An Aggressive Concurrency Control Protocol for Main Memory Databases

Mohammed Hamdi
Department of Computer Science
Southern Illinois University
Carbondale, IL, USA

Weidong Xiong
Department of Computer Science
Southern Illinois University
Carbondale, IL, USA

Feng Yu
Department of Computer Science and Information Systems
Youngstown State University
Youngstown, OH, USA

Sarah Alswedani
Department of Computer Science
Southern Illinois University
Carbondale, IL, USA

Wen-Chi Hou
Department of Computer Science
Southern Illinois University
Carbondale, IL, USA

ABSTRACT
In this paper, we propose a concurrency control protocol, called the Prudent-Precedence Concurrency Control (PPCC) protocol, for high data contention main memory databases. PPCC is prudently more aggressive in permitting more serializable schedules than two-phase locking. It maintains a restricted precedence among conflicting transactions and commits the transactions according to the serialization order established in the executions. A detailed simulation model has been constructed 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 in all ranges of system workload.

Keywords
Concurrency Control, Main Memory Database, Serializability, Serialization Graph, 2PL

1. INTRODUCTION
Most database management systems are designed assuming that data would reside on a hard disk. As hardware technology advances, computers nowadays can easily accommodate tens or even hundreds of gigabytes of memory with which to perform operations. A significant amount of data can be held in main memory without severe constraints on the memory size. Thus, the use of main memory databases is becoming an increasingly more realistic option for many applications. In light of this, database researchers can exploit main memory database features to improve real-time concurrency control protocols [15].

During the past few decades, there has been much research on currency control mechanisms in databases. The two-phase locking (2PL) [7], timestamping [3, 4, 13], and optimistic algorithms [10] represent three fundamentally different approaches and have been most widely studied. Many other algorithms are developed based on these or combinations of these basic algorithms. Bernstein et al. [2] contains comprehensive discussions on various concurrency control protocols.

Optimistic concurrency controls (OCCs) have attracted a lot of attention in distributed and real time databases [8, 9, 11, 12, 5, 6] due to its simplicity and dead-lock free nature. Transactions are allowed to proceed without hindrance until at the end - the verification phase. However, as the resource and data contention intensifies, the number of restarts can increase dramatically, and OCCs may perform much worse than 2PL [1]. As for the timestamp ordering methods, they are generally more appropriate for distributed environments with short transactions, but 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 contentions, resulted from greater numbers of concurrent transactions. Although two-phase locking (2PL) [7] has been very effective in conventional applications, its conservativeness in handling conflicts can result in unnecessary blocks and aborts, and deter the transactions in high data-contention environment.

Recently, there has been some research on concurrency control in main memory databases [16, 17, 18, 19, 20]. These works mainly focused on reducing the overheads in implementing concurrency control mechanisms, such as 2PL. In this paper, we propose a concurrency control protocol, called prudent-precedence concurrency control (PPCC), for high data contention main memory databases. The idea comes from the observations that some conflicting transactions need not be blocked and may still be able to complete serializably. This observation leads to a design that permits higher concurrency levels than the 2PL. In this research, we design a protocol that is prudently more aggressive than 2PL, permitting some conflicting operations to proceed without blocking. We prove the correctness of the proposed protocol and perform simulations to examine its performance. The simulation results verify that the new protocol performs better than the 2PL and OCC at high data contention main memory databases. This method is also simple and easy to implement.

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

2. THE PRUDENT-PRECEDENCE CONCURRENCY CONTROL

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

2.1 Observations

Our idea comes from the observation that some conflicting operations need not be blocked and they may still be able to complete serializably. 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) ..., T2: R2(a) W2(e) ...,

whereRi(x) denotes that transaction i reads item x, and Wj(y) denotes that transaction j writes item y. Consider the following schedule:

R1(b) W1(a) R2(a) ...

