A Hierarchical Precedence Concurrency Control Protocol for High Data Contention Database Environments

Weidong Xiong
Mohammed Hamdi
Feng Yu
Wen-Chi Hou

1Department of Computer Science, Southern Illinois University, Carbondale, IL 62902
2Department of Computer Science and Information Systems, Youngstown State University, Youngstown, OH 44555

(wxiong, mhamdi)@siu.edu, fyu@ysu.edu, hou@cs.siu.edu

Abstract

In this paper, we propose a family of concurrency control protocols, called the Hierarchical Precedence Concurrency Control (HPCC) protocols, for high data contention database environments. HPCC attempts to be more aggressive by permitting more serializable schedules than the two-phase locking (2PL). It maintains cycle-free precedence hierarchies for conflicting transactions. Conflicting operations are allowed to proceed only if the hierarchical orderings of precedence is not violated. Transactions commit based on the serialization order established during the executions. A detailed simulation model has been implemented and extensive experiments have been conducted to evaluate the performance of the proposed approach. The results demonstrate that the proposed algorithm outperforms the two-phase locking over a wide range of system workloads.

Keywords: Concurrency Control, Serialization Graph, 2PL, Serializability

1. Introduction

During the past few decades, there has been much research on developing concurrency control mechanisms for concurrent transaction processing in databases. In general, these mechanisms fall into four categories: locking, timestamping, certification (validation), and hybrid protocols. Among them, the two-phase locking (2PL) [7], timestamping [3, 4, 13], and optimistic algorithms [10], which represent three fundamentally different approaches, were most widely studied. Many other algorithms are developed based on these or combinations of these basic algorithms. Bernstein et al. [2] contains detailed discussions on various concurrency control protocols.

Optimistic concurrency controls (OCCs) have attracted a lot of attention in distributed and realtime databases [5, 6, 8, 9, 11, 12] due to its simplicity and dead-lock free nature. However, as the resource and data contention intensifies, the number of restarts increases dramatically, and thus OCCs can perform much worse than 2PL [1]. As for the timestamp ordering methods, they are generally more appropriate for distributed environments with short transactions; they can perform poorly otherwise [14]. 2PL and its variants have emerged as the winner in the competition of concurrency control in the conventional databases [1, 5] and have been implemented in all commercial databases.

Recent advances in wireless communication and cloud computing technology have made accesses to databases much easier and more convenient. Conventional concurrency control protocols face a stern challenge of increased data contents resulted from greater numbers of concurrent transactions. Although two-phase locking (2PL) [7] has been very effective in conventional database management systems, its conservativeness in handling conflicts can deter the transaction progress in high concurrency and high data-contention environments, resulting in unnecessary blocks and aborts.

In this paper, we propose a concurrency control protocol, called the hierarchical precedence concurrency control (HPCC), for high data contention database environments. The idea comes from the observations that some conflicting transactions need not be blocked and can still complete serializable. This observation leads us to a design that permits higher concurrency levels than 2PL. In this research, we design a protocol that is prudently more aggressive than 2PL, permitting some conflicting operations to proceed without blocking. We performed simulations to examine its performance. The simulation results verify that the proposed protocol performs better than OCC and 2PL at high levels of concurrency. This method is also simple and easy to implement.

The rest of this paper is organized as follows. In Section 2, we introduce the hierarchical precedence concurrency control protocol. In Section 3, we report on the performance of our protocol. Conclusions are discussed in Section 4.

2. The Hierarchical Precedence Concurrency Control

To avoid rollback and cascading rollback, hereafter we assume all protocols are strict protocols, that is, all writes
are performed in private workspaces and will not be written to databases until transactions have committed.

2.1 Observations

Our idea comes from the observation that some conflicting operations need not be blocked, as does in 2PL, and they may still complete serially. Therefore, we attempt to be prudently more aggressive than 2PL to see if the rationalized aggressiveness can pay off. In the following, we illustrate the observations by examples.

