

**Hybrid Protocols Using Dynamic  
Adjustment of Serialization Order  
for Real-Time Concurrency Control**

Sang H. Son,  
Juhnyoung Lee  
and Yi Lin

Computer Science Report No. TR-92-06  
March 13, 1992

# Hybrid Protocols Using Dynamic Adjustment of Serialization Order for Real-Time Concurrency Control

*Sang H. Son*  
*Juhnyoung Lee*  
*Yi Lin*

Department of Computer Science  
University of Virginia  
Charlottesville, VA 22903

## Abstract

A real-time database system differs from a conventional database system because in addition to the consistency constraints of the database, timing constraints of individual transaction need to be satisfied. Various real-time transaction scheduling protocols have been proposed which employ either a pessimistic or an optimistic approach to concurrency control. In this paper, we present new real-time transaction scheduling protocols which employ a hybrid approach, i.e., a combination of both pessimistic and optimistic approaches. These protocols make use of a new conflict resolution scheme called dynamic adjustment of serialization order, which supports priority-driven scheduling, and avoids unnecessary aborts.

Key words: real-time database system, scheduling, concurrency control, deadline

---

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

## 1. Introduction

A *real-time database system* (RTDBS) differs from a conventional database system because in addition to the consistency constraints of the database, timing constraints of individual transaction need to be satisfied. In order to provide a real-time response for queries and updates while maintaining the consistency of data, *real-time concurrency control* should involve efficient integration of the ideas from both database concurrency control and real-time scheduling. Various real-time concurrency control protocols have been proposed which employ either a pessimistic or an optimistic approach to concurrency control.

Most of the initial work in real-time concurrency control has been conducted on utilizing *Two-Phase Locking* (2PL) as the base of real-time concurrency control. The two-phase locking is termed as being *pessimistic*, because it, in anticipation of data conflicts, tends to delay the operations in order to avoid aborting them later. However, the very idea of delaying an operation is opposed to real-time systems. Besides, the degree of concurrency is low in the two-phase locking based protocols because concurrent read and write locks on the same data object by several transactions are not possible. Furthermore, two-phase locking has some inherent problems such as the possibility of deadlocks and unpredictable blocking times, which are serious problems for real-time concurrency control.

*Optimistic Concurrency Control* (OCC) is a natural alternative. An optimistic scheduler aggressively schedules all operations, hoping that nothing will go wrong, such as a non-serializable execution. Later when the transaction is ready to commit, conflicts are checked for and a conflict resolution scheme is then applied to resolve the conflicts, if any. Ideally an optimistic approach has the properties of non-blocking and deadlock freedom, which make it suitable to real-time transaction processing. In addition, it has a potential for high degree of parallelism. However, the use of aborting for conflict resolution in optimistic schedulers results in a problem that transactions can end up aborting after having paid for most of the transaction's execution. This problem can particularly be serious for real-time transaction scheduling, where timing constraints of transactions should be satisfied. For conventional database systems, it has been shown that optimal performance can be achieved by combining blocking and aborting [Yu90]. We expect the same with RTDBS.

In this paper, we present two hybrid real-time concurrency control protocols which combine pessimistic and optimistic approaches to concurrency control in order to control blocking and aborting in a more effective manner. Concurrency control protocols induce a serialization order among conflicting transactions. For a concurrency control protocol to accommodate the 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,  $T_H$  and  $T_L$  are two transactions with  $T_H$  having a higher priority. If  $T_L$  writes a data object  $x$  before  $T_H$  read 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  $T_L$  in favor of the higher priority transaction  $T_H$ . Blocking of a higher priority transaction due to a lower priority transaction 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 aborted due to another conflict. The objective of our hybrid approach is to avoid such unnecessary blocking and aborting.

The first protocol, called integrated real-time locking, is a combination of OCC and locking. This protocol uses a priority-based locking mechanism to support real-time scheduling by adjusting the serialization order dynamically in favor of high priority transactions. Also this protocol

uses the phase-dependent control of optimistic approach to support dynamic adjustment of serialization order.

The second protocol presented in this paper is a combination of OCC and timestamp ordering. This protocol also takes the advantage of the phase-dependent control of optimistic approach to apply the notion of dynamic timestamp allocation [Bay82] and dynamic adjustment of serialization order using timestamp interval [Bok87], with which the ability of early detection and resolution of nonserializable execution is improved, and unnecessary aborts are avoided. Furthermore, since this protocol is based on OCC with forward validation scheme in which the validation test is conducted against active transactions in their read phase, it can be easily extended to support real-time scheduling by adopting priority-based conflict resolution schemes such as priority abort, priority sacrifice, and priority wait [Har90, Hua91].

The remainder of the paper is organized as follows. Section 2 summarizes recent related work on the scheduling problem in RTDBS. In Section 3, we present our first protocol, integrated real-time locking. First, we describe the basic concepts and the protocol. Then we present an argument on the correctness of the protocol, and provide an example to show how the protocol works. Our second protocol, a real-time OCC protocol using timestamp, is described in Section 4. Each component of the protocol such as OCC with forward validation scheme, dynamic timestamp allocation, and dynamic adjustment of serialization order using timestamp interval is explained in each subsection. Finally, concluding remarks appear in Section 5.

## 2. Related Work

A RTDBS is a transaction processing system designed to handle workloads where transactions have completion deadlines. The objective of such a system is to meet the deadlines. The real-time performance of an RTDBS depends on several factors such as the database system organization, the underlying processors and disk speeds. For a given system configuration, however, the primary determinants for the real-time performance are the policies used for scheduling transaction accesses to system resources, because these policies determine when service is provided to a transaction.

Recently, research on the scheduling problem in real-time database systems has been active [Abb88, Buc89, Coo91, Har90, Hua90, Hua91, Lin90, Sha91, Son90]. As mentioned earlier, most of the current real-time concurrency control schemes are based on two-phase locking [Abb88, Hua90, Sha91]. Abbott and Garcia-Molina [Abb88] described a group of lock-based real-time concurrency control schemes for scheduling soft real-time transactions, and evaluated the performance of those protocols through simulation.

Some inherent problems of two-phase locking such as the possibility of deadlocks and unpredictable blocking times are serious for real-time transaction scheduling. In addition, a *priority inversion* can occur when a lower priority transaction blocks the execution of a higher priority transaction. Sha et al. [Sha91] proposed a conservative real-time concurrency control scheme called *priority ceiling*, which prevents deadlocks and transitive blocking.

Huang et al. [Hua90] developed and evaluated a group of real-time protocols for handling CPU scheduling, data conflict resolution, deadlock resolution, transaction wakeup, and transaction restart. Their study is based not on simulation but by actual implementation on a testbed system called RT-CARAT. They concluded that CPU scheduling protocols have a significant impact on the performance of RTDBS, that they dominate all other protocols, and that the overhead incurred in locking is non-negligible and cannot be ignored in real-time concurrency control analysis.

Recently, real-time concurrency control protocols based on optimistic method have been proposed and studied [Har90, Hua91, Lin90, Coo91]. Haritsa et al. [Har90] proposed a group of optimistic real-time concurrency control protocols and evaluated them on a simulation model.

They have also conducted a study on the relative performance of locking-based protocols and optimistic protocols, and concluded that OCC protocols outperform two-phase locking-based protocols over a wide range of system utilization. Huang et al. [Hua91] also conducted a similar performance study of real-time OCC protocols, but on a testbed system, not through simulation. They examined the overall effects and the impact of the overheads involved in implementing real-time OCC on the testbed system. Their experimental results contrast with the results in [Har90], showing that OCC may not always outperform a two-phase locking-based protocol which aborts the lower priority transaction when a conflict occurs. They pointed out the fact that the physical implementation schemes have a significant impact on the performance of real-time OCC protocol.

