

# Scheduling and Concurrency Control for Real-Time Database Systems

Sang H. Son and Seog Park†

Department of Computer Science, University of Virginia, Charlottesville, VA 22903, USA

† Department of Computer Science, Sogang University, Seoul, Korea

## Abstract

The design and implementation of real-time database systems presents many new and challenging problems. Compared with traditional databases, real-time database systems have a distinct feature: they must satisfy timing constraints associated with transactions. Transactions in real-time database systems should be scheduled considering both data consistency and timing constraints. In this paper we describe characteristics and requirements of real-time database systems such as timing constraints, correctness criteria and predictability. Also we address the issues associated with transaction scheduling and concurrency control and present a scheduling algorithm for distributed real-time database systems. The protocol does not assume any knowledge about the data requirements or the execution time of each transaction. This makes the protocol widely applicable, since in many actual environments such information may not be readily available.

## 1. Introduction

A real-time database system (RTDBS) has (at least some) transactions with explicit timing constraints such as deadlines. The correctness of the system depends not only on the logical results but also on the time within which the results are produced. In RTDBS, transactions must be scheduled in such a way that they can be completed before their corresponding deadlines expire. For example, both the update and query on the tracking data for a missile must be processed within given deadlines.

Real-time database systems are becoming increasingly important in a wide range of applications, such as aerospace and weapon systems, computer integrated manufacturing, robotics, nuclear power plants, network management, and traffic control systems. Unfortunately, conventional database systems are not designed for time-critical applications and they lack features required for supporting real-time transactions. They are designed to provide good average performance, while possibly

yielding unacceptable worst-case response times. It has been generally recognized that there is a lack of basic theory for RTDBS since the traditional models are not adequate for time-critical applications. Researchers pointed to the need for basic research in database systems that satisfy timing constraints in collecting, updating, and retrieving shared data [Abb92, Buc89, Gra92, Har90, Hua91, Kor90, Lin89, Sha91, Son88, Son90, Son91].

Transactions in real-time database systems can be categorized as *hard* and *soft* transactions. We define hard real-time transactions as those transactions whose timing constraints must be guaranteed. Missing deadlines of this type of transaction may result in catastrophic consequences. In contrast, soft real-time transactions have timing constraints, but there may still be some justification in completing the transactions after deadline. Catastrophic consequences do not result if soft real-time transactions miss their deadlines. Soft real-time transactions are scheduled taking into account their timing requirements, but they are not guaranteed to make their deadlines. There are many real-time systems that need database support for both types of transactions.

The reasons why conventional database systems are not used in real-time applications include their poor performance and lack of predictability. In conventional database systems, transaction processing requires access to a database stored on secondary storage; thus transaction response time is limited by disk access delays, which can be in the order of milliseconds. Still these databases are fast enough for traditional applications in which a response time of a few seconds is often acceptable. However, those systems may not be able to provide a response fast enough for high-performance real-time applications. One approach to achieve high performance is to replace slow devices (e.g., a disk) by a high speed version (e.g., a large RAM). Another alternative is to use application-specific knowledge (e.g., semantic information associated with transactions and data) to increase the degree of concurrency.

Since an RTDBS is often used in safety-critical applications, it must provide predictable performance. An unpredictable system can do more harm than good under abnormal conditions. There are many reasons why traditional database systems show unpredictable

---

This work was supported in part by ONR, by DOE, by IBM, and by CIT.

performance. For example, to ensure the data consistency, traditional database systems often block certain transactions from reading or updating certain data if these data are locked by other transactions. Blocking will cause transactions to be delayed. Even worse, it is often difficult for a transaction to predict how long the delay will be since the blocking transactions themselves in turn may be blocked by other transactions. Consequently, the response time for a transaction in conventional database systems is often unpredictable.

Hard real-time transactions must be guaranteed that their deadlines are always met. To make such a strong guarantee, we cannot simply use the *best-effort* scheduling protocols. We must have scheduling protocols which can control the locking behaviors to guarantee the locking delays. To do this, RTDBS must have advance knowledge of the resource and data requirements of transactions. The priority ceiling protocols may reject the locking requests of transactions if doing so may cause more urgent transactions to have uncontrollable locking delays [Sha91, Son90b]. Using priority-based scheduling algorithms with predefined resource usage patterns, these protocols can guarantee that all transactions with shared resources can always meet their deadlines as long as some well-defined schedulability conditions are satisfied. A drawback of the priority ceiling protocols is that they require knowledge of all transactions that will be executed in the future. This is too harsh a condition for most database systems to satisfy.

