Improving the Performance of Read-only Transactions through Asynchronous Speculation

T. Ragunathan and P. Krishna Reddy
International Institute of Information Technology, Hyderabad, India.
ragunathan@students.iiit.ac.in, pkreddy@iiit.ac.in

Keywords: Read-only transactions, speculation, concurrency control, database systems, parallel processing.

Abstract
A read-only transaction (ROT) does not modify any data. The main issues regarding processing of read-only transactions (ROTs) are correctness, data currency and performance. Even though the popular two-phase locking protocol processes ROTs correctly with no data currency related issues, its performance deteriorates with data contention. To improve the performance of ROTs, snapshot isolation-based approaches have been proposed. Even though snapshot isolation-based approaches improve the performance of ROTs, both data currency of ROTs and correctness (serializability) are compromised. In this paper, we propose an asynchronous speculative locking protocol (called as ASLR) which improves the performance of ROTs by trading extra processing resources. The simulation results show that ASLR improves the performance of ROTs significantly over two-phase locking and snapshot isolation-based approaches with manageable extra processing resources. The ASLR approach processes ROTs without any data currency and correctness issues.

1. INTRODUCTION
In the emerging web databases and e-commerce scenario, information systems should meet intensive information requirements from a large number of users. The information systems frequently process read-only transactions (ROTs) or queries. To meet the demands, efforts are being made in the literature to investigate improved approaches to process ROTs in an efficient and correct manner. The widely used two-phase locking (2PL) protocol [1][2] performs poorly in the case of ROTs. Efforts have been made to improve the performance by processing ROTs with a multi-version based approach [11] and at lower isolation levels [3][9]. To improve performance of ROTs, a new isolation level called “Snapshot Isolation (SI)” was proposed in [3]. Even though SI-based approaches improve performance, both data currency of ROTs and correctness (serializability) are compromised.

Data currency refers to how current or up-to-date we can guarantee a data object to be, for a transaction. The definition for data currency in a data warehousing environment and in a replicated environment are discussed in [6],[20] respectively. We can define data currency for a general DBMS environment 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 and the commit time of the transaction which created the version of that data object 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 noted that 2PL processes transactions at lower performance with serializability as correctness criteria, whereas SI-based protocols process transactions at higher performance, however by compromising correctness. Also, it can be noted that 2PL provides high data currency and SI-based protocols provide low data currency for ROTs. Recently, an effort has been taken to make SI-based protocols serializable by modifying the logic of the application program [4][5].

In the literature, an effort has been made to improve the transaction processing performance by using the notion of speculation [7]. By exploiting the notion of speculation, it has been shown that it is possible to improve the transaction processing performance by trading extra computing resources.

We are making efforts to improve the performance of ROT intensive environments with speculation-based approaches. The basic idea is to process ROTs with speculation and update transactions (UTs) with 2PL. As a result, there is an opportunity to improve the performance by processing ROTs with few speculative executions as compared to the approach proposed in [7].

Using speculative approach, the speculative executions of a transaction can be processed in two ways: synchronous and asynchronous. In this paper we propose an approach, in which, ROTs are processed with asynchronous speculation and UTs are processed with 2PL. In this approach, an ROT carries out multiple executions in an asynchronous manner by reading uncommitted values produced by preceding UTs. The proposed approach improves the performance of ROTs without violating serializability criteria and compromising data currency. The simulation results show that with the proposed protocol the performance is improved significantly.
over 2PL and SI-based methods by including 0.2 times additional processing resources.

1.1. System Model and Notations
A database system consists of a set of data objects which are denoted by ‘x’, ‘y’, ..., . Transactions are represented by T_i, T_j, ..., where i, j, ..., are non-negative integer values. The database management systems support components like a transaction manager and a data manager [15]. The transaction manager supervises the processing of transactions, while the data manager manages individual databases.

A transaction is a particular execution of a program that manipulates the database by means of read and write operations [15]. 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. 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_i) ∩ WS(T_j) ≠ Ø. An ROT does not contain write operations whereas a UT includes both read and write operations.

