Speculative Locking Protocols to Improve Performance for Distributed Database Systems

P. Krishna Reddy, *Member*, *IEEE Computer Society*, and Masaru Kitsuregawa, *Member*, *IEEE*

Abstract—In this paper, we have proposed speculative locking (SL) protocols to improve the performance of distributed database systems (DDBSs) by trading extra processing resources. In SL, a transaction releases the lock on the data object whenever it produces corresponding after-image during its execution. By accessing both before and after-images, the waiting transaction carries out speculative executions and retains one execution based on the termination (commit or abort) mode of the preceding transactions. By carrying out multiple executions for a transaction, SL increases parallelism without violating serializability criteria. Under the naive version of SL, the number of speculative executions of the transaction explodes with data contention. By exploiting the fact that a submitted transaction is more likely to commit than abort, we propose the SL variants that process transactions efficiently by significantly reducing the number of speculative executions. The simulation results indicate that even with manageable extra resources, these variants significantly improve the performance over two-phase locking in the DDBS environments where transactions spend longer time for processing and transaction-aborts occur frequently.

Index Terms—Distributed database, transaction processing, concurrency control, locking, performance evaluation, speculation.

1 Introduction

In distributed database systems (DDBSs), concurrency control protocols are employed to ensure the correctness when a shared database is updated by multiple transactions concurrently. To process a transaction, the two-phase locking (2PL) protocol [1], [2] or its variant is widely employed for concurrency control and the two-phase commit (2PC) protocol [2] or its variant [3] is widely employed for commit processing. In the literature [4], [5] speculation has been extended to optimistic protocol to improve the deadline performance in real-time centralized environments (see the next section on related work). In this paper, we extend speculation to 2PL to improve the transaction processing performance of DDBSs. In 2PL, it can be observed that even though a transaction produces after-images during its execution, the locks on the data objects are released only after the completion of 2PC. Parallelism could be increased by allowing the waiting transactions to access the after-images which were produced by the lock holding transaction during the execution. In the proposed speculative locking (SL) protocols, the waiting transaction is allowed to access the locked data objects whenever the lock-holding transaction produces corresponding after-images during execution. By accessing both before and after-images, the waiting transaction carries out speculative executions and retains one execution based on the termination decisions of the preceding transactions. SL improves the performance over 2PL by allowing more parallelism among the conflicting transactions without violating serializability criteria. However, SL requires extra processing resources to support multiple executions for a transaction.

The processing of a transaction T_i in DDBS is depicted in Fig. 1a. For a transaction T_i, the notations s_i, e_i, and c_i denote the start of execution, completion of execution, and completion of commit processing, respectively. The notation a_i denotes the abort of T_i. (Note that an abort can happen any time during processing.) Consider T_1 and T_2 that access (X, Y) and (X, Z), respectively. Fig. 1b illustrates the processing with conventional 2PL. In this figure, an arc $a \rightarrow b$, indicates that b happens after a. Also, $r_i[X]$ and $w_i[X]$ indicate read and write operations on X by T_i, respectively. It can be observed that even though T_1 produces the after-image of X during execution, T_2 accesses X only after the completion of T_1's commit processing. Fig. 1c illustrates the processing with SL. Whenever T_1 produces X$'$ (X$'$ denotes an after-image of X), T_2 accesses both X and X$'$ and starts the speculative executions: T_{21} and T_{22}. However, T_2 commits only after the termination of T_1. If T_1 commits, T_{22} is retained. Otherwise, if T_1 aborts, T_{21} is retained.

Under the naive version of SL, the number of speculative executions of a transaction explodes with data contention. In a database system, a submitted transaction is most likely to commit than abort. By exploiting this fact, we proposed the SL variants, SL(1) and SL(2), that process transactions efficiently by significantly reducing the number of speculative executions of a transaction. The simulation results indicate that even with manageable extra resources, both SL(1) and SL(2) significantly improve the performance over 2PL in the DDBS environments where transactions spend longer time for processing and transaction-aborts occur frequently.

The work is motivated by the fact that, in DDBSs, especially in the wide area network (WAN) environments,

- *P. Krishna Reddy is with the International Institute of Information Technology, Gachibowli, Hyderabad, 500019, India.*
E-mail: pkreddy@iiit.net.
- *M. Kitsuregawa is with the Institute of Industrial Science, University of Tokyo, 4-6-1, Komaba, Meguro-ku, Tokyo, 1538505 Japan.*
E-mail: kitsure@tkl.iis.u-tokyo.ac.jp.

Manuscript received 21 Aug. 2000; revised 7 Nov. 2001; accepted 26 Sept. 2002.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 112738.

1041-4347/04/$20.00 © 2004 IEEE Published by the IEEE Computer Society

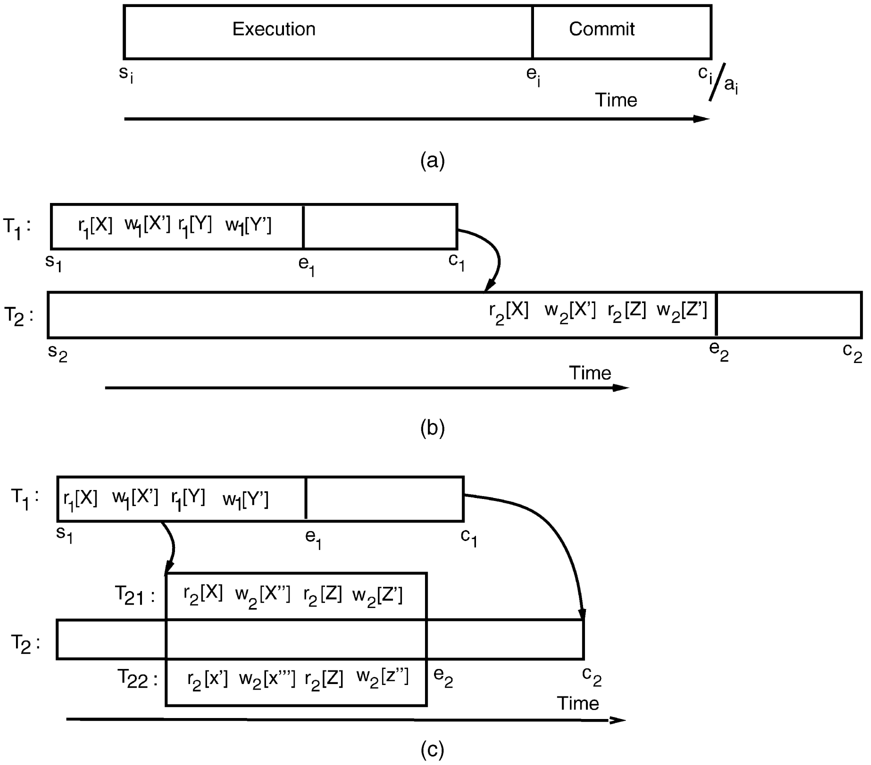
Execution
Commit
s_i
e_i
c_i/a_i
Time
(a)
T_1:
r_1[X] w_1[X'] r_1[Y] w_1[Y']
s_1
e_1
c_1
T_2:
r_2[X] w_2[X'] r_2[Z] w_2[Z']
s_2
e_2
c_2
Time
(b)
T_1:
r_1[X] w_1[X'] r_1[Y] w_1[Y']
s_1
e_1
c_1
T_21:
r_2[X] w_2[X''] r_2[Z] w_2[Z']
T_2:
T_22:
r_2[x'] w_2[x'''] r_2[Z] w_2[z'']
e_2
c_2
Time
(c)

