A Priority Ceiling Protocol with Dynamic Adjustment of Serialization Order

Kwok-wa Lam\textsuperscript{1}, Sang H. Son\textsuperscript{2} and Sheung-lun Hung\textsuperscript{1}

Department of Computer Science\textsuperscript{1}
City University of Hong Kong,
83 Tat Chee Avenue, Kowloon,
Hong Kong.
kwlam@cs.cityu.edu.hk

Department of Computer Science\textsuperscript{2}
University of Virginia,
Charlottesville, VA 22903,
U.S.A.
son@cs.virginia.edu

Abstract
The difficulties of providing a guarantee of meeting transaction deadlines in hard real-time database systems lie in the problems of priority inversion and of deadlocks. Priority inversion and deadlock problems ensue when concurrency control protocols are adapted in priority-driven scheduling. The blocking delay due to priority inversion can be unbounded, which is unacceptable in the mission-critical real-time applications. Some priority ceiling protocols have been proposed to tackle these two problems. However, they are too conservative in scheduling transactions for the single-blocking and deadlock-free properties, leading to many unnecessary transaction blockings. In this paper, we analyze the unnecessary transaction blocking problem inherent in these priority ceiling protocols and investigate the conditions for allowing a higher priority transaction to preempt a lower priority transaction using the notion of dynamic adjustment of serialization order. A new priority ceiling protocol is proposed to solve the unnecessary blocking problem, thus enhancing schedulability. We also devise the worst-case schedulability analysis for the new protocol which provides a better schedulability condition than other protocols.

Keywords: Priority ceiling protocols, concurrency control, transaction scheduling, real-time systems, database systems.

1. Introduction

In real-time database systems (RTDBS), the correctness of a result depends on not only the logical results and functional behavior of the execution, but also the temporal behavior, i.e. the time when the result is delivered [23]. Having timing constraints, usually in terms of deadlines, characterizes real-time transactions. The notion of timeliness must be considered by the schedulers of RTDBS to guarantee that the deadlines of transactions are met [15]. Failure to meet a transaction’s deadline may result in intolerable system performance degradation, or could lead to catastrophic consequences.

\* This work was supported in part by the Office of Naval Research

Many previous studies have focused on integrating priority-driven scheduling with concurrency control protocols [1,3,8,9,10,20,22]. As their applications do not require stringent timing constraints, most of their protocols adopt the best-effort approach to scheduling transactions without a guarantee of meeting transaction’s deadlines. However, not much work has been devoted to the integration of concurrency control protocols into hard RTDBS in which transactions have stringent timing constraints, i.e., the deadlines must be met. Otherwise, the consequences could be catastrophic. Hard RTDBS are used in a wide range of mission-critical applications such as avionics systems, aerospace systems, robotics and defence systems.

The difficulties of providing a guarantee of meeting hard deadlines of transactions lie in problems of priority inversion and of deadlocks [16] which could occur when concurrency control protocols are adapted to priority-driven scheduling. Concurrency control protocols normally employ transaction blocking in resolving data conflicts among transactions for data consistency [4]. Priority inversion arises when a higher priority transaction \( T_H \) blocks a lower priority transaction \( T_L \) because \( T_H \) requests to lock a data item which has been already locked by \( T_L \). Unfortunately, the duration of priority inversion can be indefinitely long because some other intermediate priority transactions can repeatedly preempt \( T_L \) [16].

The mechanism of priority inheritance has been proposed [14] to solve the priority inversion problem. This mechanism states that when a lower priority transaction \( T_L \) blocks a higher priority transaction \( \bar{T}_L \), \( T_L \) inherits and executes at the priority of \( \bar{T}_L \). \( \bar{T}_L \) will return to its original priority when it releases all of its locks on the resources on which the priority inheritance takes place. However, the priority inheritance protocol does not solve the deadlock problem [16]. Also, a higher priority transaction could be blocked by many lower priority transactions. This chained blocking can make the analysis of the worst-case blocking of a higher priority transaction too pessimistic.

There has been a class of protocols [2,6,16] that introduces the notion of priority ceiling into the priority inheritance protocol. We call this class of protocols priority ceiling protocols. A priority ceiling is defined for each resource. The
priority ceiling of a resource is the priority of the highest priority task that may access the resource. These protocols were originally devised for the synchronization of tasks on exclusive access to the shared resources in real-time systems. These protocols can avoid deadlocks and ensure that the blocking time due to priority inversion is confined to at most the duration of a single critical section. However, these priority ceiling protocols cannot directly be applied to RTDBS because they cannot guarantee the serializable executions of real-time transactions.

Therefore, some studies (e.g., [13,17]) extended the original priority ceiling protocol (PCP) [16] to scheduling transactions in hard RTDBS. While they provide a bound on transaction blocking delay and schedulability analysis, they often suffer from the problem of unnecessary transaction blockings due to their conservative scheduling of transactions in accessing the shared data. The main reason for their conservatism is their implicit assumption that if a transaction's operation conflicts that of another executing transaction, it is presumably unable to preempt the conflicting transaction. In this paper, we show that a higher priority transaction can preempt a lower priority transaction on data conflicts by using the notion of dynamic adjustment of serialization order, avoiding unnecessary blockings. The rest of the paper is organized as follows: Section 2 introduces some related work. In Section 3, we show the problem of unnecessary blockings inherent in some priority ceiling protocols. Section 4 discusses the use of dynamic adjustment of serialization order to avoid the unnecessary blockings. In Section 5, we propose a new priority ceiling protocol with dynamic adjustment of serialization order. Two working examples are used to show the advantages of the new protocol in Section 6. Two important properties, single-blocking and deadlock-free, are shown in Section 7 and its correctness is proved in Section 8. In Section 9, we devise the worst-case schedulability analysis which provides a better schedulability condition than other protocols, and conclude in Section 10.

2. Related Work

The read-write priority ceiling protocol (RW-PCP) proposed by Sha et al. [17] is the first extension of the original PCP [16] in scheduling transactions in hard RTDBS. They used two phase locking (2PL) [7] along with PCP. RW-PCP exploits the semantics of read and write operations of transactions in assigning two different priority ceilings to each data item. Due to the combination of priority ceiling assignment and priority inheritance mechanism, RW-PCP does not need to explicitly check for read/write operation conflicts. RW-PCP avoids deadlocks and guarantees that a transaction can be blocked by a single lower priority transaction only. However, it can cause transaction blockings which, we will show later, are unnecessary.