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’ and w_i[x_j] denotes that write operation is executed on a particular version of ‘x’ and ‘x_j’ is produced. The notations ‘s’, ‘c’, and ‘a’ denote the start, commit and abort of transactions, respectively. T_ij indicates the j_th speculative execution of T_i.

1.2. Organization of the paper
The rest of the paper is organized as follows. In the next section, we explain the related work. In section 3, we give an overview of 2PL, SI-based and speculative locking (SL) protocols. In section 4, we explain the proposed protocol. In section 5, we present the performance results. In section 6, we present the discussion on implementation issues. The last section contains a summary and conclusions.

2. RELATED WORK
In this section, we review the approaches proposed in the literature for improving the performance of ROTs. Speculation-based approaches are also discussed.

Four isolation levels are specified in ANSI/ISO SQL-92 standard [9] for processing transactions. These isolation levels are read uncommitted, read committed, repeatable read, and serializable. The processing of transactions is considered as correct if they are processed at serializable isolation level. The performance of ROTs can be improved by processing them at lower isolation levels by compromising both the correctness and data currency [3].

An approach has been proposed in [10] for distributed environment in which ROTs are processed with a special algorithm that is different from the one used for UTs. The ROTs are executed with specific currency requirements (strong or weak) and can read the updates produced by the preceding committed UTs by examining transaction log in reverse chronological order until the desired data are reconstructed.

A protocol is proposed in [11] for managing data in a replicated multi-version environment. In this protocol, the execution of ROTs is completely independent of the underlying concurrency control and replica control mechanisms. As a result, the data availability for ROTs increases significantly as they can be executed as long as any one of the data objects is available in the system. In [12], an approach has been discussed for maintaining multiple versions of data objects. In this technique, based on the arrival time, ROTs read particular versions of the data.

The multi-version based approach avoids undesirable interferences between ROTs and UTs, but the ROTs are not allowed to see the modifications performed by the other active UTs.

In the dual copy method proposed in [13], ROTs are separated from UTs. In this method, two copies of data are managed for each data object; a master and a slave. Master copy is used by UTs and the slave copy is used by ROTs.

Multiple versions of slave copy are maintained and these copies are synchronized by the master copy at appropriate times. In this approach, the data currency of ROTs depends on the update frequency of slave copies.

To improve the performance of ROTs, a new isolation level called “Snapshot Isolation (SI)” was proposed in [3]. (Please refer section 3.2 for details). Note that ROTs processed at SI violate serializability criteria and receive low data currency [4].

A theory, which is discussed in [4], characterizes when non-serializable executions of applications can occur under SI. A new mechanism is discussed in the paper which requires the analysis of transaction programs, without requiring any modification of the database management engine. This mechanism requires the programmers be able to detect static dependencies between the application programs and to modify the program which will lead to a semantically equivalent application program that executes serializably. In [5], automating the task of modifying the program logic is discussed.

Speculation has been extended in [14] to optimistic protocol for improving the deadline performance in centralized real-time environments. In [7], speculation has been extended to 2PL for improving the transaction processing performance (please refer section 3.3 for details).

The approaches proposed so far (other than speculation-based approaches), improve the performance of ROTs by compromising data currency and correctness. In speculation-based approaches, transactions carry out speculative executions in a synchronous manner. Mainly,
our contribution is as follows. We have proposed a notion of asynchronous speculation and proposed an improved approach to process ROTs. Through simulation experiments, it has been shown that the proposed approach improves the performance significantly over 2PL and SI-based approaches. In addition, we have compared the performance with that of other speculation-based protocols with respect to CPU utilization, and by simulating a limited resource environment.

3. OVERVIEW OF 2PL, SI-BASED, AND SL PROTOCOLS
In this section, we briefly explain 2PL, SI-based and speculation-based protocols.