### 3. Integrated Real-Time Locking Protocol

#### 3.1. Introduction

The first protocol, *Integrated Real-Time Locking*, combines locking and OCC. By using a priority-dependent locking protocol, the serialization order of active transactions is adjusted dynamically, making it possible for transactions with higher priority to be executed first so that higher priority transactions are never blocked by uncommitted lower priority transactions, while lower priority transactions may not have to be aborted even in the face of a conflict. The adjustment of the serialization order can be considered as a mechanism to support real-time scheduling.

This protocol is an integrated protocol because it uses different solutions for read/write (rw) and write/write (ww) synchronization, and integrates the solutions to the two subproblems to yield a solution to the entire problem.

The protocol is similar to OCC in the sense that each transaction has three phases, but unlike the optimistic method, there is no validation phase. This protocol's three phases are read, wait, and write phases. The read phase is similar to that of OCC wherein a transaction reads from the database and writes to its local workspace. In this phase, however, conflicts are also resolved by using transaction priority. While other optimistic real-time concurrency control protocols resolve conflicts in the validation phase, this protocol resolves them in the read phase. In the wait phase, a transaction waits for its chance to commit. Finally, in the write phase, updates are made permanent to the database.

The following is the outline of a transaction execution:

```
transaction = { tbegin();
               read phase;
               twait();
               twrite();
               }.
```

Each procedure will be described in detail later in this section. In this protocol, there are various data structures that need to be read and updated in a consistent manner. Therefore we assume critical sections to group the various data structures to guarantee mutual exclusion and to allow maximum concurrency. We also assume that each assignment statement of global data is executed atomically. Table 1 summarizes some useful notations.

The environment we assume for the implementation is a single processor with 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 criticality 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

notation	meaning
$read\_trset$	set of transactions in the read phase
$wait\_trset$	set of transactions in the wait phase
$write\_trset$	set of transactions in the write phase
$s\_count$	serialization order count
$ts(T)$	final-timestamp of transaction $T$
$priority(T)$	priority value of transaction $T$
$r_i[x]$	transaction $i$ reads data object $x$ .
$w_i[x]$	transaction $i$ writes data object $x$ .
$pw_i[x]$	transaction $i$ prewrites data object $x$ .
$rlock(T, x)$	transaction $T$ holds a read lock on data object $x$
$wlock(T, x)$	transaction $T$ holds a write lock on data object $x$

Table 1. Notations

distinguished by their start-timestamps.

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 operation, it is blocked. Once it completes, it is usually unblocked.

### 3.2. Read phase

The read phase is the normal execution of a transaction except that all writes are on private data copies in the local workspace of the transaction instead of on data objects in the database. Such write operations are called *prewrites*. The prewrites are useful when a transaction is aborted, in which case the data in the local workspace is simply discarded. No rollback is required.

In this phase read-prewrite and prewrite-read conflicts are resolved using a priority based locking protocol. A transaction must obtain the corresponding lock before it reads or prewrites. According to the priority locking protocol, higher priority transactions must complete before lower priority transactions. 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  $T_H$  and  $T_L$  are two active transactions and  $T_H$  has higher priority than  $T_L$ , there are four possible conflicts as follows.

- (1)  $r_{T_H}[x], pw_{T_L}[x]$

The resulting serialization order is  $T_H \rightarrow T_L$ , hence satisfies the priority order, and does not need to adjust the serialization order.

- (2)  $pw_{T_H}[x], r_{T_L}[x]$

Two different serialization orders can be induced with this conflict;  $T_L \rightarrow T_H$  with immediate reading, and  $T_H \rightarrow T_L$  with delayed reading. Certainly, the latter should be chosen for priority scheduling. The delayed reading in this protocol means blocking of  $r_{T_L}[x]$  by the writelock of  $T_H$  on  $x$ .

(3)  $r_{T_L}[x]$ ,  $pw_{T_H}[x]$

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

(4)  $pw_{T_L}[x]$ ,  $r_{T_H}[x]$

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

As transactions are being executed and conflicting operations occur, all the information pertaining to the induced dependencies in the serialization order needs to be retained. In order to maintain this information, we associate the following with each transaction; two sets, *before\_trset* and *after\_trset*, and a count, *before\_cnt*. The *before\_trset* (respectively, *after\_trset*) of a transaction contains all the active lower priority transactions that must precede (respectively, follow) this transaction in the serialization order. The *before\_cnt* of a transaction 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 determined and then the values of their *before\_trset*, *after\_trset*, and *before\_cnt* are changed accordingly.

By summarizing what we discussed above, we define the 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  $before\_trset_T$  then abort  $t$ ;
  else if ( $t$  is not in  $after\_trset_T$ )
    then
      include  $t$  in  $after\_trset_T$ ;
       $before\_cnt_t := before\_cnt_t + 1$ ;
    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  $after\_trset_t$ )
        then
          include  $t$  in  $after\_trset_t$ ;

```

```

        before_cntT := before_cntT + 1;
    endif
else
    if t is in wait phase /* Case 3 */
    then
        if (t is in after_trsetT)
        then abort t;
        else
            include t in before_trsetT;
        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. Figure 1 shows the lock compatibility tables in which the compatibilities are expressed by possible actions taken when conflicts occur. The compatibility depends on the priorities of the transactions holding and requesting the lock and the phase of the lock holder as well as the lock types. Even with the same lock types, different actions may be taken, depending on the priorities of the lock holder and the lock requester. Therefore a table entry may have more than one block reflecting the different possible actions.

With our locking protocol, a data object may be both read locked and write locked by several transactions simultaneously. Unlike 2PL, locks are not classified simply as shared locks and exclusive locks. Figure 2 summarizes the lock compatibility of 2PL with the *High Priority* scheme in which high priority transactions never block for a lock held by a low priority transaction [Abb88]. By comparing Figure 1 with Figure 2, it is obvious that our locking protocol is much more flexible, and thus incurs less blocking and fewer aborts. Note that in Figure 1, aborting low priority transactions in the wait phase is also included. In our locking protocol, a high priority transaction is never blocked or aborted due to conflict with an uncommitted lower priority transaction. The probability of aborting a lower priority transaction should be less than that in 2PL under the same conditions. An analytical model may be used to estimate the exact probability, but that is beyond the scope of this paper.

Transactions are released for execution as soon as they arrive. The following procedure is executed when a transaction *t* is started:

```

tbegin = (
    before_trset := ∅;
    after_trset := ∅;
    before_cnt := 0;
    include t in read_trset;
    put t in the R_Q;
).

```

Now *t* is in the read phase. When it tries to read or prewrite a data object, it requests the lock.

lock requested	lock held	
	read	write
read		
write		

lock requester has lower priority

lock requested	lock held	
	read	write
read		
write		

lock requester has higher priority




-  lock granted
-  lock requester blocked
-  lock holder aborted

Figure 1 Lock Compatibility Table

lock requested	lock held	
	read	write
read		
write		

lock requester has lower priority

lock requested	lock held	
	read	write
read		
write		

lock requester has higher priority




-  lock granted
-  lock requester blocked
-  lock holder aborted

Figure 2 Lock Compatibility Table of 2PL

The lock may be granted or not according to the locking protocol. Transactions may be aborted when lock requests are processed. To abort a transaction  $T$ , the following procedure is called:

```

tabort = (
  release all locks;
  for all transactions  $t$  in  $after\_trset_T$  do
     $before\_cnt_t := before\_cnt_t - 1$ ;
    if  $before\_cnt_t = 0$  and  $t$  is in wait phase;
    then unblock  $t$ ;
    endif
  enddo

  if  $T$  is in read phase
  then delete  $T$  from  $read\_trset$ ;
  else if  $T$  is in write phase
  then delete  $T$  from  $write\_trset$ ;
  else if  $T$  is in wait phase
  then delete  $T$  from  $wait\_trset$ ;
  endif
  endif
  endif
).

```

### 3.3. Wait Phase

The wait phase allows a transaction to wait until it can commit. A transaction in the wait phase can commit if all transactions with higher priority that must precede it in the serialization order, are either committed or aborted. Since the *before\_cnt* of a transaction is the number of such transactions, the transaction can commit only if its *before\_cnt* 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.