Fig. 1. (a) Processing of T_i. (b) Processing with 2PL. (c) Processing with SL.

communication necessitated by the remote data access occupies a significant portion of transaction execution time. It has been reported that, while it takes only a few milliseconds to deliver a message in a local area network (LAN) environment [6], it takes several hundreds of milliseconds to deliver a message in a WAN environment [7]. It has been shown in [8] that the time to commit accounts for one third of the transaction duration in a general purpose database. Also, it has been reported in [9] that the time to commit can be as high as 80 percent of the transaction time in the WAN environments. This means that, in DDBS, the performance is reduced in case of 2PL, as the data object values become unavailable to the waiting transactions for longer durations. With the continual improvement in hardware technology, we now have systems with significant amounts of processing speed and main memory. Since the cost of both CPU and main memory is falling, extra processing power can be added to the system at a reasonable cost. By trading extra processing resources, SL is able to increase concurrency without violating serializability criteria.

The rest of this paper is organized as follows: In the next section we discuss the related work. In Section 3, we present the SL protocol. In Section 4, we present and analyze the SL variants. In Section 5, we explain how SL can be extended under limited resource environments. In Section 6, we present the simulation results. In Section 7, we explain the performance and implementation issues concerning with SL. The last section consists of the summary and conclusions.

2 Related Work

In the literature, the transaction processing problem has been well-studied in the contexts of both centralized database systems and DDBSs [10], [11], [12]. In this section, we review the approaches proposed in the literature to improve the performance based on the notion of multiple versions, early release of lock and multiple executions.

In multiversion concurrency control [13], each data object is allowed to have multiple versions. The benefit of multiple versions for concurrency control is to help the data manager to avoid rejecting operations that arrive too late. When a transaction writes a data object, it creates a new version. In this scheme, a transaction carries out single execution by selecting the appropriate read version. However, if the transaction which has created a version aborts, the reader must be aborted. However, under SL, the transaction carries out multiple executions by selecting appropriate versions.

In semantics-based concurrency control [14], an invoked operation is allowed to proceed when it is recoverable with respect to uncommitted operations. The invoked operation forms a commit dependency with an uncommitted operation. In this scheme, cascading aborts are avoided, but only among recoverable operations. In the ordered sharing protocol [15], multiple transactions hold conflicting locks on data objects as long as operations are executed in the same order as that in which locks are acquired. The altruistic locking protocol [16] allows transactions to donate previously locked objects after they are done with them, but before the object is unlocked. This protocol is proposed to synchronize long lived transactions. This approach also suffers from cascading aborts. In the modified optimistic protocol [17], a transaction carries out a single execution by reading the uncommitted values produced by the conflicting transaction after its execution, up to only one level. These protocols [15], [16], [17] suffer from cascading aborts.

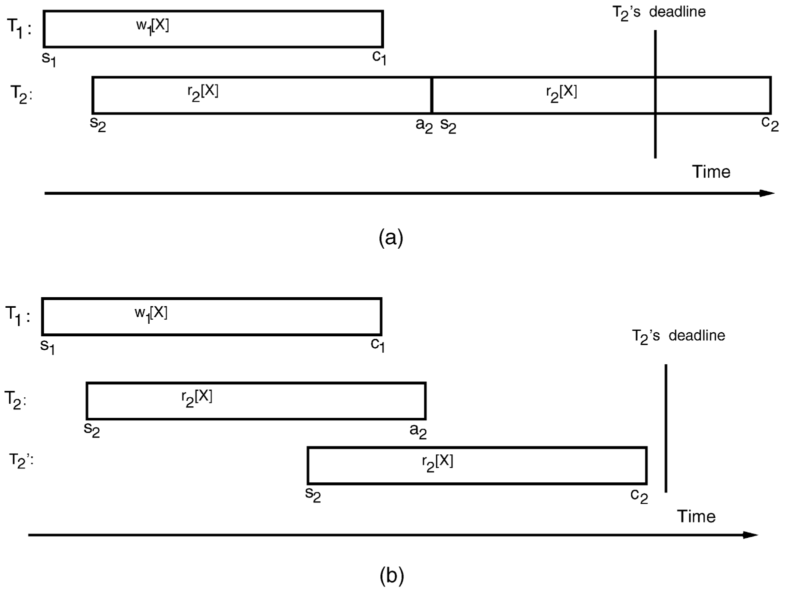

Fig. 2. Processing. (a) Optimistic approach. (b) Optimistic speculation.

In proclamation-based model [18], a cooperative transaction proclaims a set of values, one of which the transaction promises to write if it commits. The waiting transactions can access the proclaimed values and carry out multiple executions. The proclamation-based approach is mainly aimed at cooperative environments such as design databases and software engineering environments, where multiple users work on a single project/design. In the branching transaction model [19], a transaction follows alternative paths of execution in case of a conflict. In that approach the number of executions explode with data contention.