3.1. 2PL Protocol
In 2PL, a transaction should obtain all the required locks before performing any unlock operation. We consider a variation of 2PL called strict 2PL [15] for discussion and comparison. The strict 2PL scheduler releases all the locks of a transaction together, when the transaction terminates. The lock compatibility matrix for 2PL is shown in Figure 1(a). A transaction can request for Read (R)-locks to read or Write (W)-locks to write. The entry “yes” indicates that the corresponding locks are compatible and “no” indicates that the corresponding locks are incompatible.

<table>
<thead>
<tr>
<th>Lock Request by T</th>
<th>Lock Held by T</th>
<th>Lock Held by T</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>R</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Figure 1. Lock Compatibility Matrices: a) 2PL b) SL

T₁: r[x₀] w[x₁] r[y₀] w[y₁]
T₂: r[x₀] r[z₀]
T₃: r[y₀] w[x₂]

Figure 2. Depiction of transaction processing with 2PL

The processing of ROTs under 2PL is depicted in Figure 2. T₂, which is an ROT has to wait for lock on the data object ‘y’ until T₁ commits due to read-write conflict. Similarly, T₃ which is a UT has to wait for lock on ‘x’ till T₁ commits due to write-write conflict.

3.2. Snapshot Isolation
To improve the performance of ROTs, a new isolation level called “snapshot isolation (SI)” was proposed in [3]. The SI-level lies between READ COMMITTED and REPEATABLE READ isolation levels [3]. A transaction executing at SI-level reads data from the snapshot of the (committed) data as of the time the transaction started or generated the first read operation. An ROT running at SI-level is never blocked. An SI-based protocol known as “First Committer Wins Rule” (FCWR) is proposed in [3]. Let T₁ and T₃ be UTs. In FCWR, T₁ successfully commits if and only if there is no concurrent T₃ which has already committed the writes of data objects that T₁ intends to write. Otherwise T₁ has to be aborted. The SI-based protocols are not serializable [3] [4].

Figure 3 depicts the processing with FCWR. Both T₁ and T₃ are UTs, and T₂ is an ROT. It can be observed that T₂ reads the currently available values ‘y₀’ and ‘z₀’ and proceeds with the execution. Simultaneously T₁ also reads and issues update operation to ‘x’. As T₁ commits, T₃ has to be aborted as per the FCWR. 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. It can be observed that T₂ misses the updates produced by T₁ and therefore violates the serializability criteria.

T₁: r[x₀] w[x₁] r[y₀] w[y₁]
T₂: r[x₀] r[z₀]
T₃: r[y₀] w[x₂]

Figure 3. Depiction of transaction processing with FCWR

3.3. Speculative Locking Protocol
In the basic SL protocol [7], a waiting transaction carries out speculative executions with the before- and after-images and retains one of the executions based on the termination status of preceding transactions. In SL, a transaction can commit only after the termination of preceding transactions with which it formed commit dependency [7]. We assume that a transaction Tᵢ issues wᵢ[x] after completing all its work on data object ‘x’ (after the last write operation, the data object ‘x’ available in the working space of the transaction is specified as after-image of ‘x’). In SL, W-lock is partitioned into two locks: exclusive write (EW)-lock and speculative write (SPW)-lock. In SL, the EW-lock is converted into SPW-lock when the corresponding transaction produces an after-image. The lock compatibility matrix for SL is shown in Figure 1(b).
Here, the entry “sp_yes” means that the lock requesting transaction performs speculative executions by accessing before- and after-images and forms a commit dependency with the preceding transactions.

Figure 4 depicts the processing with SL. Both T_1 and T_2 are in conflict with each other. It can be observed that once T_1 produced the after-image of ‘x’ (‘x_0’ and ‘x_1’) and carried out speculative executions T_{21} and T_{22}, T_2 forms a commit dependency with T_1. If T_1 commits, T_{21} is retained, otherwise T_{22} is retained.