In addition to timing constraints of transactions such as deadlines, criticalness which represents the importance of transactions should be considered in computing the priority of transactions. Therefore, proper management of priorities and conflict resolution in real-time transaction processing are essential for predictability and responsiveness of RTDBS.

One of the challenges of RTDBS is the creation of a theory for real-time scheduling and concurrency control protocols that maximizes both concurrency and resource utilization subject to three constraints: data consistency, transaction correctness, and transaction deadlines [Stan88]. Several recent projects have investigated the issue of adding real-time constraints into database systems to facilitate efficient and correct management of timing constraints in RTDBS [Buc89, Gra92, Son88, Son91, Son92c]. There are several difficulties in achieving this goal. A database access operation, for example, takes a highly variable amount of time depending on whether disk I/O, logging, buffering, etc. are required. Furthermore, concurrency control may cause aborts or delays of indeterminate length. In this paper we address the issues associated with transaction scheduling and concurrency control, and present a scheduling algorithm for distributed real-time database systems.

## 2. Scheduling and Concurrency Control

Conventional real-time systems take into account timing constraints of individual tasks, but ignore data consistency problems. Also, they typically deal with simple tasks that have predictable data requirements. In real-time task scheduling, it is usually assumed that all tasks are preemptable. But preempting a task that uses a file resource in exclusive mode of writing may result in subsequent tasks reading inconsistent information.

In contrast to real-time systems, conventional database systems do not emphasize the notion of timing constraints or deadlines for transactions. The performance goal is to reduce response times of transactions by using a serialization order among conflicting transactions. Thus, when a decision of database scheduling is made, individual timing constraints are ignored. For example, most commonly used two-phase locking (2PL) protocol [Bern87] synchronizes concurrent data access of transactions by blocking and roll-back, and might violate timing constraints of transactions.

The goal of scheduling in RTDBS is twofold: to meet timing constraints and to enforce data consistency. Real-time task scheduling methods can be extended for real-time transaction scheduling, yet concurrency control protocols are still needed for operation scheduling to maintain data consistency. However, the integration of the two mechanisms in RTDBS is not straightforward. The general approach is to utilize existing concurrency control protocols, especially 2PL, and to apply time-critical transaction scheduling methods that favor more urgent transactions [Abb92, Gra92, Sha91, Son90]. Such approaches have the inherent disadvantage of being limited by the concurrency control protocol upon which they depend, since all existing concurrency control methods synchronize concurrent data access of transactions by the combination of two measures: blocking and roll-backs of transactions. Both are barriers to meeting time-critical schedules.

Concurrency control protocols induce a serialization order among conflicting transactions. For a concurrency control protocol to accommodate timing constraints of transactions, the serialization order it produces should reflect the priority of transactions. However, this is often hindered by the past execution history of transactions. A higher priority transaction may have no way to precede a lower priority transaction in the serialization order due to previous conflicts. For example, let  $T_H$  and  $T_L$  be two transactions with  $T_H$  having a higher priority. If  $T_L$  writes a data object  $x$  before  $T_H$  reads it, then the serialization order between  $T_H$  and  $T_L$  is determined as  $T_L \rightarrow T_H$ .  $T_H$  can never precede  $T_L$  in the serialization order as long as both reside in the execution history. Most of the current (real-time) concurrency control protocols resolve this conflict either by blocking  $T_H$  until  $T_L$  releases the writelock or by aborting  $T_H$  in favor of the

higher priority transaction  $T_H$ . Blocking a transaction may cause *priority inversion*. Priority inversion is said to occur when a high priority transaction is blocked by lower priority transactions [Sha91]. Priority inversion is contrary to the requirement of real-time scheduling. Aborting is also not desirable because it degrades the system performance and may lead to violations of timing constraints. Furthermore, some aborts can be wasteful when the transaction which caused the abort is also aborted due to another conflict.