Nakazato et al. [13] proposed the convex ceiling protocol (CCP) which is also an extension of PCP. CCP reduces the transaction blocking by unlocking the data item with the highest priority ceiling before the end of the transaction. It checks the priority ceiling of those data items to be unlocked when a transaction does not need them any more. If the transaction will not lock any data items with a higher priority ceiling, these data items are unlocked immediately. Although CCP does not adopt 2PL, serializability is still ensured in CCP. CCP reduces the worst case blocking time for some high priority transactions and thus achieves a better schedulability condition than RW-PCP. However, CCP also suffers from the similar problem of unnecessary blockings.

When ceiling blockings occur in RW-PCP and CCP, intermediate priority transactions could be blocked if a lower priority transaction holds a data item with a priority ceiling equal to the system ceiling. In that way, ceiling blocking reduces the system schedulability and increases the blocking time of transactions. Some studies [18,19,21] adopted the abortion strategy for enhancing the system schedulability and reducing the transaction blocking time. While they can reduce the blocking time of transactions at the expense of abortion and re-execution overheads, they complicate the system schedulability analysis. Some cannot even provide the schedulability analysis since they cannot bound the number of abortions that a lower priority transaction may experience [21].

3. Problem Description

In this section, we explain the problem of unnecessary transaction blockings inherent in both PCP [17] and CCP [13]. Although we use RW-PCP as an illustration example, a similar argument can also be applied to CCP. In RW-PCP, there are different priority ceilings of data items as defined as follows.

**Notations:**

- \( W_{ceil}(x) \): denotes the write priority ceiling of a data item \( x \) which is the priority of the highest priority transaction that may write \( x \).
- \( A_{ceil}(x) \): denotes the absolute priority ceiling of a data item \( x \) which is the priority of the highest priority transaction that may read or write \( x \).
- \( RW_{ceil}(x) \): denotes the r/w priority ceiling of a data item \( x \), that is dynamically determined at runtime. When a transaction write-locks \( x \), \( RW_{ceil}(x) \) is set equal to \( A_{ceil}(x) \). When a transaction read-locks \( x \), \( RW_{ceil}(x) \) is set equal to \( W_{ceil}(x) \).
- \( Sys_{ceil}(i) \): denotes the highest priority ceiling of \( RW_{ceil}(x) \) among all data items locked by transactions other than \( T_i \). \( Sys_{ceil} \) is also dynamically determined at runtime when \( T_i \) requests to lock a data item.
- \( P_i \): denotes the priority of transaction \( T_i \).

In RW-PCP, a locking condition to control transactions to access data items is that a transaction \( T_i \) cannot read-lock or write-lock a data item \( x \) unless its priority is higher than \( Sys_{ceil} \), i.e., \( P_i > Sys_{ceil} \). If \( T_i \) is blocked due to lock denial, the transaction that is blocking \( T_i \) inherits the priority of \( T_i \). Under this locking condition, when a transaction write-locks \( x \), no other transaction will be able to lock \( x \). When a transaction read-locks \( x \), only those transactions that may read-lock \( x \) and whose priorities are higher than \( W_{ceil}(x) \) may read-lock \( x \). To disallow lower priority transactions to read \( x \) concurrently is to ensure that when a higher priority transaction needs to write-lock \( x \), it will be blocked by only one lower priority transaction that has read \( x \). Two important properties of RW-PCP are deadlock-free and single-blocking. A transaction with the
property of single-blocking is confined to be blocked by at most one lower priority transaction. These two properties enable schedulability analysis of a set of periodic transactions. However, RW-PCP is too conservative in scheduling transactions to access the shared data, resulting in unnecessary blockings. Consider the following example.

Example 1: Suppose that we have three transactions $T_1$, $T_2$, and $T_3$ in descending order of priority with $T_1$ having the highest priority. Also, assume that they access the data items as follows:

$T_1$: Read(x), $T_2$: Read(y) and $T_3$: Write(x)

The absolute priority ceilings of data items $x$ and $y$:

$Aceil(x) = P_3$, $Aceil(y) = dummy$

Figure 1: Transaction execution in Example 1

Note that the dummy priority is the priority ceiling which is lower than the priorities of all transactions in the system. Figure 1 shows the executions of the three transactions. We now describe the execution of the three transactions as follows:

- At time 0, $T_1$ arrives and write-locks $x$.
- At time 1, $T_2$ arrives and requests to read-lock $y$. Although $y$ is available, $T_2$ is blocked as its priority is not higher than $Sysceil_1$ which is now equal to $P_1$. Hence, $T_3$ inherits $T_2$'s priority since $T_1$ blocks $T_2$.
- At time 2, $T_1$ arrives and requests to read-lock $x$. It is also blocked as its priority is not higher than $Sysceil_1$; either. Again, $T_3$ further inherits $T_1$'s priority.
- At time 3, after $T_1$ completes its execution and releases its lock on $x$, $T_2$ is awakened in turn and continues to execute and completes at time 5.

In Example 1, both transactions $T_1$ and $T_2$ have to wait for the completion of the lower priority transaction $T_3$'s execution. $T_2$ is blocked because $T_3$'s priority is not higher than $Sysceil_1$ though $y$ is available. This type of blocking experienced by $T_2$ is called the ceiling blocking. The ceiling blocking is essential to ensure that a higher priority transaction will be blocked by only one lower-priority transaction and that deadlocks can be avoided. On the other hand, $T_1$ is blocked because $x$ has been write-locked by $T_3$. This type of blocking is called the conflict blocking. The conflict blocking is necessary to maintain the consistency of shared data. However, these two types of blockings in RW-PCP can be avoided in many cases, which we call unnecessary blockings.

4. Background

The main reason for the unnecessary blocking problem associated with RW-PCP and CCP is the underlying transaction model adopted by their concurrency control protocols. Both RW-PCP and CCP assume the update-in-place model in which transaction update operations are in immediate effect [4]. In this model, concurrency control protocols induce a serialization order among conflicting transactions during their executions. The serialization order between two transactions is fixed upon the order of their conflicting access to the shared data. Thus, a higher priority transaction may have no way to precede a lower priority transaction in the serialization order because of previous data conflicts. In Example 1, since $T_1$ has write-locked the data item $x$ before $T_1$ requests to read-lock $x$, the serialization order between $T_1$ and $T_3$ is fixed as $T_3 \rightarrow T_1$. $T_1$ can never precede $T_3$ in the serialization order as long as both transactions exist in the execution history. Thus, $T_3$ is blocked. The blocking of $T_1$ by $T_3$ is undesirable in hard RTDBS as it contradicts the principle of priority-driven scheduling. Thus, it motivated us to employ the notion of dynamic adjustment of serialization order [3,11].

