Concurrency Control Using Priority-Based Locking

Sang H. Son and Yi Lin

Computer Science Report No. TR-90-13
June 27, 1990

Concurrency Control using Priority-Based Locking

Sang H. Son
Yi Lin

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

Time-critical scheduling in real-time database systems has two components: real-time transaction level scheduling, which is similar to task scheduling in real-time operating systems, and concurrency control, which can be considered as operation level scheduling. Most current research in this area only focuses on the transaction scheduling part and rely on the existing classical non-real-time methods for concurrency control. In this paper, a new concurrency control protocol for real-time database systems is introduced, in which real-time scheduling and concurrency control are integrated. The method is purely priority-based and does not assume knowledge of any information about data requirements or execution time of each transaction. This makes the protocol suitable even for non-real-time conventional database systems.

Key words: real-time database, concurrency control, time-critical scheduling, priority, locking

This work was supported in part by the Office of Naval Research under contract # N00014-88-K-0245, by the Department of Energy under contract # DE-FG05-88-ER25063, and by the Center for Innovative Technology under contract # CIT-INF-90-011.

1. Introduction

Compared with traditional databases, the ability to meet deadlines of transactions is vital to a real-time database. In other words, the timeliness of results can be as important as their correctness in real-time database systems. Deadline constitute the timing constraints of transactions. A deadline is said to be *hard* if it cannot be missed or the result is useless. If a deadline can be missed, it is a *soft* deadline. With soft deadlines, the usefulness of a result may decrease as the delay after the deadline increases. Another important characteristic of real-time transactions is criticality, which represents the importance of a transaction. Deadline and criticality are two orthogonal factors that play important roles in real-time database systems.

The notion of scheduling in real-time database systems is twofold: to meet timing constraints and to enforce data consistency. In real-time operating systems, scheduling is usually at task level subject to timing constraints only. Data consistency constraints are not involved. In non-real-time database systems, scheduling is at operation level in that the basic unit of a schedule is operation. The only goal of such scheduling is to guarantee data consistency. Timeliness is not an issue of consideration. The real-time database system is a combination of the two.

Real-time task scheduling methods can be extended to become real-time transaction scheduling method while concurrency control methods are still needed for operation scheduling to maintain data consistency. However, the integration of the two mechanisms in real-time database systems is not trivial, because all existing concurrency control methods synchronize concurrent data access of transactions by the combination of two measures: block and roll-back of transactions, both of which are barriers for time-critical scheduling. The conservative two phase-locking (2PL) method [Ber87, Esw76] and the optimistic methods [Kung81, Hae84, Bok87, Bas88, Hal89] are examples of the two extremes. In real-time database systems, blocking can cause priority inversion when a high priority transaction is blocked by lower priority transactions [Sha88]. The alternative is to abort the low priority transactions if they block a high priority

transaction. This implies the waste of the work done by the aborted transactions and in turn also has a negative effect on time-critical scheduling.

Most of the research work done in this area so far focuses on the time-critical transaction scheduling part of the problem while the other part, concurrency control, remains intact. The general approach is to utilize the existing concurrency control protocols, especially 2PL, and apply some time-critical transaction scheduling methods to favor more urgent transactions [Sha88, Abb88, Son89]. Such approach has the inherent disadvantage of being limited by the concurrency control method upon which it is based.

Concurrency control methods rely on the setting of a serialization order among conflicting transactions. Two transactions are said to be conflicting if they access the same data object and one of them is doing a write operation [Ber87]. In non-real-time concurrency control methods, timing constraints are not a factor in the construction of this order. This is obviously a drawback for real-time systems. For example, with the 2PL method, the serialization order is dynamically constructed and corresponds to the order in which the conflicting transactions access the shared data objects. In other words, the serialization order is bound to the past execution history with no flexibility. When a transaction T_H with a higher priority requests an exclusive lock which is being held by another transaction, T_L, with a lower priority, the only choices are either aborting T_L or letting T_H wait for T_L. Neither choice is satisfactory and thus the performance is degraded.

The situation is none the better with basic timestamp ordering protocol (TO), since the serialization order of transactions is already determined statically by their assigned timestamps even before the execution of each transaction. In TO, a high priority transaction may even be aborted due to conflict with a low priority transaction whose timestamp is smaller.

The priority ceiling protocol, which is basicly a task scheduling protocol for real-time operating systems, has been extended to the real-time database system [Sha88]. It is based on 2PL and only employs blocking, but not roll-back, to solve conflicts. This makes it a very conser-

vative approach. In non-real-time database systems, it has been shown that optimal performance may be achieved by a compromise of blocking and roll-back [Yu90]. In real-time systems, the same results may be expected. Aborting a few low priority transactions and restarting them later can usually allow more high priority transactions to meet their deadlines and thus the system performce could be improved. Another disadvantage of the priority ceiling protocol is that it requires the knowledge of all the transactions that will be executed in the future and their read sets and write sets. This is too harsh a condition and almost impossible for most database systems to satisfy. Its basic approach is to delay the release of each transaction to execution after they arrive in order to guarantee that each transaction will be blocked at most once after it begins execution. The problem is that, due to the ceiling effect, most transactions actually have already been blocked for an unpredictable period before they are released for execution. Although each transaction can only be blocked once during execution, the blocking time may be arbitrarily long. The situation is even worse if a high priority transaction is blocked by a long low priority transaction.