Abbott and Garcia-Molina have proposed a restart-based 2PL [Abb92]. It incorporates priority information in lock setting so that transactions with higher priority will be given a preference. Whenever a higher priority transaction is in conflict with a lower priority transaction, the lower priority transaction will be aborted and restarted later on. One of the weaknesses of this scheme is the impact of restarts on scheduling other transactions to meet their timing constraints. Restarting a transaction could be very costly in terms of wasted resources, and a large number of restarts will increase the workload of the system and may cause other transactions to miss their deadlines. To reduce the number of restarts, the conditional restart protocol is proposed, in which the lower priority transaction will have to be restarted only if the slack time of the higher priority transaction is smaller than the remaining execution time of the lower priority transaction that holds the lock [Abb92]. There are few problems with this protocol. First, the effectiveness of this checking is greatly affected by the probability of blocking of the lower priority transaction. Second, the scheduler should have the information such as the execution time and slack time. In real-time database systems, such information is hard to get due to the dynamic nature of resource demands and data-dependent execution path of transactions. Furthermore, priority inversion and deadlock is still possible, although they have a lesser degree of impact.

For conventional database systems, it has been shown that optimal performance may be achieved by compromising blocking and roll-back [Yu90]. For RTDBS, we may expect similar results. Aborting a few low priority transactions and restarting them later may allow high priority transactions to meet their deadlines, resulting in improved system performance.

To improve the timeliness of RTDBS, it is highly desirable to take timing requirements of transactions into consideration for scheduling decisions. An optimistic approach [Kung81] is a possible way to achieve this goal. Due to its *validation phase* conflict resolution, it can be ensured that eventually discarded transactions do not abort other transactions and timing requirements of transaction are considered.

The key component of optimistic concurrency control protocols is the *validation phase* where a

transaction's destiny is determined. In the optimistic approach, write requests issued by transactions are not immediately processed on data objects but are deferred until the transaction submits a commit request, at which time the transaction must go through the validation phase. Because write operations effectively occur at commit time, the serialization order selected by an optimistic concurrency control protocol is the order in which the transactions actually commit through validation phase. Transaction validation can be performed in one of the two ways: *forward validation* and *backward validation*.

In optimistic concurrency control protocols that perform backward validation, the validating transaction either commits or aborts depending on whether it has conflicts with transactions that have already committed. Thus, this validation scheme does not allow us to take transaction characteristics into account. In forward validation, however, either the validating transaction or conflicting ongoing transactions can be aborted to resolve conflicts. This validation scheme is advantageous in RTDBS, because it may be preferable not to commit the validating transaction, depending on the timing characteristics of the validating transaction and the conflicting ongoing transactions. A number of real-time concurrency control methods based on the optimistic approach using forward validation have been studied [Har90, Hua91, Son92]. Few of them (e.g., OPT-WAIT protocol in [Har90]) incorporate priority-based conflict resolution mechanisms, such as *priority wait*, that make low priority transactions wait for conflicting high priority transactions to complete. However, this approach of detecting conflicts during validation phase degrades system predictability, since it may be too late to restart the transaction and meet the deadline.

A scheduling algorithm can solve this problem if a lock-based concurrency control protocol supports a mechanism to adjust dynamically the serialization order of active transactions. The integrated scheduler presented in the next section integrates a priority-based locking with an optimistic approach.

Another important issue that needs further study is a notion of "correct execution" in transaction processing, different from *serializability*. As observed by Bernstein [Bern87], serializability may be too strong as a correctness criterion for concurrency control in database systems with timing constraints, because of the limitation on concurrency.

Based on the argument that timing constraints may be more important than data consistency in RTDBS, attempts have been made to satisfy timing constraints by sacrificing database consistency temporarily to some degree [Lin89, Vrb88]. It is based on a new consistency model of real-time databases, in which maintaining *external data consistency* (values of data objects represent correct values of external world outside the database) has

priority over maintaining *internal data consistency* (no data that violates consistency constraints). Although in some applications weaker consistency is acceptable, a general-purpose consistency criterion that is less stringent than serializability has not yet been proposed. The problem is that temporary inconsistencies may affect active transactions and so the commitment of these transactions may still need to be delayed until the inconsistencies are removed; otherwise even committed transactions may need to be rolled back. However, in real-time systems, some actions are irreversible.

