase locking concurrency control mechanism [4, 6, 7]. Two of the concurrency control protocols evaluated were priority-based locking (PB) and priority inheritance (PI). In Ref. [8], they provided a study of various algorithms for scheduling disk request with deadlines. The priority ceiling protocol (PC) was presented in Refs [1] and [3], and the performance of the protocol was examined in Ref. [9] by using simulations. Huang *et al.* developed and evaluated several real-time policies for handling CPU scheduling, concurrency control, deadlock resolution, transaction wakeup and transaction restart in RTDBSs [10]. Different versions of the priority-based locking (PB) policy were employed in concurrency control. Each version was different in the parameters used in determining transaction priorities. These parameters included deadline, criticalness† and remaining execution time. Later, their work was extended to the optimistic concurrency control method [12]. In Ref. [13], they compared the concurrency control protocols PB and PI, for addressing the priority inversion problem in RTDBSs, and proposed a new scheme to capitalize on the advantages of both PB and PI. Haritsa *et al.* studied, by simulation, the relative performance of two well known classes of concurrency control algorithms (locking protocols and optimistic techniques) in a RTDBS environment [14, 15]. They presented and evaluated a new real-time optimistic concurrency control protocol through simulations in Ref. [5]. Son and Chang investigated methods to apply the priority-ceiling protocol PC as a basis for real-time locking protocol in a distributed environment [16]. In Ref. [17], a new locking concurrency control protocol for RTDBSs was proposed allowing the serialization order among transactions to be adjusted dynamically in compliance with transaction timeliness and criticalness.

In presenting our findings, the basic results of each of the performance evaluation studies listed above will be provided, together with a comparison, if possible, to our results.

## 3. CONCURRENCY CONTROL PROTOCOLS

### *3.1. Locking protocols*

All locking protocols studied were assumed to have the following characteristics. Each transaction has to obtain a shared lock on each data item it reads, and an exclusive lock on each data item it writes. Two lock requests conflict if they operate on the same data item and at least one of them is an exclusive lock. A conflicting lock request may result in blocking of the requesting transaction. For the detection of a possible blocking deadlock, a wait-for graph [18] is maintained. Deadlock detection is performed each time a transaction is blocked. If a deadlock is detected, the

†Criticalness represents the importance of a real-time transaction [11]. We assume in our work that all the transactions processed in the system are equally important (i.e. the criticalness issue is not considered), and the deadline of a transaction is the only factor in determining the real-time priority of the transaction.

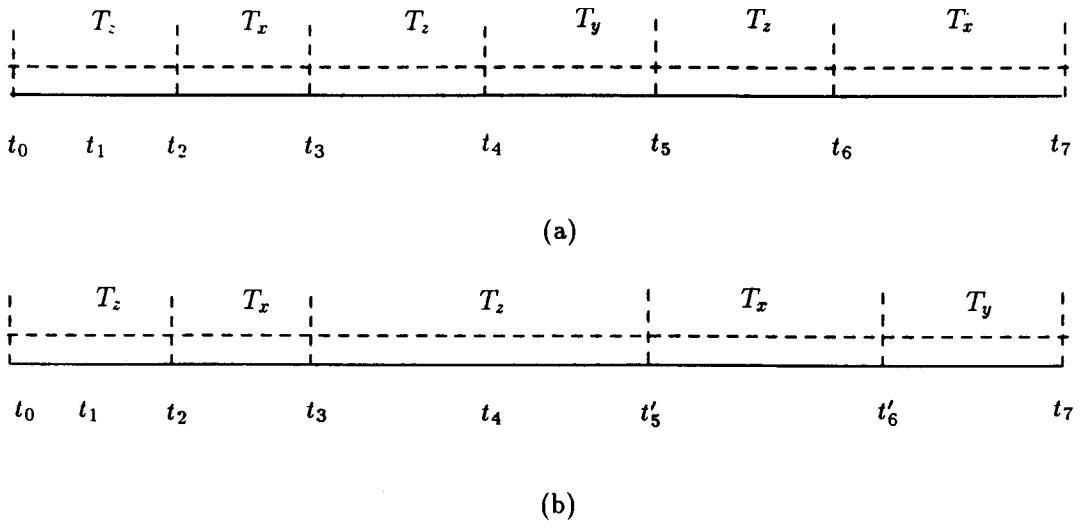


Fig. 1. Example 1: schedules produced by protocol AB and protocol PI, respectively.

lowest-priority transaction in the deadlock cycle is chosen as the victim to be aborted. Since some of the following protocols are deadlock-free, we don't have to deal with the deadlock problem for them. The locks obtained by a transaction are released all together when the transaction completes its execution. Deferred updates of written data items are performed before releasing the locks.

The first three of the locking protocols described in the following sections (i.e. AB, PI and PB) assume that the data requirements of a transaction are not known prior to the execution of the transaction, while the last two protocols (i.e. PC and DP) assume that a list of data items to be accessed is submitted by each arriving transaction. It is possible to have an advance knowledge of data access patterns in RTDBSs that process certain kinds of transactions that can be characterized by well defined data requirements [4, 14]. One such application area is the stock market. A typical transaction accesses some prespecified data items to read the current bid and prices of particular stocks. Another transaction may want to update specific data items to reflect the changes in stock prices. Banking is another application area in which the access requirements of transactions can be well defined as each transaction is typically characterized by a user account number in processing the account information of that particular user.

**Always block protocol (AB)**—This protocol is based on the strict two-phase locking scheme [18]. When a lock request results in a conflict, the transaction requesting the lock is always blocked by the transaction holding the conflicting lock. The protocol does not take the transaction priorities into account in scheduling the lock requests, thus it serves as a control against which to measure the performance gains of locking protocols that involve priorities in data access scheduling decisions.

For all the protocols presented in this paper, including protocol AB, CPU requests of the transactions are scheduled based on the real-time priorities. A high-priority transaction requiring CPU can preempt a lower priority, executing transaction.†

**Priority inheritance protocol (PI)**—Sha *et al.* proposed a scheme called "priority inheritance" to overcome the problem of priority inversion [1]. The basic idea of priority inheritance is that when a transaction blocks one or more higher priority transactions, it is executed at the highest priority of all the transactions it blocks. This idea is illustrated by the following example.

**Example 1**—Suppose that we have three transactions $T_x$, $T_y$, and $T_z$ with priorities $P(T_x)$, $P(T_y)$, $P(T_z)$, with $P(T_x) > P(T_y) > P(T_z)$. Assume that transaction $T_x$ and transaction $T_z$ both access data item $D$.

For the following sequence of events, the schedule produced by protocol AB is shown in Fig. 1a. At time $t_0$, transaction $T_z$ starts its execution and at $t_1$ it obtains a lock on data item $D$. At time $t_2$, transaction $T_x$ arrives and preempts $T_z$. It then tries to access data item $D$, but it is blocked

†The CPU scheduling method implemented in our simulations is detailed in Section 4.

by $T_z$ since $T_z$ has a conflicting lock on $D$. $T_z$ continues its execution until $T_y$ preempts it at time $t_4$. $T_y$ finishes at $t_5$ and $T_z$ resumes its processing until it commits at $t_6$. Since the lock on $D$ has been released, $T_x$ can now complete.

Figure 1b shows the schedule produced under protocol PI for the same sequence of events. When the high priority transaction $T_x$ is blocked by the lower priority transaction $T_z$, the priority of $T_x$ is inherited by $T_z$. At time $t_4$ the new transaction $T_y$ cannot preempt $T_z$ since its priority is not higher than the priority inherited by $T_z$. When $T_z$ finishes its execution, $T_x$ can continue. The blocking time of the high-priority transaction $T_x$ has been reduced by protocol PI.

**Priority-based locking protocol (PB)**—In this protocol, the winner in the case of a conflict is always the higher priority transaction [4]. In resolving a lock conflict, if the transaction requesting the lock has higher priority than the transaction that holds the lock, the latter transaction is aborted and the lock is granted to the former one. Otherwise, the lock-requesting transaction is blocked by the higher priority lock-holding transaction.

A high-priority transaction never waits for a lower priority transaction. This condition prevents deadlocks if we assume that the real-time priority of a transaction does not change during its lifetime and that no two transactions have the same priority. Thus overhead due to detection and resolution of deadlocks is prevented by this protocol.