The goal of designing our new protocol is to give critical transactions high priority in accessing the shared data so that they can complete their executions as soon as possible. The fewer the transaction blockings, the better the schedulability conditions for a transaction set. By dynamically adjusting the serialization order among conflicting transactions, the new protocol allows a higher priority transaction to preempt uncommitted lower priority transactions while it prevents lower priority transactions from being restarted even in the face of data conflicts. Although some studies [18,19,21] took the approach of restarting transactions to resolve data conflicts among transactions, restarting complicates the schedulability analysis. Thus, we assume that a transaction preemption will not entail transaction restarts. A transaction preemption means that the transaction is preempted by a higher priority transaction from utilizing the CPU and/or the shared data.

To achieve this goal, we assume that the update-in-workspace model is the underlying transaction model [4]. In the update-in-workspace model, transactions defer their updates until the end of their executions. That is, before a transaction commits, it reads and updates data items only in its private workspace, and then data items are written into the database only upon successful commit. In this model, since the serialization order can be deferred to be fixed at the commitment of transactions, it provides a greater flexibility in resolving data conflicts and in adjusting the serialization order dynamically. Since the serialization order has not yet fixed during transaction executions, higher priority transactions can have more chances to preempt lower priority transactions without causing data consistency problems. Thus, the update-in-workspace model is preferable in priority-driven scheduling. In fact, several previous studies on real-time concurrency controls have adopted this model [8,11,20].
4.1 Dynamic Adjustment of Serialization Order

We now examine the ways of enhancing the chances of a high priority transaction preempting a lower priority transaction by dynamically adjusting the serialization order induced by their data conflicts. One of the important properties of priority-driven scheduling is that a higher priority transaction $T_H$ will continue its execution to completion until $T_H$ is blocked by a lower priority transaction $T_L$ or $T_H$ is preempted by a transaction with a priority higher than $T_H$. Hence, in priority-driven scheduling, $T_L$ may block $T_H$ only when $T_H$ has accessed a data item before $T_H$ requests to access it in the conflicting mode. With this observation, we exploit this priority-driven scheduling property to enhance the preemptability of lower priority transactions by high priority transactions. Suppose $T_L$ has accessed a data item $x$. We consider three possible cases of data conflicts in which $T_L$ will be blocked in other protocols [17,13] and analyze the transaction dependencies induced in the serialization order if $T_H$ preempts $T_L$ for data conflicts.

**Notations:**
- $\text{Read}(x)$ denotes that transaction $T_i$ reads a data item $x$.
- $\text{Write}(x)$ denotes that transaction $T_i$ writes a data item $x$.
- $\text{WriteSet}(T_i)$ denotes the set of data items to be written by transaction $T_i$.
- $\text{DataRead}(T_i)$ denotes the current set of data items that transaction $T_i$ has already read.

**Case 1:** $\text{Write}(x) \cdot \text{Read}(x)$ conflict

In this case, $T_H$ can preempt $T_L$ provided that $T_H$ must commit before $T_L$. If $T_L$ commits first after $T_H$ has preempted $T_L$, $T_L$'s write will invalidate $T_H$'s read. Hence, $T_H$ has to be restarted. But, restarting $T_H$ is forbidden in the new protocol. To ensure that $T_H$ commits after $T_L$, we must make sure that $T_H$ does not have any other data conflicts with $T_L$ in which cases, $T_H$ will be blocked, either directly or indirectly. $T_H$ is able to preempt $T_L$ if $T_H$ can commit before $T_L$, i.e., when the serialization order can be constrained to be $T_H \rightarrow T_L$.

**Case 2:** $\text{Read}(x) \cdot \text{Write}(x)$ conflict

For this conflict, $T_H$ cannot preempt $T_L$ and is blocked. Suppose $T_H$ is allowed to preempt $T_L$ and write $x$. If $T_H$ commits first, $T_H$'s write will invalidate $T_L$'s read and $T_L$ will have to be restarted. In fact, the serialization order between $T_L$ and $T_H$ has to be $T_L \rightarrow T_H$. In this case, it precludes $T_H$ from preempting $T_L$.

**Case 3:** $\text{Write}(x) \cdot \text{Write}(y)$ conflict

These two write operations are blind write operations. $T_H$ can preempt $T_L$ because, whether $T_H$ or $T_L$ commits first will not compromise the data consistency and will not restart one of them. Thus, the serialization order between $T_L$ and $T_H$ can be either $T_L \rightarrow T_H$ or $T_H \rightarrow T_L$ depending on which one commits first. No constraint is induced in the serialization order between them.

Let us elaborate Case 3 further. If we consider other types of data conflicts together, i.e., $\text{Write}(y) \cdot \text{Read}(x)$ or $\text{Read}(x) \cdot \text{Write}(y)$, this write-write conflict has to conform to the serialization order constrained by these data conflicts. However, in the following example, we show that for this write-write conflict, even if $T_H$ and $T_L$ have other data conflicts, no constraint in the serialization order will be induced and the consistency of shared data will not be compromised.

**Example 2:** Suppose that $T_i$ and $T_H$ have $\text{Write}(y) \cdot \text{Write}(y)$ data conflict on a data item $y$ and that they also have other data conflicts in one of the following two types:

**Type 1:** $\text{Write}(x) \cdot \text{Read}(x)$ conflict

Consider the following two situations:

1. $\text{Write}(x) \cdot \text{Read}(x)$ preceding $\text{Write}(y) \cdot \text{Write}(y)$
   - If $T_i$ is allowed to preempt $T_H$, then the history of the committed transaction execution will be $\text{Read}(x); \text{Write}(y); \text{Write}(x); \text{Write}(y)$.
   - It is still serializable execution and the serialization order between $T_L$ and $T_H$ will be $T_H \rightarrow T_L$.

2. $\text{Write}(y) \cdot \text{Write}(y)$ preceding $\text{Write}(x) \cdot \text{Read}(x)$
   - If $T_H$ is allowed to preempt $T_i$, then the history of the committed transaction execution will be $\text{Write}(y); \text{Read}(x); \text{Write}(y); \text{Write}(x)$.
   - It is still serializable execution with the serialization order $T_H \rightarrow T_L$.

Note that as discussed in Case 1, $T_H$ must commit before $T_L$ in both situations (1) and (2). Otherwise, $T_i$'s write on $x$ will invalidate $T_H$'s read on $x$ and then $T_H$ must be restarted.

**Type 2:** $\text{Read}(x) \cdot \text{Write}(y)$ conflict

Again, consider the following two situations:

1. $\text{Write}(x) \cdot \text{Write}(y)$ preceding $\text{Write}(y) \cdot \text{Write}(y)$
   - If they had this type of data conflict, $T_H$ would have been blocked by $T_L$ and it would not be possible to have succeeding $\text{Write}(x) \cdot \text{Write}(y)$ data conflict between $T_L$ and $T_H$.

