Speculation-based Protocols for Improving the performance of Read-only Transactions

by

T. Ragunathan, P Krishna Reddy

in

International Journal of Computational Science and Engineering (To appear)

Report No: IIIT/TR/2009/25

Centre for Data Engineering
International Institute of Information Technology
Hyderabad - 500 032, INDIA
January 2009
Speculation-Based Protocols for Improving the Performance of Read-only Transactions

T. Ragunathan*
Center for Data Engineering,
International Institute of Information Technology,
Hyderabad, India.
E-mail: ragunathan@students.iiit.ac.in
*Corresponding author

P. Krishna Reddy
Center for Data Engineering,
International Institute of Information Technology,
Hyderabad, India.
E-mail: pkreddy@iiit.ac.in

Abstract: A read-only transaction (ROT) does not modify any data. The main issues regarding processing of ROTs are correctness, data currency and performance. Two-phase locking (2PL) protocol is widely used for concurrency control which processes transactions with serializability as a correctness criteria. Even though 2PL processes ROTs correctly with no correctness related issues, its performance deteriorates as data contention increases. To improve the performance over 2PL, snapshot isolation (SI)-based protocols have been proposed. Even though SI-based protocols improve the performance of ROTs, serializability is violated and data currency of ROTs is compromised. In the literature, speculative locking (SL) protocols have been proposed to improve the transaction processing performance in distributed database systems. By carrying out multiple executions for a transaction, SL increases parallelism by trading extra processing resources and without violating serializability criteria. However, in SL, the speculative executions explode with the data contention. In this paper, we have proposed efficient speculation-based protocols for ROTs by exploiting features specific to ROT environment. The proposed protocols bring significant benefits over 2PL and SI-based protocols. As compared to SL, the performance results show that the proposed protocols improve the performance with manageable extra processing resources. Further, the proposed protocols process transactions without any correctness and data currency related issues.

Keywords: transaction processing; concurrency control; read-only transactions; speculation.

Reference to this paper should be made as follows: Ragunathan, T. and Krishna Reddy, P. (2008) ‘Speculation-Based Protocols for Improving the Performance of Read-only Transactions’, Int. J. Computational Science and Engineering.

Biographical notes: Ragunathan, T. is a PhD scholar in Center for Data Engineering at International Institute of Information Technology, Hyderabad, India.

P. Krishna Reddy is a professor at Center for Data Engineering, International Institute of Information Technology (IIIT-H), Hyderabad, India since 2007. He has worked as an associate professor at the same university during 2002-2006. He has received both MTech and PhD degrees in computer Science from Jawaharlal Nehru University, New Delhi, India in 1991 and 1994, respectively. From 1994 to 1996, he worked as a faculty member at the Division of Computer Engineering, Netaji Subhas Institute of Technology, Delhi, India. During 1997-2002, he was a JSPS fellow and then research associate at the Center for Conceptual Information Processing Research, Institute of Industrial Science, University of Tokyo, Japan. His research interests include data management, transaction models, distributed computing, performance evaluation, data mining, web mining, and ICTs for agricultural and rural development. He has published about 60 refereed research papers which include 11 journal papers. He has delivered several invited talks at the reputed conferences and workshops in India and abroad. He has developed the eSagu system (An IT-based Personalized Agricultural Advisory System) which has got several recognitions including the best CSI-Nihilent e-Governance Project Award 2005-06 (overall winner), the Manthan Award 2007- India’s best e-Content Development, and Finalist in the Stockholm Challenge Award 2008.
1 INTRODUCTION

In the emerging web databases and e-commerce scenario, large number of users demand information from the corresponding information systems. The information systems frequently process read-only transactions (ROTs) or queries. A read-only transaction (ROT) does not modify any data. The main issues in processing ROTs are correctness (serializability), data currency and performance. The widely used two-phase locking (2PL) protocol (Easwaran et al., 1976; Gray and Reuter, 1993) processes ROTs with serializability as a correctness criteria. However, the performance of 2PL degrades with data contention due to increased waiting. To improve the performance, snapshot isolation (SI)-based methods (Hal Berenson et al, 1995) are widely used to process ROTs. An ROT processed with SI-based protocols reads from a snapshot of the committed data and ignores the modifications produced by the concurrent active transactions. Even though, SI-based approaches improve performance, they compromise on the aspects of both data currency and correctness (serializability).

We briefly explain about the term “data currency”. The notion of data currency is discussed for a data warehousing environment in (Dimitri and Mokrane, 1999) and for a replicated environment in (Hongfei et al, 2004). The term “data currency” refers to how current or up-to-date the system can guarantee a data object to be, for a transaction. Based on this, the term “data currency” is defined as follows. Let T_i and ‘t’ denote a transaction and time duration, respectively. The data currency of the data object provided to T_i is the value of ‘t’ which is the time difference between the commit time of the transaction which created the latest version of the data object (‘x’) and the commit time of the transaction which created the version of ‘x’ that was read by T_i. If ‘t’ is less/more, it means that transactions are provided with high/low data currency.

It can be observed that, the transactions processed with 2PL do not suffer from data currency issues whereas transactions processed with SI-based protocols suffer from data currency issues.

In the literature, a speculative locking (SL) protocol (Krishna Reddy and Masaru Kitusuregwa, 2004) has been proposed to improve the transaction processing performance in distributed database systems. In SL, a transaction releases the lock on the data object whenever it produces the corresponding after-image. The waiting transaction accesses both before- and after-images and carry out speculative executions. After the completion, the transaction retains one of the speculative execution based on the termination decision of preceding transactions. The SL protocol improves the transaction parallelism by carrying out multiple executions per transaction. The SL protocol is proposed to improve the transaction processing performance in OLTP environments by considering transactions which contain both the read and write operations. Through SL, the performance can be improved by trading extra processing resources without violating serializability criteria. However, in SL, the speculative executions of a transaction explode with data contention.

In this paper, we have proposed speculation-based protocols to improve the performance of ROTs by making appropriate modifications and extensions to SL through identifying optimizing features specific to ROT processing environments. The features are as follows. First, by observing the fact that ROTs do not cause the explosion of speculative executions, we have proposed to process update transactions (UTs) with 2PL and the ROTs with the proposed speculation-based protocols. Second, we have exploited the observation that an ROT can commit whenever it is completed without waiting for the termination of preceding transactions and without violating serializability criteria. And third, we have observed that two options exist to process the speculative executions of a transaction: synchronous and asynchronous. By exploiting these features, we have proposed two protocols: synchronous speculative locking protocol for ROTs (SSLR) and asynchronous speculative locking protocol for ROTs (ASLR). The proposed protocols bring several benefits over 2PL, SI-based and SL protocols. As compared to SL, it is possible to improve parallelism among ROTs by processing ROTs with few speculative executions. The simulation results under limited resource environments show that these protocols improve the performance significantly over 2PL and SL-based approaches by adding a fraction (0.2 times) of additional resources. Most importantly, the proposed protocols process the ROTs without any data currency and correctness issues.

1.1 System Model and Notations

A database is a collection of data objects. Users interact with the database by invoking transactions. Transactions are represented with T_i, T_j,... A transaction is a sequence of read and write operations that are executed atomically on the data objects. A transaction can read a set of data objects from the database which forms the read set (RS) of the transaction and modify the values of another set of data objects which forms the write set (WS) of the transaction. An ROT contains only read operations. A UT consists of both read and write operations. The transactions T_i and T_j are said to have a conflict, if RS(T_i) ∩ WS(T_j) ≠ Ø, or WS(T_i) ∩ RS(T_j) ≠ Ø or WS(T_j) ∩ WS(T_j) ≠ Ø. The execution of a transaction must be atomic (Easwaran et al., 1976; Gray and Reuter, 1993); i.e., a transaction either commits or aborts. The commit of a transaction results in all of its changes being applied to the database, whereas the abort results in the changes being discarded.