The use of semantic information in transaction scheduling and multiversion data is often proposed for RTDBS applications [Lin89, Son88, Song90]. Multiple versions are useful in situations that require the monitoring of data as values are changing with time. In such situations, the trends exhibited by the values of the data can be used to initiate proper actions [Kor90]. Examples include falling values in stock-market trading and rising temperature of a furnace in a nuclear reactor. Another objective of using multiple versions is to increase the degree of concurrency and to reduce the possibility of transaction rejection by providing a succession of views of data. There are several problems that must be solved in order to use multiple versions effectively. For example, the selection of old versions for a transaction must ensure the required consistency of the state seen by the transaction.

### 3. The Integrated Real-Time Locking Protocol

#### 3.1. Basic Concepts

An RTDBS is often used by applications such as tracking. Since we cannot predict how many objects need to be tracked and when they appear, we assume randomly arriving transactions. Each transaction is assigned an *initial priority* and a *start-timestamp* when it is submitted to the system. The initial priority can be based on the deadline and the criticalness of the transaction. The start-timestamp is appended to the initial priority to form the *actual priority* that is used in scheduling. When we refer to the priority of a transaction, we always mean the actual priority with the start-timestamp appended. Since the start-timestamp is unique, so is the priority of each transaction. The priority of transactions with the same initial priority is distinguished by their start-timestamps.

With two-phase locking and priority assignment, we can encounter the problem of priority inversion. What we need is a concurrency control algorithm that allows transactions to meet the timing constraints as much as possible without reducing the concurrency level of the system in the absence of any *a priori* information. The integrated real-time locking protocol presented in this paper meets these goals. It has the flavor of both locking and optimistic methods.

Transactions write into the database only after they are committed. By using a priority-dependent locking protocol, the serialization order of active transactions is adjusted dynamically, making it possible for transactions with higher priorities to be executed first so that higher priority transactions are not blocked by uncommitted lower priority transactions, while lower priority transactions may not have to be aborted even in face of conflicting operations. The adjustment of the serialization order can be viewed as a mechanism to support time-critical scheduling.

All transactions that can be scheduled are placed in a ready queue,  $R_Q$ . Only transactions in  $R_Q$  are scheduled for execution. When a transaction is *blocked*, it is removed from  $R_Q$ . When a transaction is *unblocked*, it is inserted into  $R_Q$  again, but may still be waiting to be assigned the CPU. A transaction is said to be *suspended* when it is not executing, but still in  $R_Q$ . When a transaction is doing I/O operations, it is blocked. Once it completes, it is usually unblocked.

The execution of each transaction is divided into three phases: the read phase, the wait phase and the write phase. During the read phase, a transaction reads from the database and writes to its local workspace. After it completes, it waits for its chance to commit in the wait phase. If it is committed, it switches into the write phase during which all its updates are made permanent in the database. A transaction in any of the three phases is called *active*. We take an approach of integrated schedulers in that it uses 2PL for read-write conflicts and the Thomas' Write Rule (TWR) for write-write conflicts. The TWR *ignores* a write request that has arrived late, rather than *rejects* it [Bern87].

In the protocol, there are various data structures that need to be read and updated in a consistent manner. Therefore we assume the existence of critical sections to guarantee that only one process at a time updates these data structures. We assume critical sections of various classes to group the various data structures and allow maximum concurrency. We also assume that each assignment statement of global data is executed atomically.

#### 3.2. Read Phase

The read phase is the normal execution of a transaction except that write operations are performed on private data copies in the local workspace of the transaction instead of on data objects in the database. We call such write operations *prewrites*, denoted by  $pw_T[x]$ . A write request from a transaction is performed by a prewrite operation. Since each transaction has its own local workspace, a prewrite operation does not write into the database, and if a transaction previously wrote a data object, subsequent read operations to the same data object retrieve the value from the local workspace.

The read-prewrite or prewrite-read conflicts between active transactions are synchronized during this phase by a *priority-based locking protocol*. Before a transaction can perform a read (resp. prewrite) operation on a data object, it must obtain the read (resp. write) lock on that data object first. A read (resp. write) lock on  $x$  by transaction  $T$  is denoted by  $rlock(T,x)$  (resp.  $wlock(T,x)$ ). If a transaction reads a data object that has been written by itself, it gets the private copy in its own workspace and no read lock is needed. In the rest of the paper, when we refer to read operations, we exclude such read operations because they do not induce any dependencies among transactions.