![Figure 4. Depiction of transaction processing with SL](image)

A family of SL protocols, SL(n), SL(1), and SL(2) for distributed database systems are proposed in [7]. Through simulation experiments it has been shown that the proposed protocol SL improves the performance significantly over 2PL by trading extra resources.

4. PROPOSED APPROACH

In this section, we first explain the basic idea of the proposed protocol. Next, we explain the protocol. Subsequently, we discuss the proof of correctness.

4.1. Basic Idea

The SL protocol [7] was proposed to process UTs; i.e., the transactions that contain both read and write operations. In that protocol, at a time, a data object may have multiple versions which are organized using tree data structure. Whenever a transaction executes a write operation, new uncommitted object versions are created and are added to the corresponding object trees. 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 extra processing resources. Regarding processing of ROTs, it can be observed that an ROT only reads the existing data and does not generate any new versions. So, if we process only ROTs through speculation, it is possible to improve the performance without consuming more resources as compared to the resources used for processing UTs with speculation.

We can process ROTs in two ways by employing speculation. One is synchronous speculation in which an ROT waits till the preceding transaction produces after-image. It means all the speculative executions of an ROT progress at the same pace and complete at the same time. The details of this approach are available in [8].

Alternatively, the speculative executions of ROTs can be processed in an asynchronous fashion. The basic idea is as follows. The speculative executions of an ROT can be carried out in an independent manner. The ROT is allowed to access the available data object versions and carry out speculative executions. Whenever preceding transaction produces after-image, further speculative executions can be started in a dynamic manner. The asynchronous method of processing ROTs reduces waiting and improves the performance. We call the proposed protocol as asynchronous speculative locking protocol for ROTs (ASLR).

![Figure 5. Depiction of transaction processing with ASLR](image)

Figure 5 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. Note that being UT, T_3 waits for T_1 for the release of the lock on ‘x’ as per 2PL rule.

Overall, we propose ASLR by adding three aspects to the basic SL protocol.

a) Separation of speculation for UTs and ROTs: In ASLR, only ROTs carry out speculative executions. Only one execution is carried out for each UT by following 2PL.

b) Speculative execution: The speculative executions of ROT are carried out in an asynchronous manner.

c) Commitment of ROTs: After completion, an ROT does not wait for the termination of conflicting active transactions. Whenever an ROT completes execution (say at time t), it commits by retaining appropriate speculative execution that contains the effect of the committed transactions at that instant.
(t). It will not miss the updates performed by any transaction that has committed before ‘t’.

<table>
<thead>
<tr>
<th>Lock Request by T&lt;sub&gt;i&lt;/sub&gt;</th>
<th>Lock Held by T&lt;sub&gt;j&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>RU</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>RU</td>
<td>yes</td>
</tr>
<tr>
<td>EW</td>
<td>no</td>
</tr>
</tbody>
</table>

**Figure 6.** Lock Compatibility Matrix for ASLR

The lock compatibility matrix of ASLR is shown in Figure 6. Similar to the case of speculative locking, W-lock is divided into EW-lock and SPW-lock. The UTs request EW-lock for writing the data object. The EW-lock is converted into the SPW-lock after the work on the data object is completed. We propose separate read-locks for UTs and ROTs. A UT requests RU-lock (read lock for UT) for reading a data object and an ROT requests RR-lock (read lock for ROT) for reading a data object.

### 4.2. The proposed ASLR Protocol

A ‘dependset’ data structure is maintained for each transaction to maintain the list of transactions with that the transaction has formed commit dependencies. For each data object, a FIFO queue is maintained to store the pending lock requests. The protocol is as follows.