The database management system consists of modules like transaction manager and a data manager (Bernstein et al., 1987). Processing of transactions is managed by the transaction manager component of database management systems, whereas database is managed by the data man-
Data objects are denoted with ‘x’, ‘y’,... For the data object ‘x’, ‘x_i’ (i = 0 to n) represents i_{th} version of ‘x’. The notation r_{i}[x_j] indicates that read operation is executed on ‘x_j’ by the transaction T_i and w_{i}[x_j] denotes that the transaction T_i performs a write operation on a particular version of ‘x’ and produces ‘x_j’. The notations ‘s_i’, ‘c_i’, and ‘a_i’ denote the start, commit and abort of T_i, respectively. T_{ij} indicates j_{th} speculative execution of T_i.

1.2 Paper Organization

The rest of the paper is organized as follows. In the next section, we discuss the related work. In Section 3, we briefly explain two-phase locking, snapshot isolation-based and the speculative locking protocols. In Section 4, we explain the details of the proposed protocols. The correctness proof is covered in Section 5. We present the performance evaluation results in Section 6. In Section 7, we provide discussions regarding the benefits of the proposed protocols and implementation issues. The last section contains a summary and future work.

2 RELATED WORK

In this section, we review the approaches proposed in the literature for improving the performance of ROTs. We also discuss the related approaches based on speculation.

Regarding correctness, four isolation levels are specified by ANSI/ISO SQL-92 standard (Ansi, 1992) for the processing of transactions. These isolation levels are read uncommitted, read committed, repeatable read, and serializable. These isolation levels are defined in terms of the following phenomena:

Dirty Read: Transaction T_1 modifies a data object. Another transaction T_2 then reads the data object before T_1 performs a COMMIT or ROLLBACK. If T_1 then performs a ROLLBACK, T_2 has read a data object that was never committed and so never really existed.

Non-repeatable or Fuzzy Read: Transaction T_1 reads a data object. Another transaction T_2 then modifies or deletes that data object and commits. If T_1 then attempts to reread the data object, it receives a modified value or discovers that the data object has been deleted.

Phantom: Transaction T_1 reads a set of data objects satisfying some “search condition”. Transaction T_2 then creates data objects that satisfy T_1’s “search condition” and commits. If T_1 then repeats its read with the same “search condition”, it gets a set of data items different from first read.

Transactions do not have dirty read, fuzzy read or phantom problems if they are processed with the ‘serializable’ isolation level. Moreover, the execution of a set of transactions with the ‘serializable’ isolation level is equivalent to execution of those transactions in some serial order. Transactions have phantom problem if they are processed with ‘repeatable read’ isolation level. Transactions have both phantom and fuzzy read problems if they are processed with ‘read committed’ isolation level. Transactions have dirty read, fuzzy read and phantom problems if they are processed with ‘read uncommitted’ isolation level.

Serializability is the accepted correctness criteria for transaction processing. Atul et al (2000) proposed generalized definitions for the ANSI-SQL isolation levels by covering optimistic and multi-version schemes. It has been proposed that the performance of ROTs can be improved by processing them at lower isolation levels, other than serializable isolation level. However, it can be noted that data currency is compromised by processing ROTs at lower isolation levels other than serializable isolation level. Transactions processed at lower isolation levels may read old data by not considering updates performed by concurrent transactions.

The 2PL (Easwaran et al., 1976; Gray and Reuter, 1993) protocol which processes transactions under serializable isolation level, is widely followed in database management systems. In this approach, whenever a conflict occurs for accessing a data object, the conflicting transaction is made to wait in the queue until the lock holding transaction is terminated. The performance of this approach becomes poor with the increase in data contention.

An approach has been proposed in (Hector and Gio, 1982) for distributed environment in which read-only queries are processed with an algorithm that is different from the algorithm used for UTs. The ROTs are executed with specific currency requirements (strong or weak consistency) and can read the updates produced by the preceding committed UTs by examining transaction log in the reverse chronological order until the desired data are reconstructed. A protocol is proposed in (Satyanarayanan and Agrawal, 1993) for managing data in a replicated multi-version environment. In that approach, the ROTs are processed independently of the underlying concurrency control and replica control mechanisms. As a result, the data availability for ROTs increases significantly since they can be executed as long as any one of the object is available in the system.

In (Mohan et al, 1992), an approach has been discussed by maintaining multiple versions of data objects in which ROTs read the particular versions of the data objects based on the version period during which they arrived. The approach avoids undesirable interferences between ROTs and UTs, but the ROTs cannot see the modifications performed by the other active UTs. A dual copy method has been proposed in (Bajojing et al, 2001), which separates the processing of ROTs from UTs to improve ROTs performance. In dual copy method, two copies of data are managed for each data object; a master and a slave. Master copy is used by UTs and ROTs use slave copy. Multiple versions of the slave copy are maintained and these copies are synchronized by the master copy at appropriate times. In the dual copy method, the data currency of ROTs depends on the frequency at which slave copies are updated.

An approach has been proposed in (Andre and Marc, 2003) for processing ROTs in mobile environment by con-
considering the data consistency and currency related issues. The data currency requirements of transactions are divided into three categories: transactions with strong, firm and weak requirements. In that paper only the notion of ‘firm currency’ is discussed, which means the data objects read by an ROT (T_i) must be at least as recent with reference to T_i’s starting time.

To improve the performance of ROTs, a new isolation level called “Snapshot Isolation (SI)” was proposed in (Hal Berenson et al, 1995). (refer to Section 3.2 for details). Note that ROTs processed at SI violate serializability criteria. In (Alan Fekete et al., 2005), a theory is discussed to convert non-serializable executions under SI into serializable executions by modifying the program logic of the applications. However, the approach requires programmers to detect the static dependencies between the application programs and to modify the program which will lead to a semantically equivalent application program that can be executed correctly without violating serializability criteria. In (Sudhir et al, 2007), automating the task of modifying the program logic to satisfy the serializability criteria is discussed.

The notion of speculation has been extended in (Best-varos and Braoudakis, 1995) to optimistic protocol for improving the deadline performance in centralized real-time environments. In (Krishna Reddy and Masaru Kitusuregwa, 2004), speculation has been extended to improve the performance of distributed database systems (refer to Section 3.3 for details) by considering transactions which contain both read and write operations. The SL approach improves the performance by increasing parallelism through carrying out multiple speculative executions per transaction. The transactions processed with SL do not violate serializability and do not suffer from data currency issues. The speculative locking approach improves the performance by trading extra processing resources.

The approaches proposed so far (other than speculation-based approaches), improve the performance of ROTs by compromising serializability criteria. In speculative locking approach, the speculative executions of a transaction expose with data contention.

In (Ragunathan and Krishna Reddy, 2007, 2008) we have made an effort to propose speculative-based approaches for improving the performance of ROTs by exploiting features specific to ROT environments. In this paper, we have developed improved approaches and carried out a detailed performance study. We have come up with two kinds of speculative protocols: SSLR and ASLR. Both approaches improve the performance of ROTs without correctness and data currency issues. The simulation results show that the proposed speculation-based protocols improve the performance over 2PL and SI-based approaches with manageable additional processing resources.

<table>
<thead>
<tr>
<th>Lock requested by T_j</th>
<th>Lock held by T_i</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>yes</td>
</tr>
<tr>
<td>W</td>
<td>no</td>
</tr>
</tbody>
</table>

Figure 1: Lock compatibility matrix for 2PL

### 3 CONCURRENCY CONTROL PROTOCOLS

In this section, we discuss two-phase locking, snapshot isolation-based and speculative locking protocols.

#### 3.1 Two-phase locking

Under 2PL (Easwaran et al., 1976), a transaction requests “read-lock” (R-lock) to read an object and a “write-lock” (R-lock) to write/update the data object. In 2PL, a transaction should own all the required locks before performing any unlock operation. We have considered a variation of 2PL called “strict two-phase locking protocol” (Bernstein et al, 1987). The lock compatibility matrix for 2PL is shown in Figure 1. The terms “yes” and “no” in the matrix means that the corresponding lock requests are compatible and incompatible, respectively. On any data object, the R-lock of the lock requesting transaction is compatible with the R-lock of the lock-holding transaction. If one of them is W-lock, the locks are not compatible.