The locking protocol is based on the principle that higher priority transactions should complete before lower priority transactions. That is, if two transactions conflict, the higher priority transaction should precede the lower priority transaction in the serialization order. Using an appropriate CPU scheduling policy for RTDBS, a high priority transaction can be scheduled to commit before a low priority transaction in most cases. If a low priority transaction does complete before a high priority transaction, it is required to wait until it is sure that its commitment will not lead to the higher priority transaction being aborted.

Suppose active transaction  $T_1$  has higher priority than active transaction  $T_2$ . We have four possible conflicts and the transaction dependencies they require in the serialization order as follows:

(1)  $r_{T_1}[x], pw_{T_2}[x]$

The resulting serialization order is  $T_1 \rightarrow T_2$ , which satisfies the priority order, and hence it is not necessary to adjust the serialization order.

(2)  $pw_{T_1}[x], r_{T_2}[x]$

Two different serialization orders can be induced with this conflict;  $T_2 \rightarrow T_1$  with immediate reading, and  $T_1 \rightarrow T_2$  with delayed reading. Certainly, the latter should be chosen for priority scheduling. The delayed reading means that  $r_{T_2}[x]$  is blocked by the write lock of  $T_1$  on  $x$ .

(3)  $r_{T_2}[x], pw_{T_1}[x]$

The resulting serialization order is  $T_2 \rightarrow T_1$ , which violates the priority order. If  $T_2$  is in the read phase, it is aborted because otherwise  $T_2$  must commit before  $T_1$  and thus block  $T_1$ . If  $T_2$  is in its wait phase, avoid aborting  $T_2$  until  $T_1$  commits, in the hope that  $T_2$  gets a chance to commit before  $T_1$  commits. If  $T_1$  commits,  $T_2$  is aborted. But if  $T_1$  is aborted by some other conflicting transaction, then  $T_2$  is committed. With this policy, we can avoid unnecessary and useless aborts, while satisfying priority scheduling.

(4)  $pw_{T_2}[x], r_{T_1}[x]$

Two different serialization orders can be induced with this conflict;  $T_1 \rightarrow T_2$  with immediate reading, and  $T_2 \rightarrow T_1$  with delayed reading. If  $T_2$  is in its write phase, delaying  $T_1$  is the only choice. This blocking is not a serious problem for  $T_1$  because  $T_2$  is expected to finish writing  $x$  soon.  $T_1$  can read  $x$  as soon as  $T_2$  finishes writing  $x$  in the database, not necessarily after  $T_2$  completes the whole write phase. If  $T_2$  is in its read or wait phase, choose immediate reading.

As transactions are being executed and conflicting operations occur, all the information about the induced dependencies in the serialization order needs to be retained. To do this, we retain two sets for each transaction, *before-trans-set* and *after-trans-set*, and a count, *before-count*. The set *before-trans-set* (resp. *after-trans-set*) contains all the active lower priority transactions that must precede (resp. follow) this transaction in the serialization order. The *before-count* is the number of the higher priority transactions that precede this transaction in the serialization order. When a conflict occurs between two transactions, their dependency is set and their values of *before-trans-set*, *after-trans-set*, and *before-count* will be changed accordingly.

By summarizing what we discussed above, we define the real-time locking protocol as follows:

LP1. Transaction  $T$  requests a read lock on data object  $x$ .

```

for all transactions  $t$  with  $wlock(t,x)$  do
  if ( $priority(t) > priority(T)$ 
     or  $t$  is in write phase)
    /* Case 2, 4 */
  then deny the lock and exit;
endif
enddo
for all transactions  $t$  with  $wlock(t,x)$  do
  /* Case 4 */
  if  $t$  is in before-trans-set $_T$  then abort  $t$ ;
  else if ( $t$  is not in after-trans-set $_T$ )
    then
      include  $t$  in after-trans-set $_T$ ;
       $before-count_t := before-count_t + 1$ ;
    endif
  endif
enddo
grant the lock;