In [4] and [5], speculation has been extended to optimistic concurrency control [20] to improve the deadline performance in real-time centralized environments. For the sake of discussion, let us term this approach as optimistic speculation (OS). In OS, in order to meet the deadline, the transaction optimistically starts another shadow execution as soon as conflicts that threaten the consistency of the database are detected. By carrying out multiple speculative executions for a transaction, speculation increases the probability of processing the transaction before the deadline. For example, consider two transactions T_1 and T_2, where T_2 reads data object X after T_1 has updated it. The processing is depicted in Fig. 2. If the optimistic technique is followed, T_2 is restarted at the validation phase and may miss the deadline. In OS, whenever T_2 encounters a conflict (with T_1), it starts another shadow execution T_2' by reading the before-image of the data object X written by T_1, as if T_1 can be expected to abort. If T_2 is aborted, it can select the shadow execution, T_2', which maximizes the chances of meeting the T_2s deadline. It can be noted that OS is an optimistic approach and is proposed to improve the deadline performance in centralized environments in which a transaction-abort is not expensive as in DDBSs. Whereas, SL is a locking-based approach that requires additional efforts for deadlock handling and is proposed to improve the throughput (not deadline) performance of DDBSs. In OS, the number of speculative executions explode with data contention, whereas, in the proposed SL variant, SL(1), the number of speculative executions of a transaction increase linearly with data contention.

In case of static 2PL, the transaction starts execution only after receiving all the locks. In [21], we extended speculation to static 2PL to improve the performance for DDBSs by allowing a transaction to release the locks after the execution, but before the starting of 2PC. It has been shown that the performance can be improved over 2PL in case of higher conflicts and longer transmission time values. In case of dynamic 2PL, the transaction sends a lock request whenever it requires the lock on the data object during execution. In SL, we extend speculation to dynamic 2PL by allowing the transaction to release the lock whenever it completes the work during execution. We have also extended the naive version of SL to increase concurrency in mobile environments [22] and nested environments [23]. In this paper, we exploit the fact that most of the submitted transactions commit rather than abort and mainly propose the SL variants and demonstrate that these improve the performance over the naive version of SL.

3 Speculative Locking for DDBSs

We first explain the system model and assumptions. Next, we explain the lock compatibility matrix, present a generalized version of SL for DDBS, and briefly discuss about the correctness.

3.1 System Model and Assumptions

A database system consists of a set of data objects. A data object is the smallest accessible unit of data. Each data object is stored at one site only. Data objects are represented by $X, Y, \ldots$. Transactions are represented by $T_i, T_j, \ldots$ and sites are represented by $S_i, S_j, \ldots$, where $i, j, \ldots$ are integer values. The database sites are connected by a computer network. Each site supports a transaction manager and a data manager [10]. The transaction manager supervises the processing of transactions, while the data managers manage individual databases.

Lock requested	**Lock held by T_j**	
by T_i	**R**	**W**
R	yes	no
W	no	no

(a)

Lock requested	**Lock held by T_j**		
by T_i	**R**	**EW**	**SPW**
R	yes	no	sp_yes
EW	sp_yes	no	sp_yes

(b)

Fig. 3. Lock compatibility matrix. (a) 2PL and (b) SL.

Transaction model: A transaction is defined [10] as a set of atomic operations on data objects. An operation is either a read or a write. For any T_i and X, $r_i[X]$ denotes a read executed by T_i on X. Similarly, $w_i[X]$ denotes a write executed by T_i on X. We use c_i and a_i to denote $T_i's$ Commit and Abort operations. In general, a transaction does not have to be a totally ordered sequence. When two operations are not ordered relative to each other, these can be executed in any order. However, a read and a write on the same element must be ordered. The objects to be locked by the transaction for the purpose of read and write steps are termed as read-set (RS) and write-set (WS), respectively. Transactions T_i and T_j are said to have a conflict, if $RS(T_i) \cap WS(T_j) \neq \phi$, $WS(T_i) \cap RS(T_j) \neq \phi$, or $WS(T_i) \cap WS(T_j) \neq \phi$.

Knowledge of after-image: Normally, a transaction copies data objects through read operations into private working space and issues a series of update operations. We assume that for T_i and data object X, $w_i[X]$ operation is issued whenever T_i completes work with the data object. This assumption is also adopted in [15], [16].

Commit protocol: In the literature, a variety of commit protocols have been proposed, most of which are based on the 2PC protocol. The most popular variants of 2PC are presumed abort and presumed commit [3]. To remove the blocking problem of 2PC, three-phase commit (3PC) [24] was proposed. It requires three rounds of communication. In this paper, we propose SL by employing centralized 2PC (communication is between the coordinator and the participants only, i.e., the participants do not communicate among themselves). However, any variant of 2PC or 3PC can be employed without losing generality.

3.2 Lock Compatibility Matrix

The lock compatibility matrix of 2PL is shown Fig. 3a. R and W denote read and write-locks, respectively. The terms *yes* and *no* indicate that corresponding lock requests are compatible and not compatible, respectively. In the SL approach (Fig. 3b), the W-lock is partitioned into the execution-write (EW) lock and the speculative-write (SPW) lock. Transactions request only R and EW-locks. A transaction requests the R-lock to read and the EW-lock to read and write. Consider that the transaction obtains EW-lock and later produces the after-images of the data object. The EW-lock is changed to the SPW-lock after including the after-images in the respective data object tree. We assume no lock conversion from R to EW-lock.[1]

Under SL, only one transaction holds an EW-lock on the data object at any point in time. However, multiple transactions can hold the R and SPW-locks simultaneously. SL ensures consistency by forming a *commit dependency* among transactions. If T_i forms a *commit dependency* with T_j, T_i is committed only after the termination of T_j. In Fig. 3b, the entry **sp_yes** (speculatively yes) indicates that the requesting transaction carries out speculative executions and forms the *commit dependency* with the transactions that hold the R-locks and SPW-locks. The **commit dependency rules** are as follows: 1) If T_i obtains the EW-lock while T_j is holding either an R-lock or an SPW-lock on the data object, T_i forms a commit dependency with T_j, and 2) if T_i obtains the R-lock while T_j is holding an SPW-lock on the data object, T_i forms a commit dependency with T_j.

3.3 Description of the SL Protocol

In this section, we present the generalized SL protocol after explaining the corresponding terms and data structures.