![Figure 2: Depiction of transaction processing with 2PL](image)

We explain the processing of ROTs under 2PL with an example. Consider the Figure 2. In this, both T_1 and T_3 are UTs and T_2 is an ROT. It can be observed that T_2 has to wait for a lock on the object ‘x’ until T_1 commits. Similarly, T_3 has to wait for a lock on ‘y’. (The space between the last operation and ‘c_i’ notation in the transaction diagram depicts the time required to carry out logging and commit operations. Let ‘t_1’ and ‘t_2’ are time instances. The arrow mark between ‘t_1’ and ‘t_2’ indicates that the action at ‘t_2’ starts only after the action at ‘t_1’).

In this way, in 2PL, performance of ROTs suffers as data contention increases.

#### 3.2 Snapshot isolation-based protocol

A new isolation level called snapshot isolation (SI) is proposed in (Hal Berenson et al, 1995). The SI-level lies between “read committed” and “repeatable read” isolation levels (Hal Berenson et al, 1995). In SI-based techniques,
an ROT (T₃) reads data from snapshot of the (committed) data available when Tᵢ has started or generated its first read operation. The modifications performed by other concurrent UTs, which have started their execution after Tᵢ are unavailable to Tᵢ.

Variations of SI-based protocols include “first committer wins rule” (FCWR) and “first updater wins rule” (FUWR) (Alan Fekete et al., 2005). In both the protocols, ROTs can read from snapshot of data and they are executed without blocking. But UTs are processed differently in these approaches. Let Tᵢ and Tⱼ be UTs. In FCWR, Tᵢ commits if and only if no concurrent Tⱼ has already committed writes of data objects that Tᵢ intends to write. The FUWR protocol uses lock-based approach for UTs. If Tᵢ obtains write-lock on a data object ‘x’, Tⱼ is made to wait. If Tᵢ commits, then Tⱼ will abort. If Tᵢ aborts, then Tⱼ can hold the lock and can continue the execution. The ultimate effect of FCWR and FUWR is same- one of the two concurrent UTs will abort. We consider FCWR in this paper.

\[
\begin{align*}
T₁ & \quad \text{reads } r[x₁] \text{ and writes } w[y₁] \quad \text{at } s₁ \\
T₂ & \quad \text{writes } w[x₂] \quad \text{at } s₂ \\
T₃ & \quad \text{writes } w[x₃] \quad \text{at } s₃
\end{align*}
\]

Figure 3: Depiction of transaction processing with FCWR

It can be noted that, SI-based protocols are not serializable (Alan Fekete et al., 2005). The processing of ROTs using FCWR is depicted in Figure 3. In this figure, both T₁ and T₃ are UTs, and T₂ is an ROT. It can be observed that T₂ reads the currently available values ‘y₀’ and ‘y₁’, and proceeds with the execution. Simultaneously, T₃ also reads ‘x₀’ and produces ‘x₂’. Note that, FCWR allows only one of the conflicting UTs to commit. So, T₃ has to be aborted as T₁ commits. However, as per FCWR, T₂ commits with the old values and it has not accessed the updates produced by T₁ even though T₁ commits before its completion and therefore receives low data currency.

Figure 4 shows an example for serializability violation under FCWR protocol. In this figure, both T₁ and T₂ are UTs, and T₃ is an ROT. It can be observed that T₃ reads

\[
\begin{align*}
T₁ & \quad \text{reads } r[y₀] \text{ and writes } w[y₁] \quad \text{at } s₁ \\
T₂ & \quad \text{writes } w[x₂] \quad \text{at } s₂ \\
T₃ & \quad \text{reads } r[x₃] \quad \text{at } s₃
\end{align*}
\]

Figure 4: Serializability violation under FCWR protocol

The currently available values ‘x₀’ and ‘y₁’ and proceeds with the execution. As T₁ and T₂ are not in conflict, they proceed their executions as per FCWR. We can observe that the execution of the transactions T₁, T₂ and T₃ is not equivalent to a serial order of executions and hence violates the serializability criteria. Note that, the execution of UTs T₁ and T₂ do not violate serializability criteria, But, the introduction of the ROT T₃, makes the execution of transactions T₁, T₂ and T₃ to violate serializability criteria (Alan Fekete et al., 2005).

3.3 Speculative locking protocol

In a database system, whenever a transaction Tᵢ reads data objects, these objects are copied into the private working space in the memory allocated for this transaction. Normally, Tᵢ issues wᵢ[x] (write request on the data object ‘x’) after completing all its work on data object ‘x’ (Agrawal et al, 1994; Salem et al, 1994).

In the speculative locking (SL) protocol (Krishna Reddy and Masaru Kitusuregwa, 2004), it was assumed that a transaction produces after-image whenever it completes the work with that object. A waiting transaction is allowed to access the after-images produced by the conflicting active transactions. By accessing before- and after-images of conflicting active transactions, the waiting transaction carries out multiple speculative executions. After completion, the transaction retains one of the executions based on the termination status of preceding UTs. Note that, in SL, the waiting transaction forms a commit dependency with the preceding transactions from which it has read after-images. When a transaction Tᵢ forms a commit dependency with Tⱼ, Tᵢ commits only after the termination of Tⱼ. So, in SL, the requesting transaction commits only after the termination of preceding transactions with which it has formed commit dependencies. The SL approach improves the transaction processing performance by increasing the parallelism and reducing waiting.

Figure 5 depicts the processing of transactions with SL. Tᵢ₀ indicates jᵗʰ (j > 0) speculative execution of Tᵢ. It can be observed that T₂ starts speculative executions T₂₁ and T₂₂, once T₁ produces the after-image ‘x₁’. T₂ accesses both ‘x₀’ and ‘x₁’ and starts speculative executions. Here T₂ forms a commit dependency with T₁. If T₁ commits, T₂ also commits by retaining the execution T₂₁. Otherwise, if T₁ aborts, T₂ commits by retaining T₂₁. If these transactions were processed under 2PL, T₂ could have ob-
4 SPECULATION-BASED PROTOCOLS FOR ROTs

In this section, we have proposed two speculation-based protocols to improve the performance of ROTs. We present the protocols after discussing the basic idea.

4.1 Basic idea

The SL protocol was proposed by considering UTs; i.e., the transactions that contain both read and write operations. We have made several observations to SL and ROT processing environment and identified optimizing features that are specific to ROTs which bring significant improvements to ROT performance.

- Number of data object versions: In SL protocol, at a time, a data object may have multiple versions which are organized using a tree data structure. Whenever a transaction executes a write operation, new uncommitted object versions are created and the same are added to the corresponding object trees. In SL, the number of versions of a data object explode with data contention. In the proposed protocols for ROTs, at a time only one UT can have EW-lock on a data object. So, only two versions are maintained for a data object in the object tree for the proposed protocols.

We explain the difference regarding tree growth through example. Let T₁, T₂, and T₃ be UTs and require exclusive access to ‘x’. The processing under SL is as follows (The growth of the object tree of ‘x’ is depicted in Figure 7(a)). At the beginning, the data object tree of ‘x’ contains ‘x₀’. T₁ accesses ‘x₀’ and produces ‘x₁’, which is then added to the tree. Since T₁ is processed under SL, it releases the lock after producing ‘x₁’. Next, T₂ accesses the object tree which contains two versions, ‘x₀’, ‘x₁’ and produces the corresponding new versions ‘x₂’ and ‘x₃’, respectively which are then added to the tree. T₃ accesses the tree which contains four data object versions and produces the corresponding new versions which are then included in the tree.

The processing under proposed protocols is as follows (The growth of the object tree of ‘x’ is depicted in Figure 7(b)). Note that no other UT can access ‘x’ until T₁ releases the lock as UTs are processed under 2PL. It can be noted that the tree contains only two versions as ROTs do not add any new versions.

- Number of speculative executions: Whenever a transaction accesses a data object, each of its speculative execution has to spawn the number of speculative executions equal to the number of versions in the object tree. So, the number of speculative executions depends on the number of object trees the transaction is accessing and the number of data object versions maintained in each object tree. In this way, in SL, speculative executions of a transaction explode with data contention.