There is a read-after-write (RAW) conflict on data item “a” because transaction T2 tries to read “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 can also be killed if it is blocked for too long, as 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 -> 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) ...

Similarly, W1(a) can be allowed to proceed 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 -> 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 Prudent Precedence

To allow reads to precede writes (in RAW) and writes to be preceded by reads (in WAR) without any control can yield a complex precedence graph. Detecting cycles in a complex precedence graph to avoid possible non-serializability can be quite time consuming and defeat the purpose of the potentially added serializability. Here, we present a rule, called the Prudent Precedence Rule, to simplify the graph so that the resulting graph has no cycles and thus automatically guarantees serializability.

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, …, 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 ->Tj , $1 \le i, j \le n, i \ne j$,if Ti read an item written by Tj, which has not committed yet, or Tj wrote an item (in its workspace) that has been read earlier by Ti.

Transactions in the system can be classified into 3 classes. A transaction that has not executed any conflicting operations is called an independent transaction. Once a transaction has executed its first conflicting operation, it becomes a preceding or preceded transaction, depending upon whether it precedes or is preceded by another transaction. To prevent the precedence graph from growing rampantly, once a transaction has become a preceding (or preceded) transaction, it shall remain a preceding (or a preceded) transaction for its entire life time.

Let Ti and Tj be two transactions that involve in a conflict operation. Regardless the conflict being RAW or WAR, let Ti be the transaction that performs a read on the item, while Tj the transaction that performs a write on that item. The conflict operation is allowed to proceed only if the following rule, called the Prudent Precedence Rule, is satisfied.

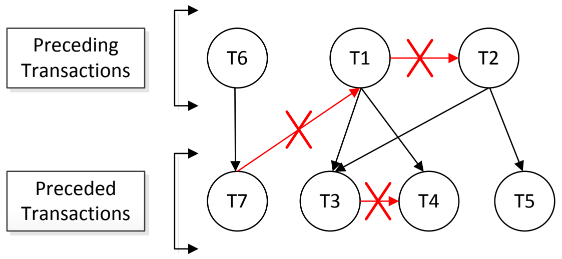

Fig 1: The Precedence Graph

2.2.1 Prudent Precedence Rule:

Ti is allowed to precede Tj or Tj is allowed to be preceded by Ti if

(i) T_i has not been preceded by any transaction and

(ii) T_j has not preceded any other transaction.

We shall use Figure 1 to explain the properties of the resulting precedence graph for transactions following the Prudent Precedence Rule. It can be observed that the first condition of the rule (denoted by (i) in the rule) states that a preceded transaction cannot precede any transaction, as illustrated by the red arcs, marked with x, T7 to T1 and T3 to T4, in the figure, while the second condition (denoted (ii)) states that a preceding transaction cannot be preceded, as illustrated by the red arcs, marked with x, T1 to T2 and T7 to T1, in the figure. Since there cannot be any arcs between nodes in the same class and there is no arc from the preceded class to the preceding class, the graph cannot have a cycle.

2.3. Prudent Precedence Protocol

Each transaction is 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. Transactions release resources in the commit phase. In the following, we describe in details each phase.

2.3.1. Read Phase

A transaction executing a conflict operation with another transaction will be allowed to proceed if it satisfies the prudent 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_1(b) W_1(a) ...

T_2: R_2(a) W_2(e) ...

T_3: R_3(e) …

Schedule: R_1 (b) W_1(a) R_2(a) W_2(e) ~~R_3(e)~~

T2 -> T1 is established when T2 reads "a", and T2 becomes a preceding transaction. Later when T3 tries to read "e" (R3(e)), the operation is suspended (denoted by R3(e) in the schedule) because T2, a preceding transaction, cannot be preceded. Thus, T3 becomes a violating transaction and needs to be blocked or aborted.