Example 1. Read-after-Write (RAW). The first few operations of transactions T₁ and T₂ are described as follows:

\[ T₁: R₁(b) W₁(a) \ldots, \quad T₂: R₂(a) W₂(e) \ldots, \]

where \( R_i(x) \) denotes that transaction \( i \) reads item \( x \), and \( W_j(y) \) denotes that transaction \( j \) writes item \( y \). Consider the following schedule:

\[ R₁(b) W₁(a) R₂(a) \ldots \]

There is a read-after-write (RAW) conflict on data item “a” because transaction T₂ reads “a” (i.e., \( R₂(a) \)) after T₁ writes “a” (i.e., \( W₁(a) \)). In 2PL, T₂ will be blocked until T₁ commits or aborts. T₂ may also be killed if it is blocked for too long because it may have involved in a deadlocked situation.

If we are a little more aggressive and allow T₂ to read “a”, T₂ will read the old value of “a”, not the new value of “a” written by T₁ (i.e., \( W₁(a) \)), due to the strict protocol. Consequently, a read-after-write conflict, if not blocked, yields a precedence, that is, T₂ precedes T₁, denoted as T₂ \( \rightarrow \) T₁. We attempt to record the precedence to let the conflicting operations proceed.

Example 2. Write-after-Read (WAR). Consider the same transactions with a different schedule as follows:

\[ R₁(b) R₂(a) W₁(a) \ldots \]

Similarly, \( W₁(a) \) of T₁ needs not to be blocked when it tries to write “a” after T₂ has read “a” (\( R₂(a) \)). If so, the write-after-read (WAR) conflict on item “a” produces a precedence T₂ \( \rightarrow \) T₁ in the strict protocol. Note that T₂ again reads “a” before T₁’s \( W₁(a) \) becomes effective later in the database.

Precedence between two transactions is established when there is a read-after-write or write-after-read conflict. Note that a write-after-write conflict does not impose precedence between the transactions unless that the item is also read by one of the transactions, in which case precedence will be established through the read-after-write or the write-after-read conflicts.

Note that either in a read-after-write or write-after-read conflict, the transaction reads the item always precedes the transaction that writes that item due to the strict protocol.

2.2 Hierarchical Precedence

To allow reads to precede writes (in RAW) and writes to be preceded by reads (in WAR) can yield a complex precedence graph. Detecting cycles in a complex precedence graph to avoid possible non-serializability of transactions can be quite time consuming and defeat the purpose of the potentially added serializability. Here, we design a protocol to constrain the growth of the precedence graph, while allowing the maximal conflicts, so that the graph cannot have cycles and thus guarantees serializability automatically.

Let \( G(V, E) \) be the precedence graph for a set of concurrently running transactions in system, where \( V \) is a set of vertices \( T₁, T₂, \ldots, Tₙ \), denoting the transactions in the system, and \( E \) is a set of directed edges between transactions, denoting the precedence among them. An arc is drawn from \( Tᵢ \) to \( Tⱼ \), \( Tᵢ \rightarrow Tⱼ \), \( 1 \leq i, j \leq n, i \neq j \), if \( Tᵢ \) read an item written by \( Tⱼ \), which has not committed yet, or \( Tᵢ \) wrote an item (in its workspace) that has been read earlier by \( Tⱼ \).

We intend to allow conflict operations to proceed as long as the precedence graph has no cycles. To help manage the transactions, transactions are grouped into precedence groups. Initially, all transactions are in the same group, called the universe precedence group denoted by \( G₀ \). When a transaction is to perform a conflict operation, we will check if it is possible to assign one transaction to a relatively higher precedence group than the other. Restrictions are applied to avoid generating cycles in the precedence graph.

Figure 1 shows a hierarchical precedence graph, in which the bounding rectangle represents the initial universal group \( G₀ \) and the ovals represent the precedence groups \( Gᵢ, i > 1 \), to which conflicting transactions are assigned. In the figure, \( T₁ \) has not performed any conflict operation with others, so it remains in the universal group \( G₀ \), while others that have performed some sorts of conflict operations are assigned to various sub-groups.

![Figure 1. A Hierarchical Precedence Graph](image-url)
T_1, T_2, and T_3 were originally in the same group, denoted by the upper outmost oval G_1. Since T_3, and T_4 later preceded T_5, T_1, and T_4 were assigned to a higher inner group G_1 than T_5 (in G_4). This is the rule (i) of the hierarchical precedence rules, to be discussed shortly. A group can repeatedly subdivide itself to accommodate deeper levels of conflicts.