Once a transaction in its wait phase finds a chance to commit, then it commits and switches to its write phase and releases all readlocks. The transaction is assigned a final timestamp which is the absolute serialization order. The wait procedure for transaction  $T$  is as follows:

```

twait = (
  include  $T$  in  $wait\_trset$ ;
  delete  $T$  from  $read\_trset$ ;
  waiting := TRUE;
  while(waiting) do
    if ( $before\_cnt_T = 0$ )
      then /* switching into write phase */
        include  $T$  in  $write\_trset$ ;
        delete  $T$  from  $wait\_trset$ ;
         $ts(T) := s\_count$ ;
         $s\_count := s\_count + 1$ ;
        for all  $t$  in  $before\_trset_T$  do
          if  $t$  is in read phase or wait phase
          then abort  $t$ ;
          endif
        enddo
      enddo

      waiting := FALSE
    enddo
  endwhile
)

```

```

        else block;
      endif
    enddo
  release all read locks;
  for all  $t$  in  $after\_trset_T$  do
    if  $t$  is in read phase or wait phase
      then  $before\_cnt_t := before\_cnt_t - 1$ ;
      if ( $before\_cnt_t = 0$  and  $t$  is in wait phase)
        then unblock  $t$ ;
      endif
    endif
  enddo
).

```

After a transaction commits, all the transactions in its *before\_trset* need to be aborted because they must commit, if they can, before this transaction. The critical section of class 1 in the procedure guarantees that transactions cannot switch into the write phase concurrently, and once a transaction is committed and assigned a final-timestamp, no transaction in its *before\_trset* can commit. Note that LP1 is also in the critical section of the same class. This achieves mutual exclusion on *before\_cnt* and *write\_trset*. The critical section of class 3 in the procedure has the same effect as that of the critical section in the procedure *tabort*.

### 3.4. Write Phase

Once a transaction is in the write phase, it is considered to be committed. All committed transactions can be serialized by the final-timestamp order. In the write phase, the only work of a transaction is making all its updates permanent in the database. Data items are copied from the local workspace into the database. After each write operation, the corresponding write lock is released. The Thomas' Write Rule (TWR) [Bern87] is applied here. The write requests of each transaction are sent to the data manager, which carries out the write operations in the database. Transactions submit write requests along with their final-timestamps. The write procedure for transaction  $T$  is as follows:

```

twrite = (
  for all  $x$  in  $wlock(T, x)$  do
    for all  $t$  in write phase do
      if  $wlock(t, x)$  and  $ts(t) < ts(T)$ 
        then release  $t$ 's write lock on  $x$ ;
      endif
    enddo
    send write request on  $x$  and wait for
    acknowledgement;
    if (acknowledgement is OK)
      then release the write lock on  $x$ ;
    else abort  $T$ ;
    endif
  enddo
  delete  $T$  from  $R\_Q$ ;
).

```

For each data object, write requests are sent to the data manager only in ascending timestamp order. After a write request on data object  $x$  with timestamp  $n$  is issued to the data manager,

no other write request on  $x$  with a timestamp smaller than  $n$  will be sent. The write requests are buffered by the data manager. The data manager can work with the first-come-first-serve policy or always select the write request with the highest priority to process. When a new request arrives, if there is another buffered write request on the same data object, the request with the smaller timestamp is discarded. Therefore for each data object there is at most one write request in the buffer. This, in conjunction with the procedure *twrite*, guarantees TWR.

### 3.5. Correctness

Having described the basic concepts and the protocol, we now prove the correctness of the protocol. First, we give the simple definitions of *history* and *serialization graph* (SG). For the formal definitions, readers are referred to [Bern87]. A history is a partial order of operations that represents the execution of a set of transactions. Any two conflicting operations must be comparable. Let  $H$  be a history. The *serialization graph* for  $H$ , denoted by  $SG(H)$ , is a directed graph whose nodes are committed transactions in  $H$  and whose edges are all  $T_i \rightarrow T_j$  ( $i \neq j$ ) such that one of  $T_i$ 's operations precedes and conflicts with one of  $T_j$ 's operations in  $H$ . To prove a history  $H$  serializable, we only have to prove that  $SG(H)$  is acyclic [Bern87].

**Theorem 1:** Every history  $H$  produced by the protocol is serializable.

**Proof:** Let  $T_1$  and  $T_2$  be two committed transactions in a history  $H$  produced by the algorithm. We argue that if there is an edge  $T_1 \rightarrow T_2$  in  $SG(H)$ , then  $ts(T_1) < ts(T_2)$ . Since  $T_1 \rightarrow T_2$ , the two must have conflicting operations. There are three cases.

Case 1:  $w_1[x] \rightarrow w_2[x]$

Suppose  $ts(T_2) < ts(T_1)$ . Therefore  $T_2$  enters into the write phase before  $T_1$ . If  $w_1[x]$  is sent to the data manager first,  $T_2$ 's write lock on  $x$  must be released before  $w_1[x]$  is sent to the data manager. If  $w_2[x]$  is sent to the data manager first, it will either be processed before  $w_1[x]$  is sent to the data manager, or be discarded when the data manager receives  $w_1[x]$ , because  $w_2[x]$  has a smaller timestamp. Therefore  $w_1[x]$  is never processed before  $w_2[x]$ . Such conflict is impossible. A contradiction.

Case 2:  $r_1[x] \rightarrow w_2[x]$

If  $T_2$  holds the write lock on  $x$  when  $T_1$  requests the read lock, we must have  $priority(T_1) > priority(T_2)$  and  $T_2$  is not in the write phase, because otherwise  $T_1$  would have been blocked by LP1. By LP1,  $T_2$  is in *after\_trset* $_{T_1}$ .  $T_2$  will not switch into the write phase before  $T_1$  does, because *before\_cnt* $_{T_2}$  cannot be zero with  $T_1$  still in the read or wait phase. Therefore  $ts(T_1) < ts(T_2)$ . If  $T_1$  holds read lock on  $x$  when  $T_2$  requests the write lock, by LP2, we have either  $T_2$  is in *after\_trset* $_{T_1}$  or  $T_1$  is in *before\_trset* $_{T_2}$ , depending on the priorities of the two transactions. In either case,  $T_1$  must commit before  $T_2$ . Hence we also have  $ts(T_1) < ts(T_2)$ .

Case 3:  $w_1[x] \rightarrow r_2[x]$

Since  $T_1$  is already in the write phase before  $T_2$  reads  $x$ , we must have  $ts(T_1) < ts(T_2)$ .

Suppose there is a cycle  $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$  in  $SG(H)$ . By the above argument, we have  $ts(T_1) < ts(T_2) < \dots < ts(T_n) < ts(T_1)$ . This is impossible. Therefore no cycle can exist in  $SG(H)$  and the algorithm only produces serializable histories.  $\square$

**Theorem 2:** There is no mutual deadlock under the real-time locking protocol.

**Proof:** In the algorithm, a high priority transaction can be blocked by a low priority transaction only if the low priority transaction is in the write phase. Suppose there is a cycle in the wait-for graph (WFG),  $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$ . For any edge  $T_i \rightarrow T_j$  in the cycle, if  $priority(T_i) > priority(T_j)$ ,  $T_j$  must be in the write phase, thus it cannot be blocked by any other

transactions and cannot appear in the cycle. Therefore we must have  $priority(T_i) < priority(T_j)$  and thus  $priority(T_1) < priority(T_2) < \dots < priority(T_n) < priority(T_1)$ . This is impossible. Hence a deadlock cannot exist.  $\square$

### 3.6. An Example

In this section, we give a simple example to show how the protocol works. The example is depicted in Figure 3. A solid line at a low level indicates that the corresponding transaction is doing I/O operation due to a page fault or in the write phase. A dotted line at a low level indicates that the corresponding transaction is either suspended or blocked, and not doing any I/O operation either. A line raised to a higher level indicates that the transaction is executing. The absence of a line indicates that the transaction has not yet arrived or has already completed.