3.3.1 Terms and Data Structures

- HS_i : Home site of a transaction T_i.
- T_{is} : The sth ($s \geq 1$) speculative execution of T_i. The notation T_{i1} is used to represent the initial execution of T_i.
- $depend\_set(T_{is})$, $depend\_set(T_i)$: Suppose T_i carries out m speculative executions. For each T_{is},

  $$depend\_set(T_{is}),$$

  $1 \leq s \leq m$, is a set of transactions from which T_{is} has formed commit dependencies. Also, $depend\_set(T_i) = \cup_{s=1}^{m} depend\_set(T_{is})$.
- $depend\_abort(T_i)$: The term $depend\_abort(T_i)$ is an integer variable which indicates the number of aborted transactions from which T_i has formed commit dependencies.
- $depend\_set(X_q)$: X_q is an uncommitted version of X. The term $depend\_set(X_q)$ is a set of (active) transactions which should commit to retain (commit) X_q.
- $tree_X$: We employ a tree data structure to organize the uncommitted versions of a data object produced by speculative executions. For a data object X, its tree is denoted by $tree_X$. The notation $X_q (q \geq 1)$ is used to represent the qth version of X. Each X_q is stored in $tree_X$ as a tuple $< X_q, depend\_set(X_q) >$. In $tree_X$, the committed version of X becomes the root and the uncommitted versions become the rest of the nodes.
- $LR_{ik}(X)$: Lock request of T_i to X that resides at S_k.
- $depend\_set(LR_{ik}(X))$: The term $depend\_set(LR_{ik}(X))$ is a set of transactions from which T_i has formed commit dependencies at S_k on the data object X.
- $queue_X$: Let S_k be the resident site of X. The incoming lock requests to X are stored in $queue_X$, which is maintained at S_k. Suppose T_i requests a lock on X.

[1] However, lock conversion can be easily incorporated.

Then, tuple $< depend\_set(LR_{ik}(X)),\ status >$ is enqueued to $queue_X$. The $status$ variable takes two values: *waiting* and *not-waiting*.

3.3.2 The SL(r) Protocol

Here, we present the generalized SL protocol, SL(r). In SL(r), if a transaction conflicts with n transactions, it is robust against r $(0 \leq r \leq n)$ transaction-aborts. That is, a transaction is to be aborted if the number of preceding transaction-aborts exceeds r.

Each data object X is organized as a tree with

$$< X_1, depend\_set(X_1) >$$

as a root with $depend\_set(X_1) = \phi$. The locks are requested dynamically one at a time. Also, the site where the object resides is referred to as an object site. Consider that T_i has entered the system. At HS_i, both $depend\_set(T_i)$ and $depend\_abort(T_i)$ are initialized to ϕ and 0, respectively. Suppose T_i requires R/EW-lock on X.

Lock acquisition

1.1 HS_i sends $LR_{ik}(X)$ to S_k, where X resides.

1.2 If $LR_{ik}(X)$ is in conflict with preceding $LR_{jk}(X)$ that is waiting for X, *depend_set* $(LR_{ik}(X))$ is initialized to ϕ, and the tuple $< depend\_set\ (LR_{ik}(X)),\ waiting >$ is enqueued to $queue_X$.

1.3 When the preceding $LR_{jk}(X)$ (lock request of T_j) acquires R/SPW-lock (status = *not-waiting*), the following steps are followed. The R/EW-lock is granted to $LR_{ik}(X)$. If T_i forms a commit dependency with T_j on X, $depend\_set(LR_{ik}(X))$ is updated as $depend\_set(LR_{jk}(X)) \cup \{T_j\}$. The status of $LR_{ik}(X)$ is changed to *not-waiting*. Next, both $tree_X$ and $depend\_set(LR_{ik}(X))$ are sent with a lock reply message to HS_i.

Execution

2.1 At HS_i, on receiving a lock reply message, T_i proceeds as follows. Suppose T_i is carrying m $(m \geq 1)$ speculative executions and $tree_X$ contains v $(v \geq 1)$versions. Based on the value of r, the processing is carried out as follows.

```
1. t=0; ($t$ is a temporary variable)
2. n=(| $depend\_set(T_i)$ | + | $\cup_{q=1}^{n} depend\_set(X_q)$ | );
3. for each $T_{is}$ (s = 1 ... m)
4. for each $X_q$ (q=1 ... v)
5. {
6.     if $r > n$ goto 8;
7.     if | $depend\_set(T_{is})$ $\cup$ $depend\_set(X_q)$ | $\geq$ $(n - r)$
8.         {
9.           t= t+1;
10.          $T_{is}$ branches to new execution, $T_{it}$ with $X_q$;
11.          depend_set($T_{it}$) = (depend_set($T_{is}$) $\cup$
             depend_set($X_q$))
12.        }
13. }
14. m=t;
15. $depend\_set(T_i) = \cup_{s=1}^{m} depend\_set(T_{is})$.
```

2.2 When T_i issues a write operation on X, for all T_{is}, $s = 1 \ldots \mathrm{m}$, the set $\{depend\_set(T_{is}) \cup T_i\}$ is copied to depend_set(X_p), where X_p is an after-image value of X produced by T_{is}. All the after-images with their *depend_sets* are sent to the object site.

At the object site, each after-image value along with its *depend_set* is included as a child to the corresponding before-image node of $tree_X$. Next, the EW-lock on X is converted to an SPW-lock.

2PC processing

3.1 On completion of T_i's execution, the coordinator (HS_i) starts 2PC by sending the PREPARE messages to the participant sites.

3.2 Suppose the participant S_k receives a $PREPARE$ message. If S_k decides to abort the transaction it sends the $VOTE\_ABORT$ message to the coordinator. Otherwise, the following actions are followed. Suppose T_i has accessed a set of data objects, say *access_set*. If $depend\_set(LR_{ik}(X)) = \phi$ for all $X \in access\_set$, a $VOTE\_COMMIT$ message along with the identities of respective root node values for the objects in the *access_set* are sent to the coordinator.

3.3 After receiving the $VOTE\_COMMIT$ messages from all the participants, the coordinator selects the speculative execution to be confirmed and sends the identity of after-images with a $GLOBAL\_COMMIT$ message to all the participants. Otherwise, if it receives a $VOTE\_ABORT$ message from any one participant, it sends a $GLOBAL\_ABORT$ message to all the participants.

3.4 When a participant S_k receives a $GLOBAL\_COMMIT$ message, the steps given below are followed.

3.4.1 The tree of the corresponding data object is replaced by the subtree with the received after-image as the root node.

3.4.2 For any T_w, if $T_i \in depend\_set(T_w)$, the identity of T_i is deleted from $depend\_set(T_w)$. Also, for all T_{ws}, s=1 ... u (T_w is carrying u number of speculative executions), if $T_i \notin depend\_set(T_{ws})$, T_{ws} is dropped.

3.4.3 If any T_w accesses the data object, say X, that was accessed by T_i at S_k, the identity of T_i is deleted from $depend\_set(LR_{wk}(X))$.