The simplest strategy to handle a violating transaction, such as T3, 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 T3 can still complete later. For example, T3 is blocked, i.e., R3(e) is postponed; if T2 eventually commits, then T3 can resume and read the new value of "e" produced by T2. The read/write with the Prudent Precedence Rule is summarized in Figure 2.

```
/* T_i is accessing an item x
if x is locked
 {      if x is locked by a transaction preceded by T_i
abortT_i;
else
          blockT_i (until x is unlocked);
  }
read/write with the Prudent Precedence Rule (Figure 2);
```

Fig 2: Read/Write with Prudent Precedence Rule

Let us elaborate on the blocking of a violating transaction a bit. By allowing a violating transaction to block, a transaction can now either be in an active (or running) state or a blocked state. Although blocking can increase the survival rate of a violating transaction, it can also hold data items accessed by the violating transaction unused for extended periods. Therefore, a time quantum must be set up to limit the amount of time a violating transaction can wait (block itself), just like the 2PL. Once the time quantum expires, the blocked (violating) transaction will be aborted to avoid building a long chain of blocked transactions.

Theorem 1. The precedence graph generated by transactions following the Prudent Precedence Rule is acyclic.

Proof. As explained in Section 2.2, there cannot be a cycle in the precedence graph following the Prudent Precedence Rule. As for violating transactions, they will either abort by timeouts or resume executions if the violation disappear due to the aborts or commits of the other transactions with which the transactions conflict. In either case, it does not generate any arcs that violate the Prudent Precedence Rule, and the graph remains acyclic.

2.3.2 Wait-to-Commit Phase

Once a transaction finishes its read phase, it enters the wait-to-commit phase, waiting for its turn to commit because transactions may finish the read phase out of the precedence order established.

First, each transaction acquires exclusive locks on those items it has written in the read phase to avoid building further dependencies. 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 shall 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. Figure 3 shows the locking when a transaction accesses a locked item.

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.

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

Fig 3: Accessing Locked Items

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 4 summarizes the wait-to-commit and the commit phases.

```
/* when a trans. T_i reaches its wait-to-commit phase */
Wait-to-Commit Phase:
  Lock items written by T_i;
T_i waits until all preceding transactions have committed;
Commit Phase:
  Flush updated items to database;
 Release locks;
 Release transactions blocked by T_i;
```

Fig 4: Wait-to-Commit and Commit Phases

Table 1: Parameter Settings

Database size	100, 500 items
Average transaction size	8± 4, 16±4 operations
Write probability	20%, 50%
Num. of CPUs	4, 16
CPU burst	15±5 time units

Example 4. Suppose that we have the following transactions, T1, T2:

T_1: $R_1(a)$, $R_1(b)$

T_2: $R_2(b)$, $W_2(a)$, $W_2(b)$

Assume that the following is the schedule:

R1 (a), R2(b), W2(a), W2(b), [wc2], R1 (b) abort1, wc2,c2

When T2 writes “a” (W2(a)), T1 ->T2 is established, due to an earlier R1(a). So, when T2 reaches its wait-to-commit phase, denoted by wc2, it locks both “a” and “b”. However, T2 has to wait until T1 has committed, denoted by [wc2], due to the established precedence T1 ->T2. Later, when T1 tries to read “b”, it is aborted, as indicated by R1(b) and abort1, because “b” is locked by T2, as stipulated in Figure 3. Now no transaction is ahead of T2, so it can finish its wait phase (wc2) and commits (c2).

2.4 Serializability

A history is a partial order of the operations that represents the execution of a set of transactions [5]. Let H denote a history. The serialization graph for H, denoted by SGH, is a directed graph whose nodes are committed transactions in H and whose edges are Ti→Tj ($i \neq j$) if there exists a Ti’s operation precedes and conflicts with a Tj’s operation in H. To prove that a history H is serializable, we only have to prove that SGH is acyclic.

Theorem 2. Every history generated by the Prudent Precedence Protocol is serializable.