Based on the argument that timing constraints may be more important than data consistency in real-time database systems, attempts have been made to satisfy the timing constraints by sacrificing database consistency to some degree [Lin89, Vrb88]. The common correctness criterion of database systems is serializability. Although in some applications a weaker criterion is acceptable [Gar83], no general-purpose consistency criterion that is less stringent than serializability has been proposed so far; neither is there any general-purpose approach that allows some kind of inconsistent state in the database and recover it later so that transactions are not delayed due to concurrency control. The problem is that the 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 not reversible. Besides, incorrect data may spread within the database. This makes inconsistency removal a difficult task. Before any breakthrough

is made in this direction, serializability seems to be the only correctness criterion for us to live with.

Satisfying the timing constraints while preserving data consistency requires the concurrency control method to accommodate timeliness of transactions as well as to maintain data consistency. This is the very goal of the our work. In real-time database systems, timeliness of a transaction is usually combined with its criticality to take the form of the priorities of that transaction. Various ways of assigning priority and their effects have been discussed in [Stan88, Hua89, Buch89].

For a concurrency control method to accommodate the timeliness 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. For example, a higher priority transaction may have no way to precede a lower priority transaction in the serialization order due to previous conflicts. The result is that either the lower priority transaction has to be aborted or the high priority transaction suffers blocking. If the information about data requirements and execution time of each transaction is available beforehand, off-line preanalysis can be done to avoid conflicts [Sha88]. This is exactly what is done in many real-time task scheduling methods. However, such approach may have to delay the starting of some transactions, even if they have high priorities, and may reduce the concurrency level in the system. This, in return, may lead to the violation of the timing constraints and degrade the system performance [Son90]. Moreover, due to blocking among transactions caused by data contention, it often takes much longer than the estimated execution time for a transaction to complete, not to mention that acquiring information such as data requirements and execution time of each transaction in a conventional database system is often impractical.

What we need is a concurrency control method 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 method presented in this paper features such ability. It has the flavor of both locking and the 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 dynamicly, making it possible for transactions with higher priorities to be executed first so that high priority transactions are never blocked by uncommitted low 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. We are making operation scheduling, which is usually for concurrency control, also time-critical. Our approach is to relax the check between the serialization order and the past execution history so that the former can be adjusted freely in favor of high priority transaction without regard to the latter. For example, T_1 and T_2 are two transactions with T_1 having a higher priority. T_2 writes a data object x before T_1 reads it. In 2PL, even in the absence of any other conflicting operations between these two transactions, T_1 has to either abort T_2 or be blocked until T_2 releases the write lock. That is because the serialization order $T_2 \rightarrow T_1$ is already determined by the past execution history. T_1 can never precede T_2 in the serialization order. In our method, when such conflict occurs, the serialization order of the two transactions will be adjusted in favor of T_1, i.e. $T_1 \rightarrow T_2$, and neither is T_1 blocked nor is T_2 aborted. The locking protocol in the method is free from dead-locks.

The remainder of the paper is organized as follows. The details of the proposed method are described in the next section. The transaction scheduling aspect of the method is discussed in Section 3. The correctness proof of the method is in Section 4. Section 5 presents an example to show how the method works. We further our discussion about the method by suggesting possible optimization in Section 6. Finally concluding remarks appear in Section 7.

2. The Proposed Method

The environment we assume is a single processor, 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. At this moment we do not discuss the priority assignment policy. 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 this 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.

All transactions that can be scheduled are put in a ready queue, R_Q. Only transactions in R_Q can be selected to occupy the CPU. 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 occupy 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. We will discuss our CPU scheduling policy in the next section.

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 is executed, only reading from the database and writing 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 it makes all its updates permanent in the database. A transaction in any of the three phases is called an *active* transaction. If an active transaction is in the write phase, then it is committed and writing into the database.