```

LP2. Transaction  $T$  requests a write lock on data object  $x$ .

```

for all transactions  $t$  with  $rlock(t,x)$  do
  if  $priority(t) > priority(T)$ 
  then /* Case 1 */

```

```

    if (T is not in after-trans-sett)
    then
        include t in after-trans-sett;
        before-countT := before-countT + 1;
    endif
else
    if t is in wait phase /* Case 3 */
    then
        if (t is in after-trans-setT)
        then abort t;
        else
            include t in before-trans-setT;
        endif
    else if t is in read phase
    then abort t;
    endif
endif
endif
enddo
grant the lock;

```

LP1 and LP2 are actually two procedures of the lock manager that are executed when a lock is requested. When a lock is denied due to a conflicting lock, the request is suspended until that conflicting lock is released. Then the locking protocol is invoked once again from the very beginning to decide whether the lock can be granted now. With our locking protocol, a data object may be both read locked and write locked by several transactions simultaneously.

### 3.3. Wait and Write Phases

The wait phase allows a transaction to wait until it can commit. A transaction can commit only if all transactions with higher priorities that must precede it in the serialization order are either committed or aborted. Since before-count is the number of such transactions, the transaction can commit only if its before-count becomes zero. A transaction in the wait phase may be aborted due to two reasons; if a higher priority transaction requests a conflicting lock, or if a higher priority transaction that must follow this transaction in the serialization order commits first. Once a transaction in the wait phase gets its chance to commit, i.e. its before-count goes to zero, it switches to the write phase and release all its read locks. The transaction is assigned a final-timestamp, which is the absolute serialization order.

Once a transaction is in the write phase, it is considered to be committed. All committed transactions are serialized by the final-timestamp order. Updates are made permanent to the database while applying Thomas' Write Rule (TWR) for write-write conflicts [Bern87]. After each operation the corresponding write lock is released.

## 4. Extension for Distributed Systems

In this section, we extend the integrated scheduler for distributed database systems. We do not consider recovery protocols for site or communication link failures in this paper. We assume that the execution of a distributed transaction *T* involves several sites and there is an agent (process) for *T* at all sites where *T* accesses data items. Each agent receives read and write requests from the home site of *T*, performs the operations and sends the results back to the home site. For a write request, the agent writes to the local workspace as in a centralized system. Similar to a centralized system, each agent maintains its local before-trans-set, after-trans-set and before-count for the transaction *T* it represents. Each site maintains its local read phase, wait phase, and write phase. Suppose each site has a unique site number. To produce unique priority for transactions originating at different sites, the home site number is appended as well as the start-timestamp to the initial priority of a transaction.

When a site receives the first read or write request of a transaction from its home site, a new agent is created for its execution. The locking protocol works in the same way as a centralized system using the local information at the site. To detect transactions that are doomed to be aborted, local before-trans-set and after-trans-set of each agent are sent back to the home agent when they are changed. Such information can be sent with the reply messages for read or write requests to reduce the communication overhead. The home agent of each transaction changes its before-trans-set and after-trans-set according to its local information and information sent back from other sites. In this way, transactions that are in both before-trans-set and after-trans-set of a transaction can be detected and aborted. The home agents of these aborted transactions may be at different sites. The common sites of two transactions are the sites at which both access some data. For one transaction to send an abort command to another transaction, it only has to send the command to one of their common sites. A common site must know the home site of the aborted transaction. Each agent can also send its local before-trans-set and after-trans-set after a certain number of changes in order to reduce the number of messages. There is a trade-off between communication cost and resource contention because it is desirable to abort a failed transaction as soon as possible without much overhead.

A transaction in a distributed database system is a logically atomic operation: it must be processed at all sites or none of them. Thus a commit or abort operation of a transaction must be processed at all sites where the transaction is executed to ensure a consistent termination. In a centralized system, a transaction in the wait phase is committed when its before-count becomes zero. In a distributed system, a transaction can be committed if before-count goes to zero at all sites where it has an

agent. This is not simple because local before-count of a transaction can be incremented by other high priority transactions at any time. If we freeze before-count for transactions in the wait phase, high priority transactions may be blocked by low priority transactions that are in the wait phase. This is obviously undesirable. Also, if we allow before-count to be changed all the time, a transaction can never be committed because the home agent may never be certain that the local before-counts of the transaction at all sites are zero. Our solution to this problem is a compromise – a short term freeze. A transaction in the wait phase can be switched into the *semicommitted* state, in which its before-count is frozen for a short period of time so that its before-count can only be decreased but not increased. High priority transactions may have to be blocked during this short period.