2. $\text{Write}(y) \cdot \text{Write}(y)$ preceding $\text{Read}(x) \cdot \text{Write}(x)$
   - $T_H$ would also be blocked by $T_i$ when it requests to write-lock $x$. Thus, $T_i$ must commit before $T_H$. Then, the committed transaction history will be $\text{Write}(y); \text{Read}(x); \text{Write}(y); \text{Write}(x)$.
   - It is still serializable execution with serialization order $T_L \rightarrow T_H$.

In the foregoing analysis, it is easy to see that two transaction's write operations can be considered non-conflicting operations as the access order of two write operations will not constrain the serialization order between the two transactions. It is also observed that transaction's write operations are preemptable operations in the sense that although they conflict the read/write operations of higher priority transactions, the higher priority transactions can still preempt them under certain condition. On the other hand, transaction's read operations are non-preemptable operations since read operations must block the write operations of higher priority transactions for data consistency. Of course, as in RW-PCP, $T_H$ can preempt $T_i$ under certain condition if $T_H$ requests to read-lock a data item which has only been read-locked by $T_i$.

<table>
<thead>
<tr>
<th>$T_i$ has locked a data item</th>
<th>$T_H$ is requesting a lock on the data item</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\text{Read}$-lock OK</td>
</tr>
<tr>
<td></td>
<td>$\text{Write}$-lock NOK</td>
</tr>
</tbody>
</table>

Table 1: The lock compatibility table

* under the condition $\text{DataRead}(T_i) \cap \text{WriteSet}(T_H) = \emptyset$
Hence, a sufficient condition to prevent $T_i$ from preempting $T_k$ is $\text{DataRead}(T_i) \cap \text{WriteSet}(T_k) \neq \emptyset$, in which case, $T_k$ will block $T_i$ due to read-write conflict as discussed in Case 2. It means that $T_i$ is not allowed to preempt $T_k$ in Case 1 unless $\text{DataRead}(T_i) \cap \text{WriteSet}(T_k) = \emptyset$ is true for the guarantee that $T_i$ commits before $T_k$.

Table 1 illustrates the lock compatibility among transactions. Note that Table 1 shows the necessary condition to allow a transaction to lock a data item for both the requirements of data consistency and of no transaction restarts. However, it is not yet a sufficient condition to maintain the single-blocking and deadlock-free properties. Thus, we also utilize the priority ceiling mechanism for these two important properties.

### 4.2 Priority Ceiling Management

We utilize the priority ceiling assignment so that lower priority transactions will be blocked earlier for the single-blocking property. In most priority ceiling protocols [13,17], two types of priority ceilings are defined for each data item for the semantics of transaction’s read/write operations. The purpose of priority ceilings is to block the operations of lower priority transactions that may conflict and block those of higher priority transactions in advance, so that a higher priority transaction will only be blocked by a single lower priority transaction.

In our new protocol, only one priority ceiling is needed for each data item although the system utilizes the context of transaction’s read/write operations. As discussed in Section 4.1, the transaction’s write operations are preemptable operations. That is, under no circumstances will they block the read/write operations of higher priority transactions. However, the read operations are non-preemptable operations that may block the write operations of higher priority transactions. Therefore, it is not necessary to define a priority ceiling to control the write operations. On the other hand, read operations need to be controlled to prevent two concurrent read operations from blocking the write operation of a higher priority transaction. Hence, one priority ceiling, called write priority ceiling, is used to control read operations of transactions.

### 5. Protocol Description

We assume that the system has a single processor with a memory resident database and that all transactions are periodic transactions with rate monotonic priority assignment [12]. That is, a transaction with a shorter period is assigned a higher priority and the deadline of a transaction is at the end of its period. The priorities of transactions are of a total order. This initial priority of a transaction is called the original priority. Due to the mechanism of priority inheritance, a transaction’s priority may change during its execution. It could monotonically increase until it commits at the end of its execution. Transactions are assigned to the processor based on their current running priorities that are initially equal to their original priorities. If a transaction blocks a higher priority transaction, its running priority will inherit that of the higher priority transaction. The transaction with the highest running priority ready to execute is scheduled to run on the processor.

Before a transaction can read or write a data item $x$, it must first obtain the read-lock or write-lock on the data item $x$ respectively. All the locks will be released after the commitment of the transaction.

Although we use the terms write priority ceiling of data items and the system priority ceiling, their semantics are slightly different from those of RW-PCP. Like RW-PCP, each data item $x$ is statically assigned a priori a write priority ceiling, $W(x)$ which is the priority of the highest priority transaction that may write $x$. $W(x)$ will come into effect only if $x$ is read-locked by a transaction. The Sysceil denotes the highest $W(x)$ among all data items read-locked by transactions other than the executing transaction $T_i$.

**Notations:**

$T_1,..., T_n$ denotes an ordered set of transactions which are listed in descending order of priority, with $T_1$ having the highest priority.

$\text{Rlock}(x)$ denotes that $T_i$ requests to read-lock $x$.

$\text{Wlock}(x)$ denotes that $T_i$ requests to write-lock $x$.

$O_i(x)$ denotes a requesting locking operation of $T_i$ on a data item $x$ such that $O_i(x) = \text{Rlock}(x)$ or $O_i(x) = \text{Wlock}(x)$.

$\text{No_Rlock}(x)$ denotes a data item $x$ that is not being read-locked by transactions other than $T_i$ when $T_i$ requests to lock $x$.

$\text{HPW}(x)$ denotes the highest priority of transactions that may write $x$.

$T^*$ denotes the transaction holding the data item $x$ whose write priority ceiling is equal to Sysceil.

Note that for simplicity of exposition, we use one more variable $\text{HPW}(x)$ instead of $W(x)$ because $W(x)$ comes into effect only when $x$ is read-locked. Since the new protocol employs the notion of dynamic adjustment of serialization order, we refer to it as PCP-DA. Now we define the locking conditions of PCP-DA. A transaction $T_i$ is allowed to read-lock or write-lock a data item $x$ if one of the locking conditions is true.

**Locking Conditions:**

**LC1:** $T_i$ requests a write-lock on $x$ and $x$ is not being read-locked by other transactions, i.e., $O_i(x) = \text{Wlock}(x)$ and $\text{No_Rlock}(x)$.

**LC2:** $T_i$ requests a read-lock on $x$ and $T_i$’s priority is higher than the highest write priority ceiling of data items read-locked by other transactions, i.e., $O_i(x) = \text{Rlock}(x)$ and $P_i > \text{HPW}(x)$ and $x \notin \text{WriteSet}(T^*)$.