**Protocol for ROTs:**

1. **Lock acquisition:** Let T<sub>i</sub> be an ROT and requests for RR-lock to read ‘x’. The lock request is entered into the queue.
   - a) If no transaction holds EW- or SPW-locks, the RR-lock is allocated to T<sub>i</sub>. The step 2(a) is followed.
   - b) If RR-lock is granted. The identifiers of preceding transactions that hold lock on ‘x’ are included in the T<sub>i</sub>’s dependset.
   - c) If the preceding transaction is holding SPW-lock, then step 2(b) is followed. If the preceding transaction is holding EW-lock, then step 2(c) is followed.
2. **Execution:**
   - (a) T<sub>i</sub> continues with the current executions by accessing ‘x’. Step 2(d) is followed.
   - (b) Each execution of T<sub>i</sub> is split into two speculative executions: one is with the before-image and the other is with the after-image. Step 2(d) is followed.
   - (c) T<sub>i</sub> continues with the current executions by accessing ‘x’. Whenever the preceding UT converts EW-lock into SPW-lock, T<sub>i</sub> starts additional speculative executions by accessing the after-image of ‘x’ produced by the UT.
   - (d) If there are no pending lock requests for T<sub>i</sub>, then step 3 is followed. Otherwise, step 1 is followed.
3. **Commit/Abort Rule:** Suppose one of the speculative executions T<sub>ij</sub> of T<sub>i</sub> has completed at time ‘t’. If the read set of T<sub>ij</sub> contains the effect of all the conflicting transactions that have committed before ‘t’, T<sub>ij</sub> is retained and T<sub>i</sub>’s other speculative executions are aborted. If T<sub>i</sub> is aborted, then all of its speculative executions are also aborted. Also the locks allocated to T<sub>i</sub> are released. The dependset of T<sub>i</sub> is deleted.

**Protocol for UTs:**

4. **Lock acquisition:** Let T<sub>j</sub> be a UT and requests for RU- to read ‘x’ or EW-lock to write ‘x’. The lock request is entered into the queue.
   - (a) T<sub>j</sub> obtains RU-lock if no transaction holds EW-lock or SPW-lock. Step 5 is followed.
   - (b) T<sub>j</sub> obtains EW-lock on ‘x’, if no transaction holds RU-, RR-, EW-, and SPW-locks.
5. **Execution:** During execution, whenever T<sub>j</sub> produces the after-image for a data object, EW-lock on the data object is converted into SPW-lock. If there are no pending lock requests for T<sub>j</sub>, then step 6 is followed. Otherwise step 4 is followed.
6. **Commit/Abort Rule:** Whenever T<sub>j</sub> commits, the speculative executions of ROTs that have been carried out with before-images of T<sub>j</sub> are terminated. Whenever T<sub>j</sub> aborts, the speculative executions of ROTs carried out with after-images of T<sub>j</sub> are terminated. The information regarding T<sub>j</sub> is deleted from the dependsets of ROTs. All the related lock entries are deleted.

### 4.3. Correctness

We briefly argue that the schedules produced by ASLR are serializable [15]. Under ASLR, the UTs are handled using 2PL rules which capture all Read-Write and Write-Write conflicts. The ASLR rules capture all the Write-Read conflicts. For each Write-Read conflict, ASLR rules ensure that Read operation reads from the preceding Write operation. Suppose, let T<sub>j</sub> be an ROT and 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 ASLR rules, T<sub>j</sub> is committed by including the effects of all the transactions in CS. So, T<sub>j</sub>’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<sub>j</sub>. It means that the execution is equivalent to the serial order CS << T<sub>j</sub> << US (“<<” denotes the partial order). So, it can be easily proved that ASLR produces serializable schedules.

### 5. SIMULATION RESULTS

A discrete event simulator, based on a closed-queuing model, has been developed. We have used a pool of CPU servers and I/O servers, all having identical capabilities. Both the servers are serving one global queue of requests of transactions in FCFS order. Two I/O servers are managed by one CPU server. The I/O model is a probabilistic model of a database that is spread out across all 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 [16].