Proof. The precedence graph is acyclic as proved in Theorem 1. The wait-to-commit phase enforces the order established in the precedence graph to commit. So, the serialization graph has no cycle and is serializable.

3. SIMULATION RESULTS

This section reports the performance evaluation of 2PL, OCC, and the Prudent Precedence Concurrency Control (PPCC) by simulations.

3.1 Simulation Model

We have implemented 2PL, OCC and PPCC 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. 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

Our goal is observe the performance of the algorithms under data contentions. The write operations cause conflicts and thus the data contentions. Therefore, we shall experiment with different write probabilities, 20% (moderate), and 50% (the highest), to observe how the two algorithms adapt to 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 as shown in Table 1.

Transactions may be blocked in 2PL, OCC and PPCC to avoid generating cycles in the precedence graphs. 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.

3.3 Experimental Results

In this section, we report the simulation results on the two protocols based on the above setups.

3.3.1 Data Contention

As mentioned earlier, the data contention is mainly caused by the write operations. If transactions have no writes, there will be no conflicts and all three protocols will have identical performance.

Given the same write probability, the greater the transaction sizes, the greater the numbers of write operations are in the system, and thus the higher the data contentions are. On the other hand, given the same number of write operations, the smaller the database size, the greater the chance of conflicts, and thus the higher the data contentions are. Here, we will see how these factors affect the performance of the three protocols.

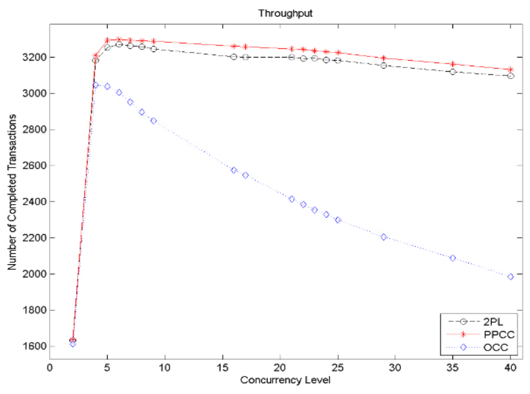

Fig 5(a): DB size 500

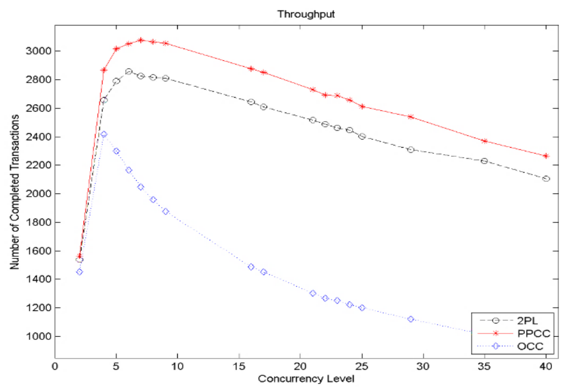

Fig 5(b): DB size 100

Fig 5: Write probability 0.2, Transaction Size 8

We experimented with two database sizes, 100 items and 500 items, and two transaction sizes, averaged 8 and 16 operations in each transaction. The experimental results in this subsection were obtained with the setup of 4 CPUs. The simulation time for each experiment is 100,000 time units.