It can be observed that write operations are the cause for the generation of new uncommitted versions. As a result, the number of speculative executions that are to be carried out by waiting transactions and the number of versions stored in the trees explode with the
increase in data contention. As a result, we require more memory and processing resources.

Regarding processing of ROTs, it can be observed that an ROT only reads the data and does not generate any new versions. So, if we process only ROTs through speculation and UTs with 2PL, it is possible to improve the performance by consuming less resources as compared to the resources required for processing UTs with speculation.

**Commitment:** In SL, a transaction can commit only after the termination of transactions with which it has formed commit dependency. Note that, in SL, after completing execution, UTs must wait for the termination of preceding transactions to decide which speculative execution to retain.

In case of ROTs, it can be observed that the commit dependency formed by ROTs is different from the commit dependency formed by UTs in SL. Suppose an ROT is processed under speculative mode. After its completion it can commit without waiting for the termination of UTs with which it has formed commit dependency: It can simply commit by retaining appropriate execution which contains the effect of committed transactions at that instant. Note that, this processing is correct as per serializability theory.

Let T_i be an ROT and T_j be a UT. Suppose T_i forms commit dependency with T_j. In SL, T_i commits only after the termination of T_j. Whenever T_i completes its execution it would be possible to commit T_i by retaining one of the speculative executions without waiting for T_j to terminate. However, it can be noted that, a UT waits for the termination of preceding UTs and ROTs.

Consider that, we process ROTs with speculation and UTs with 2PL protocol. In this approach, we commit ROTs without waiting for preceding UTs. We briefly argue that the schedules produced by this approach are serializable. Under the proposed approach, UTs are handled using 2PL rules which capture all Read-Write and Write-Write conflicts. The speculation rules capture all the Write-Read conflicts.

For each Write-Read conflict, speculation rules ensure that Read operation reads from the preceding Write operation. Suppose, T_i is an ROT which conflicts with ‘n’ transactions and commits at time ‘t’. We can divide ‘n’ transactions into two sets. One set is “Committed Set (CS)” which includes the transactions which have committed before ‘t’ and another set is “Uncommitted Set (US)” which includes the transactions which are not committed at time ‘t’. As per speculation rules, T_i is committed by including the effects of all the transactions in CS. So, T_i’s execution is equivalent to the serial execution produced after CS. The execution of each transaction in US is equivalent to the serial execution after T_i. It means that the execution is equivalent to serial order CS ≪ T_i ≪ US (“≪” denotes the partial order). So, it can be easily proved that the proposed approach produces serializable schedules.

**Processing speculative executions:** In SL, all speculative executions of a transaction are processed synchronously. That is, the waiting transaction waits for the preceding transaction to produce after-image and starts speculative executions synchronously. However, in case of ROT environment, an ROT can read available version and can proceed with the existing speculative executions without waiting for the preceding transaction to produce after-image in asynchronous manner. Whenever preceding transaction produces after-image, the ROT starts further speculative executions. So, there are two ways to carry out speculative executions: one is synchronous method and another is asynchronous method.

Based on this, we propose speculative locking protocols for ROTs by adding three features to the basic SL protocol.

1. We apply separate protocols for UTs and ROTs. UTs are processed with 2PL and ROTs are processed with proposed protocols.
2. An ROT commits whenever it completes.
3. Two methods of carrying out speculative executions: synchronous method and asynchronous method.

By modifying SL with preceding features specific to ROTs, we propose two protocols: one is synchronous speculative locking protocol (SSLR) and the other is asynchronous speculative locking protocol (ASLR).

The proposed SL-based protocols improve the performance of ROTs due to reduced waiting. Due to speculative processing and an autonomy to commit whenever the processing is completed, the proposed protocols reduce the waiting time without compromising the correctness and data currency.

4.2 The SSLR Approach

We first explain the basic idea of SSLR and then present the protocol.
4.2.1 Synchronous SL for ROTs

In SSLR, the basic aspect is that the speculative executions of ROT are processed in a synchronous manner. UTs are processed with 2PL and ROTs are processed with speculation. The details are as follows.

1. Types of Locks: The lock compatibility matrix of SSLR is shown in Figure 8. The W-lock is divided into EW-lock and SPW-lock. The EW-lock is requested by UTs for writing the data object. The RU-lock (Read lock for UT) is requested by UTs for reading a data object. The RR-lock (read lock for ROT) is requested by ROT for reading a data object. The entry “ssp_yes” (synchronous speculation yes) indicates that the requesting ROT carries out speculative executions and forms a commit dependency. Note that the notion of commit dependency in SSLR is different as compared to the notion of commit dependency in SL. In SSLR, if Ti carries out speculative executions and forms a commit dependency with Tj, Ti commits whenever it is completed (say at time ‘t’) by retaining appropriate speculative execution based on the termination status of Tj at time ‘t’. That is, even though Tj is not completed, Ti can retain one of the speculative execution and proceed to commit whenever Tj completes execution. This is possible, only if Ti is an ROT. Whereas in SL, Ti has to wait for Tj’s termination as SL is proposed for UTs.

2. Processing of UTs: The UTs are processed with 2PL, but with a slight difference. The UTs request RU-lock for read and EW-lock for write/update. After producing after-image, the EW-lock is converted into SPW-lock. (Note that the conversion from EW-lock to SPW-lock is an atomic operation.) Note that EW-lock is incompatible with all other locks. Also, RU-lock is incompatible with EW- and SPW-locks.

3. Processing of ROTs:

- **Processing:** In SSLR, speculative executions of an ROT are processed in a synchronous manner. Suppose, an ROT is carrying out ‘n’ speculative executions and requests RR-lock on a data object. Whenever preceding UT holds EW-lock, the ROT holds all speculative executions till the preceding UT converts EW-lock into SPW-lock on a conflicting data object. As RR-lock is speculatively compatible to SPW-lock, the ROT accesses data objects and carries out speculative executions and forms a commit dependency with the preceding UTs.

- **Commitment:** In SSLR, an ROT commits whenever it completes execution. Whenever an ROT completes execution, it retains one of the execution by selecting appropriate speculative execution which contains the effects of the transactions which have committed at that instant.

![Figure 8: Lock compatibility matrix for SSLR](attachment:image1)

The processing of ROTs under SSLR is illustrated in Figure 9. Here, T1 and T3 are UTs which are processed with 2PL and T2 is an ROT which is processed with SSLR. T1 obtains EW-lock on data object ‘x’. It reads ‘x0’ and produces ‘x1’ and converts the EW-lock on ‘x’ to SPW-lock. T3, being a UT, waits till T1 releases the lock on ‘x’. The ROT T2 is processed as follows. Note that, even though both T1 and T2 have arrived at the same instant, T2 waits till T1 produces after-image ‘x1’, T2 carries out two executions T21 and T22 by accessing ‘x0’ and ‘x1’ respectively. Note that, T21 and T22 are carried out synchronously. After T2’s completion, T21 is retained even though T1 is not yet committed. We can observe that, T2 is committed without waiting for the termination of T1. Also, the transactions are serialized as per the order T2 ≪ T1 ≪ T3.

![Figure 9: Depiction of transaction processing with SSLR](attachment:image2)

4.2.2 The SSLR protocol

We use the following data structures to explain SSLR protocol.

- **dependset(Tij)** is a list maintained to store the commit dependency details of “ith” speculative execution of a transaction Tj. This list is maintained for each speculative execution of transactions.

- **lockqueue** is a FIFO queue maintained for each data object to store the pending lock requests.

**Protocol for UTs:**

In SSLR, a UT requests for RU-lock to read and EW-lock to write. The protocol for UTs is as follows. Let Ti be a UT.

1. **Lock acquisition.** Let Ti requests for RU- or EW-lock on ‘x’. The lock request is entered into the lockqueue.
1.1 $T_i$ obtains RU-lock if no transaction holds EW-lock or SPW-lock. Step (2) is followed.
1.2 $T_i$ obtains EW-lock on ‘x’, if no transaction holds RU-, RR-, EW-, and SPW-locks.

2. Execution. During execution, whenever $T_i$ produces the after-image for a data object, EW-lock on the data object is converted into SPW-lock. If $T_i$ obtains all the locks, step (3) is followed. Otherwise, step (1) is followed.