The description of parameters along with values is given in Table 1. The database size is assumed to be “dbSize”. The parameters “cpuTime” and “ioTime” are amounts of CPU and I/O time associated with reading and writing an object (equivalent to an operating system page).
Regarding transaction size, we have chosen different parameter values for ROTs and UTs by considering the load character in modern information systems [19]. The parameters “rotMaxTranSize” and “rotMinTranSize” are the maximum and minimum number of data objects respectively present in an ROT. The maximum and minimum number of data objects present in a UT is represented by the parameters “utMaxTranSize” and “utMinTranSize” 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 “percentageOfUts” denote percentage of UTs. The parameter “mpl” denotes multiprogramming level, which indicates the number of active transactions existing in the system.

The value for “dbSize” is chosen as 1000 data objects [16]. The value for “cpuTime” is chosen as 5 ms by considering the speed of modern processors [17]. The value for “ioTime” is fixed as 10 ms by considering the speed of recent hard disk drives [18]. The values for “rotMaxTranSize” and “rotMinTranSize” are fixed at 20 and 15 respectively and the values for “utMaxTranSize” and “utMinTranSize” are 15 and 5 objects respectively [19]. 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 and CPU utilization. Throughput is the number of transactions completed per second. Let ‘c’ and ‘t’ denote CPU idle time and total simulation time respectively. Then, CPU utilization = 100(1 – (c/t)).

Protocols: We have compared ASLR with 2PL, FCWR, SI-2PL and SL. In 2PL, SL and ASLR transactions request locks in a dynamic manner, one by one. For SL and ASLR we have assumed that all the speculative executions of a transaction are carried out in parallel. In FCWR, the conflicts between UTs are managed by aborting the transactions. Aborted transactions are resubmitted after the time duration equals to average response time. We also consider SI-2PL approach. SI-2PL is a variation to the approach proposed in [11],[12]. In SI-2PL, ROTs are processed with snapshot isolation and UTs are processed with 2PL.

We assume that the cost of performing concurrency control operations is negligible as compared to the cost of accessing objects. 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 were plotted with a mean of 95 percent confidence intervals. These confidence intervals are omitted from the graphs.

**Table 1. Simulation Parameters, Meaning and Values**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Values</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>rotMaxTranSize</td>
<td>Size of largest ROT transaction</td>
<td>20 objects</td>
</tr>
<tr>
<td>rotMinTranSize</td>
<td>Size of smallest ROT transaction</td>
<td>15 objects</td>
</tr>
<tr>
<td>utMaxTranSize</td>
<td>Size of largest UT transaction</td>
<td>15 objects</td>
</tr>
<tr>
<td>utMinTranSize</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>Number of active transactions (10 – 100)</td>
<td>10-100</td>
</tr>
</tbody>
</table>

5.1. Performance results under unlimited resources

Figure 7 shows the variation of throughput performance for 2PL, FCWR, SL, ALSR and SI-2PL with MPL. It can be noted that the performance of ASLR is significantly higher than that of 2PL and FCWR. The 2PL protocol performs poorly as the waiting time of the transactions is more. In FCWR, due to its “first committer wins rule”, more number of UTs gets aborted as data contention increases. This causes the performance of FCWR to deteriorate. Note that ASLR performance is better than that of SL as a result of starting speculative executions without waiting for lock conversion from EW-lock to SPW-lock. It can be observed that the performance of ASLR is more than that of SI-2PL once MPL increases beyond 50. (Note that both SI-2PL and FCWR suffer from correctness and data currency problems.)
In Figure 8, the details regarding the percentage of transactions which consumed 1, 2, 4, 8 and above 8 speculative executions in case of SL and ASLR are shown. It can be observed that, in ASLR, about 23 percent of transactions require two speculative executions whereas in SL about 25 percent of transactions require 4 speculative executions. The average number of speculative executions for SL and ASLR comes to 4.2 and 1.5 respectively.

Figure 8. Details of Speculative Executions

Figure 9 shows the details regarding CPU utilization. Note that, CPU utilization of ASLR is higher than 2PL. As ROts are processed without waiting for the locks, the CPU utilization of ASLR is high. Also the CPU utilization of FCWR is very high as ROTs need not wait for locks. It can be noted that even though the CPU utilization of FCWR is better than that of ASLR and 2PL protocols, more UTs are aborted as data contention increases. Note that aborted transactions are resubmitted. So, the performance of FCWR is less, even though its CPU utilization is high.