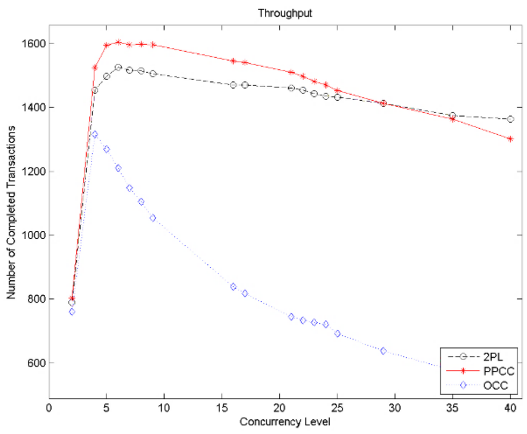

Fig 6(a): DB size 500

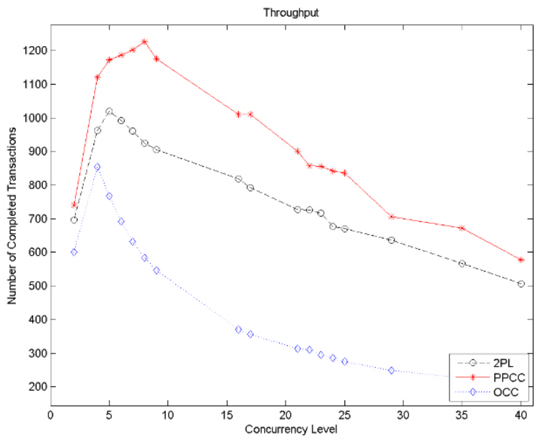

Fig 6(b): DB size 100

Fig 6: Write probability 0.2, Transaction Size 16

- **Write probability 0.2**

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

Figures 5 shows the performance for transactions with averaged 8 (8 ± 4) operations for two databases of sizes 500 (Figure 5(a)) and 100 (Figure 5(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 a bit. After a particular point, each protocol reached its peak performance and started to drop, known as thrashing.

For database size 500 (Figure 5(a)), the highest numbers of transactions completed in the given 100,000 time unit period were 3,299 for PPCC, 3,271 for 2PL, and 3,046 for OCC, that is, a 0.86% and 8.31% improvements over 2PL and OCC, respectively.

In Figure 5(b), the database size was reduced to 100 items to observe the performance of these protocols in a high data contention environment. The highest numbers of completed transactions were 3,078, 2,857, and 2,417 for PPCC, 2PL, and OCC, respectively, i.e., an 7.74% and 27.35% higher throughputs than 2PL and OCC. This indicates that PPCC is more effective in high data contention environments than in low data contention environments, which is exactly the purpose that we design the PPCC for.

Now, we increase the average number of operations in each transaction to 16 while maintaining the same write probability 0.2. Figure 6 shows the results.

For database size 500 (Figure 6(a)), the highest throughput obtained by PPCC was 1,605, while 2PL peaked at 1,527 and OCC at 1,316. PPCC had a 5.11% and 21.96% higher throughputs than 2PL and OCC. As for database size 100 (Figure 6(b)), the highest throughputs obtained were 1,226, 1,019, and 854 for PPCC, 2PL and OCC, respectively. PPCC had a 20.31% and 43.56% higher throughputs than 2PL and OCC.

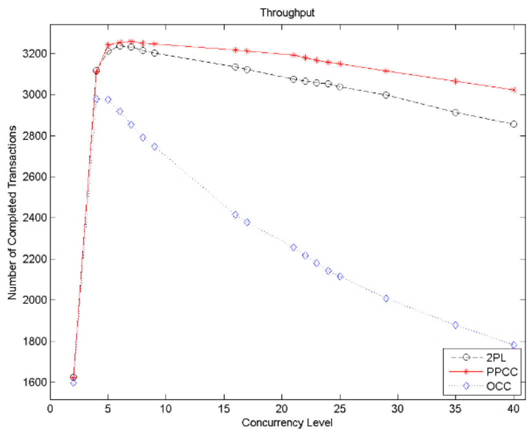

Fig 7(a): DB size 500

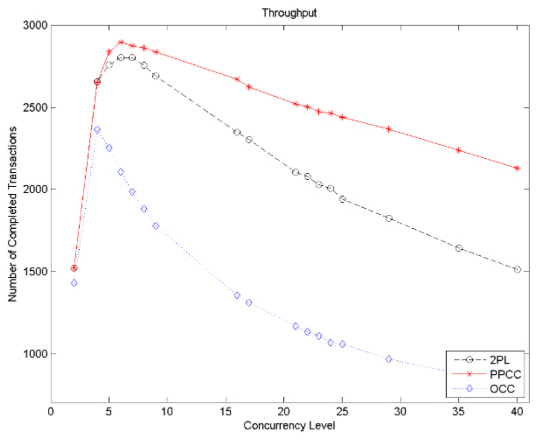

Fig 7(b): DB size 100

Fig 7: Write probability 0.5, Transaction Size 8

In general, as the data contention intensifies, PPCC has greater improvements over 2PL and OCC in performance.