There are three transaction in the example.  $T_1$  has the highest priority and  $T_3$  has the lowest.  $T_3$  arrives at time  $t_0$  and reads data object  $a$ . This causes a page fault. After the I/O operation, it prewrites  $b$ . Then  $T_2$  comes in at time  $t_1$  and preempts  $T_3$ . At time  $t_2$  it reads  $c$  and causes another page fault. So it is blocked for the I/O operation and  $T_3$  executes. After  $T_3$  prewrites  $d$ ,  $T_2$  finishes I/O and preempts  $T_3$  again. It prewrites  $d$  which is only write locked by  $T_3$ . At time  $t_3$ ,  $T_1$  arrives and preempts  $T_2$ .  $T_1$  first reads  $d$ , which is write locked by both  $T_2$  and  $T_3$ . Therefore,  $before\_trset_{T_1}$  becomes  $\{T_2, T_3\}$  and both  $before\_cnt_{T_2}$  and  $before\_cnt_{T_3}$  become 1. Then  $T_1$  reads  $b$ , which is write locked by  $T_3$ . Since  $T_3$  is already in  $before\_trset_{T_1}$ , nothing is changed. Then  $T_1$  prewrites  $b$  and prewrites  $d$ . Since these two data objects are not read locked by any other transactions, the write locks are granted to  $T_1$  directly. At time  $t_4$ ,  $T_1$  switches into the write phase. Both  $before\_cnt_{T_2}$  and  $before\_cnt_{T_3}$  go back to 0. Now  $T_2$  should be executed, but it needs to read  $b$ , which is being write locked by  $T_1$ ; hence  $T_3$  is executed instead. It reads  $c$ , which is read locked by  $T_2$ . At time  $t_5$ ,  $T_1$  finishes writing  $b$  and releases the write lock so that  $T_2$  can preempt  $T_3$  to continue its work. It reads  $b$ , which is write locked by  $T_3$ . Now  $before\_trset_{T_2}$  becomes  $\{T_3\}$  and  $before\_cnt_{T_3}$  becomes 1. After  $T_2$  prewrites  $b$ , it switches into the write phase and  $before\_cnt_{T_3}$  becomes 0 again. Then  $T_3$  executes and also

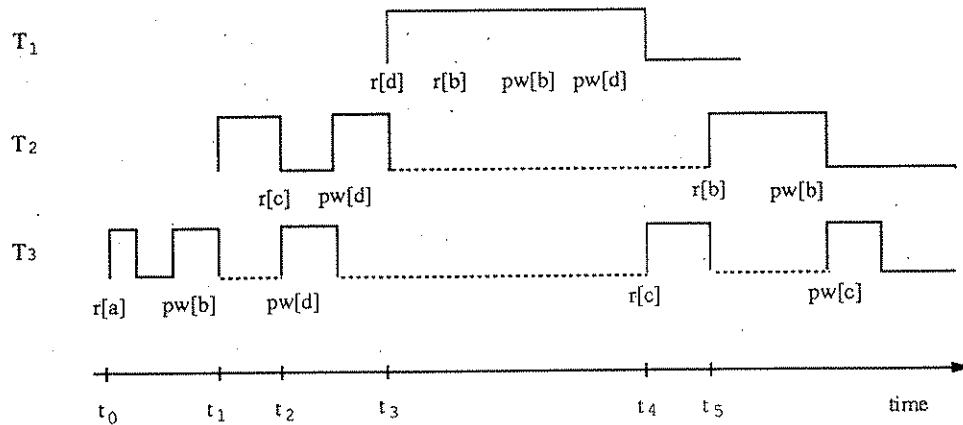


Figure 3 An Example

switches into write phase after prewriting  $c$ .

In this example,  $T_1$ , which is supposed to be the most urgent transaction, finishes first although it is the last to arrive.  $T_3$ , which is supposed to be the least urgent one, is the last one to commit. None of the three transactions need to be aborted. Assume we use 2PL in the above example. When a high priority transaction requests a lock which is held by a low priority transaction, we either let the high priority transaction to wait or abort the low priority transaction. Suppose we choose the first alternative, then both  $T_1$  and  $T_2$  would be blocked by  $T_3$  because  $T_3$  holds a write lock on  $d$ . If we choose the second alternative,  $T_3$  will be aborted by  $T_2$  when  $T_2$  prewrites  $d$  and then  $T_2$  will be aborted by  $T_1$  when  $T_1$  reads  $d$ .

## 4. A Real-Time Optimistic Concurrency Control Using Timestamp

### 4.1. Introduction

This protocol is a combination of OCC and timestamp ordering. As discussed earlier, one major problem of OCC is *wasted resources* and *time*. Because OCC is usually dependent on restart-based conflict resolution, and data conflicts are detected and resolved only at transaction commit time, i.e., validation phase, transactions can end up aborting after having paid resources and time for most of the transaction's execution. The situation becomes even worse when the transaction is restarted because previously performed work has to be redone. The problem of the wasted resources and time becomes even more serious for real-time transaction scheduling, because it reduces the chances of meeting deadlines of transactions.

Another problem of OCC is *unnecessary aborts*. When a transaction is ready to commit, it is checked whether this transaction is involved in any nonserializable execution. This validation test is usually conducted based on the read sets and write sets of transactions, rather than on actual execution order. Hence sometimes the validation process using the read sets and write sets erroneously concludes that a nonserializable execution has occurred, though it has not in actual execution. For example, in a forward validation [Haed84, Har90, Hua91], a conflict is said to occur between a validating transaction  $T_i$  and an active transaction  $T_j$  if

$$\text{readset}(T_j) \cap \text{writeset}(T_i) \neq \emptyset,$$

and conflicts are resolved using aborts. However, as we will see later, this validation condition can sometimes abort transactions whose execution does not necessarily violate serializability. Thus this validation condition solely based on read sets and write sets can induce unnecessary aborts. The problem of unnecessary aborts is serious because it results in waste of resources and time.

The problem of wasted resources is partly remedied with forward validation scheme, because the validation test is conducted against active transactions in their read phase [Har90, Hua91]. Early detection and resolution of conflicts can reduce the wasted resources and time. Our protocol presented here also utilizes OCC with forward validation to take the advantage of the early detection and resolution of nonserializable executions. Furthermore, this protocol employs the notion of dynamic timestamp allocation [Bay82] and dynamic adjustment of serialization order using timestamp interval [Bok87], with which the ability of early detection and resolution of nonserializable execution is improved, and unnecessary aborts are avoided as much as possible.

It should be pointed out that the proposed protocol is completely different from OCC protocols that use timestamp-based validation technique such as one proposed in [Car87]. In those protocols, timestamp is used for set intersection operation to make the complexity of validation test independent of the number of committed transactions, and does not provide any ability to adjust the serialization order dynamically.

Before we provide a procedural description of the protocol, we explain briefly the idea of each component of the protocol.

### (1) OCC with Forward Validation

This protocol is an OCC protocol. The execution of each transaction in this protocol consists of three phases; read, validation, and write phase, as in other OCC protocols. This protocol uses forward validation scheme, rather than backward validation scheme. As mentioned earlier, in forward validation, the validation test is conducted against active transactions in their read phase. When a conflict is detected, either the validating transaction or the conflicting active transaction can be aborted. It is this property that makes OCC with forward validation flexible and easily combined with the priority mechanism. The phase-dependent control of OCC and the property of forward validation scheme provide a framework for the following components of the protocol.