The higher and more outer the group, the higher the precedence it has. A transaction with a higher precedence is allowed to precede a transaction with a lower precedence, or the latter is allowed to be preceded by the former, such as T_1 -> T_2. This is the rules (ii) and (iii) of the hierarchical precedence rules.

A group G_i is a parent group of another group G_j, i ≠ j, if G_i contains G_j. Similarly, G_i is the ancestor group of G_j if G_i is the parent group of, …, parent group of G_j. For example, G_1 is an ancestor group of G_3 and G_4, but it is not an ancestor group of G_2, G_5, and G_6. Transaction T_9 is said to be an independent transaction in group G_1 as it has no conflict with any other transactions in G_1 (and its descendants), T_1, T_2, and T_3. Similarly, T_2 is an independent transaction in the group G_2. An independent transaction can join or subdivide a descendent group if it executes a conflict operation with transactions in the descendent groups (rule (iv)).

Let us now formally state the precedence rules. T_i and T_j be two transactions that involve in a conflict operation. Regardless the conflict being RAW or WAR, let T_i be the transaction that performs a read on the item and T_j the transaction that performs a write on that item, that is, T_i attempts to precede T_j or T_j attempts to be preceded by T_i. The conflict operation is allowed to proceed only if one of the following rules, called the Hierarchical Precedence Rules, is satisfied.

Hierarchical Precedence Rules:

(i) T_i and T_j are in the same precedence group
(ii) T_j is in a higher precedence group than T_i
(iii) T_i is an outer group of T_j
(iv) T_j is an independent transaction in an ancestor group of T_i

In order not to make the precedence too deeply nested, one can pre-determine the level of nesting.

It can be observed that a transaction is allowed to precede another one only if the former has or can be assigned a higher precedence than the later. Therefore, the precedence graph cannot have a cycle and guarantees the serializability. Due to space limitation, we shall not give a formal proof here. Readers are referred to an extended version of the paper [15] for formal proofs and how to assigned appropriate precedence.

2.3. Hierarchical Precedence Protocol

Transaction are executed in three phases: read, wait-to-commit, and commit phases. In the read phase, transactions proceed following the precedence rule. Once a transaction finishes all its operations, it enters the wait-to-commit phase, waiting for its turn to commit following the precedence established in the read phase. Updates are written to the disk and transactions release resources in the commit phase.

2.3.1. Read Phase

A transaction executing a conflict operation with another transaction will be allowed to proceed if it satisfies the hierarchical precedence rules; otherwise, it will be either blocked or aborted. The transaction that violates the precedence rules is hereafter called a violating transaction.

In the following, we show a situation with a violating transaction.

Example 3. There are three transactions. Their operations and schedule are as follows.

\[ T_1: R_i(b); W_i(a); R_j(e); \ldots; T_2: R_2(a); W_2(e); \ldots; T_3: R_3(e); \ldots; \]

Schedule: \( R_1(b); W_i(a); R_2(a); W_2(e); R_3(e); R_4(e); \)

T_3 is allowed to precede T_1 when T_2 reads “a” by rule (i). Then, T_3 is allowed to precede T_2 when it tries to read “e” (R_3(e)) by rule (iii). But when T_1 tries to read e (R_1(e)), T_1 becomes a violating transaction, denoted by \( R_2(e) \) in the schedule and needs to be blocked or aborted.

The simplest strategy to handle a violating transaction, such as T_1, is to abort it. Unfortunately, aborts may waste the efforts already spent. Therefore, we prefer blocking with the hope that the violation may later resolve and the violating transaction can still complete later. The read phase with the Hierarchical Precedence Rules is summarized in Figure 2.

```
if there is a RAW or WAR conflict
{
    if the hierarchical precedence rule is satisfied,
    proceed with the operation;
    else
        abort or block;
}
```

Figure 2. Read Phase with Hierarchical Precedence Rules

A time quantum must be set up to limit the amount of time a violating transaction can wait (block itself). Once the time quantum expires, the blocked (violating) transaction will be aborted to avoid building a long chain of blocked transactions.