3. Commit/Abort Rule. Whenever $T_i$ commits, the speculative executions of ROTs which have been carried out with the before-images of $T_i$ are terminated. Whenever $T_i$ aborts, the speculative executions of ROTs carried out with the after-images of $T_i$ are terminated. The information regarding $T_i$ is deleted from the dependset maintained for each of the speculative execution of ROTs which are dependent on $T_i$. Also, all the related lock entries of $T_i$ are deleted.

### Protocol for ROTs:

In SSLR, an ROT requests for RR-lock to read. The protocol for ROTs is as follows. Let $T_j$ be an ROT.

4. Lock acquisition. Let $T_j$ requests for RR-lock on ‘x’. The lock request is entered into the lockqueue.

4.1 If no transaction holds EW- or SPW-locks, the RR-lock is allocated to $T_j$. Step (5.1) is followed.
4.2 If the preceding transaction is holding SPW-lock, lock is granted in speculative mode (ssp_Yes). The identifiers of preceding transactions which hold lock on ‘x’ are included in the $T_j$’s dependset. Step (5.2) is followed.

5. Execution.

5.1 $T_j$ continues with the current executions by accessing ‘x’. If $T_j$ obtains all the locks, step (6) is followed. Otherwise, step (4) is followed.
5.2 Each execution of $T_j$ is split into two executions; one with the before-image and the other one with the after-image. If no further lock requests for $T_j$, step (6) is followed. Otherwise, step (4) is followed.

6. Commit/Abort. Once $T_j$’s processing is completed, one of its speculative executions is chosen as follows: Suppose $T_j$ has completed at time ‘t’. $T_j$ retains that speculative execution which contains the effect of all committed transactions which have committed before ‘t’. If $T_j$ is aborted, then all its speculative executions are also aborted. The locks allocated to $T_j$ are released after the termination.

### 4.3 The ASLR approach

We first explain the basic idea of ASLR approach and then present the protocol.

<table>
<thead>
<tr>
<th>Lock requested by $T_j$</th>
<th>Lock held by $T_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>yes</td>
</tr>
<tr>
<td>RU</td>
<td>yes</td>
</tr>
<tr>
<td>EW</td>
<td>no</td>
</tr>
<tr>
<td>SPW</td>
<td>no</td>
</tr>
</tbody>
</table>

Figure 10: Lock compatibility matrix for ASLR

### 4.3.1 Description of ASLR

In ASLR, the basic aspect is that the speculative executions of an ROT are processed in an asynchronous manner. UTs are processed with 2PL and ROTs are processed with speculation. The details are as follows.

1. **Types of Locks**

   The lock compatibility matrix of ASLR is shown in Figure 10. The W-lock is divided into EW-lock and SPW-lock. The EW-lock is requested by UTs for writing the data object. The RU-lock (Read lock for UT) is requested by UTs for reading a data object. The RR-lock (read lock for ROT) is requested by ROTs for reading a data object.

   Similar to SSLR, EW-lock is incompatible with the other locks. Also, RU-lock is compatible with both RR- and RU-locks, but incompatible to both EW- and SPW-locks. However, there is a difference with SSLR, on the aspect of compatibility of RR-lock with other locks. In SSLR, RR-lock is speculatively compatible only with SPW-lock whereas in ASLR, RR-lock is speculatively compatible with both EW-lock and SPW-lock. But, the nature of compatibility is different from SSLR. So, a different name called “asp_Yes” (asynchronous speculative yes) is used here which means that the ROT carries out the possible speculative executions by accessing available versions of data object and forms a commit dependency with the preceding UTs. The method of commitment is also different (refer to discussion regarding “commit processing”).

2. **Processing of UTs**

   The UTs are processed with 2PL. The EW-lock is incompatible with all other locks and RU-lock is incompatible with both EW- and SPW-locks.

3. **Processing of ROTs in ASLR**

   - **Processing**

     In ASLR, speculative executions of ROT are processed in an asynchronous manner. Whenever an ROT involves in a conflict, it can continue the current speculative executions by accessing the available data object versions, without waiting
for the preceding transaction to produce after-image. Whenever the after-image becomes available, further speculative executions are started dynamically.

In ASLR, RR-lock is speculatively compatible with EW-lock. Suppose, an ROT is carrying out 'n' speculative executions. Whenever preceding UT holds EW-lock, the ROT accesses the available object versions and continues 'n' speculative executions. Whenever, preceding UT converts EW-lock into SPW-lock, the ROT starts additional 'n' speculative executions. In this manner, the speculative executions of an ROT progress asynchronously.

- **Commit processing**
  
  In ASLR, an ROT tries to commit whenever it completes one of the speculative execution. Whenever one or more speculative executions of an ROT completes execution (say at time 't') the following procedure is followed for each completed speculative execution. If a speculative execution \((T_{ij})\) contains the effect of all the committed transactions at 't', it retains that execution and aborts all other speculative executions of that transaction. Otherwise, \(T_{ij}\) becomes obsolete and therefore aborted.

Figure 11 depicts the processing under ASLR. Here \(T_2\) accesses the before-image \(x_0\) and other available values of data objects \(y_0\) and \(z_0\) and starts speculative execution \(T_{21}\). Once the after-image \(x_1\) becomes available, another speculative execution \(T_{22}\) is started. Note that \(T_{21}\) and \(T_{22}\) are executed in a parallel manner. Whenever the processing is completed for any one of the speculative execution, the ROT can be committed provided it contains the effect of committed transactions at that instant. We can observe that, \(T_{21}\) does not depend on \(T_1\) and it is committed once it completes the execution without waiting for \(T_1\). So, \(T_{22}\) is aborted. Note that being UT, \(T_3\) waits for \(T_1\) for the release of the lock on 'x' as per 2PL rule.

![Figure 11: Depiction of Transaction processing with ASLR](image)

It can be noted that, in ASLR, each speculative execution of a transaction progresses at a different pace. As a result, some of the speculative executions may complete early. However, there is a chance that the speculative execution which may complete early may not commit as it may not contain the effect of all transactions which have committed at the time of its completion. Overall, for a transaction the speculative execution which started early can commit with high probability. As a result, ASLR reduces the waiting time and improves performance over SSLR. However, in the worst case, the performance is equivalent to SSLR for certain transactions.

4.3.2 The ASLR protocol

The following data structures are used to explain ASLR protocol.

- \(\text{dependset}(T_{ij})\) is a list maintained to store the commit dependency details of \(j^{th}\) speculative execution of a transaction \(T_i\). This list is maintained for each speculative execution of transactions.
- \(\text{lockqueue}\) is a FIFO queue maintained for each data object to store the pending lock requests.

In ASLR, a UT requests for RU-lock to read and EW-lock to write. The protocol for UTs is as follows.

**Protocol for UTs:**

1. **Lock acquisition.** Let \(T_i\) be a UT and requests for RU-to read 'x' or EW-lock to write 'x'. The lock request is entered into the lockqueue.
   
   1.1 \(T_i\) obtains RU-lock if no transaction holds EW-lock or SPW-lock. Step (2) is followed.
   
   1.2 \(T_i\) obtains EW-lock on 'x', if no transaction holds RU-, RR-, EW-, and SPW-locks.

2. **Execution.** During execution, whenever \(T_i\) produces the after-image for a data object, EW-lock on the data object is converted into SPW-lock. If \(T_i\) obtains all the locks, step (3) is followed. Otherwise step (1) is followed.

3. **Commit/Abort Rule.** Whenever \(T_i\) commits, the speculative executions of ROTs that have been carried out with before-images of \(T_i\) are terminated. Whenever \(T_i\) aborts, the speculative executions of ROTs which have been carried out with after-images of \(T_i\) are terminated. The information regarding \(T_i\) is deleted from the \(\text{dependset}\) maintained for each of the speculative execution of ROTs which are dependent on \(T_i\). Also, all the related lock entries of \(T_i\) are deleted.

**Protocol for ROTs:**

In ASLR, an ROT requests for RR-lock to read. The protocol for ROTs is as follows.