When a transaction has finished its read phase, the home agent sends a read phase termination message to all participating agents. This message also contains the site addresses of all agents who will participate in the termination protocol. Upon receiving this message, each agent switches into the wait phase and sends back an acknowledgment along with the local before-count. After receiving acknowledgments from all agents, the home agent then waits until the before-count of all agents becomes zero in order to initiate an attempt to commit. During this waiting period, each agent reports to the home site whenever its before-count switches from one to zero or from zero to one.

Each *commit attempt* is a variant of the two-phase commit (2PC) protocol. If before-counts of all agents including the home agent are zero, then the home agent initiates a commit attempt by sending a TRY-COMMIT message to all agents. If an agent finds its before-count to be zero, it switches into the semicommitted state and sends back a reply to the home agent. Only when all agents are in the semicommitted state, can the home agent decide to commit the transaction. Then the home agent sends a COMMIT message to all agents. The commit attempt will fail if at least one agent finds its before-count greater than zero. If it fails, then the failed agent sends an ATTEMPT-FAIL message to all other agents, which takes them back to the wait phase. Then the home agent has to wait for the next chance to initiate another commit attempt. This process goes on until either a commit attempt succeeds or the transaction is aborted by a high priority transaction.

Only the transactions in the wait phase can be switched into the semicommitted state. A transaction in the semicommitted state is different from one that is in the wait phase in two aspects. First, when an agent is in the semicommitted state, agents in the read phase for other transactions cannot increase its before-count. If the commit attempt fails, then this agent is changed back from the semicommitted state to the wait phase, and the

higher priority transaction is unblocked and the locking protocol is invoked again to decide if the lock is available now. In this way, high priority transactions may be blocked by uncommitted low priority transactions only for their commit attempt periods. Second, a transaction in the semicommitted state cannot be aborted. This may require high priority transactions to delay commit operations. The only reason to cause a transaction to be aborted in the semicommitted state is the commitment of another higher priority transaction whose before-trans-set contains this transaction.

To commit a transaction, the home agent sends the commit command and a final-timestamp to all agents. With serial commitment of conflicting transactions, a correct timestamp assignment policy only has to ensure that at each site, if transaction  $T_1$  commits and is assigned a final-timestamp before another transaction  $T_2$ , the final-timestamp of  $T_2$  will be greater than that of  $T_1$ . Each site should maintain the largest final-timestamp, *max\_ts*, ever assigned to a committed transaction on that site. When an agent switches into the semicommitted state, it sends this *max\_ts* to the home agent. The home agent assigns a final-timestamp that is greater than any *max\_ts* of the agents. This timestamp can always be made unique by appending either the transaction identifier or home site number to it.

## 5. Conclusions

In this paper, we have discussed characteristics and requirements of RTDBS such as correctness criteria, predictability, and timing constraints. We also addressed the issues associated with transaction scheduling and concurrency control for RTDBS. We focused on the operation scheduling aspect of time-critical scheduling, and introduced an integrated scheduler for conflict resolution, which integrates a priority-based locking with an optimistic approach.

In the integrated scheduler, the execution of a transaction is divided into read, wait, and write phases, in a way similar to optimistic concurrency control mechanisms. By delaying write operations of transactions, the restrictions imposed by past execution history on the serialization order can be relaxed. We introduced the priority-dependent locking protocol, which dynamically adjusts the serialization order of active transactions. We assume the priority of a transaction reflects its timing constraints such as deadline and criticalness.

This integrated scheduler incurs less blocking and aborts than the 2PL protocol with high priority scheme [Son92b]. Also it can reduce the number of late restarts and the number of conflicting transactions over that of the Wait-50 protocol [Har90], since it analyzes read-write conflicts among transactions as early as possible, and resolves them by giving preference to higher priority transactions without unnecessarily delaying conflict

resolution. It features the ability to allow transactions to meet their timing constraints as much as possible without reducing the concurrency level of the system. It works in applications that require handling of unpredictable data. The scheduler has been extended for distributed database systems. We are currently working on the implementation details of the distributed version of the scheduler. Other issues that need further investigation include scheme for efficient dynamic priority assignment, extensions to periodic transaction model, and methods to combine it with time-critical CPU scheduling.