The following example presents the difference between protocols AB and PB.

**Example 2**—Suppose that transactions $T_x$ and $T_y$ both access data item $D$, and $T_y$ has higher priority than $T_x$. The schedule produced by protocol AB for the following sequence of events is shown in Fig. 2a.

At time $t_0$, transaction $T_x$ arrives and starts its execution. At time $t_1$, it obtains a lock on data item $D$. Later transaction $T_y$ arrives and preempts $T_x$. At time $t_3$ $T_y$ attempts to obtain a lock on $D$, but it is blocked by $T_x$, which has already locked $D$. $T_x$ resumes its execution, and at time $t_4$ it releases all the locks it has obtained and commits. Transaction $T_y$ can now lock data item $D$ and continue processing. At time $t_5$ $T_y$ commits.

Figure 2b shows the schedule produced by protocol PB. In this case, at time $t_3$ transaction $T_x$ is aborted and transaction $T_y$ obtains the lock it requires, since it has higher priority than $T_x$. At time $t'_4$ $T_y$ commits and $T_x$ is restarted. Compared to the previous schedule, the high-priority transaction $T_y$ has an earlier commit time.

**Priority ceiling protocol (PC)**—The priority ceiling method proposed by Sha *et al.* is an extension to the priority inheritance protocol (PI) [1, 3]. It eliminates the deadlock problem from PI and attempts to reduce the blocking delays of high-priority transactions. 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. In order 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 the transaction $T$ is blocked by the transaction which holds the lock on the data item of the highest priority ceiling.

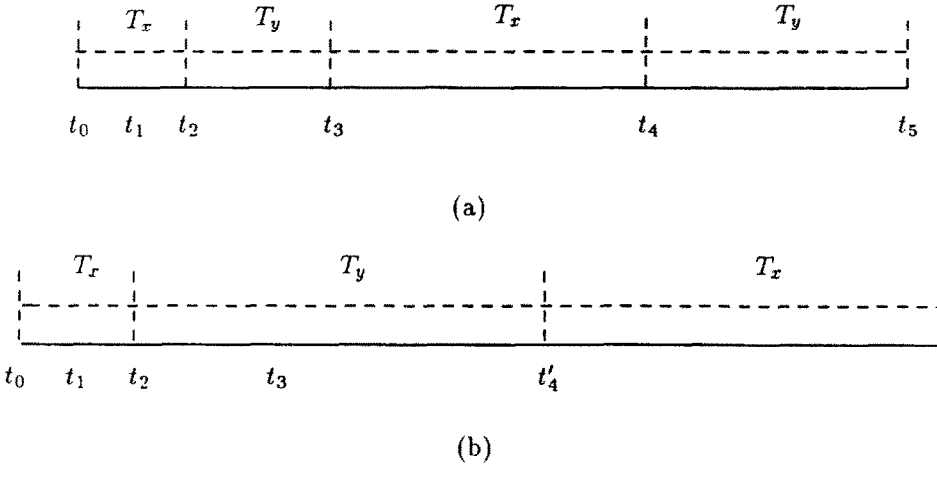


Fig. 2. Example 2: schedules produced by protocol AB and protocol PB, respectively.

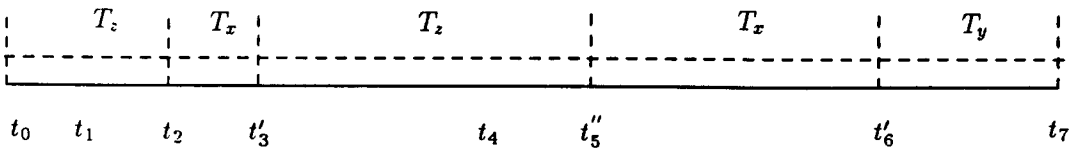


Fig. 3. Example 3: schedule produced by protocol PC.

**Example 3**—Consider again Example 1 with protocol PI. Suppose that transactions $T_x$ and $T_z$ have another common item $D'$ in their access lists. Assume that $T_x$ obtains a lock on data item $D'$ within time interval $[t_2, t_3]$. At time $t_3$ $T_x$ starts waiting for $T_z$ to release the lock on $D$. Later, $T_z$ will attempt to lock data item $D'$ and be blocked by $T_x$. Deadlock!

The schedule produced by protocol PC for the same sequence of events is shown in Fig. 3. The priority ceilings of data items $D$ and $D'$ are equal to the priority of $T_x$, assuming that $T_x$ is the highest priority transaction that can lock these items. At time $t_1$ $T_z$ obtains a lock on $D$; later $T_x$ preempts $T_z$ and attempts to lock $D'$ at time $t'_3$. Since its priority is not strictly higher than the priority ceiling of the locked data item $D$, the lock request is rejected. $T_z$ now inherits the priority of $T_x$ and resumes execution. A potential deadlock between $T_x$ and $T_z$ is prevented.

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 access item $D$. The list is sorted based on the transaction priorities; the highest priority transaction determines the priority ceiling of $D$. The scheduler is responsible for keeping track of the current highest priority ceiling value of the locked data items and the id of the transaction holding the lock on the item with the highest priority ceiling. To determine the current values of these two variables, the scheduler maintains a sorted list of (priority ceiling, lock-holding transaction) pairs of all the locked data items in the system.

In order to include read/write semantics in the protocol, the read/write priority ceiling rule introduced in Ref. [9] is used. The read/write priority ceiling of a read-locked data item is set to the priority of the highest priority transaction 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 transaction which will read from or write into that item. In this case, the transaction list maintained for each data item includes the type of data access (i.e. read or write) as well as transaction ids and priorities. In order to obtain a read or write lock on any data item, the protocol requires that a transaction $T$ must have a priority strictly higher than the highest read/write priority ceiling of data items locked by the transactions other than $T$.

There are several reasons that make protocol PC impractical in RTDBSs. One of those reasons is the pessimistic nature of the priority ceiling blocking rule. Even if no data conflict exists among the concurrent transactions, most of the access requesting transactions are blocked to avoid deadlock and chained blocking. Scheduling the accesses on an individual data item cannot be performed independently from the accesses on other data items, since the scheduler requires priority information from all locked data items in the system. Another drawback of the protocol is the CPU-scheduling policy adopted, which assumes 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.† 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.

**Data-priority-based locking protocol (DP)**—In this section, we propose a new deadlock-free locking protocol based on prioritizing data items. Each data item carries a priority which is equal to the highest priority of all transactions currently 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. The protocol assumes that a list of data items that are going to be accessed is submitted to the scheduler by the arriving transaction. When a transaction terminates, each data item that carries the priority of that

†The implementation of protocol PC in our simulations followed this assumption, while implementation of the other protocols included the CPU scheduling method described in Section 4.

```
if priority(T) = priority(D)
  if D was locked by a transaction T'
    T' is aborted;
  Lock on D is granted to T;
else
  T is blocked by the transaction that determines
  the current priority of D;
```

Fig. 4. Lock request handling in protocol DP.

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.

A transaction list is maintained for each individual data item. The list contains the id and priority of the active transactions requiring access to 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/abort of relevant transactions.

The procedure presented in Fig. 4 describes how the lock requests of transactions are handled by protocol DP. Assume that transaction $T$ is requesting a lock on data item $D$. In order to obtain the lock, the priority of transaction $T$ must be equal to the priority of $D$; in other words, it should be the transaction responsible for the current priority of $D$. Otherwise (if the priority of $T$ is less than that of $D$), transaction $T$ is blocked.

A data item $D$ may have been locked by a transaction $T'$ when a new transaction $T$, containing the item $D$ in its access list and carrying a priority higher than the priority of $D$, arrives at the system and updates the priority of $D$. If $D$ is still locked by $T'$ at the time $T$ needs to access it, the lower priority transaction $T'$ is aborted and $T$ obtains its lock on $D$. The assumption that each transaction has a unique priority makes DP deadlock-free, since a high-priority transaction is never blocked by lower priority transactions.