4. **Lock acquisition.** Let \(T_j\) be an ROT and requests for RR-lock to read 'x'. The lock request is entered into the lockqueue.

4.1 If no transaction holds EW- or SPW-locks, the RR-lock is allocated to \(T_j\). The step (5.1) is followed.
5 CORRECTNESS

In this section, we present the correctness proof of SSLR on the lines of 2PL (Bernstein et al, 1987). The correctness of ASLR can be proved similarly.

The terms transaction, history over a set of transactions, and serializability theorem are formally defined in (Bernstein et al, 1987).

Under 2PL, Three types of conflicts occur: R-W, W-R and W-W conflicts. However, under SSLR, the following conflicts occur (refer to Figure 8): RR-EW, RR-SPW, RU-EW, RU-SPW, EW-RR, EW-RU, EW-EW, EW-SPW conflicts. (Note that in SSLR, in place of W-lock both EW-lock and SPW-lock are employed. Also, UTs request RU-lock for read and EW-lock for write. The UTs convert EW-lock into SPW-lock after completing the work on the data object. ROTs request RR-lock to read.)

Let \( p_i[x] \) denote an operation (RR, RU, EW) requested by transaction manager (TM) for \( T_i \) and \( pl_i[x] \) denotes a lock (RR-, RU-, EW- and SPW-locks). The notation ‘\( \ll \)’ indicates partial order. The notation ‘\( T_i \ll T_j \)’ indicates \( T_i \) precedes \( T_j \) in the history. We use the notations ‘\( o \)’ to denote an operation (RR, RU, EW), ‘\( i \)’ to denote locking operations, ‘\( l_i \)' to denote all the locking operations of \( T_i \), ‘\( u_i \)’ to denote unlocking operations, ‘\( c_i \)’ to denote all the unlocking operations of \( T_i \) and \( c_i \) to denote the commitment of \( T_i \).

The SSLR protocol manages locks using the following rules.

1. A transaction has to acquire lock on a data object in order to perform an operation on it.

2. When the scheduler receives an operation \( p_i[x] \) from the TM, the scheduler tests the conflict between \( pl_i[x] \) with some \( ql_j[x] \) that is already set. If there is no conflict, the lock is set for \( T_i \).

3. Once the scheduler has set a lock for \( T_i \), say \( pl_i[x] \), it may not release that lock at least until \( T_i \) is committed or aborted.

4. Once a transaction completes the work on a data object, the scheduler converts the EW-lock into the SPW-lock for that object in an atomic manner.

5. Once the scheduler has released a lock for a transaction, it may not subsequently obtain any more locks for that transaction (on any data object).

6. When the scheduler receives an operation \( p_i[x] \) from the TM, the scheduler tests the conflict between \( pl_i[x] \) with some \( ql_j[x] \) that is already set. Note that, two types of conflicts can occur in SSLR namely “no” and “ssp-yes”.

(a) If the conflict is of “no”, it delays \( p_i[x] \) by forcing \( T_i \) to wait until it can set the lock it needs. This is equivalent to “\( ql_j[x] \ll pl_i[x] \)”.  

(b) If the conflict is of “ssp-yes”, \( T_i \) forms commit dependency with \( T_j \) and it carries out speculative executions. After its completion, \( T_i \) retains one of the speculative executions based on the \( T_j \)’s termination status. If \( T_j \) has already committed, it will retain that execution which is being carried out by reading the after-images produced by \( T_j \). This is equivalent to the order “\( ql_j \ll pl_i[x] \)” or (“\( c_j \ll c_i \)” or “\( u_j \ll c_i \)”). Otherwise \( T_i \) retains that execution which is carried out by reading the before-images of \( T_j \). This is equivalent to the order “\( pl_i[x] \ll ql_j[x] \)” or (“\( c_i \ll c_j \)” or “\( u_i \ll c_j \)”).

Based on the preceding rules, we propose the following propositions.

Proposition 1. Let \( H \) be a history produced by an SSLR scheduler. If \( o_i[x] \) is in \( C(H) \), then \( ol_i[x] \) and \( ou_i[x] \) are in \( C(H) \), and “\( ol_i[x] \ll o_i[x] \ll ou_i[x] \)”.

There are three kinds of operations: read by ROTs, read by UTs and write by UTs. Whenever TM requests read operation on behalf of an ROT or UT, the operation is executed after obtaining the corresponding lock as per rules (1) and (2). When a UT requests a write operation, the
operation is executed after obtaining the EW-lock. After the completion of work on the data object, the EW-lock is converted into SPW-lock atomically. All the locks are released after the commit/abort of transactions as per rule (3). So, lock is obtained for every operation and released after the completion of the operation i.e. \( ol_i[x] \ll q_i[x] \ll ou_i[x] \).

**Proposition 2.** Let H be a complete history produced by an SSLR scheduler. If \( p_i[x] \) and \( q_i[y] \) are in C(H), then \( "p_i[x] \ll q_i[y]".\)

As per the rule (5), a transaction cannot obtain any lock after releasing any other lock. It means every locking operation is executed before any unlocking operation. (However, it can be noted that conversion of EW-lock into SPW-lock which is not equivalent to the unlocking operation.)

**Proposition 3.** Let H be a history produced by an SSLR scheduler. If \( c_i \) and \( u_i \) operations are in C(H), then \( c_i \ll u_i \).

As per rule (3), a transaction may not release the locks until it commits or aborts. So, commit operation of a transaction precedes all unlocking operations of that transaction.

**Proposition 4.** Let H be a history produced by an SSLR scheduler. If \( p_i[x] \) and \( q_i[x] \) ( \( i \neq j \)) are conflicting operations in C(H), then either \( u_i \ll c_j \) or \( u_j \ll c_i \).

Suppose we have two operations \( p_i[x] \) and \( q_i[x] \) that are in conflict, then according to commit dependency rule \((6\text{b})\), either \( u_i \ll c_j \) or \( u_j \ll c_i \).

**Theorem 1.** Let H be a history of the committed transactions under SSLR. Then, H is serializable.

**Proof:** To prove H is serializable, we have to prove that SG (H) is acyclic. Suppose an edge \( T_i \rightarrow T_j \) is in SG (H). As per the propositions (1),(2) and (3), \( u_i \ll l_j \text{ or } u_i \ll c_j \) (unlocking operation of \( T_i \) precedes locking operations of \( T_j \) on data objects) or \( u_i \ll c_j \) (unlocking operations of \( T_i \) on data objects precedes commitment of \( T_j \)) as per the proposition 4. Suppose \( T_i \rightarrow T_j \rightarrow T_k \) is in SG (H). This means that \( u_j \ll l_k \text{ or } u_j \ll c_k \). By transitivity, \( u_k \ll c_j \). By induction, this argument extends to long paths. For any long path \( T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \), \( u_i \ll l_n \text{ or } u_i \ll c_n \). Suppose SG (H) contains a cycle \( T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1 \). This means \( u_i \ll l_n \) or \( u_i \ll c_n \) and in turn \( u_n \ll l_i \) or \( u_n \ll c_i \). By transitivity, \( u_i \ll l_j \) or \( u_i \ll c_i \). The term \( u_i \ll l_j \) is a contradiction as per the propositions (1) & (2). The term \( u_i \ll c_i \) is a contradiction as per the proposition 3. Thus SG (H) has no cycles and therefore H is serializable.

### 6 PERFORMANCE EVALUATION

In this section, we first explain the simulation model. Next, we present simulation results.

6.1 Simulation model

We have developed a discrete event simulator based on a closed-queuing model. We have a pool of CPU servers, all having identical capabilities and are serving one global queue of transactions. Each CPU manages two I/O servers. A CPU server serves the requests placed in the CPU queue in FCFS order. The I/O model is a probabilistic model of a database that is spread out across all of the disks. A separate queue is maintained for each I/O server. Whenever a transaction needs service, it randomly (uniform) chooses a disk and waits in the I/O queue of the selected I/O server (Rakesh et al., 1987).