2.3.2 Wait-to-Commit Phase

First, each transaction acquires exclusive locks on those items it has written in the read phase to avoid building further precedence. Any transaction in the read phase wishes to access a locked item shall be blocked. If such a blocked
transaction already preceded a wait-to-commit transaction, it will be aborted immediately in order not to produce a circular wait, that is, wait-to-commit transactions wait for their preceding blocked transactions to complete or vice versa. Otherwise, the blocked transaction remains blocked until the locked item is unlocked.

A transaction can proceed to the commit phase if no transactions, either in the read or the wait-to-commit phase, precede it. Otherwise, it has to wait until all its preceding transactions commit.

2.3.3 Commit Phase

As soon as a transaction enters the commit phase, it flushes updated items to the database, releases the exclusive locks on data items obtained in the wait-to-commit phase, and also releases transactions blocked by it due to violations of the precedence rule. Figure 3 summarizes the wait-to-commit and the commit phases.

```c
/* when a trans. T reaches its wait-to-commit phase */
Wait-to-Commit Phase:
  Lock items written by T;
  T waits until all preceding transactions have committed;

Commit Phase:
  Flush updated items to database;
  Release locks;
```

Figure 3. Wait-to-Commit and Commit Phases

3. Simulation Results

This section reports the performance evaluation of OCC, 2PL and the Hierarchical Precedence Concurrency Control (HPCC) by simulations. We have experimented with HPCC with 2, 3, and 4 levels of precedence, denoted by HPCC2, HPCC3, and HPCC4, respectively.

3.1 Simulation Model

We have implemented, OCC, 2PL and HPCC in a simulation model that is similar to [1]. Each transaction has a randomized sequence of read and write operations, with each of them separated by a random period of a CPU burst of 15±5 time units on average. The randomized disk access time is 35±10. All writes are performed on items that have already been read in the same transactions. All writes are stored in private work space and will only be written to the database after commits following the strict protocol.

3.2 Parameter Settings

The write operations cause conflicts and thus data contentions. Therefore, we shall experiment with different write probabilities, 20% (moderate) and 50% (the highest), to observe how these algorithms adapt to the conflicts. Other factors that affect the data contentions are database sizes and transaction sizes. Therefore, two database sizes of 100 and 500 items, and two transaction sizes of averaged 8 and 16 operations will be used in the simulation. Table 1 summarizes the parameter settings for the simulation.

Transactions may be blocked in 2PL and HPCC. Blocked transactions are aborted if they have been blocked longer than specified periods. We have experimented with several block periods and select the best ones to use in the simulations.

The primary performance metric is the system throughput, which is the number of transactions committed during the period of the simulation. This is an overall performance metric.

Table 1: Parameter Settings

<table>
<thead>
<tr>
<th>Database size</th>
<th>100, 500 items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. transaction size</td>
<td>8±4, 16±4 operations</td>
</tr>
<tr>
<td>Write probability</td>
<td>20%, 50%</td>
</tr>
<tr>
<td>Num. of CPUs/Disks</td>
<td>4/8, 16/32</td>
</tr>
<tr>
<td>CPU burst</td>
<td>15±5 time units</td>
</tr>
<tr>
<td>I/O access time</td>
<td>35±10 time units</td>
</tr>
</tbody>
</table>

3.3 Experimental Results

3.3.1 Data Contention

Data contention is caused by the write operations. The higher the write probabilities, the higher the data contentions are. Similarly, the higher the concurrency levels and the smaller the database sizes, the greater the data contention. Here, we will see how these factors affect the performance of the two protocols.

We experimented with two database sizes, 100 items and 500 items, and two transaction sizes, averaged 8 and 16 operations in each transaction. The simulation time for each experiment is 100,000 time units.

- Write probability 0.2

  Given the write probability 0.2, each transaction has on average one write operation for every four reads.

  Figures 4 shows the performance for transactions with average 8 (8±4) operations for two databases of sizes 500 (Figure 4(a)) and 100 (Figure 4(b)). As observed, as the level of concurrency increased initially, the throughput increased. At low concurrency levels, all protocols had similar throughputs because there were few conflicts. But as the concurrency level increased further, conflicts or data contention intensified and the increase in throughput slowed down. After a particular point, each protocol reached its peak performance and started to drop, known as thrashing.

