Concurrency Control and Recovery of Long Lived Transaction Processing in Virtualized Environment

Nazifa Noor, Motoyasu Nagata
Department of computer science, Osaka Kyoiku University, Osaka, Japan

Abstract - This paper explores long lived transaction processing in the virtualized environment. The employment of the virtualization is motivated by the information sharing among database servers, which enhances concurrency of long lived transaction processing system. Our long lived transaction processing system is characterized by the combination of concurrency control and recovery. Our protocol exploits priority ceiling instead of traditional locking method, while employing and extending the concept of ‘wait’ inherent to altruistic locking approach. The recovery is efficient due to information sharing log, active list, commit list and abort list. Also, we have evaluated the performance of our approach and shown its effectiveness.

Keywords: Long Lived Transaction Processing, Priority Ceiling, Two-phase-locking, Virtual Environment, Concurrency Control, Recovery techniques.

1 Introduction

As the database technologies are adapted to a wide range of applications, long lived transaction processing models are inevitably required to support various semantics associated with the applications for throughput enhancement. The ability to maintain concurrency control, and recover from erroneous execution or failure is essential requirement for long lived transaction processing system.

Long Lived Transactions (LLT), have long duration compared to the other transactions, therefore these transactions cause significant delays for Short transactions in the database systems, this compromises concurrency of the system.

In this paper, our goal is developing protocols to motivate concurrency of those database systems that contain long lived transactions. Our approach is exploiting priority ceiling to the long lived transactions in a virtual environment. Furthermore enhancement in the properties of altruistic locking is proposed. We claim that our proposed method brings more concurrency to transaction processing system. The concurrency control is the activity of coordinating concurrent accesses to database in multi-user database system. Concurrency in virtual environment permits users to access a database in a multi-programmed fashion with the illusion that access to databases of various location and servers is possible from one and nearest server. The concurrency control has been actively investigated several times for the past decade. One standard solution accepted to cope with the problem is Two-Phase-Locking method, which is applicable to any type of transaction processing system.

Considering the fact that long lived transactions (LLTs) conduct lengthy computation over database objects and their process interval is longer than others, applying two-phase-locking sounds too rigid.

By two-phase-locking method transactions encountering lock conflict are blocked for long period of time, which slow down the processing system. Basic Altruistic locking is a pioneering attempt to cope with concurrency control of LLTs; this concept sets the term of donating database objects (release before unlock). Database objects which are no longer needed by the transaction are released during the process interval of that transaction before it’s terminated. Using altruistic locking, short lived transactions (SLT) can go into the state called wake of LLTs which means they can access the database objects donated by LLT.

2 Proposed Approach

Database servers from distinct and remote locations are virtually centralized in a way that gives user the illusion as they are located together in one and closest server. In virtual environments transaction manager TM, scheduler and database manager DM are centralize and share. This makes long lived transactions processing smoother than the distributed system with many TM, DM and servers.

2.1 Concurrency in virtual environments

Virtual environmet is clearly structured by figure1.

By the time that the LLT (long lived transaction) reaches the terminal of TM (Transaction Manager), its structure is
revealed to the TM, the requested data within a list is pitched to the transaction scheduler. Transaction scheduler checks for the availability of the data, while negotiating with Database manager if the data is available, the DM (Database manager) creates a temporary cache, this temporary cache contains all the requested data, at this time scheduler acknowledges TM the acceptance of the transaction, thus the transaction can accesses the required data within the temporary cache.

By the time that the transaction is terminated (commit or abort) the temporary cache is also terminated. And the updated data is flushed to the stable storage.

This way if a transaction comes in the wake of LLT, it will access the temporary cache created for the LLT.

Transaction Scheduler also has a pending queue which contains all the conflicting transactions.

2.2 Enhancement of Altruistic Locking (EAL)

We want to further enhance the properties of altruistic locking:

2.2.1 Properties of Basic Altruistic locking (BAL)

1. Two transactions can’t hold simultaneous locks on same database object, unless one of them has locked and released the object before the other locked it (second lock holder is in the wake of the releasing transaction).
2. If a transaction (SLT) is in the wake of other transaction(LLT) it should be completely in the wake of that transaction. It means that SLT can’t access any data object outside of the wake for that transaction.

2.2.2 EAL (Enhanced Altruistic Locking) properties

In order to represent our terms let’s take a look at the following example.

Consider two virtually shared databases: z[A,B,C,D,E] and y[F,G,H,I], LLT with two SLT.