**Example 4**—The deadlock situation in Example 3 can also be handled by protocol DP. However, in this case the lock requests of transaction $T_x$ on data items $D$ and $D'$ are granted since the priority of $T_x$ is equal to the priority of data items $D$ and $D'$. Figure 5 presents the schedule produced by protocol DP. Obtaining a lock on $D$ (at time $t_3$) results in aborting transaction $T_z$ that has locked $D$ before. Comparing Fig. 5 with Fig. 3, we see that DP (because of the abort) delays completion of $T_z$, but completes the higher priority transactions earlier.

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 transaction $T$ must be larger than or equal to the write priority of $D$. A write lock request of transaction $T$ on data item $D$ is honored if $T$ has a priority equal to the write priority of $D$, and larger than or equal to the read priority of $D$. The procedures that handle read and write requests are detailed in Fig. 6.

Compared to protocol PC, DP is a more practical policy to implement in RTDBSs. The major drawbacks of PC, discussed in the preceding section, are eliminated by our new protocol. In ordering the access requests of the transactions on a data item, only the priority of that data item is considered; the scheduling decisions are independent of the priorities of other data items. Moreover, only in the case of a data conflict is a transaction blocked at its data access request.

### *3.2. Timestamp-ordering protocols*

**Basic timestamp-ordering protocol (TO)**—Protocol TO is the basic version of the timestamp-ordering concurrency control protocol [18]. Each transaction is assigned a timestamp when it is

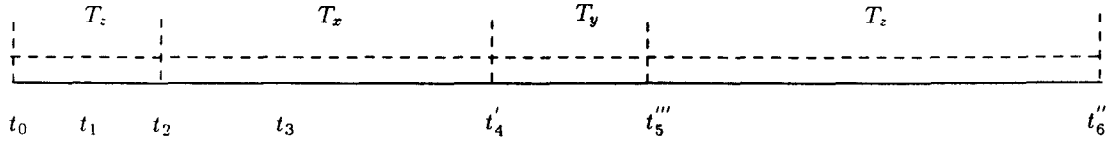


Fig. 5. Example 4: schedule produced by protocol DP for the sequence of events in Example 3.

*Handling READ LOCK requests*

if priority($T$) $\geq$ write-priority($D$)
  if $D$ was *write locked* by a transaction $T'$
    $T'$ is aborted;
  *Read lock* on $D$ is granted to $T$;
else
  $T$ is blocked by the transaction that has assigned
  the *write priority* of $D$;

*Handling WRITE LOCK requests*

if (priority($T$) = write-priority($D$) and priority($T$) $\geq$ read-priority($D$))
  if $D$ was *read or write locked* by any transaction $T'$
    $T'$ is aborted;
  *Write lock* on $D$ is granted to $T$;
else
  $T$ is blocked by the transaction that has assigned
  the *maximum of read and write priorities* of $D$;

Fig. 6. Extending protocol DP with read/write locks.

submitted to the system. Conflicting operations are ordered based on transaction timestamps. A read request for a data item is honored if no other transaction with a larger timestamp has updated that data item. A write attempt is satisfied if no other transaction with a larger timestamp has read or written that item. Maximum timestamps of reads and writes are maintained for each data item. If a read/write request of a transaction is not accepted, the transaction is restarted with a larger timestamp.

As discussed in Ref. [18], if each granted write access has an immediate effect on the database, although serializability is enforced by TO rules, recoverability is not ensured. Also, the schedules produced may lead to cascading aborts of transactions. To provide strictly serializable executions (i.e. to guarantee recoverability and avoid cascading aborts), granted writes of a transaction are deferred until the commit time of the transaction. Read requests on data items are not granted until the updates on those items become visible. Effectively, a granted write on a data item acts like a lock since subsequent access requests with later timestamps on the same data item do not actually take place until the updating transaction commits. However, that does not mean that strict TO and two-phase locking protocols produce the same schedules for a given sequence of transaction operation.†

**Priority-based timestamp-ordering protocol (PTO)**—In the basic timestamp-ordering protocol, scheduling decisions for conflicting operations are all based on the timestamp values assigned to transactions at startup time. Each transaction is assigned a timestamp based on its submission (or resubmission) time to the system. In a prioritized transaction execution environment, a high-priority transaction $T$ may be aborted at its access to a data item, when a lower priority transaction $T'$, carrying a timestamp higher than the timestamp of $T$, has accessed that data item previously. This problem can be called *priority inversion for timestamp-ordering protocols.*

We propose a new protocol that attempts to control the priority inversion problem in the basic timestamp-ordering protocol. One possible way to make use of real-time priorities of transactions during scheduling is to involve real-time constraints in the timestamp assignment procedure. The new protocol PTO categorizes the transactions into timestamp groups based on their arrival times. The time is divided into intervals of a certain length $\Delta$ and the transactions that arrive at the system within the same interval are placed in the same timestamp group.‡ The basic idea is to schedule the transactions of the same timestamp group based on their real-time priorities.

Each transaction is assigned a two-level timestamp made up of a group timestamp and a real-time timestamp. The transactions within the same timestamp group are assigned the same group timestamp which is the arrival time of the first transaction in that group. Real-time timestamps of transactions within the same group are determined based on the real-time priorities of

†See the examples provided in Ref. [18] revealing the differences between those two schemes.
‡At each time interval $\Delta$, a timestamp group is formed only if at least one transaction arrives during that interval.

```
if (group_timestamp(T) > group_timestamp(D))
  T accesses D;
  group_timestamp(D) = group_timestamp(T);
  real-time_timestamp(D) = real-time_timestamp(T);
else
  if (group_timestamp(T) = group_timestamp(D)
  and real-time_timestamp(T) > real-time_timestamp(D))
    T accesses D;
    real-time_timestamp(D) = real-time_timestamp(T);
  else
    T is aborted;
```

Fig. 7. Data access handling in protocol PTO.

transactions. The transaction with the highest priority obtains the largest real-time timestamp, so it cannot be aborted by any other transaction in the same group in the case of a data access conflict. Real-time timestamps are used in ordering the access requests of the transactions from the same group, while the group timestamp is used in ordering the transactions from different groups. Between any two timestamp groups, the one which is formed earlier has the smaller group timestamp.†

Each data item is associated with both the group and real-time timestamps of the most recent transaction that has performed an access on the item. Figure 7 describes how the data access requests of transactions are handled by protocol PTO. Assume that a transaction $t$ requires an access on data item $D$. When transaction $T$ attempts to access $D$, the group timestamp of $T$ is compared with that of $D$. If $T$ has a larger group timestamp, the access request is granted and the group and real-time timestamps of $D$ are updated. If, on the other hand, $T$'s group timestamp is smaller than $D$'s group timestamp, $T$ is aborted. If both have equal group timestamps, the real-time timestamps are compared. $T$ must have a larger real-time timestamp for a successful access attempt, otherwise it is aborted. An aborted transaction obtains new group and real-time timestamps when it is restarted. As can be seen easily, the protocol differs from basic timestamp-ordering timestamps when it is restarted. As can be seen easily, the protocol differs from basic timestamp-ordering (TO) when the transactions from the same timestamp group have conflicting data accesses. The following example illustrates this situation.

**Example 5**—Suppose that transactions $T_x$ and $T_y$ are placed in the same group and $T_x$ has a higher real-time priority; thus it is assigned a higher real-time timestamp even though it arrives earlier than $T_y$. Assume that they have a conflicting access on data item $D$, and $T_y$ accesses the item before $T_x$. Later, transaction $T_x$'s request for $D$ is accepted since its real-time timestamp is larger than that of $D$. However, under protocol TO, the higher priority transaction $T_x$ would get a smaller timestamp of it arrived earlier than $T_y$. In this case, for the same access sequence, $T_x$ is aborted at its attempt to access $D$.

Read/write access semantics can simply be included in the protocol by maintaining the maximum group/real-time timestamps of both read and write accesses for each data item. For a read request, the group/real-time timestamps of the requesting transaction are checked against the group/real-time write timestamps of the data item while for a write request, the transaction timestamps are checked against both read and write timestamps of the item.