  Among the various levels of precedence for HPCC, HPCC4 is slightly better than HPCC2 and HPCC3.
Therefore, we shall use the figures of HPCC4 only in the following discussions. Since OCC in general performed the worst, we shall not discuss its performance specifically.

For database size 500 (Fig. 4(a)), the highest numbers of transactions completed in the 100,000 time unit period were 2,315 for HPCC4 and 2,232 for 2PL, that is a 3.72% improvement for the proposed HPCC4.

In Figure 4(b), the database size was reduced to 100 items. The highest numbers of completed transactions were 1,638 and 1,474 for HPCC4 and 2PL, respectively, i.e., a 11.13% higher throughput than 2PL. This indicates that HPCC is more effective in high data contention environments than in low data contention environments, which is exactly the purpose that we design the HPCC for.

For database size 500 (Fig. 5(a)), the highest throughput obtained by HPCC4 was 906, while 2PL peaked at 814. HPCC4 had a 11.30% higher throughput than 2PL. As for database size 100 (Fig. 5(b)), the highest throughputs obtained were 406 and 331 for HPCC4 and 2PL, respectively. HPCC4 had a 22.66% higher throughput than 2PL.

Notice that the performance improvement over 2PL increased from 3.72% in Fig. 4(a) to 11.30% in Fig. 5(a) for database size 500, and from 11.13% in Fig. 4(b) to 22.66% in Fig. 5(b) for database sizes 100. These improvements were due to the increases in transaction sizes from 8 to 16, which increased the data contention. As expected, as the data contention intensifies, HPCC has greater improvement over 2PL in performance.

Now, we increase the average number of operations in each transaction to 16 while maintaining the same write probability 0.2. Figure 5 shows the results.

For database size 500 (Fig. 5(a)), the highest throughput obtained by HPCC4 was 906, while 2PL peaked at 814. HPCC4 had a 11.30% higher throughput than 2PL. As for database size 100 (Fig. 5(b)), the highest throughputs obtained were 406 and 331 for HPCC4 and 2PL, respectively. HPCC4 had a 22.66% higher throughput than 2PL.

Notice that the performance improvement over 2PL increased from 3.72% in Fig. 4(a) to 11.30% in Fig. 5(a) for database size 500, and from 11.13% in Fig. 4(b) to 22.66% in Fig. 5(b) for database sizes 100. These improvements were due to the increases in transaction sizes from 8 to 16, which increased the data contention. As expected, as the data contention intensifies, HPCC has greater improvement over 2PL in performance.

With the write probability 0.5, every item read in a transaction is later written too in that transaction. Figure 6 shows the throughput of the two protocols with the average number of operations set to 8 per transaction.

The highest numbers of transactions completed during the simulation period (Fig. 6(a)) were 2,312 for HPCC4 and
2,310 for 2PL for database size 500, a slight (0.09%) improvement over 2PL like Fig. 4(a). As the database size decreased to 100 (Fig. 6(b)), the highest numbers of completed transactions were 1,607 and 1506 for HPCC4 and 2PL, respectively, that is, a 6.71% higher throughput than 2PL, due to higher data contentions.

Figure 7 shows the throughput of the two protocols with the number of operations per transaction increased to 16.

The highest numbers of transactions completed during the simulation period (Fig. 7(a)) were 819 for HPCC4 and 793 for 2PL for database size 500, a 3.28% (vs. 0.09% (Fig 6(a)) improvement over 2PL. As the database size decreased to 100 (Fig. 7(b)), the highest numbers of completed transactions were 365 and 328 for HPCC4 and 2PL, respectively, that is, a 11.28% higher throughput than 2PL.

4. Conclusions

The proposed protocol can resolve many conflicts. It performed better than OCC and 2PL in all situations. It has the best performance when conflicts are not extremely severe, for example, in situations where transactions are not very long and write probabilities are not extremely high. Further research is still needed for resolving more complex conflicts while keeping the protocols simple.

References