3.5 When a participant S_k receives a $GLOBAL\_ABORT$ message, the steps given below are followed.

3.5.1 The after-images included by the transaction are deleted along with the subtrees.

3.5.2 For any T_w, if $T_i \in$ $depend\_set(T_w)$, the identity of T_i is deleted from $depend\_set(T_w)$ and the variable $depend\_abort$ $(T_w) > r$ is incremented. If $depend\_abort(T_w) > r$, T_w is aborted. Otherwise, for all T_{ws}, $s = 1 \ldots u$ (T_w is carrying u number of executions), if $T_i \in$ $depend\_set(T_{ws})$, T_{ws} is dropped.

3.5.3 This step is similar to the Step 3.4.3.

Deadlocks

The process of checking deadlocks [25], [26] that occur due to waiting for the locks and resolving commit dependencies can be achieved using a single wait-for-graph [14]. This graph, known as dependency graph, contains both wait-for edges and commit-dependency edges. Whenever a lock request is made to wait for a lock or forms a commit dependency, an edge is added to the wait-for-graph and deadlock detection is initiated. If a cycle is found, the transaction which is making the request is aborted.

3.4 Correctness

Under SL, each transaction follows commit dependency rules if it carries out speculative executions. Based on commit dependency rules, it can be easily proved that SL histories are serializable similar to 2PL histories [10].

4 The SL Variants

We first discuss the SL variants. Next, we discuss and analyze the processing of transactions under these variants. Then, we explain the processing when a transaction conflicts on multiple data objects.

4.1 The SL(n), SL(0), SL(1), and SL(2) Protocols

We present and analyze the SL variants by considering a situation where T_i conflicts with n transactions that are accessing data object X. We first present extreme pessimistic and optimistic variants of SL by setting $r = n$ and 0 in SL(r), respectively. We then discuss SL(1) and SL(2) by setting $r = 1$ and 2 in SL(r), respectively.

SL(n): It is the naive (extreme pessimistic) version of SL. In SL(n), T_i is robust against n transaction-aborts. Given that each transaction has two possibilities of termination (commit or abort), when T_i conflicts with n transactions, these bring into being 2^n termination possibilities. Therefore, the total number of speculative executions carried out by T_i comes to 2^n. From binomial expansion, we can express 2^n as follows:

$$\binom{n}{0} + \binom{n}{1} + \ldots + \binom{n}{n} = 2^n. \quad (1)$$

In (1), the term $\binom{n}{k}$ indicates the number of combinations (i.e, executions of T_i) of n transactions, when k transactions commit and $n - k$ transactions abort. For a transaction, let p be probability of commit and q be probability of abort $(p + q = 1)$. Then, the probability that k transactions commit and $n - k$ transactions abort is $\binom{n}{k} p^k q^{n-k}$. Also,

$$\sum_{k=1}^{n} \binom{n}{k} p^k q^{n-k} = 1. \quad (2)$$

Under SL(n), as T_i carries out 2^n executions, the probability that T_i commits if any of k $(1 \leq k \leq n)$ transactions abort is equal to 1. Let $SL(n)(abort)$ be the probability of abort (cascading abort probability) of T_i, when k $(1 \leq k \leq n)$ preceding transactions abort. Then,

$$SL(n)(abort) = 0. \quad (3)$$

SL(0): It is the extreme optimistic version of SL. In SL(0), a transaction carries out one execution by reading only the after-images of the preceding transactions. So, T_i is robust against zero transaction-aborts, i.e., T_i has to abort even one of the preceding transactions aborts. Given p, the probability that all transactions commit = p^n. Therefore, the probability that at least one transaction aborts = $1 - p^n$. Let $SL(0)(abort)$ be the probability that T_i aborts when k $(1 \leq k \leq n)$ transactions abort. Then,

p	q	SL(0)	SL(1)	SL(2)	SL(n)
0.90	0.10	0.65	0.263	0.069	0
0.95	0.05	0.40	0.086	0.017	0
0.99	0.01	0.095	0.0042	0.00011	0

Fig. 4. Cascading abort probabilities when a transaction conflicts with 10 transactions on X.

$$SL(0)(abort) = 1 - p^n. \quad (4)$$

From (4), we can observe that, unless p is very close to 1, SL(0)(abort) increases significantly with n. As a result, the performance of SL(0) decreases as data contention increases.

SL(1): In a database system, it can be observed that a submitted transaction is more likely to commit than abort, i.e., $p \gg q$. Based on this fact, we propose SL(1) and SL(2) by processing transactions with moderate robustness against cascading aborts.

In SL(1), a transaction is robust against one transaction-abort. For a transaction, to be robust against one transaction-abort, it is sufficient to support $n + 1$ (by covering only last two terms in (1)) executions. Under SL(1), if the number of the preceding transaction-aborts exceeds one, T_i has to be aborted. Since $p \gg q$, by supporting $n + 1$ executions, cascading abort probability in SL(1) can be significantly reduced as compared to SL(0).

The probability that $n - 1$ transactions commit = $\binom{n}{n-1} p^{n-1} q = np^{n-1}q$. Therefore, the probability that at least $n - 1$ transactions commit = $p^n + np^{n-1}q$. Let $SL(1)(abort)$ be the probability of abort of T_i when k $(1 \leq k \leq n)$ preceding transactions abort. Then,

$$SL(1)(abort) = 1 - p^n - np^{n-1}(1 - p). \quad (5)$$

From (5), it can be observed that if p is close to one, the value of $SL(1)(abort)$ comes close to zero. Also, in SL(1), the number of speculative executions of the transaction increases linearly with n.

SL(2): Under this protocol, a transaction is robust against two transaction-aborts. If the number of the preceding transaction-aborts exceeds two, T_i has to be aborted. To be robust against two transaction-aborts, it is sufficient to support $\sum n + 1$ executions for a transaction. This can be computed by considering the last three terms in (1). The probability that $n - 2$ transactions commit = $\binom{n}{n-2} p^{n-1} q^2$. Therefore, the probability that at least $n - 2$ transactions commit = $p^n + np^{n-1}q + \binom{n}{n-2} p^{n-2} q^2$. Let SL(2)(abort) be the probability of abort of T_i when k $(1 \leq k \leq n)$ preceding transactions abort. Then,

$$SL(2)(abort) = 1 - p^n - np^{n-1}q - \binom{n}{n-2} p^{n-2} q^2. \quad (6)$$

The last term in (6) reduces the value of SL(2)(abort) significantly under high abortive environments. Therefore, SL(2) possesses the potential to improve the performance under high abortive situations (see Fig. 4).