Critical sections are used in the method. There are several different classes of critical sections. Only one transaction can be in any critical section of a particular class at a time. For example, C_1, C_2 and C_3 are three critical sections of the same class. If transaction T_1 is in one of the

three critical sections, say C_2, no other transaction can be in any one of the three critical sections. Therefore it is impossible for another transaction T_2 to be in one of the three critical sections, say C_3, at the same time. However, transactions can be in critical sections of different classes. For example, C_4 is a critical section of another class different from that of C_2. T_2 can be in C_4 while T_1 is in C_2. We denote critical sections of different classes by angular brackets with the number of the class as their subscripts, e.g. "$<_1\ _1>$" means the first class critical section. In our pseudo-code, we assume that each assignment statement of global data is executed atomically. The following are some useful notations:

id:	id of this transaction
read_trset:	set of ids of transactions in the read phase
wait_trset:	set of ids of transactions in the wait phase
write_trset:	set of ids of transactions in the write phase
tscnt:	final-timestamp count of the system
ts:	final-timestamp of this transaction
ts(T):	final-timestamp value 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.
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

2.1. Read Phase

The read phase is the normal execution of the transaction except that write operations are performed on the private data copies in the local workspace of the transaction instead of on the data objects in the database. We call such write operations *prewrite*. One advantage of this prewrite operation is that when a transaction is aborted, all that have to be done for recovery is to simply discard the data in its local workspace. No rollback is needed because no changes have been made in the database.

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 do a read (resp. prewrite) operation on a data object, it must obtain the read (resp. write) lock on that data object

first. If a transaction reads a data object that has been written by itself, it gets the private copy in its own workspace immediately and no read lock is needed. In the remaining part of the paper, when we refer to read operations, we exclude such read operations because they do not incur any dependency among transactions.

Each lock contains the priority of the transaction holding the lock as well as other usual information. The locking protocol is based on the *principle* that high priority transactions should complete before lower priority transactions. That means if two transactions conflict, the higher priority transaction usually should precede the lower priority transaction in the serialization order. With our CPU scheduling policy, which will be described in the next section, we can expect a high priority transaction to commit before a low priority transaction most of the time. 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 abort of a higher priority transaction. Since transactions do not write into the database during the read phase, write-write conflicts need not be considered here.

Suppose active transaction T_1 has a higher priority than active transaction T_2. We have the following four possibilities of conflict and the transaction dependencies they set in the serialization order:

(1) $read_{T_1}[x] \rightarrow prewrite_{T_2}[x]$ => $T_1 \rightarrow T_2$

(2) $prewrite_{T_1}[x] \rightarrow read_{T_2}[x]$ => $T_1 \rightarrow T_2$ (delayed reading)
$T_2 \rightarrow T_1$ (immediate reading)

(3) $read_{T_2}[x] \rightarrow prewrite_{T_1}[x]$ => $T_2 \rightarrow T_1$

(4) $prewrite_{T_2}[x] \rightarrow read_{T_1}[x]$ => $T_1 \rightarrow T_2$ (immediate reading) or
$T_2 \rightarrow T_1$ (delayed reading)

Case (1) meets the principle. In case (2), following our principle, we should choose delayed reading, i.e. T_2 should not read x until T_1 has committed and written x in the database. Case (3) violates our principle. In this case, unless it is already committed, T_2 is usually aborted because

otherwise T_2 must commit before T_1 and thus will block T_1. However, if T_2 has already finished its work, i.e. in the wait phase, we should avoid aborting it because aborting a transaction which has completed its work imposes a considerable penalty on the the system performance. In the meantime, we still do not want T_1 to be blocked by T_2. Therefore when such conflict occurs and T_2 is in the wait phase, we do not abort T_2 until T_1 is committed, hoping that T_2 may get a chance to commit before T_1 commits. In case (4), if T_2 is already committed and in the write phase, we should delay T_1 so that it reads x after T_2 writes it. This blocking is not a serious problem for T_1 because T_2 is already in the write phase and 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. Therefore T_1 will not be blocked for a long time. Otherwise, if T_2 is not committed yet, i.e. either in the read phase or in the wait phase, T_1 should read x immediately because that is in accordance with the principle.

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 associate with each transaction two sets, *f_trset* and *b_trset*, and a count, *fcnt*. The set *f_trset* (resp. *b_trset*) contains all the active lower priority transactions that must precede (resp. follow) this transaction in the serialization order. *Fcnt* 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 *f_trset*, *b_trset*, and *fcnt* will be changed correspondingly.

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

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

$<_1$ **for** $t \in \{T_i \mid wlock(T_i,x) \wedge Ti \neq T\}$ **do**
if *(priority (t) > priority (T)* $\vee$ $t \in write\_trset$*)*
then *deny the lock;*
else if $(t \in f\_trset_T)$ **then** *abort t;*
else if $(t \notin b\_trset_T)$

```
                then (b_trset_T := b_trset_T ∪ {t};
                      fcnt_t := fcnt_t + 1;
                     )
    grant the lock;
1>
```

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

```
<2 for t∈ {T_i | rlock(T_i,x) ∧ Ti≠T} do
     if (priority (t) > priority (T) )
     then ( if (T ∈ f_trset_t) then abort T;
            else if (T ∉ b_trset_t)
                 then (b_trset_t := b_trset_t ∪ {T};
                       fcnt_T := fcnt_T + 1;
                      )
          )
2>
       else if (t ∈ wait_trset)
            then ( if (t ∈ b_trset_T) then abort t;
                   else f_trset_T := f_trset_T ∪ {t};
                 )
            else if (t ∈ read_trset) then abort t;

   grant the lock;
```

Lock requests are processed sequentially by a lock manager. 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 decided whether the lock can be granted now. The critical sections will be discussed later. Fig. 1 shows the lock compatibility tables. The first columns of the two tables indicate the types of the locks being requested. The first rows of the two tables indicate the types of the locks being held by other transactions. The compatibility depends on the priorities of the transactions holding and requesting the lock as well as the lock types. Even with the same lock types, locks may be compatible or not, depending on the priorities of the lock holder and the lock requester. For incompatible locks, the locking protocol defines the actions to be adopted.