### (2) Categories of Conflicting Transactions

Since this protocol uses forward validation that is conducted against active transactions, when a validation test is performed for a transaction, say  $T_v$ , active transactions in the system can be divided into several sets according to their execution history (with respect to that of  $T_v$ ). First, the set of the active transactions are divided into two sets; *conflicting set*, that contains transactions in conflict with  $T_v$ , and *nonconflicting set*, that contains transactions not in conflict with  $T_v$ . The conflicting set can be further divided into two sets; *Reconcilably Conflicting* (RC) set and *Irreconcilably Conflicting* (IC) set. Transactions in the RC set are in conflict with  $T_v$ , but the conflicts are reconcilable, i.e., serializable. However, transactions in the IC set are in conflict with  $T_v$ , and the conflicts are irreconcilable, i.e., nonserializable. The formal description of the conditions to categorize these sets of active transactions and the definitions of the terms such as reconcilable conflict and irreconcilable conflict will be given in the later sections. For now, intuitive meaning of these terms is sufficient to understand the protocol.

The RC transactions do not have to be aborted, but their execution history have to be adjusted with timestamp interval facility of this protocol. This is the topic of the subsection (4) below. The IC transactions should be handled with priority-based real-time conflict resolution. This is the topic of the subsection (5) below.

### (3) Dynamic Timestamp Allocation

Another important aspect of this protocol is dynamic timestamp allocation. Most timestamp-based concurrency control protocols use static timestamp allocation scheme, i.e., each transaction is assigned a timestamp value at its startup time, and a total ordering instead of a partial ordering is built up. This total ordering does not reflect any actual conflict. Hence, it is possible that a transaction is aborted even when it requests its first data access [Bay82]. Besides, the total ordering of all transactions is too restrictive, and degrades the degree of concurrency considerably. With dynamic timestamp allocation, serialization order among transactions are dynamically constructed on demand whenever actual conflicts are occurring. Only necessary partial ordering among transactions is constructed instead of a total ordering from the static timestamp allocation.

This dynamic timestamp allocation scheme is possible, because OCC provides a phase-dependent structure of transaction execution. During the read phase, a transaction builds gradually its serialization order with respect to committed transactions on demand whenever a conflict with such transactions occurs. Only when the transaction commits (after passing the validation test), its permanent timestamp, i.e., the final serialization order is determined.

### (4) Dynamic Adjustment of Serialization Order with Timestamp Interval

The dynamic timestamp allocation scheme is made further efficient with timestamp interval facility [Bok87]. More flexibility to adjust serialization order can be obtained using a timestamp interval (initially, the entire range of the timestamp space) assigned to each transaction instead of single value for timestamp. The timestamp intervals of active transactions preserve the partial

ordering constructed by serializable execution. The timestamp interval of each transaction is adjusted (shrunk) whenever the transaction reads or writes a data object to preserve the serialization order induced by committed transactions. When the timestamp interval of a transaction shuts out, it means the transaction has been involved in a nonserializable execution, and the transaction should be restarted. With this facility, it is possible to detect and resolve nonserializable execution early in read phase.

Besides, when a transaction, say  $T_v$ , commits after its validation phase, the timestamp intervals of those transactions categorized as *reconcilably conflicting* are adjusted, i.e., the serialization order between the validating transaction  $T_v$  and its RC transactions are determined. Since the permanent serialization order (final timestamp) of these active transactions is not determined, all we have to do is just determine the partial ordering between  $T_v$  and these active transactions by adjusting their timestamp intervals. Thereby these transactions do not have to be aborted though they are in conflict with the committed transaction, i.e., unnecessary aborts are avoided, unlike other OCC protocols.

### (5) Real-Time Conflict Resolution

In order to resolve irreconcilable conflicts of active transactions with validating transaction, either one has to be aborted, because an irreconcilable conflict means a nonserializable execution. As mentioned, since this protocol is based on OCC with forward validation scheme, either the validating transaction or the conflicting active transaction can be aborted. Hence to determine which transaction to abort, we can employ priority-based conflict resolution schemes such as *priority abort*, *priority sacrifice*, and *priority wait* [Har90, Hua91]. With these schemes, we can extend our protocol to real-time concurrency control protocols, that are corresponding to the real-time OCC protocols presented in [Har90, Hua91].

## 4.2. Procedural Description

We now provide a more detailed, procedural description of the proposed protocol. To execute the proposed protocol, the system maintains an *object table* and a *transaction table*. The object table entries maintain the following information:

*RTS*: the largest timestamp of the committed transactions that read the data object; and  
*WTS*: the largest timestamp of the committed transactions that wrote the data object.

The transaction table entries maintain the following information:

*RS*( $T$ ): read set of transaction  $T$ ;  
*WS*( $T$ ): write set of transaction  $T$ ; and  
*TI*( $T$ ): timestamp interval of transaction  $T$ .

We assume that the write set of a transaction is a subset of its read set and there is no "blind write". In addition to the timestamp interval assigned to each active transaction, a final timestamp, denoted as  $TS(T)$ , is assigned to each committed transaction,  $T$  that has passed the validation test.

Figure 4 provides a procedural description of the read, validation and write phase of transaction execution with the proposed protocol.

At the start of the execution of a transaction,  $T$ , its timestamp interval  $TI(T)$  is initialized as  $[0, \infty)$  (the entire range of timestamp space). For each read or write of a data object made by  $T$ ,  $TI(T)$  should be adjusted to represent the dependencies induced by the operation, i.e., the adjustment of  $TI(T)$  should preserve the order induced by the timestamps of all committed transactions which have accessed that data object. One method to accomplish this adjustment is that when  $T$  reads a data object, the order of the read operation is adjusted to place after all the write

**Read Phase:**

```

for every data object  $x$  in  $RS(T)$  do
  read( $x$ )
   $TI(T) := TI(T) \cap [WTS(x)+1, \infty)$ 
  if  $TI(T) = \emptyset$  then restart( $T$ )
endif
enddo

```

```

for every data object  $x$  in  $WS(T)$  do
  write( $x$ )
   $TI(T) := TI(T) \cap [WTS(x)+1, \infty) \cap [RTS(x)+1, \infty)$ 
  if  $TI(T) = \emptyset$  then restart( $T$ )
endif
enddo

```

**Validation and Write Phase:**

```

find  $RC\_set(T)$  from active_transaction_set
find  $IC\_set(T)$  from active_transaction_set
if  $IC\_set(T) \neq \emptyset$ 
  then invoke real-time_conflict_resolution( $IC\_set(T)$ )
endif
if not aborted( $T$ )
  then choose  $TS(T)$  from  $TI(T)$ 
    update  $RTS(x)$  and  $WTS(x)$  for every  $x$  in  $RS(T)$  and  $WS(T)$ 
    adjustment( $RC\_set(T)$ )
    execute write phase
  endif
endif

```

Figure 4. A Real-Time OCC Using Timestamp

operations made by committed transactions, and when  $T$  writes a data object, the order of the write operation is adjusted to place after all the read and write operations made by committed transactions. As we see in the given procedure, this adjustment is done by intersection operation on timestamp intervals. We assume timestamp intervals contain only integers. Also this adjustment is efficiently accomplished using the final timestamps of committed transactions retained in data objects as  $RTS$  and  $WTS$ .

Any operation of an active transaction  $T$  which introduces a nonserializable execution can be detected by checking whether the execution of the operation results in  $TI(T) = \emptyset$ . A transaction  $T$  must be restarted if  $TI(T) = \emptyset$ .

When a transaction,  $T$  finishes the read phase and reaches the validation phase, the  $RC$  set and the  $IC$  set of  $T$  can be found, using the information in the transaction table. This categorization procedure is the topic of the next section.

If there is one or more  $IC$  transaction in the system, the protocol invokes a real-time conflict resolution scheme to resolve the conflict between  $T$  and the active transaction. Application of real-time conflict resolution scheme will be discussed in detail in Section 4.4.

If  $T$  is not aborted during the real-time conflict resolution (if any), then it is validated and committed. Now the execution of  $T$  should be reflected to the serialization order of committed transactions. Thus a final timestamp for  $T$  should be chosen such that the order induced by the final timestamp should not destroy the serialization order constructed by the already committed transactions. In fact, any timestamp in the range of  $TI(T)$  satisfies this condition because  $TI(T)$  preserves the order induced by all committed transactions. Hence any timestamp from  $TI(T)$  can be chosen for the final timestamp. Then  $RTS$  and  $WTS$  for data objects  $T$  accessed should be updated, if necessary, and finally, the timestamp intervals of all the RC transactions should be adjusted. The adjustment of the timestamp intervals of RC transactions will be discussed in detail in the next section.