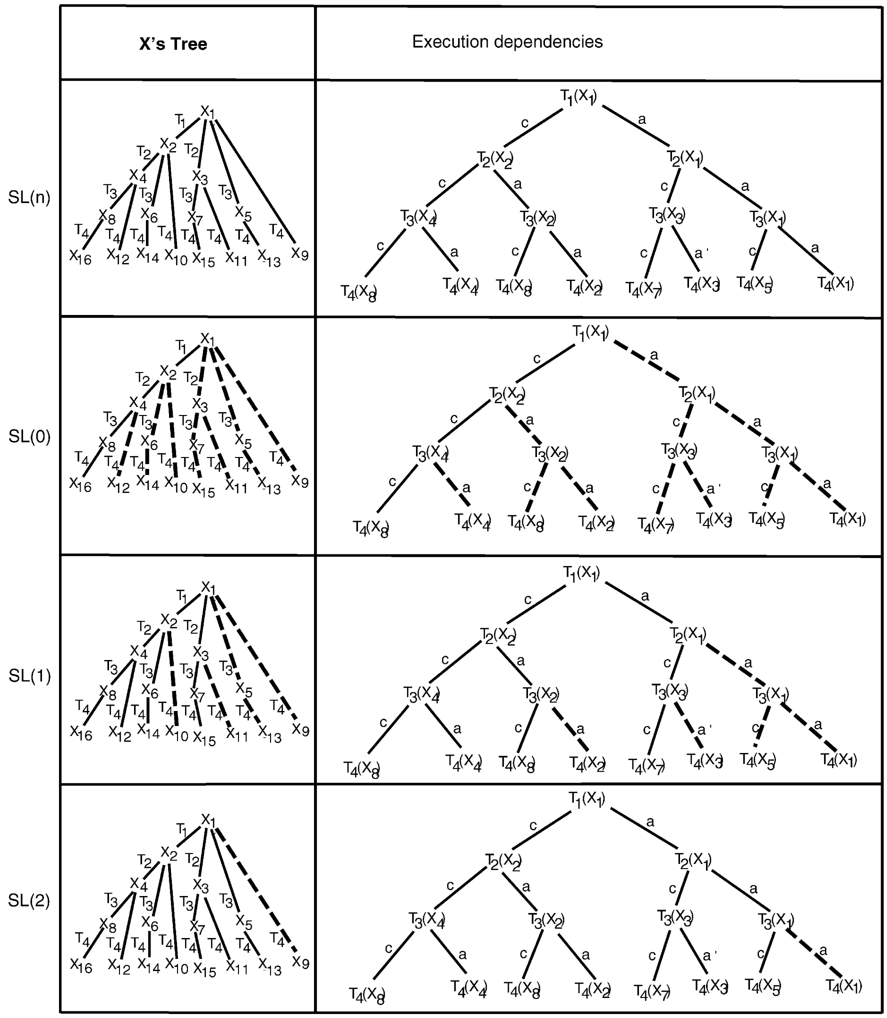

Fig. 5. Depiction of tree growth and the speculative executions under SL(n), SL(0), SL(1), and SL(2) when four transactions conflict on X. The notations c and a denote commit and abort, respectively. Also, a node of $tree_X$ is represented with corresponding version without its $depend\_set$.

4.2 Discussion

We first explain the processing under the SL variants through an example and discuss the feasibility of other SL variants.

4.2.1 Processing under SL Variants

Suppose T_1, T_2, T_3, and T_4 conflict on a data object X, in that order. Fig. 5 depicts the speculative executions of the transactions and the number of versions in the Xs tree under SL(n), SL(0), SL(1), and SL(2). A commit generates an after-image of a data object where an abort causes no change to the before-image value. The transactions update the Xs tree as follows: T_1 reads X_1 and produces X_2, which is then included to the Xs tree. Next, T_2 carries out two speculative executions: One execution reads X_1 and produces X_3, and the other execution reads X_2 and produces X_4. Both X_3 and X_4 are then included to the Xs tree. Similarly, T_3 and then T_4 carry out speculative executions and corresponding new versions of X are included to the Xs tree.

In SL(n), a transaction carries out 2^n executions and need not be aborted even all the preceding transactions abort. For example, if T_1, T_2, and T_3 abort, T_4 can retain the execution carried out by reading X_1. In SL(0), a transaction carries out one execution by reading only after-images produced by the preceding transactions and has to be aborted even if one among the preceding transactions aborts. (In Fig. 5, thick dotted lines indicate invalid options.) In SL(1), the transaction carries out $n+1$ executions and is not aborted if any one of the preceding transactions abort. However, if the number of preceding transaction-aborts exceeds one, the transaction has to be aborted. For instance, if T_1 commits and both T_2 and T_3 abort, then T_4 has to be aborted in case of SL(1). In SL(2), the transaction carries out $\sum n+1$ number of executions and has to be aborted if more than two transactions abort. For instance, if T_1, T_2, and T_3 abort, then T_4 has to be aborted. It can be observed that T_4 carries out eight executions under SL(n) and only four executions under SL(1).

4.2.2 SL(3), SL(4), ... Approaches

Note that, by varying the value of r in SL(r), we can have more variants such as SL(3), SL(4), etc. However, as we increase the robustness (or the value of r), the number of speculative executions of the transaction also increases accordingly. For instance, if we consider SL(3), the number of executions comes to $(n^3 + 5n + 6)/6$. In this paper, we do

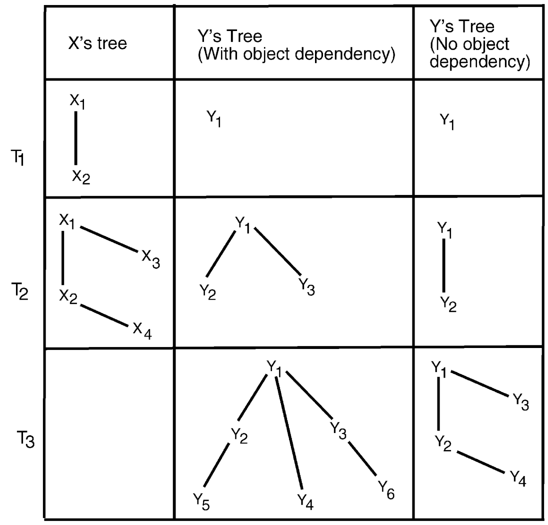
X's tree
Y's Tree (With object dependency)
Y's Tree (No object dependency)
T1
T2
T3
X1
X2
X3
X4
Y1
Y2
Y3
Y4
Y5
Y6

Fig. 6. Depiction of the growth of the object trees in case of an object dependency.

not consider the other variants as SL(2) is enough to improve the performance in high abort situations (see experiment results).