The major issue in constructing a timestamp group for the transactions is the selection of the time interval $\Delta$. The effect of the $\Delta$ value on the performance of protocol PTO has been studied and the results are provided in Section 5.1. The value of $\Delta$ should be neither too small nor too large. If $\Delta$ is too small, protocol PTO behaves very much like protocol TO; each transaction group usually contains one transaction, leading to the situation that transaction priorities are hardly being used in scheduling decisions. For too large values of $\Delta$, the protocol was seen to exhibit worse performance than TO, even though the number of data conflicts resolved based on transaction priorities increases with larger group sizes. One reason for this result might be the so-called 'starvation' problem, i.e. a transaction can be aborted repeatedly at each access to the same data

†Note that accesses to a data item by the transactions from "different" groups are ordered based on transactions' arrival times to the system, just as in protocol TO, without taking the priorities into account.

item if it is placed in the same transaction group (and thus assigned the same timestamp as before) when it is restarted. Another reason for the worse performance might be unnecessary transaction aborts as a result of the grouping idea. Consider the following situation. A transaction $T$ is submitted after transaction $T_y$ of the same group has been committed, and carries a priority less than that of $T_y$. Transaction $T_x$ will be aborted if it attempts to access a data item $D$ formerly accessed by $T_y$. It cannot be acceptable to abort a transaction $T$ (even if it has a low priority) due to a data conflict with another transaction that had already been committed when $T$ was submitted to the system. Based on these observations, it can be concluded that the time interval $\Delta$ should not be too small in order to make use of transaction priorities in access scheduling, and not too large in order to prevent the waste of resources due to unnecessary transaction aborts. With a reasonable selection of $\Delta$, we have shown that protocol PTO outperforms protocol TO for the majority of the transaction load ranges employed in our performance experiments.

### 3.3. *Optimistic protocol (OP)*

In the optimistic concurrency control protocol proposed by Haritsa *et al.* [5], the validation check for a committing transaction is performed against the other active transactions and the transactions that are in conflict with the committing transaction are aborted. The proposed protocol uses a"50%" rule as follows: If half or more of the transactions conflicting with a committing transaction are of higher priority, the transaction is blocked until the high-priority transactions complete; otherwise, it is allowed to commit while the conflicting transactions are aborted.

The protocol described above is the *broadcast commit* (or *forward-oriented*) variant of optimistic concurrency control. A validating transaction is always guaranteed to commit [14]. Another variant is the *classical* (or *backward-oriented*) optimistic concurrency control in which transactions are allowed to executed without being blocked or aborted due to conflicts until they reach their commit point [19]. A transaction is validated at its commit point against transactions that committed since it began execution. If the transaction fails its validation test (i.e. if it has a conflict with any of those committed transactions), it is restarted. The real-time performance of a classical optimistic concurrency control protocol was also evaluated using our RTDBS model.† The performance results were similar to what we obtained (and present in Section 5 of this paper) with the optimistic concurrency control protocol (OP) detailed in the previous paragraph.

## 4. RTDBS MODEL

This section briefly presents the RTDBS simulation model that we used to evaluate the performance of the protocols. The model is based on an open queuing model of a single-site database system. It contains two physical resources shared by the transactions, CPU and disk. The simulator has five components: a workload generator, a transaction manager, a scheduler, a buffer manager and a resource manager.

The workload generator simulates transaction arrivals and determines the type, deadline and data access list of each new transaction by using the system parameter values.

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. All the concurrency control protocols, except PC and DP‡ assume that "deadline" is the only information provided by the arriving transaction to be used in scheduling decisions. Each transaction is assigned a unique real-time priority based on its deadline. Unless otherwise indicated, the *Earliest Deadline First* (EDF) priority assignment policy is employed; i.e. a transaction with an earlier deadline has higher priority than a transaction with a later deadline. 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.

†The performance results are provided in Ref. [20].
‡Protocols PC and DP assume that the data access list of the transaction is also available prior to transaction's execution.

Each transaction performs one or more database operations (read/write) on specified data items. Concurrent data access requests of the transactions are controlled by the scheduler. The scheduler orders the data accesses based on the concurrency control protocol executed. Depending on its real-time priority, an access request of a transaction is either granted or results in blocking or abort or the transaction. If the access request is granted, the transaction attempts to read the data item from the main memory. If the item cannot be found in memory, the transaction waits until the item is transferred from the disk into the memory. Data transfer between disk and main memory is provided by the buffer manager. The FIFO page replacement strategy† is used in the management of memory buffers. Following the access, the data item is processed. A write operation is handled similar to a read except for the IO activity to write the updated data on the disk. Whenever a write operation is performed by a transaction, the new values of the updated data item is placed in a memory buffer. It is assumed that the buffer space is large enough so that a transaction does not have to write its updates on the disk until after commit. If a transaction is aborted at any point of its execution, all the data updates performed by it, if any, are simply ignored. Thus, aborting a transaction does not involve any disk writes. The major cost of an abort is the waste of resources (i.e. processing and IO times) already used by the aborted transaction. The aborted transaction is restarted after a certain restart delay.‡ The purpose of delaying the transaction before resubmitting it to the system is to prevent the same conflict from occurring repeatedly. A restarted transaction accesses the same data items as before (i.e. the items in its access list).

When a transaction completes all its data access and processing requirements, it can be committed. If it is a query (read-only), it is finished. If it has updated one or more data items during its execution, it goes to the IO queue to write its updates into the database. The IO delay of writing the modified data items into the database constitutes the basic cost of a commit operation.

The resource manager is responsible for providing IO service for reading/updating data items, and CPU service for processing data items and performing various concurrency control operations (e.g. conflict check, locking, etc.). Both CPU and IO queues are organized on the basis of the transactions' real-time priorities. Preemptive-resume priority scheduling is used by the CPU; 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. Besides preemption, the CPU can be released by a transaction as a result of a lock conflict or for IO.

We do not model a recovery scheme as we assume a failure-free database system.

The following set of parameters is used to specify the system configuration and workload. The parameter *db_size* corresponds to the number of data items stored in the database, and *mem_size* is the number of data items that can be held in main memory. *iat* is the mean interarrival time of transactions. Arrivals are assumed to be Poisson. The type of a transaction (i.e. query or update) is determined randomly using the parameter *tr_type_prob* which specifies the update type probability. The number of data items to be accessed by the transaction is determined by the parameter *access_mean*. The distribution of the number of data items is exponential. Accesses are randomly distributed across the whole database. 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 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*. CPU and IO times for transactions are considered separately for the purpose of achieving CPU–IO overlap. The CPU time for processing a data item is specified by the parameter *cpu_time*, while the time to access a data item on the disk is determined by *io_time*. These parameters represent constant service time requirements. To prevent the possibility of transaction overload in the system, the total number of active transactions in the system is limited by the parameter *max_act_tr*. When this limit is reached, transaction arrivals are temporarily inhibited.§

†The FIFO page replacement strategy was used in the simulation because it is simple to implement. This choice should not affect the comparative performance of the concurrency control protocols.

‡The restart delay used in the simulations was the average lifetime of a transaction in an unloaded system.

§This did not occur often enough in our simulations to affect results.

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 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.
- Validation test (in optimistic protocol OP): performing the validation test for a transaction against each of the active transactions (i.e. checking for a common element between the read set of the validating transaction and the write set of each of executing transaction).

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 for a locking protocol 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. For a timestamp-ordering protocol, the conflict check operation executes a table lookup operation to get the timestamp value of the requested data item, and a comparison operation to compare the timestamp of the requesting transaction to that of the data item. 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. In optimistic protocol OP, during the validation phase of a transaction, each validation test is performed by intersecting two data sets to determine if a common element exists in these two sets. The intersection algorithm requires a number of comparison operations. The number of validation tests performed for a transaction is equal to the number of actively executing transactions.

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*.