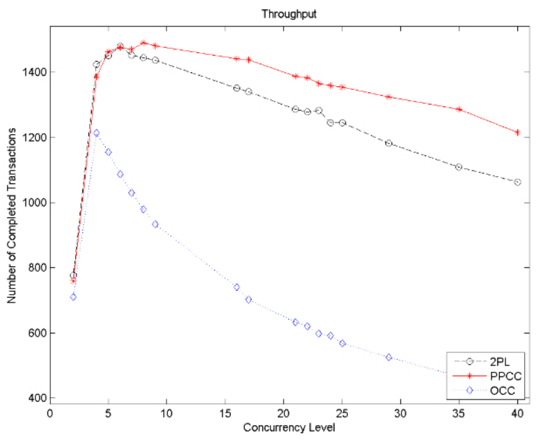

Fig 8(a): DB size 500

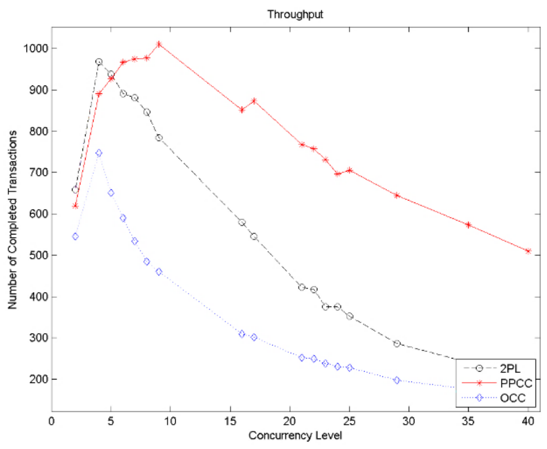

Fig 8(b): DB size 100

Fig 8: Write probability 0.5, Transaction Size 16

- **Write probability 0.5**

With the write probability 0.5, every item read in a transaction is later written too in that transaction. Figure 7 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 (Figure 7(a)) were 3,258 for PPCC, 3,237 for 2PL, and 2,978 for OCC for database size 500, a slight improvement over 2PL(0.65%), but a much larger improvement over OCC (9.40%). As the database size decreased to 100 (Figure 7(b)), the highest numbers of completed transactions were 2,898, 2,803, and 2,365 for PPCC, 2PL, and OCC, respectively, that is, a 3.39% and 22.54% higher throughput than 2PL and OCC, due to the higher data contentions.

Figure 8 shows the throughput of the three protocols with the number of operations per transaction increased to 16.

The highest numbers of transactions completed during the simulation period (Figure 8(a)) were 1,490 for PPCC, 1,480 for 2PL, and 1,213 for OCC for database size 500, a 0.68% and 22.84% improvements over 2PL and OCC. As the database size decreased to 100 (Figure 8(b)), the highest numbers of completed transactions were 1,011, 969, 747 for PPCC, 2PL, and OCC, respectively, that is, a 4.33% and 35.34% higher throughputs than 2PL and OCC.

In very high data contention environments, few transactions can succeed, as illustrated in Figure 8(b). This indicates that there is still room for improvement in designing a more aggressive protocol that allows more concurrent schedule to complete serializably.

4. CONCLUSIONS

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

5. REFERENCES

[1] R. Agrawal, M. J. Carey, and M. Livny, “Concurrency Control Performance Modeling: Alternatives and Implications,” ACM Transactions on Database Systems, 12(4), pp. 609-654, 1987.