4.3 Multiple Conflicts

We discuss the case where a transaction conflicts with other transactions on multiple data objects. In this case, *object dependency* between the data objects accessed by the transaction affects the number of speculative executions of the subsequent transactions. Suppose T_i accesses two data objects X and Y. We denote the object dependency between X and Y with depend(X,Y), which is either true or false. If $w_i[Y]$ is an arbitrary function of $r_i[X]$ or $w_i[X]$, depend(X,Y) is true. Otherwise, depend(X,Y) is false. Note that $depend(X,Y) \neq depend(Y,X)$.

If the object dependency exists between any two objects accessed by the transaction, the subsequent transaction may form indirect dependencies. For example, consider three transactions T_1, T_2, and T_3 that arrive to access (X), (X,Y), and (Y), respectively, in that order. The progress of the trees of both X and Y is depicted in Fig. 6. T_1 accesses X_1 and then produces X_2. When T_2 accesses both X_1 and X_2, it starts two speculative executions and each execution then reads Y_1. We explain the processing by considering two cases. First, consider depend(X,Y) is true. In this case, T_2 adds two new versions to the Y's tree as both executions of T_2 read the same version Y_1, but produce different versions Y_2 and Y_3. When T_3 accesses Y's tree, it carries out three executions since the Y's tree contains three versions. So, the selection of the speculative execution of T_3 depends on the termination mode of T_1 even though there is no direct conflict between T_3 and T_1. In this situation, T_3 forms an indirect dependency with T_1. And, second, if depend(X,Y) is false, T_2 adds only one new version Y_2 to the Y's tree. So, T_3 carries out only two executions and selects the appropriate execution based only on the T_2's termination. If object dependency does not exist among the data objects accessed by a transaction, a transaction carries out 2^n, $n+1$, and $\sum n+1$ executions in case of SL(n), SL(1), and SL(2), respectively. However, if object dependency exists among the data objects accessed by the transaction, the SL protocols have to carry out additional executions to cover indirect dependencies. Suppose a transaction conflicts with n transactions directly and m transactions indirectly. Then, it carries out at most 2^{n+m}, $n+m+1$, and $\sum(n+m)+1$ executions in case of SL(n), SL(1), and SL(2), respectively. In this paper, however, we carry out simulation experiments assuming that object dependency exists among all the data objects accessed by a transaction. However, the issue of how object dependency affects performance will be carried out as part of future work.

5 Extension under Limited Resources

We explain how the SL protocols can be extended for limited resource environments.

In the SL protocols, since each speculative execution needs separate work space, the size of the main memory available in the system limits the number of speculative executions carried out by the transaction. With this limitation, processing cost may not be considered as a significant overhead as current technology provides high speed parallel computers at low cost. According to size of the main memory, the number of speculative executions carried out by a transaction can be controlled with two variables: $executions\_limit$ and $versions\_limit$.

- *Executions_limit* (*EL*). Suppose the amount of memory to carry out a single execution is one memory unit. This also includes the space to keep both the before and after-images of an execution. The EL value is set to the maximum number of speculative executions that can be carried out by the system for a transaction.
- *Versions_limit* (*VL*). The VL value specifies the maximum number of versions allowed in the object's tree. If the number of versions in the tree of a data object exceeds VL, the lock request is made to wait; i.e., the lock request waits for the termination of the preceding transactions. If the number of nodes in the tree becomes less or equal to the VL value, the lock is granted to the waiting lock request.

Determining the EL value: Given the fixed amount of resources in the system, we present a simple strategy to determine the EL value. We assume that all the transactions require the same amount of main memory for execution. Also, for the transaction, all of its speculative executions require the same amount of main memory. Let each execution requires one unit of main memory. Let MPL/site denotes the number of active transactions at a site and $memory\_units$ (mus), where $mus \geq MPL$, denotes the number of memory units available at a site. In the SL approach, the EL value for the transaction is fixed based on the mus value. By knowing the values of both mus and MPL at a site, we determine EL as $\lfloor \frac{mus}{MPL} \rfloor$ units of memory to each active transaction. For instance, with $mus = 20$ and MPL/Site = 10, two units of memory is allotted to each transaction. However, if MPL is reduced to five, with $mus = 20$, four units of memory can be allocated to each transaction.

TABLE 1
Model Parameters with Settings

Parameter	Meaning	Values
db_size	Number of objects in the database	1000 and 10,000
num_sites	Number of sites in the DDBS	5
trans_size	Mean_size of transaction	8 objects
max_size	Size of largest transaction	12 objects
min_size	Size of smallest transaction	4 objects
write_prob	Pr (write X/read X)	1
local_to_total	Ratio of local requests to total requests	0.6
res_cpu	Time required to carry out CPU request	15 msec
res_io	Time required to carry out I/O request	35 msec
rus	Number of resource units per site	1, 5, and 10
mus	Number of memory units per site	Simulation variable
MPL	Multiprogramming level per site	Simulation variable
trans_time	Transmission time between two sites	Simulation variable

Note that this method suffers from the problem of underutilization of processing resources. We leave the problem of optimal distribution of extra memory resources among transactions by considering transaction specific properties (such as priority and length) for further investigation.

6 Simulation Study

We first explain the simulation model, and then discus the simulation results of 2PL and optimistic locking protocols. Next, we discuss the simulation results under the limited resource and skewed environments.

6.1 Simulation Model

We developed a discrete event simulator on a closed queuing model of a DDBS. For the sake of simplicity, the communication network is modeled as a fully-connected network. Any site can send messages to all the sites at the same time. Each site is modeled with multiple CPU servers and I/O servers [30]. When a transaction needs CPU service, it is assigned a free CPU server. Thus, at each site, we have a pool of CPU servers, all identical and serving one global CPU queue. Requests in the CPU queue are served in the FCFS manner. The I/O model at each site is a probabilistic model of a database that is spread out across all the disks. There is a queue associated with each of the I/O servers. When a transaction needs service, it randomly (uniform) chooses a disk and waits in an I/O queue with the selected disks. The service discipline for the I/O queue is also FCFS.