5.2. Performance results under limited resources

Figure 10 shows the performance of 2PL, SL and ASLR protocols under limited resources environments. The resources are allocated in terms of memory units (MUS). We assumed that each memory unit carries out one speculative execution. If sufficient number of MUS is not available to carry out speculative executions, the transaction is put to wait. From Figure 10, it can be observed that the performance of ASLR reaches maximum value and saturates at MUS values equal to 1.2*MPL. Note that the performance of SL does not reach the performance of ASLR even after doubling the MUS values equal to 2*MPL. Note that the performance of 2PL is not affected with the number of additional MUS.

Table 2 shows the comparison of ASLR with SL, 2PL and FCWR protocols. From the table, we can understand that the speculative locking protocols ASLR and SL outperform both the 2PL and SI-based protocols. Note that, ASLR performs better than SL in both the limited and unlimited resources environment. Overall, ASLR protocol exhibits better performance than the remaining protocols.

![Figure 9. MPL versus CPU Utilization](image_url)

![Figure 10. Throughput Performance with Limited Resources](image_url)

Table 2. Comparison of ASLR, SL, 2PL and FCWR

<table>
<thead>
<tr>
<th>Parameter</th>
<th>FCWR</th>
<th>2PL</th>
<th>SL</th>
<th>ASLR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput Performance</td>
<td>Medium</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Device utilization</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Performance variation with additional computing resources</td>
<td>Performance cannot be improved</td>
<td>Performance cannot be improved</td>
<td>Performance increases slowly</td>
<td>Performance increases</td>
</tr>
<tr>
<td>Additional resource requirement</td>
<td>Not required</td>
<td>Not required</td>
<td>Very High</td>
<td>Manageable (0.2 times)</td>
</tr>
<tr>
<td>Data Currency</td>
<td>Minimum</td>
<td>Maximum</td>
<td>Maximum</td>
<td>Maximum</td>
</tr>
<tr>
<td>Correctness</td>
<td>Not Serializable</td>
<td>Serializable</td>
<td>Serializable</td>
<td>Serializable</td>
</tr>
</tbody>
</table>
6. DISCUSSION
In this section, we discuss the implementation issues regarding processing of ROTs in ASLR. However, the detailed investigation on these issues will be carried out as a part of future work.

(i) Pre-compilation. In this paper, we assume that a UT releases the lock whenever it produces the after-image. We assume that it is possible to put markers for each data object to indicate when the transaction finishes work on that object. Since the transactions are stored procedures, we believe that it is possible to put the lock conversion markers by analyzing the stored procedures.

(ii) Speculative executions. We have assumed that speculative executions of transaction are carried out in parallel by considering multi-processor environment. It can be noted that additional memory can be added to the system at lesser cost. Since CPU speed is high in the orders of magnitude than the disk I/O, even in a single processor environment, the CPU time can be used productively to improve the performance of ROTs.

7. SUMMARY AND CONCLUSIONS
In this paper we have proposed an asynchronous speculation based protocol to improve the performance of ROTs. The proposed approach does not suffer from any data currency and correctness issues. Through simulation results, it has been shown that the proposed protocol improves the performance significantly over 2PL and SI-based protocols with a fraction (0.2 times) of additional resources.

As a part of future work, we are planning to implement ASLR protocol in DERBY, a prototype DBMS, and evaluate the performance with TPC-C benchmark. We are also planning to investigate the handling of indexes and scans when modifications are performed under ASLR protocol as a part of future work.

Currently multi-core CPUs are available with high processing power. Main memory cost is also coming down. Also, data currency of ROTs is a crucial factor in several environments like stock marketing, airline operating systems and other crucial web services. Asynchronous speculation-based ROT algorithm provides the scope for improving the performance of ROTs in such environments.

REFERENCES