*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 is determined by the following formula:

$$deadline = start\ time + processing\ time\ estimate + slack\ time$$

where

$$slack\ time = expon\,(slack\_rate \times processing\ time\ estimate)$$

$$processing\ time\ estimate = CPU\ requirement + IO\ requirement$$

Table 1. Performance model parameter values

| Configuration Parameters | |
|---|---|
| *db_size* | 200 |
| *mem_size* | 50 |
| *cpu_time* | 12 ms (constant) |
| *io_time* | 12 ms (constant) |
| *max_act_tr* | 20 |
| *pri_assign_cost* | 1 ms (constant) |
| *basic_op_cost* | 0.1 ms (constant) |
| **Transaction Parameters** | |
| *iat* | 100 ms (exponential) |
| *tr_type_prob* | 0.5 |
| *access_mean* | 6 (exponential) |
| *data_update_prob* | 0.5 |
| *slack_rate* | 5 (exponential) |

Let *items* denote the actual number of data items accessed by the transaction:

$$CPU\ requirement = items \times cpu\_time \tag{1}$$

For a query,

$$IO\ requirement = items \times (1 - mem\_size/db\_size) \times io\_time \tag{2}$$

and for an update transaction:

$$IO\ requirement = items \times (1 - mem\_size/db\_size) \times io\_time + items \times data\_update\_prob \times io\_time. \tag{3}$$

## 5. PERFORMANCE EVALUATION

The values of the configuration and transaction parameters that are common to all concurrency control protocols are given in Table 1. Unless otherwise stated, these are the values used in the experiments. It was not intended to simulate a specific database; instead the parameter values were chosen to yield a system load and data contention high enough to observe the differences between 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 is 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 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* where chosen to obtain almost identical CPU and IO utilizations in the system.‡ The range of *iat* values used in the experiments corresponds to high levels of CPU and IO load. 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. The results and discussions for some other settings of the value of this parameter can be found in Ref. [20].

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 [21], which is a process-oriented simulation language based on the C programming language. Each data item in the database is simulated individually. The program also keeps track of the list of data items resident in main memory. Each

†The results of the experiments performed with larger database size values are provided in Section 5.3.
‡The calculations of expected average CPU and IO utilizations based on the system parameters are provided in Ref. [20].

run of the following experiments was continued until 500 transactions were executed. The "independent replication' method was used to validate the results by running each configuration 25 times with different random number seeds and using the averages of the replica means as final estimates. 90% confidence intervals were obtained for the results with the assumption of independent observations [22]. The following sections discuss only statistically significant performance results. The width of the confidence interval of each statistical data point is less than 3% of the point estimate. In displayed graphs, only the mean values of the performance results are plotted.

The basic performance metrics used were *success-ratio*, and *average-lateness*. *success-ratio* is the fraction of transactions that satisfy their deadline constraints. *average-lateness* is the average lateness of the transactions that missed 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).

In the following sections we present those experimental results that were most interesting and best illustrate the performance of the protocols. The remainder of the performance results are summarized where necessary.

### *5.1. Effects of transaction load*

In this experiment, the real-time performance characteristics of concurrency control protocols were investigated for varying transaction loads in the system. The parameter *iat* was varied from 75 to 135 ms in steps of 10. These values correspond to expected CPU and IO utilizations of about 0.96–0.53 [20]. We are basically interested in the evaluation of control protocols under high levels of resource utilization. The results of the experiment are presented in two categories; the locking protocols (AB, PI, PB, PC and DP) are placed in the first category, and the timestamp-ordering and optimistic protocols (TO, PTO and OP) are grouped in the second category. The performance results for the locking protocols are shown in Figs 8 and 9. An interesting result obtained in this experiment is the relatively poor performance exhibited by priority ceiling protocol PC. Even protocol AB, which does not take real-time priorities into account in scheduling the data accesses, provides better performance than PC. The implementation drawbacks of PC, which were summarized when we described the protocol, have led to this unsatisfactory performance. The restrictive nature of the priority ceiling blocking rule does not allow many of the transactions to obtain data locks without being blocked. A high level of *conflict-ratio* is observed with protocol PC due to priority ceiling conflicts (average number of times each transaction is blocked on the priority ceiling rule) rather than data conflicts. An excessive number of transaction blocks results in low concurrency and low resource utilization. Especially at high transaction loads, many transactions miss their deadlines.

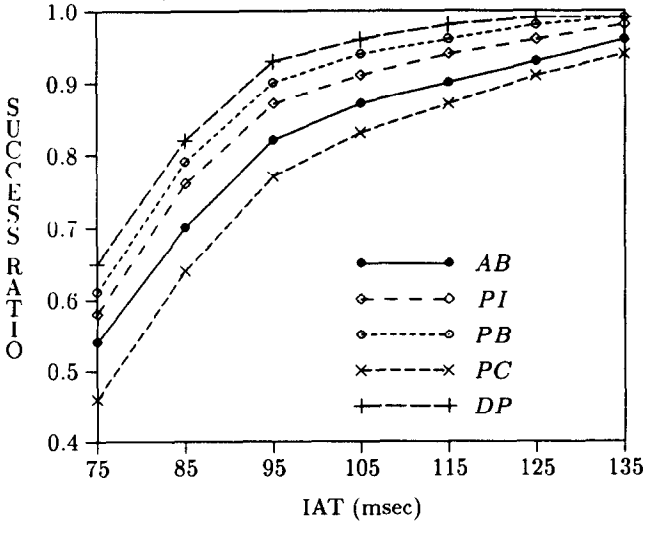


Fig. 8. *Success-ratio* vs *iat* [average transaction interarrival time (ms)] for the locking protocols.

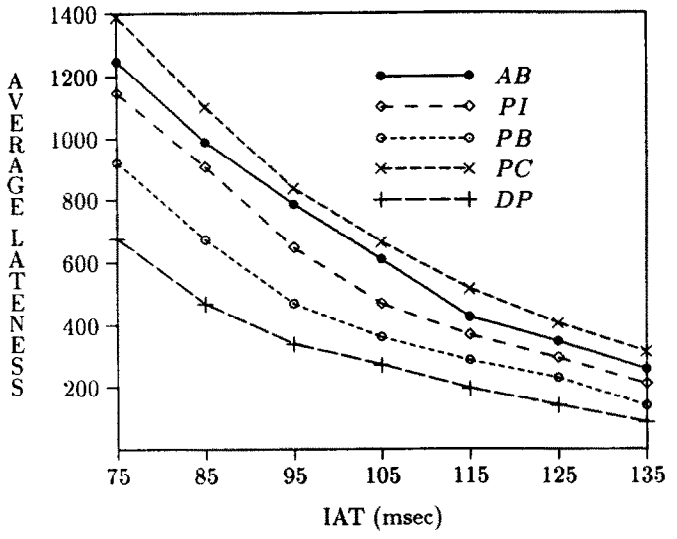


Fig. 9. *Average-lateness* (ms) vs *iat* [average transaction interarrival time (ms)] for the locking protocols.

The performance of the priority ceiling protocol in RTDBSs was also studied by Son and Chang [16] and Sha *et al.* [9]. Son and Chang evaluated two different versions of the priority ceiling protocol (PC) in a distributed system environment, and observed that the performance of both protocols were 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 performance model are soft; i.e. all the transactions are processed to completion, regardless of whether they are late or not.

Sha *et al.* 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.

We found that protocol PI provides a considerable improvement over the base protocol AB. This improvement is due to reducing the blocking time of high-priority transactions by the priority inheritance method. However, PIs performance cannot reach the level achieved by protocol PB. Remember that PB never blocks higher priority transactions, but instead aborts low priority transactions when necessary. PB also eliminates the possibility and cost of deadlocks. We can conclude that aborting a low priority transaction is preferable in RTDBS's to blocking a high priority one, even though aborts lead to a waste of resources. These results are similar to what Huang *et al.* obtained in Ref. [13], where it was shown that protocol PB works better than protocol PI in RTDBSs. They also found in Ref. [10] that the performance obtained by employing real-time policies based on the PB concurrency control protocol was better, in general, than that obtained in a nonreal-time transaction processing system. Abbott and Garcia-Molina also found that both protocols PI and PB perform better than protocol AB [6]. Their results indicated that no protocol is the best under all conditions; the comparative performance of the protocols PI and PB 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

†Remember that in protocol PC, when a 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 until transaction $T$ is reexecuted or a transaction with higher priority arrives at the CPU queue.