Finally, there are two notes related with this process of validation. First, the choice of final timestamp  $TS(T)$  from timestamp interval  $TI(T)$  can be conducted in favor of high priority transactions. When choosing the final timestamp for a validating transaction, the protocol can check the priority of its conflicting transactions, and decide the timestamp in such a way that higher priority transactions are left with larger timestamp intervals. Because a larger timestamp interval means less possibility of restarting the transaction in some sense, a transaction with higher priority needs to have a larger timestamp interval than a transaction with lower priority. Second, the final timestamp does not have to be a specific number. The entire timestamp interval can be the final timestamp of a committed transaction. Then  $RTS$  and  $WTS$  can also be intervals of timestamps, instead of a specific timestamp. This generalization of the final timestamp may provide more flexibility for adjustment of serialization order.

### 4.3. Categories of Conflicting Transactions

Let  $T_i$  be a validating transaction and  $T_j$  ( $j = 1, 2, \dots, n, j \neq i$ ) be transactions in their read phase. Then, the testing for conflicts can be done by looking at intersections of read sets and write sets of  $T_i$  and  $T_j$  as follows:

1.  $RS(T_j) \cap WS(T_i) \neq \emptyset$  (read-write conflict)
2.  $WS(T_j) \cap WS(T_i) \neq \emptyset$  (write-write conflict)
3.  $WS(T_j) \cap RS(T_i) \neq \emptyset$  (write-read conflict)

Before discussing the categorization of transactions in the system with respect to the validating transaction  $T_i$ , we first explain how to identify the state of an active transaction using these conditions. For notational convenience, we introduce a simple notation as the following. When a conflict condition,  $c$ , is satisfied, it is denoted just as  $c$ , and when a condition,  $c$ , is not satisfied, it is denoted as  $\sim c$ . Then the state of an active transaction is denoted as a triple, in which each element indicates whether the corresponding conflict condition is satisfied or not. For example, if a transaction satisfies conditions 1 and 3, but does not satisfy condition 2, i.e., it has both read-write and write-read conflicts with the validating transaction, but it does not have any write-write conflict, then it is denoted as  $(1, \sim 2, 3)$ .

Using this notation and Venn diagrams, Figure 5 shows the six possible states of active transactions under the assumption that the write set of a transaction is a subset of its read set. In the figure,  $RS_i$  and  $WS_i$  indicate  $RS(T_i)$  and  $WS(T_i)$  of the validating transaction  $T_i$ , respectively. Also  $RS_j$  and  $WS_j$  indicate  $RS(T_j)$  and  $WS(T_j)$  of an active transaction  $T_j$ , respectively. There are two Venn diagrams of the combination  $(\sim 1, \sim 2, \sim 3)$  in (a) and (b). They indicate (a) no access to common data objects at all and (b) only read operation on common data objects, respectively. Both cases do not produce any conflict between  $T_j$  and  $T_i$ . The combination  $(1, \sim 2, \sim 3)$  in (c) means that there is only read-write conflict, and no write-write or write-read conflict between  $T_j$  and  $T_i$ . The combination  $(\sim 1, \sim 2, 3)$  in (d) indicates the case when there is only write-read conflict between  $T_j$  and  $T_i$ . The combination  $(1, \sim 2, 3)$  in (e) indicates the case when there are read-write conflict and write-read conflict, but not write-write conflict between  $T_j$  and  $T_i$ .

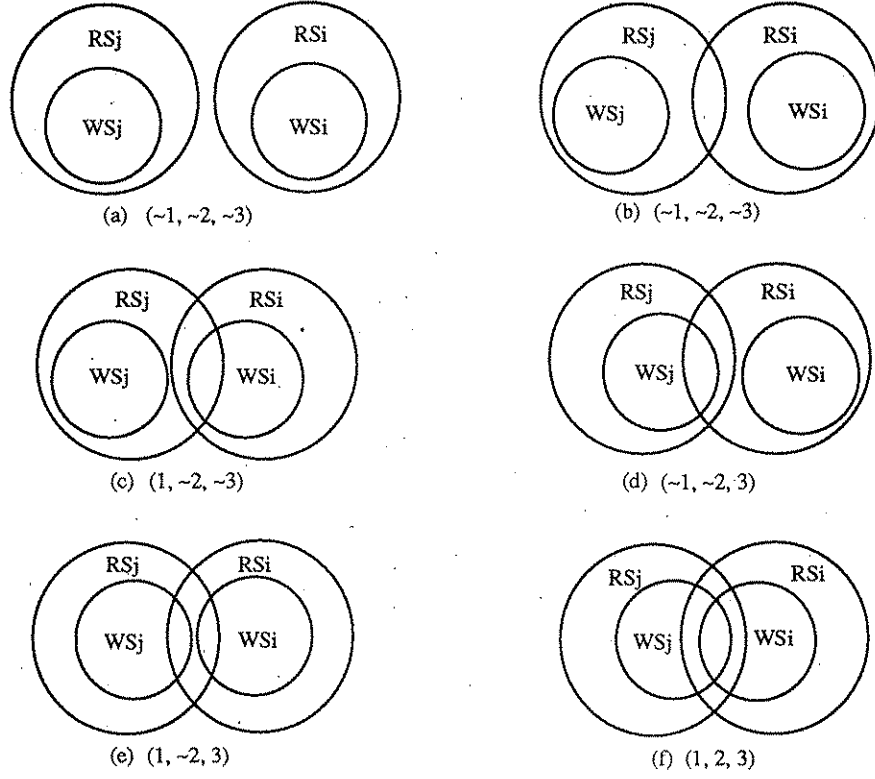


Figure 5. Six Possible States of Active Transaction

Finally, the combination (1, 2, 3) in (f) means that between  $T_j$  and  $T_i$ , all the three possible conflicts, i.e., read-write, write-write, and write-read conflicts exist. Besides these combinations, there are three more combinations of conditions,  $(\sim 1, 2, \sim 3)$ ,  $(1, 2, \sim 3)$ , and  $(\sim 1, 2, 3)$  which are impossible to occur under the given assumption for the relationship of read set and write set of transactions, and hence they are not included in the figure.

Now we can categorize the active transactions in the system according to their states. First, the transactions are divided into two sets;

*set of nonconflicting transactions, and*  
*set of conflicting transactions.*

Obviously, transactions whose state satisfies the condition  $(\sim 1, \sim 2, \sim 3)$  belong to the first set. The latter contains the rest,  $(1, \sim 2, \sim 3)$ ,  $(\sim 1, \sim 2, 3)$ ,  $(1, \sim 2, 3)$  and  $(1, 2, 3)$ .

As mentioned, the set of conflicting transactions are further divided into two sets;

*set of reconcilably conflicting transactions (RC), and*  
*set of irreconcilably conflicting transactions (IC).*

Because of the dynamic serialization order determination mechanism we employed in the proposed protocol, we can make some conflicts reconcilable, i.e., we can serialize these conflicts without aborting. Before discussing how to distinguish RC transactions from IC transactions, let

us consider how each conflict type given above can be serialized.

(1)  $RS(T_j) \cap WS(T_i) \neq \emptyset$  (read-write conflict)

The read-write conflict between the validating transaction,  $T_i$ , and the active transaction,  $T_j$ , can be serialized by the following timestamp interval adjustment of  $T_j$ ;

$$TI(T_j) \leftarrow TI(T_j) \cap [0, TS(T_i)-1].$$

This adjustment of the timestamp interval of  $T_j$  makes a partial ordering between  $T_i$  and  $T_j$  as  $T_j \rightarrow T_i$ , and is called *forward adjustment*. The meaning of this operation is that the read of  $T_j$  precedes the write of  $T_i$  on the same data object, i.e., the data read by  $T_j$  have been not written by  $T_i$ .