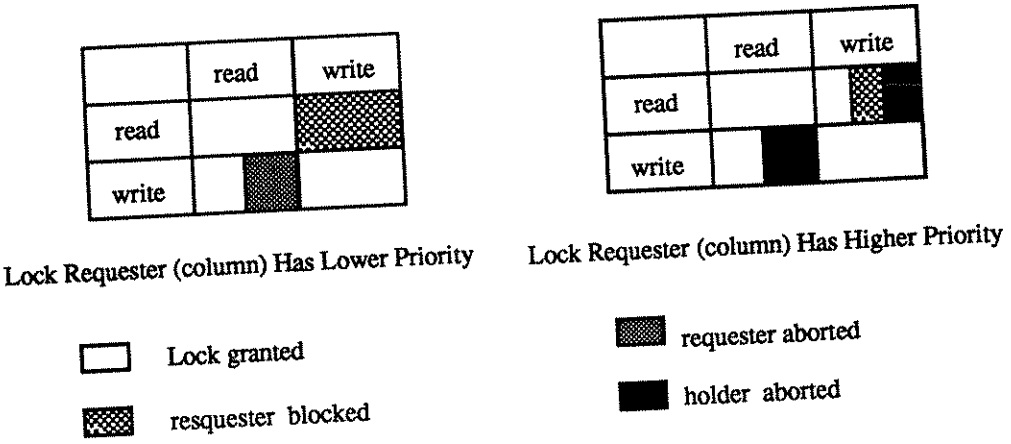

Fig. 1 Lock Compatibility Table

Note that a data object may be both read locked and write locked by several transactions simultaneously with our locking protocol. Unlike 2PL, locks are not classified simply as shared locks and exclusive locks. Fig. 2 summarizes the lock compatibility of 2PL. By comparing Fig. 1 with Fig. 2, it is obvious that our locking protocol is much more flexible, thus incurs less blocking and abort. In our locking protocol, a high priority transaction is never blocked or aborted due to conflict with a lower priority transaction. The probability of aborting a lower priority transaction is roughly less than half of 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 is started:

```
tbegin = (
  f_trset := ∅;
  b_trset := ∅;
  fcnt := 0;
  read_trset := read_trset ∪ {id};
  R_Q := R_Q ∪ {id};
```

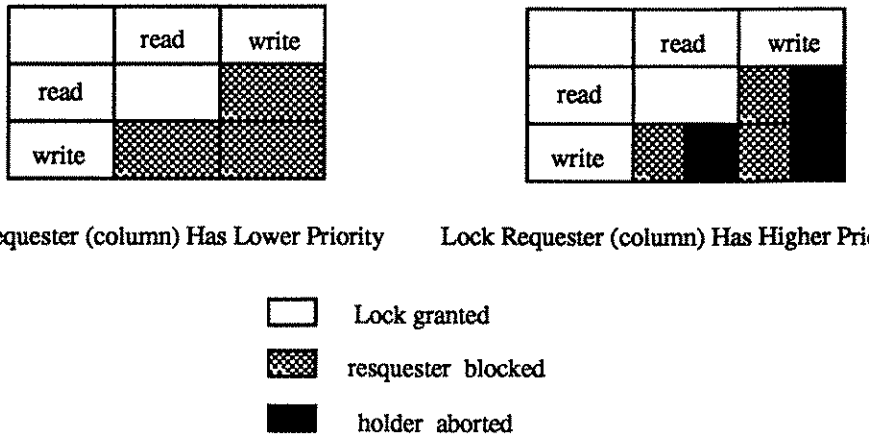

Fig. 2 Lock Compatibility Table for 2PL

) .