The description of parameters with values is shown in Table 1. The database size is assumed to be “dbSize”. The parameters “cpuTime” and “iotime” denote the I/O and CPU time associated with reading and writing an object (equivalent to an operating system page). The parameters “rotMaxTransSize” and “rotMinTransSize” are the maximum and minimum number of objects in ROT respectively. The maximum and minimum number of objects in UT is represented by the parameters “utMaxTransSize” and “utMinTransSize” respectively. Each resource unit (RU) constitutes 1 CPU and 2 I/O servers by considering that one CPU can drive two I/O servers. The parameter “noResUnits” represents the number of resource units. The parameter “MPL” denotes the number of active transactions exist in the system.

The value for “dbSize” is chosen as 1000 data objects (Rakesh et al., 1987). This value is chosen to create a situation in which conflicts are more frequent. The value for “cpuTime” is chosen as 5 ms by considering the speed of modern processors (Kam-Yiu et al., 2002). The value for “iotime” is fixed as 10 ms by considering the speed range of current hard disk drives (Barracuda, 2006). Regarding transaction size, we have chosen different parameter values for ROTs and UTs by considering the load character in modern information systems. The values for “rotMaxTransSize” and “rotMinTransSize” are fixed at 20 and 15 respectively and the values for “utMaxTransSize” and “utMinTransSize” are 15 and 5 objects respectively (Kwok-Wa et al., 1998). The size of a ROT is a random number between 15 and 20 (both inclusive) and UT is a random number between 5 and 15 (both inclusive). We conducted the experiments by varying “MPL” from 10 to 100.

**Performance Metrics.** We have employed the following performance metrics: throughput, % of transaction aborts, % of CPU utilization and % of I/O device utilization. Throughput is the number of transactions completed per second. Percentage of transaction aborts is the ratio of the number of aborted transactions to the number of committed transactions. Let ‘ct’, ‘it’ and ‘st’ denote CPU idle time, I/O device idle time and total simulation time.
Table 1: Simulation Parameters, Meaning and Values

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbSize</td>
<td>Number of objects in the database</td>
<td>1000</td>
</tr>
<tr>
<td>cpuTime</td>
<td>Time to carry out CPU request</td>
<td>5ms</td>
</tr>
<tr>
<td>ioTime</td>
<td>Time to carry out I/O request</td>
<td>10ms</td>
</tr>
<tr>
<td>rotMaxTransSize</td>
<td>Size of largest ROT transaction</td>
<td>20 objects</td>
</tr>
<tr>
<td>rotMinTransSize</td>
<td>Size of smallest ROT transaction</td>
<td>15 objects</td>
</tr>
<tr>
<td>utMaxTransSize</td>
<td>Size of largest UT transaction</td>
<td>15 objects</td>
</tr>
<tr>
<td>utMinTransSize</td>
<td>Size of smallest UT transaction</td>
<td>5 objects</td>
</tr>
<tr>
<td>noResUnits</td>
<td>Number of RUs (1 CPU, 2 I/O)</td>
<td>8</td>
</tr>
<tr>
<td>MPL</td>
<td>Multiprogramming Level (10-100)</td>
<td>Simulation Variable</td>
</tr>
</tbody>
</table>

respectively. Then, percentage of CPU utilization is equal to \(100(1-\frac{ct}{ct})\) and percentage of I/O device utilization is equal to \(100(1-\frac{it}{it})\).

Protocols. We have compared SSLR and ASLR with 2PL, FCWR, SI-2PL, and SL protocols. In all these protocols, we have assumed that aborted transactions are resubmitted again after the time duration equals to average response time in order to reduce repeated aborts. For SL, SSLR and ASLR, we have assumed that all the speculative executions are carried out in parallel. We believe that with the availability of multi-core CPUs, parallel processing of speculative threads is feasible (refer to section 7.3). Also, we have not taken into account the cost of deadlock detection as it is same for all locking-based protocols.

In the experiments, the graphs show the mean results of 20 experiments; each experiment was carried out for 10,000 transactions. The results are plotted with a mean of 95 percent confidence intervals. These confidence intervals are omitted from the graphs. The following protocols are implemented.

- **2PL**: The lock requests are generated by a transaction in a dynamic manner, one by one. We have considered a variation of 2PL called “strict two-phase locking” here. The strict 2PL scheduler releases all locks of a transaction together, when the transaction terminates.

- **FCWR**: This is a variation of SI-based protocol. In this protocol, the read requests of the transactions are carried out by accessing the snapshot of the committed data and UTs are executed as per the FCWR rule.

- **SI-2PL**: This is similar to the approach proposed in (Satyanarayanan and Agrawal, 1993; Mohan et al, 1992). Using snapshot isolation, the read requests of the transactions are carried out without any waiting. The UTs follow 2PL procedure. So, the lock requests are issued dynamically, one by one, by the UTs.

- **SL**: This is the protocol proposed in (Krishna Reddy and Masaru Kitusuregwa, 2004). The lock requests are issued dynamically. Both UTs and ROTs speculate. Whenever a transaction produces after-image for a data object, the EW-lock on the data object is changed to SPW-lock. All the speculative executions of a transaction are executed in a synchronous manner.

- **SSLR**: This is one of the proposed protocol. The lock requests are issued dynamically one by one by the transactions. UTs follow 2PL and ROTs follow synchronous speculation. The speculative executions of ROTs are carried out in a synchronous manner.

- **ASLR**: This is the proposed protocol. The lock requests are issued dynamically one by one by the transactions. UTs follow 2PL and ROTs follow asynchronous speculation. The EW-lock on the data object is converted into SPW-lock, once the after-image becomes available. In this protocol, all the speculative executions are carried out in an asynchronous manner.

6.2 Experiments under unlimited resources

![Figure 12: MPL versus Throughput](image)

In the following experiments, we have reported results by simulating unlimited resources environment; i.e., there is no restriction on the number of speculative executions for a transaction.

Figure 12 shows how throughput performance for 2PL, FCWR, SL, SSLR, ASLR and SI-2PL varies with MPL.
in an unlimited resources environment. The performance of 2PL protocol is low due to more lock waiting time. In FCWR, the performance deteriorates due to increased number of aborts. It can be noted that the performance of SL, SSLR and ASLR is significantly higher than that of 2PL and FCWR because of improved parallelism due to speculation. Among the SL protocols, the performance of SL and SSLR is close. This is due to the fact that under unlimited resources environment, both SL and SSLR perform in a similar manner. The ASLR protocol exhibits more performance over SL and SSLR due to increased parallelism. It can be noted that SI-2PL also exhibits higher performance. However, it can be noted that both SI-2PL and FCWR suffer from correctness and data currency problems. Overall, speculative protocols exhibit better performance over other protocols.

![Figure 13: % of UTs versus transaction aborts](image)

Figure 13 shows how abort performance of 2PL, SSLR, ASLR, FCWR and SI-2PL protocols vary with the increase in number of UTs. It can be observed that the number of transaction aborts under FCWR increases with the increase in data contention. However, the number of transaction aborts under 2PL, SI-2PL, SSLR and ASLR protocols is very less in comparison with FCWR.

In Figure 14, the details regarding percentage of transactions which consumed 1, 2, 4, 8 and above 8 speculative executions in case of SL, SSLR and ASLR are shown. It can be noted that about 60 percent of transactions carry out single execution in SL, whereas about 70 percent of transactions carry out single execution in SSLR and ASLR. This is due to the fact that both UTs and ROTs speculate in SL whereas only ROTs speculate in SSLR and ASLR. As more number of transactions carry out speculation, the number of transactions which carry out single execution is less in SL over SSLR and ASLR. Only three percent of transactions carry out two speculative executions in SL, whereas it is 24 percent in SSLR and ASLR. This can be observed that 24 percent of transactions carry out four speculative executions in SL, whereas it is 5 percent in SSLR and ASLR. Also, in SL, about 10 percent transactions carry out more than 8 speculative executions, whereas the percentage value is very less in case of SSLR and ASLR. The results show that a transaction carries out more speculative executions in SL as compared to SSLR and ASLR. The average number of speculative executions for SL, SSLR and ASLR comes to 4.2, 1.5, and 1.5 respectively. This experiment shows that the proposed protocols require very few speculative executions as compared to SL. So, by consuming less extra processing resources, it is possible to improve the performance in ROT environment.