**LC3:** $T_i$ requests a read-lock on $x$ and $T_i$’s priority is higher than the highest priority of transaction that may write $x$ and $x$ is not in the write set of $T^*$, i.e., $O_i(x) = \text{Rlock}(x)$ and $P_i > \text{HPW}(x)$ and $x \notin \text{WriteSet}(T^*)$.

**LC4:** $T_i$ requests a read-lock on $x$ and $T_i$’s priority is equal to the highest priority of transaction that may write $x$ and $x$ is not being read-locked by other transactions and $x$ is not in the write set of $T^*$, i.e., $O_i(x) = \text{Rlock}(x)$ and $P_i = \text{HPW}(x)$ and $x \notin \text{WriteSet}(T^*)$.

For data consistency requirements, LC1 needs to explicitly check for the write/read lock conflicts according to the lock compatibility table in Table 1. If there has been a read lock on data item $x$ by another transaction, no subsequent write-lock on $x$ can be granted to $T_i$. LC2 ensures that none of the data items
already read-locked will be write-locked by any transaction with a priority higher than or equal to $T_i$. LC3 ensures that $T_i$‘s priority is higher than the highest priority transaction that may write-lock the data item $x$ and that $x$ will not be write-locked by $T^*$. The latter condition guarantees that $T_i$ will not block $T^*$ because $T^*$ may inherit a priority level higher than $T_i$‘s priority when $T^*$ blocks another transaction with a priority higher than $T_i$‘s priority. As we will prove later, $T^*$ will be unique with respect to $T_i$. Note that neither LC2 nor LC3 need to explicitly check the condition $\text{DataRead}(T^*) \cap \text{WriteSet}(T_i) = \emptyset$ for the read/write lock conflicts as required in the lock compatibility table in Table 1. That is because in both LC2 and LC3, $T_i$ will not request a write-lock on the existing read-locked data items. Since $P_i = \text{HPW}(x)$ in LC4, $T_i$ is the highest priority of the transaction which writes the data item $x$. In that situation, we need to check the truth of the condition.

It can be observed that PCP-DA enhances transaction preemptibility in two ways. First, no priority ceiling is needed for write operations in which case Sysceil, with respect to a transaction $T_i$, may not be raised as high as other priority ceiling protocols do. Second, it provides more locking conditions, particularly LC3 and LC4, for transactions to access data. Having no priority ceiling for controlling transactions’ write operations has a significant impact on the system schedulability which is greatly affected by the priority ceiling levels that have been raised for the locked data items. The lower priority ceiling levels are raised, the more transactions can be scheduled. In RW-PCP, for any data item $x$, $\text{Accel}(x) \geq \text{Wcel}(x)$. Once $x$ is write-locked, with respect to $T_i$, $\text{Sysceil}_i = \text{Accel}(x)$. Hence, no transaction including $T_i$ can access $x$ any more. However, in PCP-DA, no priority ceiling will be raised for write operations. Sysceil will only be raised as high as $\text{Wcel}(x)$.

These enhancements can reduce the number of both 
\textit{ceiling blockings} and \textit{conflict blockings} occurring in RW-PCP, as discussed in Section 3. Thus, compared to RW-PCP, some transaction blockings that may happen under RW-PCP can be avoided. On the other hand, transaction blocking that happens under PCP-DA must happen under RW-PCP.

6. Working Examples

\textbf{Example 3 :} Suppose that we have two transactions $T_1$ and $T_2$ with $T_1$ having a higher priority. Without loss of generality, we assume the period of $T_1$ is 5. The data items accessed by the two transactions are as follows:

$T_1: \text{Read}(x), \text{Read}(y) \quad T_2: \text{Write}(x), \text{Write}(y)$

The write priority ceilings of $x$ and $y$ are:

$\text{Wcel}(x) = P_2, \text{Wcel}(y) = P_2$

The execution of the two transactions under PCP-DA is shown in Figure 2. We now describe the execution of the two transactions as follows:

- At time 0, $T_2$ arrives and requests to write-lock $x$. Since no other transaction is holding a read-lock on $x$, LC1 is true. Then, $T_2$ is allowed to write-lock $x$.
- At time 1, $T_1$ arrives and requests to read-lock $x$. Since $\text{Sysceil}_1 = \text{dummy}$ and $P_i > \text{Sysceil}_1$, LC2 is true. Thus, $T_1$ is allowed to read-lock $x$ and preempts $T_2$.

- At time 2, since LC2 is true again, $T_1$ read-locks $y$ subsequently.
- At time 3, $T_1$ finishes its execution and releases its locks. $T_2$ resumes to continue its execution.
- At time 5, $T_2$ requests to write-lock $y$. Since LC1 is true, $T_2$ write-locks $y$.
- At time 6, $T_1$ arrives in its next period. $T_1$ can read-lock both $x$ and $y$ at time 6 and 7 respectively because LC2 is true.
- At time 8, $T_1$ completes its execution and releases its locks. $T_2$ resumes again to continue its execution and competes its execution at time 9.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{Transaction Execution in Example 3 under PCP-DA}
\end{figure}

For comparison, we also show the execution of the two transactions under RW-PCP in Figure 3. We will not describe the execution under RW-PCP. Interested readers are referred to [17]. It can be easily seen that under RW-PCP, the worst case effective blocking time of $T_1$ by $T_2$ is 4 time units. In the worst case, $T_1$ may be blocked after $T_2$ completes 1 time units of execution until $T_1$ completes 5 time units of execution. The first instance of $T_1$ is blocked by $T_2$ from time 1 to 5 and $T_1$ misses its deadline at time 6. In contrast, under PCP-DA, the worst case blocking time of $T_1$ is avoided because $T_1$ is allowed to gain access to $x$ and $y$ although $x$ and $y$ are being write-locked by $T_2$. That is, $T_1$ does not experience blocking in this transaction set. This example shows that the schedulability condition of PCP-DA is better than that of RW-PCP.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.png}
\caption{Transaction Execution in Example 3 under RW-PCP}
\end{figure}

In Example 3, under RW-PCP, $T_1$ experiences the \textit{conflict blocking} when it requests to read-lock $x$ and $y$ which are being write-locked by $T_2$. PCP-DA can avoid this unnecessary \textit{conflict blocking}. In the following example, we will show that PCP-DA can also avoid some \textit{ceiling blockings} occurring in RW-PCP. Moreover, we will also illustrate that Sysceil with
respect to a transaction $T_i$ under PCP-DA will not be as high as in RW-PCP.

**Example 4**: Suppose that we have four transactions $T_1$, $T_2$, $T_3$, and $T_4$ in descending order of priority with $T_1$ having the highest priority. The data items accessed by the four transactions are as follows:

- $T_1$ : Read(x)
- $T_2$ : Write(y)
- $T_3$ : Read(z), Write(z)
- $T_4$ : Read(y), Write(x)

The write priority ceilings of $x$, $y$, and $z$:

$Wceil(x) = P_1$; $Wceil(y) = P_2$; $Wceil(z) = P_3$.