Then the transaction is in the read phase. When it tries to read or prewrite a data object, it requests the lock. 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, the following procedure is called:

```
tabort = (
  release all locks;
 <2for t ∈ b_trset do
     ( fcnt_t := fcnt_t−1;
       if (fcnt_t = 0 /\ t ∈ wait_trset) then unblock t;
     )
 2>
  if (id ∈ read_trset)   then read_trset := read_trset - {id};
  else if (id ∈ write_trset)   then write_trset := write_trset - {id};
       else if (id ∈ wait_trset)   then wait_trset := wait_trset - {id};
  )
```

2.2. Wait Phase

The wait phase allows a transaction to wait until they can commit. A transaction T can commit only if all transactions with higher priorities that must precede it in the serialization order are either committed or aborted. Since *fcnt* is the number of such transactions, T can commit only if its *fcnt* becomes zero. A transaction in the wait phase may be aborted due to two reasons. The first one is that since T is not committed yet and still holding all the locks, by the locking protocol it may be aborted due to a conflicting lock request by a higher priority transaction. The second reason is the commitment of a higher priority transaction that must follow T in the serialization order. When such a transaction commits, it finds T in *b_trset* and aborts T. Once a transaction in the wait phase gets its chance to commit, i.e. its *fcnt* goes to zero, it switches into the write phase and release all its read locks. A final-timestamp is assigned to it, which is the absolute serialization order. The procedure is as follows:

```
twait = (
   wait_trset := wait_trset ∪ {id};
   read_trset := read_trset - {id};
   waiting := TRUE;
   while(waiting)
    ( <1 if (fcnt = 0)
        then ( wait_trset := wait_trset - {id};     /* switching into write phase */
               write_trset := write_trset ∪ {id};
               ts := tscnt;
               tscnt := tscnt + 1;
               for t ∈ f_trset do
                  if (t∈read_trset ∨ t∈wait_trset ) then abort t;
     1>
               waiting := FALSE
             )
         else block;
    )
   release all read locks;
<2 for t ∈ b_trset do
        if (t∈read_trset ∨ t∈wait_phase )
        then ( fcnt_t := fcnt_t−1;
               if (fcnt_t = 0 ∧ t ∈ wait_trset)
               then unblock t;
             )
2>
 )
```

After a transaction commits, all the transactions in its *f_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 *f_trset* can commit. Note that LP1 is also in the critical section of the same class. This achieves mutual exclusion on *fcnt* and *write_trset*. The critical section of class 2 in the procedure also appears in LP2 and the procedure *tabort*. This ensures the mutual exclusion on *b_trset*. To be precise, mutual exclusion is only needed between LP2 and the two procedures. Transactions can be in the critical sections of class 2 in the two procedures, or even of the same procedure simultaneously, but not in the critical sections of class 2 in LP2 and one of the two procedures.

2.3. 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 are copied from the local workspace into the database. After each write operation, the corresponding write lock is released. The Thomas' Write Rule (TWR) is applied. 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 is as follows:

```
twrite = (
 <3  for x∈ { x_i | wlock(id,x_i) } do
        ( for T∈write_trset do
             if ( wlock(T,x) /\ ts(T) < ts(id) )
             then release T's write lock on x;
          send write request on x and wait for acknowledgement;
 3>
          if (acknowledgement is ok)
          then release the write lock on x;
          else  abort;
        )
    R_Q := R_Q - {id};
   )
```

The purpose of the critical section is to achieve mutual exclusion on write locks. 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 that 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. CPU scheduling

Although the focal point of this paper is on concurrency control, i.e. operation level scheduling, we still need to discussion a little about the transaction scheduling, or CPU scheduling, aspect of our method. In non-real-time database systems, CPU scheduling is usually done by the underlying operating systems, because there are no timing constraints. Data consistency is the only concern. In real-time database systems, however, CPU scheduling should take into account the timeliness of transactions.

In our protocol, R_Q contains all transactions that can be scheduled. These transactions can be in any phase. We need a policy to determine the CPU scheduling priority for transactions in different phases. Transactions in their wait phase are those that have finished their work and are waiting for their chances to commit. We would like to avoid aborting such transactions as much as possible. Therefore transactions in this phase are given higher CPU scheduling priority than those in the read phase so that they can commit as soon as they get the chance. Transactions in the read phase are scheduled according to their assigned priority. If in the R_Q, there are several transactions in the read phase, the one with the highest priority is always selected to execute.

For transactions in the wait phase, the lower the priority is, the higher the CPU scheduling priority is. Because low priority transactions are more vulnerable to conflicts, once there is a chance, they should be committed as soon as possible to avoid being aborted later. Moreover, when a high priority transaction T_H is committed, it may have to abort a low priority transaction T_L if T_L is in T_H's *f_trset*. If T_L is also ready to commit and we allow it to commit before T_H, both T_L and T_H can be committed.

4. Properties and Correctness

In this section, we prove some properties and the correctness of the method. First, we give the simple definitions of *history* and *serialization graph* (SG). For the formal definitions, see [Ber87]. 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 [Ber87].

Property 1: For a transaction T, f_trset_T and b_trset_T can be changed only if T is not in the write phase.

Proof: The Property is derived directly from LP1 and LP2. In LP1 and LP2, T must be in the read phase, for it is requesting a read or write lock. In LP2, b_trset_t may be changed. Since t is holding some read locks and only transactions in the read or wait phase hold read locks, t must not be in the write phase.