(2)  $WS(T_j) \cap WS(T_i) \neq \emptyset$  (write-write conflict)

The write-write conflict between  $T_i$  and  $T_j$ , can be serialized by

$$TI(T_j) \leftarrow TI(T_j) \cap [TS(T_i)+1, \infty).$$

This adjustment of the timestamp interval of  $T_j$  makes a partial ordering between  $T_i$  and  $T_j$  as  $T_i \rightarrow T_j$ , and is called *backward adjustment*. This operation implies that the write of  $T_i$  precedes the write of  $T_j$  on the same data object, i.e., the write of  $T_j$  is not overwritten by  $T_i$ . (In fact, write-write conflicts are automatically resolved in OCC, if I/O operations in the write phase are done sequentially.)

(3)  $WS(T_j) \cap RS(T_i) \neq \emptyset$  (write-read conflict)

The write-read conflict between  $T_i$  and  $T_j$ , can be serialized by

$$TI(T_j) \leftarrow TI(T_j) \cap [TS(T_i)+1, \infty).$$

Similarly, this adjustment of the timestamp interval of  $T_j$  makes a partial ordering between  $T_i$  and  $T_j$  as  $T_i \rightarrow T_j$ , and is a backward adjustment. This operation means that the data read by  $T_i$  have not been written by  $T_j$ .

Now we can find the set of RC transactions with this adjustment of serialization order in mind. RC transactions are transactions involved in conflicts that can be serialized by either forward adjustment only or backward adjustment only, but not both. That is, transactions whose state is either (1, ~2, ~3) or (~1, ~2, 3) among other possible states, belong to the set of RC transactions.

We are left with the set of IC transactions. Transactions involved in conflicts that cannot be serialized by either forward adjustment only or backward adjustment only belong to IC set. Obviously, a need of both forward and backward adjustment of timestamp interval for serialization results in the timestamp interval shut out. The transactions belonging to IC set are involved in nonserializable execution with the validating transaction. Transactions whose state is either (1, ~2, 3) or (1, 2, 3) among other possible states, belong to the set of IC transactions.

We employ real-time conflict resolution schemes to resolve these irreconcilable conflicts. The real-time conflict resolution will be discussed in detail in the next section. For the real-time conflict resolution, it is useful to divide the IC set into two set;

*set of conflicting transactions with higher priority (CHP), and*  
*set of conflicting transactions with lower priority (CLP).*

CHP set contains transactions that are in the IC set and have a higher priority than the validating transaction. Similarly, CLP set contains transactions that are in the IC set and have a lower priority than the validating transaction.

Figure 6 summarizes the relationship among the categories of active transactions in the system explained in this section.

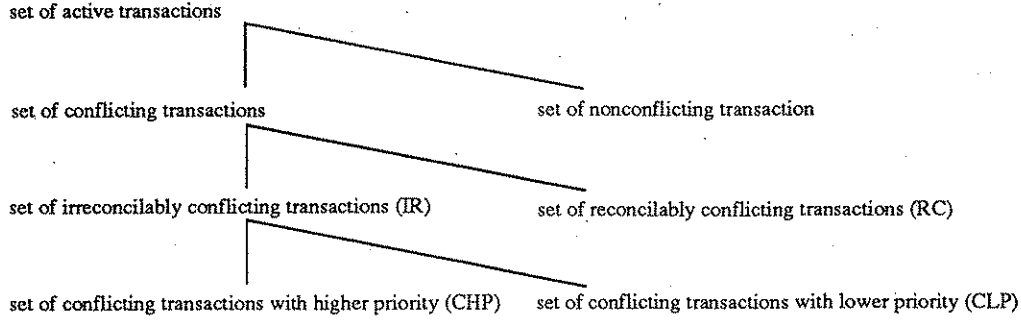


Figure 6. Categories of Active Transactions

#### 4.4. Real-Time Conflict Resolution Schemes

In order to resolve irreconcilable conflicts, we employ real-time conflict resolution schemes. In this section, we describe several real-time conflict resolution schemes, and discuss informally their potential strengths and weaknesses.

These schemes are not basically different from those proposed in [Haed84, Har90, Hua91]. However, they are different in that they handle different sets of conflicting transactions. Those schemes in [Haed84, Har90, Hua91] are performed on the set (denoted as  $S_1$ ) of active transactions,  $T_j$ 's, that are in conflict with validating transaction  $T_i$  in the sense that

$$RS(T_j) \cap WS(T_i) \neq \emptyset.$$

However, the real-time conflict resolution schemes of the proposed protocol are performed on the IC transactions as categorized in the previous section. These are transactions that satisfy either the condition (1, ~2, 3) or the condition (1, 2, 3), i.e., either

$$(RS(T_j) \cap WS(T_i) \neq \emptyset) \wedge (WS(T_j) \cap WS(T_i) = \emptyset) \wedge (WS(T_j) \cap RS(T_i) \neq \emptyset)$$

or

$$(RS(T_j) \cap WS(T_i) \neq \emptyset) \wedge (WS(T_j) \cap WS(T_i) \neq \emptyset) \wedge (WS(T_j) \cap RS(T_i) \neq \emptyset).$$

Let  $S_2$  be the set of these IC transactions. Then obviously, the set,  $S_2$  is a subset of the set,  $S_1$ . This is the reason why the proposed protocol can avoid unnecessary aborts in other real-time OCC protocols using dynamic adjustment of serialization order mechanism with the refined categorization of active transactions.

##### (1) Commit

This scheme leads the protocol to an OCC with forward validation [Rob82]. With this scheme, when a transaction reaches the validation phase, it commits and notifies all the IC transactions. These IC transactions are immediately restarted. A transaction that has reached its validation

phase is guaranteed to commit. The advantage of this scheme is that less resources are wasted because the resources consumed by a validating transaction are never wasted. However, this scheme does not utilize transaction priority during data conflict resolution.

#### (2) Priority Abort [Hua91]

With this policy, when a transaction reaches its validation phase, it is aborted if its priority is less than that of all the IC transactions. If not, it commits and all the IC transactions are restarted immediately as with Commit scheme. Though this scheme utilizes transaction priority, it behaves still like Commit policy most of the time, because the condition for the restart of validating transaction is too strong and inflexible.

#### (3) Priority Sacrifice [Har90]

With this policy, when a transaction reaches its validation phase, it is aborted if at least one IC transaction has a higher priority than the validating transaction; otherwise it commits and all the IC transactions are restarted immediately. This scheme uses transaction priority in such a way that the validating transaction sacrifices itself for the sake of IC transactions with higher priority. With this scheme, many validating transactions can end up aborting after having paid for most of the transaction's execution, because of a IC high priority transaction still in its read phase. Another problem of this scheme is *wasted sacrifice*. A wasted sacrifice occurs when a transaction is sacrificed on behalf of another transaction that is later discarded.

#### (4) Priority Wait [Har90, Hua91]

With this scheme, when a transaction reaches its validation phase, if its priority is not the highest among the IC transactions, it waits for the IC transactions with higher priority to complete. This scheme gives the higher priority transactions a chance to make their deadlines first. While the validating transactions is waiting, it may be restarted due to the commit of one of the IC transactions with higher priority.

This scheme keeps the original goal of real-time conflict resolution in that precedence is given to high priority transactions. There is no wasted restart in this scheme, because all restarts are made on demand. The blocking effect derived from delaying the validating transaction instead of immediately committing or aborting, results in the conservation of resources. The blocking effect also brings drawbacks as well as advantages. If a transaction finally commits after waiting for some time, it causes all its IC transactions with lower priority to be restarted at a later point in time, hence decreasing the chance of these transactions meeting their deadlines, and thus wasting resources. This problem can become worse because with this scheme, a chained blocking is possible. Moreover, while a validating transaction waits, new conflicts can occur and the number of IC transactions is increased, hence resulting in more restarts.