![Figure 4: Transaction Execution in Example 4 under PCP-DA](image)

Figure 4 shows the execution of the four transactions under PCP-DA. The description of the execution of the four transactions is as follows:

- At time 0, $T_1$ arrives and requests to read-lock $y$. Since no data item is being locked, $Sysceil_4$ is dummy. Thus, $T_4$ is allowed to read-lock $y$.
- At time 1, $T_1$ arrives and requests to read-lock $z$. Since $P_1 < Sysceil_4 = P_3$, LC2 is false. However, LC4 is true because $T^* = T_4$ and $z \not\in \text{WriteSet}(T_4)$. Therefore, $T_2$ preempts $T_4$ and is allowed to read-lock $z$.
- At time 2, since no other transaction is holding a read-lock on $z$ except $T_1$ itself, LC1 is true. Thus, $T_2$ write-locks $z$.
- At time 3, $T_3$ completes its execution and releases its locks. Then, $T_4$ resumes to continue its execution. Since LC1 is true, $T_4$ write-locks $x$.
- At time 4, $T_1$ arrives and requests to read-lock $x$. Since $P_1 > Sysceil_4 = P_2$, LC2 is true. Then, $T_1$ preempts $T_4$ and read-locks $x$.
- At time 6, $T_1$ completes its execution and releases its locks. Then, $T_4$ is awakened to continue its execution.
- At time 9, $T_4$ completes its execution and releases its locks. Also, $T_3$ arrives and requests to write-lock $y$. Since LC1 is true, $T_4$ write-locks $y$ and completes its execution at time 11.

Figure 5 also shows the execution of the four transactions under RW-PCP. It can be easily seen that $T_3$ encounters ceiling blocking since its priority is not higher than $Sysceil_4$, which is currently equal to $P_3$. $T_1$ experiences conflict blocking since $x$ has already been write-locked by $T_1$. The effective blocking times of $T_1$ and $T_3$ blocked by $T_4$ are 1 and 4 time units respectively. Again, PCP-DA can avoid these two types of unnecessary blockings.

The dotted lines in both Figure 4 and Figure 5 indicate the maximum system priority ceiling, $Max_{Sysceil}$, during transaction executions, such that $Max_{Sysceil} \geq Sysceil_i$ for all transaction $T_i$. It is easily seen that $Max_{Sysceil}$ under PCP-DA is as high as $P_2$ and is equal to dummy when no data item is being read-locked at time 9. On the other hand, under RW-PCP, $Max_{Sysceil}$ is as high as $P_1$ and will only be equal to dummy when no transaction is being executed. The level of $Max_{Sysceil}$ has negative impact on the system schedulability.

The higher it is, the more restrictions are placed on transactions accessing the shared data. Thus, the push-down of $Max_{Sysceil}$ is one of the main advantages of PCP-DA over RW-PCP.

![Figure 5: Transaction Execution in Example 4 under RW-PCP](image)

### 7. Single-blocking and Deadlock-free

It is shown that RW-PCP [17] has two important properties: single-blocking and deadlock-free. With these properties, one can determine the schedulability conditions for a given transaction set before the system actually executes the transaction set. We now show that PCP-DA also has these two properties. In the following proofs, when the priority of a transaction $T_i$ is being referred to, it always refers to the priority level at which $T_i$ is currently executing, i.e., its running priority, unless explicitly stated otherwise ($T_i$’s original priority).

**Lemma 1**: Suppose that transaction $T_i$ write-locks a data item $x$ because LC1 is true. Then $x$ write-locked by $T_i$ cannot block a higher priority transaction $T_H$.

**Proof**: Since the transaction’s write operations are preemptable operations and from the definition of PCP-DA, no priority ceiling will take effect for the data items that have been write-locked, the Lemma follows.

**Lemma 2**: A transaction $T_i$ can block a higher priority transaction $T_H$ only if $T_i$ has read-locked at least one data item.

**Proof**: From the definition of PCP-DA, if $T_i$ has not read-locked a data item $x$, $Wceil(x)$ will not come into effect and $Sysceil_i$ with respect to $T_H$ will not be raised subsequently. Hence, $T_H$ can always preempt $T_i$. Since priority inheritance is
in effect and the highest priority transaction that is ready will always be executing. \( T_i \) cannot resume its execution until \( T_h \) completes. The Lemma follows.

**Lemma 3**: The priority inherited by a transaction cannot be higher than the highest write priority ceiling of the data items the transaction has read-locked.

**Proof**: By Lemma 1, a transaction \( T_i \)’s write operations will not block a higher priority transaction \( T_h \). Thus, no priority inheritance occurs for \( T_i \’s \) write operations. By Lemma 2, \( T_i \) can block \( T_h \) only when \( T_i \) has read-locked some data items. With the priority inheritance mechanism, when \( T_i \) read-locks a set of data items, the highest priority level \( L \) can inherit is equal to the highest write priority ceiling of data items read-locked by \( T_i \).

**Lemma 4**: If a transaction \( T_i \) blocks a higher priority transaction \( T_h \), \( T_i \) must have read-locked at least one data item \( x \) such that \( Wceil(x) \geq P_h \) when \( T_h \) is denied to lock a data item.

**Proof**: By Lemma 2, to block other transactions, \( T_i \) must have read-locked some data items. Let \( h \) be the data item that has the highest write priority ceiling among all data items read-locked by \( T_i \). Assume that \( Wceil(h) < P_h \), and when \( T_h \) is still executing, consider the four locking conditions:

**LC1**: Suppose \( T_h \) will write-lock a data item \( x \) which has been read-locked by \( T_i \), then \( Wceil(x) \geq P_h \). However, it contradicts our assumption \( Wceil(h) < P_h \). It implies that \( T_i \) will not write-lock any data items which have been read-locked by \( T_i \). Thus, \( T_i \) will not block \( T_h \).

**LC2**: Suppose \( T_i \) read-locks a data item \( x \) because \( LC3 \) is true. Then, \( P_h > HPW(x) = Wceil(x) \). It implies that \( T_i \) will not write-lock \( x \). For \( Wceil(h) < P_h \), it means that \( T_i \) cannot block \( T_h \).

**LC3**: Suppose \( T_i \) read-locks a data item \( x \) because \( LC4 \) is true. Then, \( P_h = HPW(x) = Wceil(x) \). It implies that \( T_i \) is the highest priority transaction that will write-lock \( x \). Since LC4 ensures that \( x \) is only being write-locked by other transactions, it means that \( T_i \) has not read-locked \( x \). Thus, \( T_i \) cannot block \( T_h \).