Long Lived Transaction list, \( LLT \) locks \( \{Read(A), Write(B), Reading(C), Read(E)\} \)

(The underlined \( C \) means LLT is now operating on \( C \), and has successfully finished \( A \) and \( B \), but has not yet reached \( E \)).

First Short Lived Transaction list, \( SLT1 \) try to lock \( \{Read(E), Write(F), Read(A)\} \)

Second Short Lived Transaction list, \( SLT2 \) try to lock\( \{Write(A), Write(B)\} \).

See also figure (3) bellow.
If we look at SLT1’s request list, its approach is inside and outside process interval of LLT, according to altruistic locking rule SLT’s lock request can’t be granted.

However SLT2 can be granted, since SLT2’s approach is only inside the wake of LLT.

According to our conducted experiment to invoke enhanced altruistic locking (EAL), let’s consider that the system grants lock request by SLT1.

The result is,

Because SLT1 is a commute transaction with LLT

\[ SLT(reqData) \cap LLT(reqData) = Commute \]

It means the action committed by one of these transactions will not interrupt the other.

This proves that: commute Short-Lived-Transaction scan enter inside the wake of long-lived transactions, but they don’t need to be restricted during LLT’s process interval (wake).

Based on the above assumption we can enhance the Basic Altruistic Locking (BAL) as following:

Enhanced Altruistic Locking’s (EAL) properties for long lived transactions in virtual environments:

1. Acquire a floppy lock before every read of database object by transaction.
2. Acquire a solid lock before every write of database object by transaction.

The floppy-lock is a kind of lock by which, the second transaction can enter the wake of the first transaction without following the BAL rule (restricted during LLT’s process interval), for example, commute transactions.

The solid-lock is the one by which the second transaction has to be completely under wake of the first transaction and its termination state (commit/abort) depends on first transaction, for example, conflict transactions.

EAL properties for short transactions in virtual environment

1. Acquire access to the release of floppy-locked data objects without entering the wake of LLT.
2. Acquire access of SLT to the release of solid-locked-data objects only by entering the wake of LLT.

2.3 Protocol

The priority between transactions can be implemented further as below:

Pursuing our method EAL combined with Priority ceiling between simultaneous transactions in virtual Environments, The overall protocol is listed below in a set of 3 steps:

Step 1. The structure of all transactions becomes known to the TM, by the time the transaction arrives at the TM terminal.
Step 2. TM assigns priority based on the arrival time of T.
Step 3. Scheduler initiates and periodically updates the array-Table of current simultaneous transactions prior to their access to manage the priority of transactions. This array-Table contains notation as \( T[PC(i,j)=k] \), where T is the specific transaction, (i) shows priority of transaction, (j) is the innings of holding the lock on a single database object and (k) is the number of times T wants to access one specific object (as in table a LLT accesses data object (b) two times so k(b)=2).

Table A. shows the transaction priority and innings.

<table>
<thead>
<tr>
<th>Transactions</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLT[PC(i,j)=k]</td>
<td>PC(1.1)=0</td>
<td>PC(1.2)=0</td>
<td>PC(1.3)=1</td>
<td>PC(1.4)=0</td>
<td>PC(1.5)=0</td>
</tr>
<tr>
<td>SLT1[PC(i,j)=k]</td>
<td>PC(2.1)=0</td>
<td>PC(2.2)=0</td>
<td>PC(2.3)=0</td>
<td>PC(2.4)=0</td>
<td>PC(2.5)=0</td>
</tr>
<tr>
<td>SLT2[PC(i,j)=k]</td>
<td>PC(3.1)=0</td>
<td>PC(3.2)=0</td>
<td>PC(3.3)=0</td>
<td>PC(3.4)=0</td>
<td>PC(3.5)=0</td>
</tr>
</tbody>
</table>

Furthermore, the acceptance of transaction’s attempt of accessing objects based on the following priority ceiling terms:

1. No higher priority (T) exists for the same object.
2. Access times (k) of transaction over data is zero PCT (i,j)=0. Commit in this case belongs to the easy calculation of T1(i,j) and T2 (i,j).

For example, PCT1(1,1)>PCT2(2,2) therefore T1 which is LLT gets to commit first.

However, in case of PCT2 (2,2)=1 and PCT3 (3,1)=1 don’t care since their timing and Required list is different.

The chart below shows the overall structure of our methodology.
3 Recovery

Recovery rule is obtained by rollback procedure. Our proposed system contains periodical checkpoints, each checkpoint is in charge to backup the state of data (under manipulation by transaction) and over all states of the system from the last the checkpoint. The logging is essential part of this task. If any type of failure occurs in the system, recovery manager rolls back the system to the previous checkpoint in order to omit the aftermaths of failure and to recover the consistent state of the system.