Property 2: For a transaction T, $fcnt_T = n$ iff there exist n transactions T_i, $i = 1,2,\cdots,n$, such that T_i is not committed, and $T \in b\_trset_{T_i}$.

Proof: By LP1 and LP2, $fcnt_T$ is increased if and only if after T is inserted into $b\_trset_{T_H}$, where T_H is another transaction. By *twait()* and *tabort()* , $fcnt_T$ is decreased if and only if T_H is aborted or enters into the write phase. The Property follows.

Property 3: Let T_1 and T_2 be two committed transactions. If $T_2 \in b\_trset_{T_1}$, then $ts(T_1) < ts(T_2)$, i.e. T_1 enters into the write phase before T_2.

Proof: Suppose T_2 enters into the write phase before T_1. Since $T_2 \in b\_trset_{T_1}$, at the time T_2 enters into the write phase, $fcnt_{T_2}>0$. In *twait()* , T_2 can enter into the write phase if and only if $fcnt_{T_2} = 0$. A contradiction.

Property 4: Let T_1 and T_2 be two committed transactions. If $T_1 \in f\_trset_{T_2}$, then $ts(T_1) < ts(T_2)$, i.e. T_1 enters into the write phase before T_2.

Proof: Suppose T_2 enters into the write phase before T_1. Since $T_1 \in f\_trset_{T_2}$, in *twait ()* , when T_2 enters into the write phase, all transactions in $f\_trset_{T_2}$ are aborted. By Property 1, T_1 must be in $f\_trset_{T_2}$ at that time, because T_1 must be added into $f\_trset_{T_2}$ before T_2 enters the write phase. Therefore T_1 is also aborted. This contradicts the fact that T_1 is committed. The Property follows.

Theorem 1: Let T_1 and T_2 be two committed transactions in a history H produced by the method. If there is an edge $T_2 \rightarrow T_1$ in SG(H), then $ts(T_2) < ts(T_1)$.

Proof: Suppose $ts(T_1) < ts(T_2)$. Therefore T_1 enters into the write phase before T_2. Since $T_2 \rightarrow T_1$, The two must have conflicting operations. There are three cases.

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

If $w_2[x]$ is sent to the data manager first, T_1's write lock on x must be released before $w_2[x]$ is sent to the data manager in *twrite()*. If $w_1[x]$ is sent to the data manager first, it will either be processed before $w_2[x]$ is sent to the data manager, or

be discarded when the data manager receives $w_2[x]$, because $w_1[x]$ has smaller timestamp. Therefore $w_2[x]$ is never processed before $w_1[x]$. Such conflict is impossible.

Case 2: $r_2[x] \rightarrow w_1[x]$

If T_1 holds write lock on x when T_2 requests the read lock,

(1) $priority(T_1) < priority(T_2)$

Since T_2 reads x before T_1 writes it in the database, T_1 is not in the write phase when T_2 requests the read lock. By LP1, $T_1 \in b\_trset(T_2)$. By Property 3, $ts(T_2) < ts(T_1)$. A contradiction.

(2) $priority(T_1) > priority(T_2)$

By LP1, T_2 is blocked until T_1 releases the write lock on x. This contradicts the fact that T_2 reads x before T_1 writes it.

If T_2 holds read lock on x when T_1 requests the write lock,

(1) $priority(T_1) < priority(T_2)$

By LP2, $T_1 \in b\_trset_{T_2}$. By Property 3, $ts(T_2) < ts(T_1)$. A contradiction.

(2) $priority(T_1) > priority(T_2)$

By LP2, $T_2 \in f\_trset_{T_1}$. By Property 4, $ts(T_2) < ts(T_1)$. A contradiction.

Case 3: $w_2[x] \rightarrow r_1[x]$

Since T_1 enters into the write phase before T_2, such conflict is impossible.

None of the three cases is true. The edge $T_2 \rightarrow T_1$ cannot exists. A contradiction. Hence the Theorem follows.

Corollary 1: Every history H produced by the method is serializable.

Proof: Suppose there is a cycle $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$ in SG(H). By Theorem 1, we have $ts(T_1) < ts(T_2) < \cdots < ts(T_n) < ts(T_1)$. This is impossible. The Corollary follows.

Property 5: Let T_H and T_L be two transactions with $priority(T_H) > priority(T_L)$. T_H can be blocked by T_L only if T_L is in the write phase.

Proof: By LP1, if T_H is in the read phase, it can be blocked by T_L only if T_L is in the write phase. When T_H is in the wait phase, it is blocked if and only if $fcnt_{T_H} \equiv n > 0$. By Property 2, that means there exist n transactions T_i, i=1,2, $\cdots$,n, such that $T_H \in b\_trset_{T_i}$. By LP1 and LP2, we must have $priority(T_i) > priority(T_H)$. Therefore $T_L \neq T_i$. T_L cannot block T_H when T_H is in the wait phase. We know that when a transaction is in the write phase, it will not be blocked by any other transactions. Therefore T_H can be blocked by T_L only when T_H is in the read phase and T_L is in the write phase. Hence the Property follows.