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

[3] P. Bernstein, N. Goodman, “Timestamp-based Algorithm for Concurrency Control in Distributed Database Systems”, Proc. VLDB 1980, pp. 285 – 300.

[4] P. Bernstein, N. Goodman, J. Rothnie, Jr., C. Papadimitriou, “Analysis of Serializability in SDD-1: a System of Distributed Databases”, IEEE Transaction on Software Engineering SE-4:3, 1978, pp. 154 -168.

[5] M. Carey, M. Livny, “Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication”, Proc. 14th VLDB Conference, pp. 13-25, 1988.

[6] S. Ceri, S. Owicki, “On The Use Of Optimistic Methods for Concurrency Control in Distributed Databases”, Proc. 6th Berkeley Workshop, pp. 117-130, 1982.

[7] P. Eswaran , J. N. Gray , R. A. Lorie , I. L. Traiger, (1976) The Notions of Consistency and Predicate Locks in a Database System, Communications of the ACM, Vol. 19, No. 11, p.624-633.

[8] T. Haerder, (1984) Observations on Optimistic Concurrency Control Schemes. Information Syst., 9, 111-120.

[9] J. R. Haritsa, , M. J. Carey, and M. Livny, (1990) Dynamic Real-Time Optimistic Concurrency Control. In Proc. 11th Real-Time Systems Symp., Lake Buena Vista, FL,5-7 December, pp. 94-103.

[10] H. Kung, and J. Robinson, (1981) On Optimistic Methods for Concurrency Control. ACM Trans. Database Syst., 6, 213-226.

[11] K.W. Lam, K.Y. Lam and S.L. Hung. Distributed Real-time Optimistic Concurrency Control Protocol. In Proceedings of International Workshop on Parallel and Distributed Real-time Systems, Hawaii, pp. 122-125, IEEE Computer Society Press (1996).

[12] S. Mullender and A. S. Tanenbaum. ”A Distributed File Service Based on Optimistic Concurrency Control,” Proc. 10th ACM Symp. On Operating System Principles, 1985, pp. 51-62.

[13] D. Reed, “Implementing Atomic Actions on Decentralized Data”, ACM Transactions on Computer Systems, 1,1, pp. 3-23, 1983.

[14] I. Ryu, A. Thomasian, “Performance Evaluation of Centralized Databases with Optimistic Concurrency Control”, Performance Eval. U, 3, 195, 211, 1987.

[15] Özgür, U and Alejandro, B. "Exploiting main memory DBMS features to improve real-time concurrency control protocols." ACM SIGMOD Record 25.1 (1996): 23-25.

[16] Ren K, Thomson A, Abadi DJ. "Lightweight locking for main memory database systems". In Proceedings of the VLDB Endowment 2012; 6(2): 145-156. VLDB Endowment.https:/doi.org/10.14778/2535568.2448947

[17] Larson PÅ, Blanas S, Diaconu C, Freedman C, Patel JM, Zwilling M. "High-performance concurrency control mechanisms for main-memory databases". Proceedings of the VLDB Endowment 2011; 5(4), 298-309.https:/doi.org/10.14778/2095686.2095689

[18] Neumann T, Mühlbauer T, Kemper A. Fast serializable multi-version concurrency control for main-memory database systems. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data 2015; 677-689. ACM.https:/doi.org/10.1145/2723372.2749436

[19] Yu X, Bezerra G, Pavlo A, Devadas S, Stonebraker M. Staring into the abyss: An evaluation of concurrency control with one thousand cores. Proceedings of the VLDB Endowment 2014; 8(3): 209-220.https:/doi.org/10.14778/2735508.2735511

[20] Jones EP, Abadi DJ, and Madden S. Low overhead concurrency control for partitioned main memory databases. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data 2010; 603-614. ACM.https:/doi.org/10.1145/1807167.1807233