Since \( T_i \) cannot block \( T_h \) in the four locking conditions, it contradicts our assumption. The Lemma follows.

**Lemma 5**: When a transaction \( T_h \) is executing, at most one transaction \( T_i \) among the currently executing transactions with priorities lower than \( P_h \) has read-locked the data item \( x \) such that \( Wceil(x) \geq P_h \).

**Proof**: Suppose that another lower priority transaction \( T_k \) has read-locked a data item \( y \) such that \( Wceil(y) \geq P_h \) and that, without loss of generality, \( T_k \) read-locked \( y \) first. When \( T_i \) requests to read-lock \( x \), it notes that \( y \) has been read-locked by \( T_k \) and that \( P_i < P_h \leq Wceil(x), Wceil(y) \). Hence, \( LC2, LC3 \) and \( LC4 \) are all false and \( T_i \) will not be allowed to read-lock \( x \). It contradicts our assumption. The Lemma follows.

**Lemma 6**: Suppose that when transaction \( T_i \) requests to read-lock a data item \( x \), \( LC2 \) is false. Then, the transaction holding \( x \), \( T^* \), must be unique and \( x = Sysceil \).

**Proof**: If \( LC2 \) is false, then \( Wceil(x) \geq P_h \). By Lemma 5, \( T^* \) must be unique and \( x = Sysceil \).

**Theorem 1**: A transaction \( T_h \) can be blocked by at most a single lower priority transaction.

**Proof**: By Lemma 4, \( T_h \) can only be blocked by those lower priority transactions which have read-locked a data item \( x \) such that \( Wceil(x) \geq P_h \). By Lemma 5, when \( T_h \) is executing, at most one lower priority transaction \( T_i \) has read-locked a data item \( x \) such that \( Wceil(x) \geq P_h \). It implies that at most one lower priority transaction can block \( T_h \). The Theorem follows.

Based on the above proofs, the following two locking conditions suffice to maintain the single-blocking property:

1. \( P_i > Sysceil \).
2. \( P_i \geq HPW(x) \).

The first condition ensures that none of the data items already read-locked will be write-locked by any transaction with a higher priority than \( T_i \). The second condition ensures that data item \( x \) will not be write-locked by any transaction with a priority higher than \( P_i \). Therefore, either condition can ensure that a higher priority transaction can be blocked by at most a single lower priority transaction and thus no chained blocking occurs. However, condition (2) cannot avoid deadlocks in the system. The following example shows an occurrence of a deadlock:

**Example 5**: Suppose that there are two transactions \( T_h \) and \( T_l \) with \( P_h > P_l \). The data items accessed by them are as follows: \( T_h \): Read\((y)\), Write\((x)\) \( T_l \): Read\((x)\), Write\((y)\). Thus, \( Wceil(x) = P_h \) and \( Wceil(y) = P_l \). Suppose \( T_i \) read-locks \( y \) first because condition (1) is true. Then when \( T_h \) arrives, it preempts \( T_l \) and read-locks \( y \) because condition (2) is true. When \( T_h \) requests to write-lock \( x \), \( T_i \) is blocked by \( T_l \) since \( x \) is being read-locked by \( T_l \). Then, \( T_l \) inherits \( T_h \)'s priority and continues its execution. When \( T_l \) requests to write-lock \( y \), it is in turn blocked by \( T_h \) thus resulting in a deadlock. Therefore, condition (2) needs to be more restrictive to avoid deadlocks, leading to the derivations of the LC3 and LC4 of the locking conditions.

In the following, we prove the deadlock-free property. To have a deadlock, there must be a circular-wait situation. In a circular-wait situation, each transaction in the cycle has locked some data items while waiting to lock a data item which is being locked by another transaction. In PCP-DA, write operations are preemptable operations. Hence, by Lemma 2, a transaction can block other transactions only if it has read-locked at least one data item.

**Lemma 7**: Suppose that a transaction \( T_h \) read-locks a data item \( x \) when \( LC2 \) is true. Then \( T_h \) cannot be blocked by a lower-priority transaction until \( T_h \) completes its execution.

**Proof**: Since \( LC2 \) is true, \( P_h > Wceil(x) \). It means that none of the data items already read-locked will be write-locked by any transaction with a priority higher than or equal to \( T_h \) (including \( T_h \)). By Lemma 3, no lower priority transaction can inherit a priority level greater than or equal to \( P_h \). It implies that no lower priority transaction can block \( T_h \) or any other higher priority transaction. Moreover, in the priority-driven scheduling, no arriving transaction with priority lower than \( P_h \) can preempt \( T_h \). Thus, \( T_h \) cannot be blocked by lower priority transactions before \( T_h \) completes its execution.

**Lemma 8**: Suppose that a transaction \( T_i \) read-locks a data item \( x \) since either \( LC3 \) or \( LC4 \) is true while \( LC2 \) is false. Then, the read-lock on \( x \) by \( T_i \) cannot block \( T^* \).
Consider the following two situations:

(1) If LC3 is true, \( P_i > Wceil(x) \). Hence, no higher priority transaction (including \( T_i \)) will write-lock \( x \).

(2) If LC4 is true, \( P_i = Wceil(x) \). Hence, no transaction with a priority higher than \( T_i \) will write-lock \( x \). However, \( T_i \) is the highest transaction that will write-lock \( x \). For No_Rlock(\( x \)) is true, LC4 ensures that \( T_i \) will not be blocked by the current executing lower priority transactions.

For both situations, \( T^* \) is the only transaction that can inherit a block \( T^* \) even if \( T^* \) has inherited a higher priority. The proof follows.

**Theorem 2**: PCP-DA prevents deadlocks.

**Proof**: By Lemma 2, a lower priority transaction \( T_L \) can block a higher priority transaction \( T_H \) only when \( T_L \) has read-locked at least a data item. By Theorem 1, the number of transactions in the blocking cycle can only be two. Suppose that \( T_L \), which had read-locked a data item was preempted by \( T_H \) which later attempted to read-lock another data item. For \( T_H \) to read-lock a data item, consider the following two situations:

(1) If LC2 is true, by Lemma 7, \( T_L \) cannot block \( T_H \) even if \( T_L \) has inherited a higher priority. The Lemma follows.

(2) If either LC3 or LC4 is true, by Lemma 6, \( T_i = T^* \). By Lemma 8, \( T_H \) cannot block \( T_i \) even if \( T_i \) has inherited a priority higher than \( P_H \).

Hence, a deadlock cannot occur. The Theorem follows.