6.3 Experiment under limited resources

In this experiment we report results by simulating the limited resource environment. The number of speculative executions are restricted for a transaction. We assume that a fixed number of resources are available in the system. Each processing unit is termed as one memory unit (MU) which can be used to process one speculative execution. We assume that a fixed number of MUs are available in the system. The MUs are allocated dynamically based on the requirement of the transaction. If the requested number of MUs are not available, the transaction is put to wait. Figure 15 shows the performance of 2PL, SL, SSLR and ASLR protocols by simulating limited resources environments. The experiment is carried out by fixing MUs equal to MPL. We have plotted the results by varying MUs from MPL to 2*MPL and evaluated the performance. It can be observed that the performance of both SSLR and ASLR reaches the maximum value and saturates at MUs value equal to 1.2*MPL. Whereas the performance of SL linearly increases with MUs and the performance is significantly less over SSLR and ASLR even at MUs=2*MPL. Also, note that the 2PL performance does not vary with the increase in MUs. This experiment shows that the performance of ROTs can be improved significantly with a fraction of (0.2 times) additional resources.
6.4 Experiments for resource utilization

In Figure 16, the CPU utilization of 2PL, FCWR, SSLR and ASLR protocols is shown. It can be noted that CPU utilization is high in case of FCWR. This is due to the fact that more number of UTs are aborted and resubmitted. Among SSLR and ASLR, CPU utilization of ASLR is high. Note that in SSLR, a transaction waits for the lock till the preceding transaction produces after-image. Whereas in ASLR, a transaction accesses the available version of the data object and carries out processing. Due to more waiting, the CPU utilization is less in SSLR. Also, the CPU utilization in case of 2PL is much less than other protocols due to more waiting.

In Figure 17, the performance regarding I/O device utilization of 2PL, FCWR, SSLR and ASLR protocols is shown. The trend is similar to the case of CPU utilization graphs in Figure 16.

7 DISCUSSION AND IMPLEMENTATION ISSUES

In this section, we discuss how speculation-based protocols improve the performance over the 2PL, FCWR, SI-2PL and SL protocols. We also discuss the implementation issues of the proposed protocols. We discuss the handling of speculative executions subsequently.

7.1 Comparison of Protocols

Table 2 shows the performance of the proposed protocols SSLR, ASLR and other protocols SL, 2PL, SI-2PL, FCWR by considering several aspects such as throughput, device utilization, performance variation with additional resources, data currency and correctness criteria. Regarding throughput performance, all the protocols except 2PL perform well. However, regarding data currency both FCWR and SI-2PL provide low data currency to transactions whereas SL, ASLR, SSLR and 2PL provide high data currency. Regarding correctness, it can be observed that both FCWR and SI-2PL violate serializability criteria whereas SL, ASLR, SSLR and 2PL follow serializability criteria. By adding additional resources, the performance of SL-based protocols can be improved whereas the performance of other protocols do not change. Regarding amount of additional resources, SSLR and ASLR require less resources as compared to SL. Note that, the resource utilization of ASLR is better than SSLR and 2PL protocols. Even though the resource utilization of FCWR is better than the remaining protocols, its performance is less than both ASLR and SSLR.

Overall, if we consider all aspects together, the proposed SSLR and ASLR protocols perform better over other protocols regarding performance, device utilization, data currency, correctness and trading of extra resources with performance. Among SSLR and ASLR, ASLR performs better in terms of throughput and device utilization.
Table 2: Performance summary of ASLR, SSLR, SL, 2PL, FCWR and SI-2PL protocols

<table>
<thead>
<tr>
<th>Parameter</th>
<th>ASLR</th>
<th>SSLR</th>
<th>SL</th>
<th>2PL</th>
<th>FCWR</th>
<th>SI-2PL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput Performance</td>
<td>More than</td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Correctness</td>
<td>Serializable</td>
<td>Serializable</td>
<td>Serializable</td>
<td>Not Serializable</td>
<td>Not Serializable</td>
<td>Not Serializable</td>
</tr>
<tr>
<td>Data currency</td>
<td>Maximum</td>
<td>Maximum</td>
<td>Maximum</td>
<td>Maximum</td>
<td>Minimum</td>
<td>Minimum</td>
</tr>
<tr>
<td>Device Utilization</td>
<td>More than</td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Performance variation with</td>
<td>Increases</td>
<td>Increases</td>
<td>Increases</td>
<td>Cannot be</td>
<td>Cannot be</td>
<td>Cannot be</td>
</tr>
<tr>
<td>additional computing resources</td>
<td></td>
<td></td>
<td>slowly</td>
<td>improved</td>
<td>improved</td>
<td>improved</td>
</tr>
<tr>
<td>Additional resource requirement</td>
<td>Manageable</td>
<td>Manageable</td>
<td>High</td>
<td>Not required</td>
<td>Not required</td>
<td>Not required</td>
</tr>
<tr>
<td></td>
<td>(0.2 times)</td>
<td>(0.2 times)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7.2 Implementation issues

In this section, we discuss two implementation issues: lock conversion and index locking issues.

There is a notion of ‘lock conversion’ under SL. We have considered that a UT converts EW-lock to SPW-lock whenever it produces the after-images. We believe that since transactions are stored procedures, it is possible to put the lock conversion markers in the transactions by analyzing the corresponding stored procedures. The lock conversion marker indicates when the transaction finishes work on that object. During execution, when lock conversion marker is encountered, the corresponding EW-lock is converted to SPW-lock.

We briefly discuss how index locking should be carried out under the proposed protocols. In database systems, hierarchical locking schemes are used to support concurrent access of index structures. A node in the index structure can be locked either in shared mode (for reading) or exclusive mode (for writing). We have to investigate the issue of converting exclusive lock held by a node in the index structure into speculative lock, so that ROTs can speculatively read that node to access the relevant data page.

7.3 Speculative executions

In simulation experiments, the additional resource requirement to carry out speculative executions is equated with extra memory and extra processing power. We have added extra memory in proportion to the number of speculative executions in terms of memory units. However, it is assumed in case of processing power that the speculative executions of a transaction are carried out in parallel. The assumption is justified as follows. Currently, multi-core CPUs are available with high processing power which can perform parallel executions. So, it is feasible to implement the proposed protocols in such processing environments.

Further, the CPU processing speed is increasing significantly at higher rate than the main memory/disk access time. For the proposed protocols, the simulation results indicate that additional processing power is needed to support only 0.5 number of additional executions. Further, the speculative executions of a transaction could easily be processed in parallel, using multi-threading computing environments. So, it is easy for modern high speed computing environments to support additional executions either by utilizing CPU idle time or by adding more CPUs.

8 SUMMARY AND FUTURE WORK

Speculation-based protocols improve the performance of transaction processing by carrying out multiple executions for a transaction and trading extra processing resources. In this paper, we have investigated how speculation can be exploited to improve the performance of ROTs and proposed two speculation-based protocols. By analyzing the features specific to ROTs, we have shown that the processing of ROTs under speculation provides more opportunities of parallelism as compared to the processing of UTs under speculation. Mainly, we have exploited the fact that an ROT contains only read operations and does not cause the explosion of speculative executions. In addition, the commit processing of ROT provides more opportunities to improve the parallelism under speculation. Besides exploiting these properties, one of the proposed protocol carries out speculative executions of transaction in a synchronous manner and the another proposed protocol carries out speculative executions in an asynchronous manner. The simulation results show that the proposed protocols improve the performance significantly over two-phase locking and snapshot-isolation based protocols with a small fraction (0.2 times) of additional resources. The
proposed protocols do not suffer from any correctness and data currency issues.

As a part of future work we are going to investigate implementation issues of proposed protocols in database environments. We are also planning to investigate the extension of proposed protocols in distributed environments and real-time database systems.

Improving the performance of ROTs without correctness and data currency issues is a crucial factor in several e-commerce environments like stock marketing, airline operating systems and other web services. Also, currently multi-core CPUs are available with high processing power. Main memory cost is also coming down. The proposed speculation-based protocols provide the scope for improving the performance of ROTs in such environments by trading extra processing resources.

REFERENCES