#### (5) Wait-50 [Har90]

Wait-50 scheme is a variation of priority wait strategy. With this policy, a validating transaction will wait if at least 50% of IC transactions have a higher priority over the validating transaction, but otherwise it commits and all the IC transactions are restarted immediately as Commit scheme. The purpose of this scheme is to maximize the beneficial effects of blocking, while reducing the effects of its drawbacks.

### 4.5. Dynamic Adjustment of Serialization Order

In Section 4.3, we have already discussed the set of RC transactions, and the dynamic serialization order adjustment using dynamic adjustment of the timestamp intervals of RC transactions. In this section, we just provide a pseudo code for the **adjustment** procedure, which performs the timestamp interval adjustment for the RC transactions in a validation phase of a transaction. Let  $T_i$  be a validating transaction and  $T_j$  ( $j = 1, 2, \dots, n, j \neq i$ ) be transactions in their read phase.

adjustment(RC\_set( $T_i$ ))

```
{
  for every data object  $x$  in  $RS(T_i)$  do
    for every active transaction  $T_j$  that has written  $x$  do
       $TI(T_j) := TI(T_j) \cap [TS(T_i)+1, \infty)$ 
      if  $TI(T_j) = \emptyset$  then restart( $T_j$ )
    endif
  enddo
enddo

  for every data object  $x$  in  $WS(T_i)$  do
    for every active transaction  $T_j$  that has read  $x$  do
       $TI(T_j) := TI(T_j) \cap [0, TS(T_i)-1]$ 
      if  $TI(T_j) = \emptyset$  then restart( $T_j$ )
    endif
  enddo

  for every active transaction  $T_j$  that has written  $x$  do
     $TI(T_j) := TI(T_j) \cap [TS(T_i)+1, \infty)$ 
    if  $TI(T_j) = \emptyset$  then restart( $T_j$ )
  endif
enddo
enddo
}
```

## 5. Conclusions

Time-critical scheduling in real-time database systems has two components: real-time scheduling and concurrency control. While both concurrency control and real-time scheduling are well-developed and well-understood, there is only limited knowledge about the integration of concurrency control and real-time scheduling. Though recently the problem has been studied actively, the solution proposed is still at an initial stage. A major source of problems in integrating the two is the lack of coordination in the development. They are developed on different objectives and incompatible assumptions [Buc89].

Most of the proposed work for real-time concurrency control employ a simple method to utilize one concurrency control scheme such as 2PL, TO and OCC, and to consider the priority of operations inherited from the timing constraints of transactions in operation scheduling. This method has an inherent disadvantage of being limited by the concurrency control method used as the base. Since neither of pessimistic nor optimistic concurrency control is satisfactory by itself for real-time scheduling [Son90], this simple method using only one control can hardly satisfy the timing requirements of RTDBS. Problems such as excessive blocking, wasted restarts, and priority inversion are serious in RTDBS.

In this paper, we proposed two real-time transaction scheduling protocols which employ a hybrid approach, i.e., a combination of both pessimistic and optimistic approaches. These protocols make use of a new conflict resolution scheme called dynamic adjustment of serialization order, which supports priority-driven scheduling, and avoids unnecessary aborts.

The first protocol is a concurrency control protocol which integrates a priority-dependent locking mechanism with an optimistic approach. It works when no information about data requirements or execution time of each transaction is available. By delaying the write operations of transactions, the restraint of past transaction execution on the serialization order is relaxed, allowing the serialization order among transactions to be adjusted dynamically in compliance

with transaction timeliness and criticality. The proposed protocol allows transactions to meet timing constraints as much as possible without reducing the concurrency level of the system or increasing the restart rate significantly. With this protocol, high priority transactions are never blocked by an uncommitted lower priority transaction, while low priority transactions may not have to be aborted even in face of conflicts with high priority transactions.

The second protocol is a real-time OCC protocol in which serializability is guaranteed by dynamic timestamp ordering mechanism as well as validation test of OCC. This protocol has several advantages over other real-time OCC protocols proposed in [Har90, Hua91]. First, in the proposed protocol, the ability to detect and resolve nonserializable execution is further improved by timestamp interval facility. The ability of early detection and resolution of nonserializable execution is one of the primary merit of forward validation-based OCC. Early restart is especially important for real-time database systems, because it can increase the chances of meeting deadlines of transactions. The proposed protocol utilizes forward validation, and with the timestamp interval facility, it can check easily and detect early nonserializable execution. Second, the proposed protocol can avoid unnecessary aborts in other real-time OCC protocols with a refined conflict type categorization, a timestamp interval facility, and a dynamic adjustment of serialization order mechanism.

However, these advantages have brought some overhead such as overhead of managing the object table, especially maintaining the read and write timestamp for every data object, and the overhead associated with identifying the reconcilably conflicting transactions and the irreconcilably conflicting transactions from the transaction table. The second overhead requires many set intersection operations. The performance of the proposed protocol can be improved by efficient implementation that reduces the effect of those overheads. For example, the overhead associated with set intersection operation can be significantly reduced by maintaining the transaction table as a bitmap, in which each bit represents a data object in database. In such implementation of transaction table, a set intersection operation can be efficiently implemented as a bit operation.

## REFERENCES

- [Abb88] R. Abbott, H. Garcia-Molina, "Scheduling Real-time Transactions: A Performance Evaluation," *Proceedings of the 14th VLDB Conference*, 1988.
- [Bay82] Bayer, R., K. Elhardt, J. Heigert and A. Reiser, "Dynamic Timestamp Allocation for Transactions in Database Systems," *Proc. 2nd Int. Symp. Distributed Data Bases*, September 1982, pp 9-20.
- [Bern87] Bernstein, P. A., V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, Mass., 1987.
- [Bok87] Boksenbaum, C., M. Cart, J. Ferrie, and J. Pons, "Concurrent Certifications by Intervals of Timestamps in Distributed Database Systems," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 4, April 1987.
- [Buc89] Buchmann, A. et al., "Time-Critical Database Scheduling: A Framework for Integrating Real-Time Scheduling and Concurrency Control," *5th Data Engineering Conference*, February 1989.
- [Car87] Carey, M. J., "Improving the Performance of an Optimistic Concurrency Control Algorithm Through Timestamps and Versions," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 6, June 1987. pp. 746 - 751.

- [Coo91] Cook, R. P., S. H. Son, H. Y. Oh, and J. Lee, "New Paradigms for Real-Time Database Systems," *8th IEEE workshop on Real-Time Operating Systems and Software (in Conjunction with) IFAC/IFIP Workshop on Real-Time Programming*, May 1991.
- [Hae84] Haeder, T., "Observations on Optimistic Concurrency Control Schemes," *Information Systems*, Vol. 9, No. 2, 1984.
- [Har90] J. R. Haritsa, M. J. Carey, and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control," *IEEE Real-Time Systems Symposium*, Orlando, Florida, December 1990.
- [Hua90] Huang, J., J. A. Stankovic, D. Towsley, and K. Ramamritham, "Real-Time Transaction Processing: Design, Implementation, and Performance Evaluation," Univ. of Massachusetts, COINS Technical Report 90-43, May, 1990.
- [Hua91] J. Huang, J. A. Stankovic, K. Ramamritham, and D. Towsley, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes," *VLDB Conference*, Barcelona, Spain, Sept. 1991.
- [Lin90] Y. Lin and S. H. Son, "Concurrency Control in Real-Time Database Systems by Dynamic Adjustment of Serialization Order," *IEEE Real-Time Systems Symposium*, Orlando, Florida, December 1990.
- [Rob82] Robinson, J. T., "Experiments with Transaction Processing on Multiprocessors," *IBM Research Report*, RC9725, Yorktown Heights, NY, December 1982.
- [Sha91] Sha, L., R. Rajkumar, S. Son and C. Chang, "A Real-Time Locking Protocol," *IEEE Trans. on Computers*, Vol. 6, No. 7, July 1991.
- [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.
- [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.