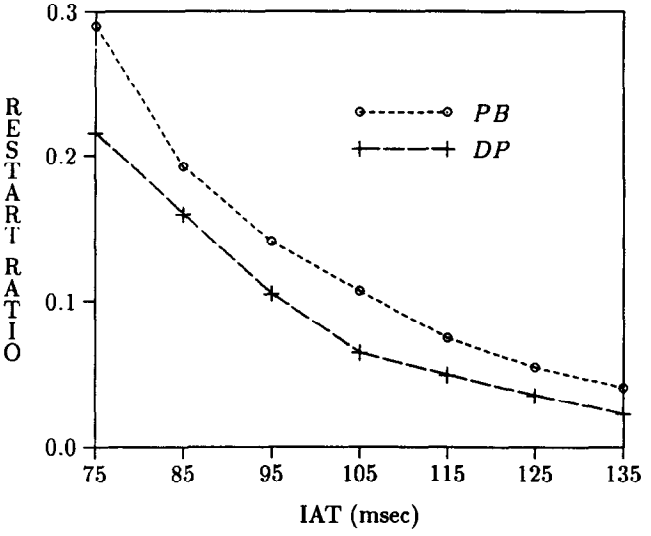


Fig. 10. *Restart-ratio* vs *iat* [average transaction interarrival time (ms)] for protocols PB and DP.

of protocol PB. 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.

The performance of our new protocol DP is better than that of protocol PB, especially at high-transaction load. Some of the transaction aborts, and the resulting resource waste, experienced with protocol PB can be prevented by protocol DP, which employs prior knowledge of data requirements in access scheduling. Protocol DP restricts the possibility of aborts to only the situation that between any two conflicting transactions, the low priority transaction accesses the data item before the high-priority transaction is submitted to the system (before the priority of the data item is adjusted by the entry of the high-priority transaction). For this special case the lower priority transaction is aborted when and if the higher priority transaction accesses the data item before the commitment of the low priority transaction. Figure 10 shows the *restart-ratio* obtained by executing these two protocols under varying levels of transaction load.

The timestamp-ordering and optimistic concurrency control protocols were the next group of protocols that we evaluated by our simulation model. Before conducting the experiments to compare the performance of the priority-based timestamp-ordering protocol PTO with others', a set of tests was performed to select a reasonable value for the parameter $\Delta$ of the protocol (see Section 3.2). Table 2 shows the improvement obtained with protocol PTO over protocol TO for a wide range of values of $\Delta$, in terms of *success-ratio* on a percentage basis. Each $\Delta$ value was tested for different values of *iat* corresponding to different levels of transaction load. The best performance results for PTO with our parameter settings were obtained when $\Delta$ was assigned a value of around 500 ms. The reasons of the poor performance results obtained for too small values of $\Delta$ (i.e. $\Delta \leqslant 50$ ms), and too large values of $\Delta$ (i.e. $\Delta \geqslant 2000$ ms) were already discussed in Section 3.2. The experiment results for protocol PTO presented below were obtained with $\Delta = 500$ ms.

Figures 11 and 12 show the performance results of the timestamp-ordering and optimistic concurrency control protocols. The optimistic protocol (OP) performs very well for large values

Table 2. Improvement in *success-ratio* by PTO over TO. Row: *iat* (in ms), Column: $\Delta$ (in ms)

| | 20 | 50 | 100 | 250 | 500 | 750 | 1000 | 2000 |
|---|---|---|---|---|---|---|---|---|
| 75 | 0% | 0% | 5.5% | 10.9% | 12.7% | 9.1% | 1.8% | −3.6% |
| 85 | 0% | 0% | 2.7% | 5.5% | 6.9% | 6.9% | 1.4% | −1.4% |
| 95 | 0% | 0% | 1.2% | 3.7% | 4.9% | 3.7% | 0% | 0% |
| 105 | 0% | 0% | 1.2% | 3.5% | 3.5% | 4.7% | 3.5% | −1.2% |
| 115 | 0% | 0% | 0% | 1.1% | 1.1% | 2.2% | 1.1% | −1.1% |
| 125 | 0% | 0% | 0% | 0% | 0% | 0% | 0% | −1.1% |
| 135 | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |

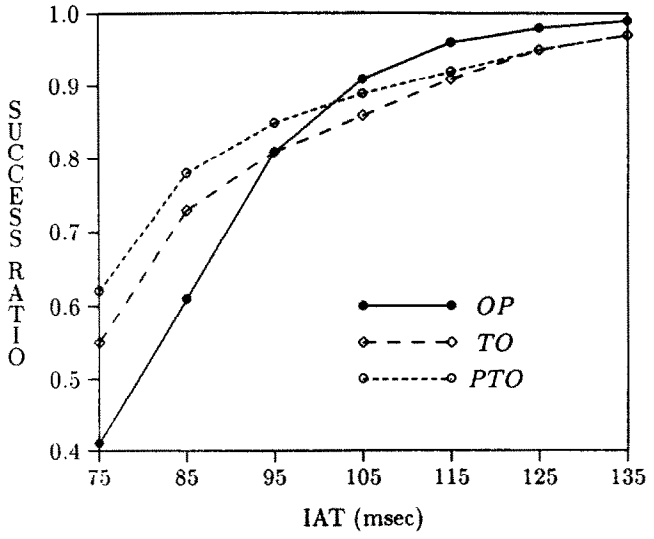


Fig. 11. *Success-ratio* vs *iat* [average transaction interarrival time (ms)] for the timestamp-ordering and optimistic protocols.

of interarrival time; i.e. when the system is lightly loaded. There is no overhead of checking for conflicts at each data access. Since the number of conflicts is small under low load levels, only a few transactions fail to be validated at commit time. However, under high load levels the performance of protocol OP is worse than other protocols. This can be explained by the increased number of restarts due to increased conflicts among transactions. The transactions that are in conflict with a committing transaction are aborted and restarted from the beginning. The wasted execution time due to the large number of restarts substantially increases the number of missed deadlines and the lateness of the tardy transactions.

Protocol TO performs better than protocol OP when the system is highly loaded (small *iat*). However, the situation is reserved under low load levels. Our new protocol PTO was designed to improve the real-time performance of protocol TO by involving real-time priorities in scheduling decisions. For high transaction load, protocol PTO provides a considerable improvement over the performance of protocol TO. Under low transaction loads, corresponding to $iat \geqslant 125$, further improvement of the performance is not possible with this method. As the time interval between

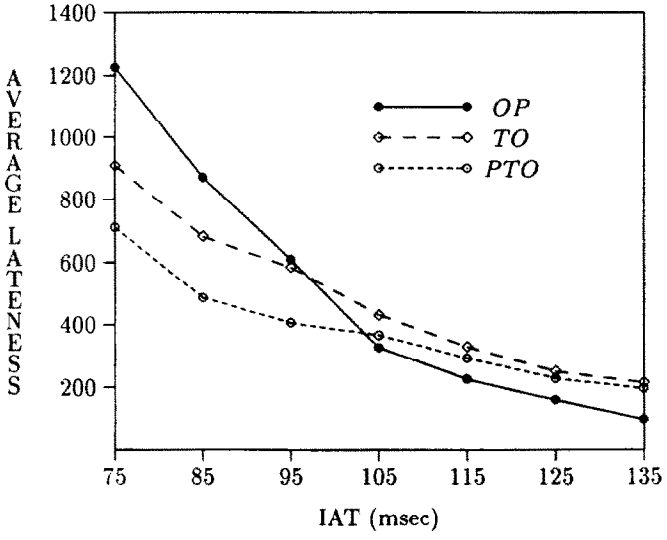


Fig. 12. *Average-lateness* (ms) vs *iat* [average transaction interarrival time (ms)] for the timestamp-ordering and optimistic protocols.

transaction arrivals increases, the number of transactions constituting a group becomes smaller. This results in a reduced chance of using transaction priorities in access scheduling decisions.† Thus, protocol PTO behaves more like basic timestamp-ordering protocol TO.

If we compare all types of protocols together, the best locking protocol (protocol DP) is the best overall. Under high transaction loads, the performance of the timestamp-ordering protocol PTO is close to that of protocol DP. The optimistic protocol (OP) performs well only under light transaction loads; this observation agrees with the result obtained by Huang *et al.* who showed in Ref. [12] that the optimistic scheme performs better than the two-phase locking scheme when data contention is low, and vice versa when data contention is high.

Haritsa *et al.* also compared locking and optimistic protocols in Ref. [14], where it was shown that the optimistic concurrency control technique is superior to the two-phase locking protocol. However, their experiments were performed in a RTDBS that discards late transactions (i.e. the deadlines are firm) and most of the simulation results were obtained under the assumption that the system has infinite resources. These assumptions, most probably, are the source of the difference between their results and ours; because, when they processed the transactions in a finite resource system, with soft deadlines (as in our model), they found that the locking protocol performs better than the optimistic one, which confirms our findings.