## 8. Correctness of PCP-DA

In this section, we prove the correctness of PCP-DA. For the correctness of PCP-DA, we prove that all histories produced by PCP-DA are serializable. That is, we have to show that there is no cycle in the serialization graph, \( SG(H) \) in any history \( H \) produced by PCP-DA [4].

**Lemma 9**: A successfully-committed transaction cannot have write-read conflicts with the currently executing transactions.

**Proof**: Suppose \( T_H \) has write-read conflicts with a currently executing transaction \( T_i \). The condition, DataRead(\( T_L \)) \( \cap \) WriteSet(\( T_H \)) \( \neq \emptyset \), suffices to prevent \( T_H \) from preempting \( T_L \). In this case, \( T_H \) would be blocked and would not have committed before \( T_L \), which contradicts our assumption. Hence, the Lemma follows.

**Theorem 3**: If \( H \) is a history produced by PCP-DA, then \( H \) is serializable.

**Proof**: Suppose that \( T_i \to T_j \) (\( j \neq i \)) is an edge of \( SG(H) \). Then, \( T_i \) and \( T_j \) are committed in \( H \) and there are conflicting operations \( q(x) \) and \( p(x) \) such that \( q(x) \) precedes \( p(x) \). For the same edge, a conflict is not considered to occur if \( T_i \) is committed before \( T_j \) accesses \( x \). As discussed in Section 4.1, two write operations are non-conflicting operations. Hence, \( q(x) \) and \( p(x) \) can only be either Read(x) and Write(x) or Write(x) and Read(x). By Lemma 9, \( q(x) \) and \( p(x) \) should be Read(x) and Write(x) respectively. We claim that the commitment of \( T_i \) precedes that of \( T_j \). Suppose \( T_i \) was committed first. When \( T_j \) commits, Write(x) must have been processed. Hence Write(x) precedes Read(x) and conflicts with it. Since \( T_j \) is currently executing, \( T_j \) cannot commit since it violates Lemma 9. It contradicts our assumption that \( T_i \) is committed before \( T_j \) in \( H \). Therefore, it can be shown that if \( T_j \to T_i \) is in \( SG(H) \), \( T_j \) must be committed before \( T_i \) in \( H \). By induction, if a cycle exists in \( SG(H) \), every transaction on that cycle would have to be committed before itself which is impossible. Thus, \( SG(H) \) is acyclic and \( H \) is serializable. The Theorem follows.

## 9. Worst Case Schedulability Analysis

**Notations**:  
- \( C_i \) denotes the execution time of \( T_i \); 
- \( Pd_i \) denotes the period of \( T_i \); 
- \( B_i \) denotes the worst-case blocking time of transaction \( T_i \); 
- \( BTS \) denotes the set of transactions that may block \( T_i \).

With the single-blocking and deadlock-free properties, one can perform the schedulability analysis of a transaction set which uses the rate-monotonic priority assignment and PCP-DA. For schedulability condition, it has been proved in [17] that a set of \( n \) periodic transactions using rate-monotonic priority assignment under RW-PCP can always meet their deadlines if the following conditions are satisfied:

\[
\forall i, 1 \leq i \leq n, \frac{C_i}{Pd_i} + \frac{C_j}{Pd_j} + \ldots + \frac{C_n}{Pd_n} + \frac{B_i}{Pd_i} \leq \left( \frac{1}{2^i - 1} \right)
\]

It can be easily seen that the above schedulability conditions are also applicable to PCP-DA. The schedulability condition for a transaction set depends on the value of \( B_i \). The smaller the value of \( B_i \) is, the better the schedulability condition. We now determine the value of \( B_i \) in PCP-DA and compare it with that in RW-PCP as follows:

In PCP-DA, since write operations are preemptable, only read operations of lower priority transactions may block the write operations of higher priority transactions. From Lemma 4, a transaction \( T_i \) with a priority lower than \( P_i \) may block \( T_i \) if \( T_i \) reads a data item \( x \) such that \( Wceil(x) \geq P_i \). Hence, we can define the blocking transaction set, \( BTS_i \), of \( T_i \) to be a set of transactions with priorities lower than \( P_i \) that may read a data item \( x \) such that \( Wceil(x) \geq P_i \). We have

\[
BTS_i = \{ T_L | P_L < P_i \text{ and } T_L \text{ reads } x \text{ and } Wceil(x) \geq P_i \}
\]

On the other hand, RW-PCP as shown in [17] has \( BTS_i \):

\[
BTS_i = \{ T_L | P_L < P_i \text{ and } (T_L \text{ reads } x \text{ and } Wceil(x) \geq P_i \text{ or } T_L \text{ writes } x \text{ and } Aceil(x) \geq P_i) \}
\]

For both PCP-DA and RW-PCP, the worst case blocking time of \( T_i \): \( B_i = \text{Maximum of } C_i \) for all \( T_i \in BTS_i \).

It can be observed that \( BTS_i \) in RW-PCP is a superset of that in PCP-DA. If the worst case blocking time \( B_i \) occurs in RW-PCP when \( T_i \) only writes \( x \in Aceil(x) \) which is greater than \( P_i \), the value of \( B_i \) can be reduced in PCP-DA because \( T_i \) will not be included in \( BTS_i \) in PCP-DA.
10. Conclusions

Hard real-time transactions have stringent timing constraints in RTDBS. For the serializability of transaction executions, concurrency control protocols employ blockings to resolve data conflicts among transactions when transactions concurrently access the shared data. The concurrency control protocols adapted in priority-driven scheduling pose the priority inversion problems. Unfortunately, transaction blockings due to priority inversion can be indefinitely long. This unpredictability of transaction execution is unacceptable in most mission-critical applications. Thus, a good scheduling protocol must provide a tight bound on the priority inversion delay of transactions.

Several priority ceiling protocols [13,17] have been proposed to provide this bound and the worst-case schedulability analysis. They avoid deadlocks and guarantee that a transaction is blocked at most once by a lower priority transaction. However, they are too conservative in scheduling transactions to access the shared data because they implicitly assume no transaction preemptions upon data conflicts between transactions. As a result, some transaction blockings may be unnecessary, thus reducing the system schedulability.

In this paper, we have analyzed two types of unnecessary transaction blockings inherent in these priority ceiling protocols. We have investigated the conditions under which a higher priority transaction can preempt a lower priority transaction by dynamically adjusting the serialization order between them. A new priority ceiling protocol with dynamic adjustment of serialization order, PCP-DA, is proposed to enhance the system schedulability in hard RTDBS. We have shown that PCP-DA can avoid these two types of unnecessary blockings. We have also shown that PCP-DA is deadlock-free and ensures that a transaction can be blocked at most once by a lower priority transaction. We have devised the worst-case schedulability analysis using PCP-DA, which provides a better schedulability condition compared to other protocols.

References