Each checkpoint must maintain three important lists:
1. active list which contains live operating transactions and live manipulated data object.
2. commit list, list of the transactions that were committed by the time they reached the specific checkpoint.
3. Abort list, is a list of transactions which were aborted by the time they reached particular checkpoint.

1. Commit
When the transaction processing is completed at the final server, contents of the cache slot are flushed to the corresponding database at each server.

2. Abort
The abort is caused by the following two cases.
Case 1, the fault of the transaction itself occurs at the executing server, despite on the way or final servers.

Case 2, the committed transaction read the object x which transaction T1 wrote, at this time transaction T1 is executed normally. However, later, transaction T1 becomes a situation where this transaction T1 must be aborted.

3. Restart
The restart must be carried out for two cases.
Case 1, the restart must be executed by the normal periodical checkpoint.
Case 2, the restart is also executed in emergency for fault occurrence.

Table 2, concurrency control and recovery in two different situation (Global Commit and Local Commit).

<table>
<thead>
<tr>
<th>Concurrency Control</th>
<th>Global Commit</th>
<th>Local Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency Control</td>
<td>Keep updated data at cache before commit.</td>
<td>Flush updated data to DB before commit.</td>
</tr>
<tr>
<td>Transaction exists in active list.</td>
<td>Transaction exists in local commit list and local active list.</td>
<td></td>
</tr>
<tr>
<td>Updated log is generated.</td>
<td>Updated log is generated.</td>
<td></td>
</tr>
<tr>
<td>When commit is issued at the final server, commit log is broadcast.</td>
<td>When commit is issued at the final server, commit log is broadcast.</td>
<td></td>
</tr>
<tr>
<td>Flush is done at every server simultaneously.</td>
<td>Flush is done at each server step by step.</td>
<td></td>
</tr>
</tbody>
</table>

3.1 Index
Based on the proposed protocol using priority ceiling for the Long Lived Transaction processing, we simulated the concurrency control and obtained experimental results. The concurrency of transaction processing with respect to varying parameters was evaluated, in this case, number of objects located at different servers.

The concurrency of transaction processing is defined by the following equation:
Concurrency of transaction = (number of object) x (issued number of transactions) — (observed access number to objects)

\[(N \times I) - O\]

Where we assume that, every transaction accesses the same number of objects.

3.2 Experimental Results of Concurrency

The experimental result is illustrated in Figure 5 and 6 also shown in Table 3. As the result of performance evaluation, the concurrency ratio is 27.42 percent for three objects, the concurrency ratio is 28.94 percent for four objects, and the concurrency ratio is 29.02 percent for five objects. The simulation result showed that concurrency meaning the simultaneous accesses for different objects at different server’s increases according to the increase of parameters. Meanwhile, for fixed five objects, the concurrency ratios are 28.96 %, 29.02 % and 28.94 %, when the issued numbers of transactions are 1000, 2000 and 3000, respectively. This experimental result indicates the concurrency of transaction processing almost same in spite of increase in the issued number of transactions’.

![Figure 5](image)

*The figure 5, shows concurrency ratio as per the number of transactions*

![Figure 6](image)

*Figure 6, Comparison of concurrency ratios for different number of objects.*

<table>
<thead>
<tr>
<th>Number of objects</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency ratio</td>
<td>0.2742</td>
<td>0.2864</td>
<td>0.2902</td>
</tr>
</tbody>
</table>

Table 3 Comparison of concurrency ratios for different number of objects

4 Conclusions

The enhanced altruistic locking for Long Lived Transaction processing was proposed in virtual environment. The motive was to promote concurrency in virtual environments with Long Lived Transactions. The performance evaluation showed tendency of the concurrency ratio. The concurrency of Long Lived Transaction Processing in Virtual Environment was proposed. Priority Ceiling of transactions based on their arrival time was proposed as best option to bring more management in the system. Aside from that, Recovery mechanisms were highlighted and over viewed.

5 Acknowledgment

Sincere gratitude is hereby extended to those who continued their help until this research was accomplished:

Mr. Kasuke Shuda, Mr. Noriho Okazaki, for conducting and sharing the programing expertise.
Special thanks to the laboratory members of transaction processing, my friends Ryuta Ono, Yuko Nishimura.

For the moral support, thanks to my family and friends who never give up on me.

Above all, utmost appreciation to the almighty God for the divine intervention in this academic endeavor.

6 References