### *5.2. Priority assignment policies*

This experiment was conducted to determine whether changing the priority assignment policy would affect the results. In the results reported above, the Earliest Deadline First (*EDF*) policy was used. Another well-known priority assignment heuristic for real-time scheduling algorithms is the Least Slack First (*LSF*) policy.

For this experiment, the static version‡ of the LSF policy was implemented. This policy assigns the priority of a transaction based on its slack time when it is submitted to the system as a new transaction or when it is restarted. The LSF policy assumes that each transaction provides its processing time estimate.

Comparing the EDF and LSF priority assignment policies, we observe different performance characteristics for the blocking-based locking protocols AB, PI and PC. The LSF policy results in a decrease in the number of missed deadlines and an increase in the average lateness of tardy transactions. Since the *restart-ratio* in these protocols is quite low (0 in PC), most of the transactions keep the static priority assigned at the beginning of their execution. As a result, as is not the case in the EDF policy, the priority of a tardy transaction is not always higher than the priority of a nontardy transaction. This situation gives rise to an increase in lateness of the tardy transactions as the nontardy transactions get a better chance to satisfy their deadlines. Our observation agrees with the result obtained by Abbott and Garcia-Molina in comparing EDF and LSF [6]. Figure 13 shows the performance results in terms of *average-lateness* for the protocol PI under both the EDF and LSF policies. The reason for showing PI results is that it is one of the blocking-based protocols affected by the selection of the priority assignment policy.

The difference between the two priority assignment policies is not striking for the other protocols, which are based on resolving the conflicts by restarts. Since the priority of a transaction is changed each time it is restarted, tardy transactions with adjusted priority are more likely to be scheduled before the nontardy transactions, which is always the case with the EDF policy.

### *5.3. Summary of the results of other experiments*

The size of the database in our RTDBS model was set to 200 data items in all experiments described in the preceding sections. The purpose of assuming a very small database size was to obtain a high number of data conflicts among the transactions. About 3% of the database was accessed by each transaction which provides *conflict-ratio* high enough to be able to observe the relative effectiveness of the various conflict resolution strategies. The small database used in our experiments can also be considered as the "hot spot" of a larger database. For a more complete

†Remember that only the transactions that belong to the same transaction group are ordered based on their priorities. Transactions from different groups are scheduled based on their arrival times.

‡The dynamic version of this policy evaluates the transaction priorities continuously. We did not implement it because of the considerable overhead incurred by calculation of the priorities whenever needed.

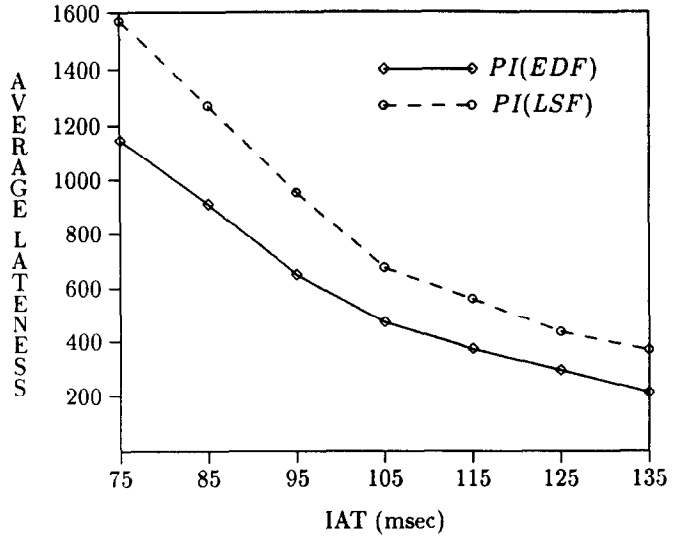


Fig. 13. *Average-lateness* (ms) vs *iat* [average transaction interarrival time (ms)] for protocol PI with different priority assignment policies.

study of the protocols' performance for different levels of data conflict, the protocols were evaluated under varying values of the database size (*db_size*). As database size got larger, less data contention (due to fewer data access conflicts) occurred as expected, and the real-time performances of the protocols became closer [20].

To justify the assumption of considering a small-sized database (i.e. 200 data items) as the hot spot of a larger database, we performed an experiment by setting the database size to 1000 data items and employing the 20/80 data access model which assumes that 80% of data operations are performed on 20% of the database. Figure 14 illustrates the performance results of three representative concurrency control protocols (DP, OP and PTO) in terms of the fraction of satisfied deadlines under different transaction load levels. Comparing the results to those presented in Figs 8 and 11 (which were obtained with *db_size* of 200 items and random data access model), one can see that the change in data access model did not affect the relative performances of the protocols. This observation confirms that our hot spot assumption is reasonable.

Another experiment was conducted of evaluate the performance of the protocols under the assumption that there exists some locality of data reference. As the locality of reference (i.e. the

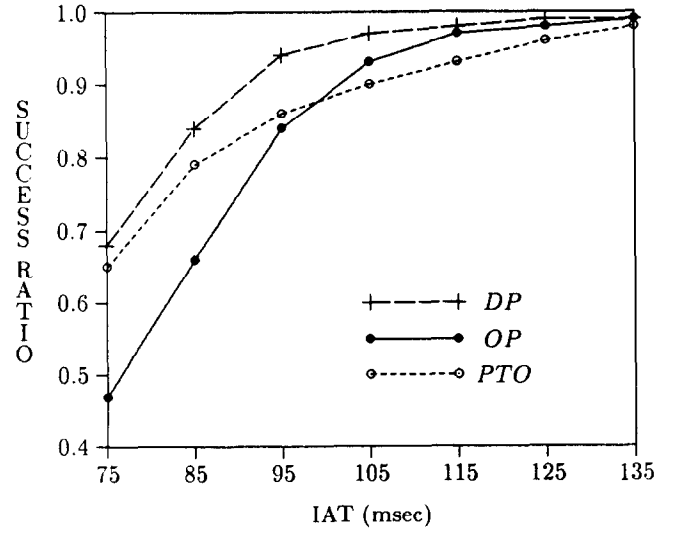


Fig. 14. *Success-ratio* vs *iat* [average transaction interarrival time (ms)] with *db_size* of 1000 data items and 20/80 data access model.

probability of referring to a recently accessed data item) was increased, the number of data conflicts among transactions became reduced as a result of the decrease in the average number of distinct data items accessed by each transaction. The average IO delay experienced by each transaction also dropped due to the fact that recently accessed data items can more likely to be found in main memory. As a result, better performance was achieved by each concurrency control protocol with higher levels of locality. Figure 15 illustrates the performance results, again for three representative protocols. *iat* value was set to 75 ms in this experiment. As can be seen from the figure, the protocol which benefits most from increasing locality is OP. This result is in agreement with our previous observation for this protocol which states the OP performs well under low conflict levels. The figure presents the results up to the locality level of 0.6; for higher locality values the performances were observed to be indistinguishable (statistically insignificant).

The performance model used in our evaluation of concurrency control protocols employed both data and resource contention. Data contention exists due to the conflicting data access requests of the transactions, which results in either transaction blocking or transaction aborts to resolve the conflict. The effects of data conflicts, and thus data contention, have been studied in detail by the experiments presented above. Resource contention, on the other hand, exists in a system due to the limited number/capacity of system resources (e.g. CPU, disk). It results in queuing delay of transactions at each of those resources. In order to observe the performance of the protocols under different levels of resource contention we varied the number of processors (CPUs). The performance of the protocols improved with increasing number of processors; however the optimistic protocol (OP) outperformed the other protocols when large numbers of processors existed in the system (i.e. under low levels of CPU contention). This result is similar to what we obtained for low levels of data contention; that is not surprising, because shorter transaction lifetime (due to less queuing delay) results in fewer data conflicts among the transactions.