Theorem 2: The method prevents deadlock.

Proof: Suppose there is a cycle in the wait-for graph (WFG),

$$T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1.$$

For each edge $T_i \rightarrow T_j$ in the cycle, we prove that $priority(T_j) > priority(T_i)$. Suppose $priority(T_i) > priority(T_j)$, by Property 5, 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 have

$$priority(T_1) < priority(T_2) < \cdots < priority(T_n) < priority(T_1).$$

This is impossible. The Theorem follows.

The strictness of the histories produced by the method follows obviously from the fact that a transaction applies the results of its write operations from its local workspace into the database only after it commits.

5. An Example

In this section, we give a simple example to show how the method works. The example is depicted in Fig. 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 transactions in our example. T_1 has the highest priority and T_3 has the lowest. T_3 arrives at time t0 and reads data object *a*. This causes a page fault. After the I/O operation, it pre-writes *b*. Then T_2 comes in at time t1 and preempts T_3. At time t2 it reads *c* and causes another page fault. So it is blocked for the I/O operation and T_3 executes. After T_3 pre-writes *d*, T_2 finishes I/O and preempts T_3 again. It pre-writes *d* which is only write locked by T_3. At time t3, T_1 arrives and preempts T_2. T_1 first reads *d*, which is write locked by both T_2 and

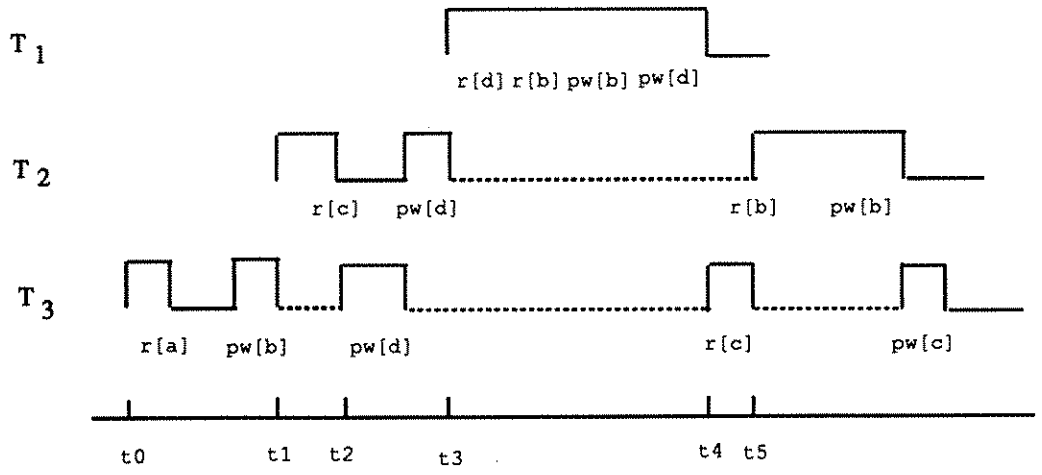

Fig. 3 An Example

T_3. Therefore, $b\_trset_{T_1}$ becomes $\{T_2, T_3\}$ and both $fcnt_{T_2}$ and $fcnt_{T_3}$ become 1. Then T_1 reads b, which is write locked by T_3. Since T_3 is already in $b\_trset_{T_1}$, nothing is changed. Then T_1 pre-writes b and pre-writes 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 t4, T_1 switches into the write phase. Both $fcnt_{T_2}$ and $fcnt_{T_3}$ 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 t5, 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 $b\_trset_{T_2}$ becomes $\{T_3\}$ and $fcnt_{T_3}$ becomes 1. After T_2 pre-writes b, it switches into the write phase and $fcnt_{T_3}$ becomes 0 again. Then T_3 executes and also switches into write phase after pre-writing c.

In the above 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, 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 pre-writes d and then T_2 will be aborted by T_1 when T_1 reads d. This example illustrates the advantage of the proposed method over 2PL.

6. Further optimization

In the previous algorithm, a transaction T_L in the read phase is always aborted if it holds a read lock on a data object O and another transaction , T_H, with a higher priority requests a write lock on O. The reason is that since the CPU scheduling gives high priority to T_H, the probability for T_H to commit before T_L is very high. Therefore T_L will most probably be aborted at last. Based on such an assumption, we would rather abort T_L earlier. However, if T has already

completed most of its work, we should avoid aborting T. The solution is to set a point in the transaction. If the transaction reaches this point, it must have finished most of its work and enters into a protection state. This point can be set easily by the compiler near the end of the transaction. LP2 should be changed now as follows:

LP2'. A transaction T requests a write lock on a data object x.

```
<2 for t∈ {Ti | rlock(Ti,x) /\ Ti≠T} do
      if (priority (t) > priority (T) )
      then ( if (T ∈ f_trset_t) then abort T;
             else if (T ∉ b_trset_t)
                  then (b_trset_t := b_trset_t ∪ {T};
                        fcnt_T := fcnt_T + 1;
                       )
           )
2>
      else if (t ∈ wait_trset \/ state (t) = protection)
           then ( if (t ∈ b_trset_T) then abort t;
                  else f_trset_T := f_trset_T ∪ {t};
                )
           else if (t ∈ read_trset) then abort t;

   grant the lock;
```

In this optimized locking protocol, a transaction in the protection state is treated in the same way as one in the wait phase and is not aborted immediately.