## REFERENCES

- [Abb92] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *ACM Trans. on Database Systems*, vol. 17, no. 3, pp 513-560, Sept. 1992.
- [Bern87] Bernstein, P., V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.
- [Buc89] Buchmann, A. et al., "Time-Critical Database Scheduling: A Framework for Integrating Real-Time Scheduling and Concurrency Control," *5th Data Engineering Conference*, Feb. 1989.
- [Gra92] Graham, M., "Issues in Real-Time Data Management," *Journal of Real-Time Systems*, vol. 4, Sept. 1992, pp 185-202.
- [Har90] Haritsa, J., M. Carey, and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control," *Real-Time Systems Symposium*, Orlando, Florida, Dec. 1990.
- [Hua91] Huang, J., J. Stankovic, K. Ramamritham, and D. Towsley, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes," *VLDB Conference*, Sept. 1991.
- [Kung81] Kung, H. and J. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. on Database Syst.*, vol. 6, no. 2, pp 213-226, June 1981.
- [Kor90] Korth, H., "Triggered Real-Time Databases with Consistency Constraints," *16th VLDB Conference*, Brisbane, Australia, Aug. 1990.
- [Lin89] Lin, K., "Consistency issues in real-time database systems," *Proc. 22nd Hawaii Intl. Conf. System Sciences*, Hawaii, Jan. 1989.
- [Sha91] Sha, L., R. Rajkumar, S. H. Son, and C. Chang, "A Real-Time Locking Protocol," *IEEE Transactions on Computers*, vol. 40, no. 7, July 1991, pp 793-800.
- [Son88] Son, S. H., guest editor, *ACM SIGMOD Record 17, 1*, Special Issue on Real-Time Database Systems, March 1988.
- [Son90] Son, S. H. and J. Lee, "Scheduling Real-Time Transactions in Distributed Database Systems," *7th IEEE Workshop on Real-Time Operating Systems and Software*, Charlottesville, Virginia, May 1990, pp 39-43.
- [Son90b] Son, S. H. and C. Chang, "Performance Evaluation of Real-Time Locking Protocols using a Distributed Software Prototyping Environment," *10th International Conference on Distributed Computing Systems*, Paris, France, June 1990, pp 124-131.
- [Son91] Son, S. H., C. Iannacone, and M. Poris, "RTDB: A Real-Time Database Manager for Time-Critical Applications," *Euromicro Workshop on Real-Time Systems*, Paris, France, June 1991, pp 207-214.
- [Son92] Son, S. H., J. Lee, and Y. Lin, "Hybrid Protocols using Dynamic Adjustment of Serialization Order for Real-Time Concurrency Control," *Journal of Real-Time Systems*, vol. 4, Sept. 1992, pp 269-276.
- [Son92b] Son, S. H., S. Park, and Y. Lin, "An Integrated Real-Time Locking Protocol," *Eighth IEEE International Conference on Data Engineering*, Phoenix, Arizona, February 1992, pp 527-534.
- [Son92c] Son, S. H., R. Cook, J. Lee, and H. Oh, "New Paradigms for Real-Time Database Systems," in *Real-Time Programming*, K. Ramamritham and W. Halang (Editors), Pergamon Press, 1992.
- [Song90] Song, X. and J. Liu, "Performance of Multiversion Concurrency Control Algorithms in Maintaining Temporal Consistency," *COMP-SAC '90*, pp 132-139, October 1990.
- [Stan88] Stankovic, J., "Misconceptions about Real-Time Computing," *IEEE Computer 21*, 10, October 1988, pp 10-19.
- [Vrb88] Vrbsky, S. and K. J. Lin, "Recovering Imprecise Transactions with Real-Time Constraints," *Symp. Reliable Distributed Systems*, Oct. 1988, pp 185-193.
- [Yu90] Yu, P. and D. Dias, "Concurrency Control using Locking with Deferred Blocking," *6th Intl. Conf. Data Engineering.*, Los Angeles, Feb. 1990, pp 30-36.