Other experiments were conducted to evaluate the effect of various other system parameters on the protocols' performance. These parameters included *tr_type_prob* (fraction of upgrade type transactions), *slack_rate* (ratio of the slack time of a transaction to its execution time) and *mem_size* (number of data items that can be held in main memory). The results, that can be found in Ref. [20], are not displayed here since the relative performance of the protocols was not sensitive to varying the values of these parameters. In another experiment, we studied the protocols under different transaction length environments by varying the parameter *access_mean* (i.e. average number of data items accessed by each transaction). This was another way of varying the *conflict-ratio* experienced by the transactions. As expected from the results obtained from the experiments presented above, protocol OP performed the best, in general, when the system was dominated by short transactions (i.e. when data conflict was low); on the other hand, its performance was worse than the others when the majority of the transactions were long-lived.

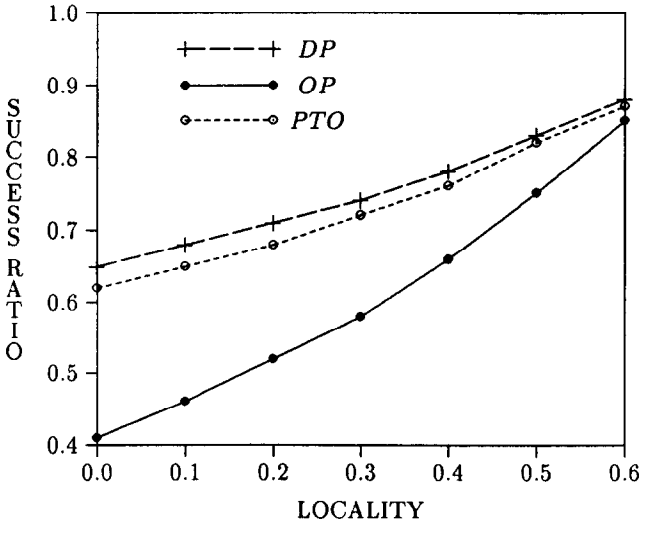


Fig. 15. *Success-ratio* vs locality (the probability of referring to a recently accessed data item).

## 6. CONCLUSIONS

In this paper, we have provided a study of the relative performance of different approaches to the concurrency control problems in RTDBSs. Unlike conventional database systems, the transactions processed in a RTDBS have timing constraints, typically in the form of deadlines. Crucial to the concurrency control problem in RTDBSs is processing the transactions within their deadlines, as well as maintaining the consistency of data accessed by the transactions.

Each protocol studied in this paper belongs to one of the three basic classes of concurrency control: locking, timestamp-ordering and optimistic. Two new protocols have been introduced in this paper to provide an improvement in the performance of the existing ones. One of the new protocols is based on locking, and the other on timestamp-ordering. Various simulation experiments were conducted to study the relative performance of the protocols under many possible real-time and database environments. The performances of the protocols were evaluated in terms of the fraction of transactions that satisfy their deadlines, and the average lateness of the tardy transactions. The basic conclusions obtained from the experiments can be summarized as follows:

- In terms of the real-time performance of the locking protocols, the Priority Inheritance protocol (PI) and Priority-Based conflict resolution protocol (PB) both dominated the Always Block protocol (AB), which is the basic version of two-phase locking. PI allows a low priority transaction to execute at the highest priority of all the high-priority transactions it blocks. PB aborts a low priority transaction when one of its locks is requested by a higher priority transaction. PB appeared to perform better than PI, leading to a conclusion that aborting a low priority transaction instead of blocking a higher priority on can help more transactions to commit within their deadlines. The Priority Ceiling protocol (PC) was outperformed by all the protocols we tested. The priority ceiling rule that might block the lock-requesting transactions even without a data conflict is too restrictive to be implemented in RTDBSs. Our new Data Priority-based locking protocol (DP) proved that an improvement in the real-time performance level achieved by PB is possible if the data access list of each transaction is available to the scheduler prior to the execution of the transaction. DP outperformed all other locking protocols under various levels of transaction load, and resource contention.
- The basic Timestamp-Ordering protocol (TO) did not perform well compared to other concurrency control protocols for most of the parameter ranges we tested. We investigated the effects of involving real-time priorities in TO by employing our new Priority-based Timestamp-Ordering protocol PTO. What we observed is that it is possible to improve the real-time performance of TO, especially under high load and high data conflict conditions, by using real-time priorities in assigning timestamps to transactions, but the improvement is not enough to bring its performance up to that of the good locking protocols. The Optimistic Protocol (OP) performed well only under light transaction loads, or when the data/resource contention in the system was low.
- When the Least Slack First priority assignment policy was employed instead of Earliest Deadline First, a decrease in the number of missed deadlines and an increase in average lateness of tardy transactions were observed for the blocking-based locking protocols AB, PI and PC, but the other protocols were little affected.

## REFERENCES

[1] L. Sha, R. Rajkumar and J. P. Lehoczky. Concurrency control for distributed real-time databases. *ACM SIGMOD Record* **17**, 82–98 (1988).
[2] Ö. Ulusoy and G. G. Belford. Real-time lock based concurrency control in a distributed database system. *Proc. 12th Int. Conf. on Distributed Computing Systems*, pp. 136–143 (1992).
[3] L. Sha, R. Rajkumar and J. P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Trans. Comput.* **39**, 1175–1185 (1990).
[4] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. *Proc. 14th Int. Conf. on Very Large Data Bases*, pp. 1–12 (1988).
[5] J. R. Haritsa, M. J. Carey and M. Livny. Dynamic real-time optimistic concurrency control. *Proc. 11th Real-Time Systems Symp.*, pp. 94–103 (1990).
[6] R. Abbot and H. Garcia-Molina. Scheduling real-time transactions with disk resident data. *Proc. 15th Int. Conf. on Very Large Data Bases*, pp. 385–396 (1989).
[7] R. Abbot and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. *ACM Trans. Database Syst.*, **17**, 513–560 (1992).

[8] R. Abbot and H. Garcia-Molina. Scheduling I/O requests with deadlines: a performance evaluation. *Proc. 11th Real-Time Systems Symp.*, pp. 113–124 (1990).
[9] L. Sha, R. Rajkumar, S. H. Son and C. H. Chang. A real-time locking protocol. *IEEE Trans. Comput.* **40**, 793–800 (1991).
[10] J. Huang, J. A. Stankovic, D. Towsley and K. Ramamritham. Experimental evaluation of real-time transaction processing. *Proc. 10th Real-Time Systems Symp.*, pp. 144–153 (1989).
[11] S. R. Biyabani, J. A. Stankovic and K. Ramamritham. The integration of deadline and criticalness in hard real-time scheduling. *Proc. 9th Real-Time Systems Symp.*, pp. 152–160 (1988).
[12] J. Huang, J. A. Stankovic, K. Ramamritham and D. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. *Proc. 17th Int. Conf. on Very Large Data Bases*, pp. 35–46 (1991).
[13] J. Huang, J. A. Stankovic, K. Ramamritham, D. Towsley and B. Purimetla. Priority inheritance in soft real-time databases. *Real-Time Syst.* **4**, 243–268 (1992).
[14] J. R. Haritsa, M. J. Carey and M. Livny. On being optimistic about real-time constraints. *ACM SIGACT-SIGMOD-SIGART*, pp. 331–343 (1990).
[15] J. R. Haritsa, M. J. Carey and M. Livny. Data access scheduling in firm real-time database systems. *Real-Time Syst.* **4**, 203–242 (1992).
[16] S. H. Son and C. H. Chang. Performance evaluation of real-time locking protocols using a distributed software prototyping environment. *Proc. 10th Int. Conf. on Distributed Computing Systems*, pp. 124–131 (1990).
[17] Y. Lin and S. H. Son. Concurrency control in real-time databases by dynamic adjustment of serialization order. *Proc. 11th Real-Time Syst.*, pp. 104–112 (1990).
[18] P. A. Bernstein, V. Hadzilacos and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, New York (1987).
[19] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.* **6**, 213–226 (1981).
[20] Ö. Ulusoy. Concurrency control in real-time database systems. Technical Report, UIUCDCS-R-92-1762, Department of Computer Science, University of Illinois at Urbana-Champaign (1992).
[21] H. Schwetman. CSIM: a C-based, Process-oriented simulation language. *Proc. Winter Simulation Conf.*, pp. 387–396 (1986).
[22] D. Ferrari. *Computer Systems Performance Evaluation.* Prentice-Hall, Englewood Cliffs, NJ (1978).