7. Conclusions

Time-critical scheduling in real-time database systems consists of two scheduling mechanisms: transaction scheduling and operation scheduling. To find new concurrency control methods in which timing constraints of transactions are taken into account, we have investigated solutions to the operation scheduling aspect of time-critical scheduling.

In this paper, a priority-based concurrency control method for real-time database systems is presented which employs a priority-dependent locking mechanism. By delaying the write operations of each transaction, 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 new method features the ability that allows transactions to meet the timing constraints as much as possible without reducing the concurrency level of the system or increasing the abort rate significantly. In the method, high priority transaction is never blocked by an uncommitted lower priority transaction, while the low priority transaction may not have to be aborted even in face of conflicting operations. In conjunction with a time-critical transaction scheduling policy, or CPU scheduling policy, the method is expected to improve the system performance significantly. The work to extend the method to the distributed environment and the performce evaluation of the method is under way.

Acknowledgment

The authors wish to thank Prof. John Pfaltz for his kind help.

REFERENCES

[Abb88] R. Abbott, and H. Garcia-Molina, "Scheduling real-time transactions: a performance evaluation," in *Proc. 14th VLDB Conf.*, Los Angeles, Aug. 1988.

[Bas88] M. A. Bassiouni, "Single-site and distributed optimistic protocols for concurrency control," *IEEE Trans. Software Eng.*, vol. 14, no. 8, pp. 1071-1080, Aug. 1988.

[Ber87] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesley, 1987.

[Bok87] C. Boksenbaum, M. Cart, J. Ferrie, and J. F. Pons, "Concurrent certifications by intervals of timestamps in distributed database systems," *IEEE Trans. Software Eng.*,vol. SE-13, no. 4, pp. 409-419, Apr. 1987.

[Buch89] A. P. Buchmann, D. R. McCarthy, M. Hsu, and U. Dayal, "Time-critical database scheduling: a framework for integrating real-time scheduling and concurrency control," in *Proc. Data Engineering Conf.*, Los Angeles, Feb. 1989.

[Esw76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, no. 11, pp. 624-633, Nov. 1976.

[Gar83] H. Garcia-Molina, "Using semantic knowledge for transaction processing in a distributed database," *ACM Trans. Database Syst.*, vol. 8, no. 2, pp. 186-213, June 1983.

[Hae84] T. Haerder, "Observations on optimistic concurrency control schemes," *Inform. Syst.*, vol. 9, no. 2, pp. 111-120, 1984.

[Hal89] U. Halici and A. Dogac, "Concurrency control in distributed databases through time intervals and short-term locks," *IEEE Trans. Software Eng.*, vol. 15, no. 8, pp. 994-1003, Aug. 1989.

[Hua89] J. Huang, J. A. Stankovic, D. Towsley, and K. Ramamritham, "Experimental evaluation of real-time transaction processing," in *Proc. Real-time Systems Symp.*, Dec. 1989.

[Kung81] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213-226, June 1981.

[Lin89] K. J. Lin, "Consistency issues in real-time database systems," in *Proc. 22nd Hawaii Intl. Conf. System Sciences*, Hawaii, Jan. 1989.

[Sha88] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Concurrency control for distributed real-time databases," *ACM SIGMOD Record*, vol. 17, no. 1, Mar. 1988.

[Son89] S. H. Son and R. P. Cook, "Scheduling and consistency in real-time database systems," in *Proc. 6th IEEE Workshop Real-Time Operating Systems and Software*, Pittsburgh, Pennsylvania, May 1989, pp 42-45.

[Son90] S. H. Son and C. Chang, "Performance evaluation of real-time locking protocols using a distributed software prototyping environment," to appear in *Proc. 10th Intl. Conf. Distributed Computing Syst.*, Paris, France, June 1990.

[Stan88] J. A. Stankovic and W. Zhao, "On real-time transactions," *ACM SIGMOD Record*, vol. 17, no. 1, Mar. 1988.

[Vrb88] S. V. Vrbsky and K. J. Lin, "Recovering imprecise transactions with real-time constraints," in *Proc. Symp. Reliable Distributed Syst.*, Oct. 1988, pp. 185-193.

[Yu90] P. S. Yu and D. M. Dias, "Concurrency control using locking with deferred blocking," in *Proc. 6th Intl. Conf. Data Engineering.*, Los Angeles, Feb. 1990, pp. 30-36.