The meaning of each model parameter for simulation is given in Table 1. The size of the database is assumed to be db_size data objects. We equate each data object with a page. The database is uniformly distributed across the num_sites sites. A new transaction is assigned an arrival site which is chosen randomly over num_sites. The parameter $trans\_size$ is the average number of data objects requested by the transaction. It is computed as the mean of a uniform distribution between max_size and min_size (inclusive). The probability that an object read by a transaction will also be written is determined by the parameter $write\_prob$. The parameter $trans\_time$ is the time required to transmit a message between sites. The WAN behavior is realized by varying $trans\_time$. The parameter $local\_to\_total$ is the ratio of the number of local requests to the number of total requests for a transaction. The CPUs and disks are allocated at each site in terms of resource units; each resource unit consists of one CPU and two disk units. The parameter rus is the number of resource units at each site. The parameter res_io is the amount of time taken to carry out an i/o request and the parameter res_cpu is the amount of time taken to carry out a CPU request. Accessing a data object requires res_io and res_cpu. To force-write a log record (no data) requires res_cpu and then res_io. Also, to write each data object to the disk requires res_cpu and then res_io. The total number of concurrent transactions active in the system at any time is specified by the multiprogramming level (MPL). Each transaction execution consumes a memory unit. The parameter mus indicates the amount of memory units available at the site.

The settings for res_io, res_cpu, db_size, $trans\_size$, min_size, and max_size parameters are given in Table 1 [30]. The variable num_sites is fixed at 5. The $local\_to\_total$ ratio for a transaction is fixed at 0.6 [31]. Thus, 60 percent of the data objects are randomly chosen from the local database and 40 percent of the data objects are randomly chosen from the remaining database sites. We have set $write\_prob$ parameter to 1; i.e., a transaction writes all the data objects it reads. For simplicity, we do not consider shared accesses. With respect to shared accesses, it has been shown that a certain fraction of shared accesses can be taken into account by modifying the effective database size [32]. We assume that all the data is accessed from the disk.

The graphs show mean values that have relative half-widths about the mean of less than 10 percent at the

90 percent confidence level. Each experiment has been run 10 times; each time with 10,000 transactions. Only statistically significant differences are discussed here.

Whenever a lock request waits, deadlock is checked. In case of a deadlock, the waiting transaction is selected as a victim. The aborted transaction is resubmitted after a delay and makes the same data accesses as its original incarnation. The length of the delay is equal to the average transaction response time which is used in most transaction management studies [30], [17]. We neglect the overheads for detecting deadlocks since they are usually negligible compared to the overall cost of accessing data [33]. Also, since the cost of detecting deadlocks is the same for all the approaches, it will not affect the relative performance.

The primary performance metric of our experiments is throughput, that is, the number of transactions completed per second. In the simulation experiments we vary MPL/Site from one to 20 (since there are five sites, overall MPL is varied from five to 100). We vary transmission time from 0 milliseconds (msec) to 500 msec to take care of both the LAN and WAN environments [7], [9].

We investigate the performance of the following approaches.

- **2PL:** We follow the dynamic 2PL. On arrival, the transaction starts its execution by issuing a lock request on the first data object. After receiving a lock reply, subsequent lock request is issued. After execution, 2PC is followed for commit processing. The locks are released on a data object when the corresponding site receives final commit messages.
- **WDL:** In the literature, by varying the blocking policy of 2PL, *wait depth limited (WDL)* locking algorithm [34] was proposed for centralized environments. In the WDL approach, the number of transactions waiting for a data object is limited based on some properties of transactions. Here, we consider a variant of WDL, in which one transaction is allowed to wait for a data object at any time. Suppose T_1 has acquired a lock on X. If T_2 arrives for object X, it is put to wait. Next, if T_3 arrives for object X, then we compare the progress of T_2 with T_3. The progress of a transaction is defined as the number of locks held by it. If T_3s progress is more than T_2 then T_2 is aborted; otherwise T_3 is aborted.
- **SL(0) or Optimistic locking:** We have investigated the performance of SL(0) at levels 1 and 2 (termed as SL(0)(level=1) and SL(0)(level=2)). In SL(0)(level=1), a transaction is put to wait if more than one transaction has already accessed the data object, whereas, in SL(0)(level=2), a transaction is put to wait if more than two transactions have already accessed the data object. Otherwise, the locks are granted to the requesting transaction.
- **SDTP:** In speculative distributed transaction processing (SDTP) protocol [21], the EW-lock on a data object is converted to SPW-lock when the transaction finishes execution (before the starting of 2PC).
- **SL(unlimited), SL(n), SL(1), SL(2):** Similar to 2PL, lock requests are issued dynamically. When the transaction receives a lock reply, corresponding object tree is updated with the new versions equal to the number of executions at that instant. The after-images are sent to the remote site. Next, the EW-lock is changed to an SPW-lock. When a transaction terminates the respective data object trees and $depend\_sets$ of proceeding transactions are modified. In SL(unlimited), we show the maximum (upper bound) performance that could be obtained with SL(n) assuming unlimited resources are available in the system.

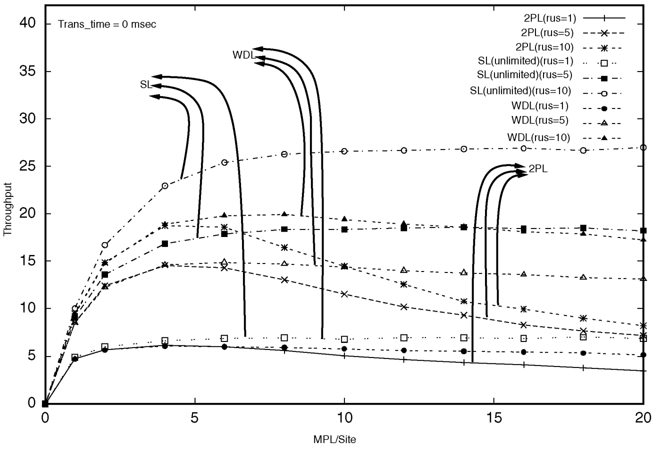

Fig. 7. MPL versus throughput at $trans\_time = 0$ msec.

In addition, we consider the case in which lock requests are always granted by assuming no contention. This gives upper bound on the throughput obtainable from a given system.

We consider the WDL approach because it has been proposed to improve the throughput performance in high data contention environments over immediate-abort (2PL aborting on a conflict) and the optimistic approaches. SL(0) represents the group of protocols [15], [16], [17] which are based on the notion of early release of lock (with cascading abort problem).

6.2 Simulation Results: 2PL, WDL, SL(0), and SL(unlimited)

Here, we discuss the performance of the 2PL, WDL, SL(0), and SL(unlimited) approaches.

At different MPL values, Fig. 7 shows the performance of the 2PL, WDL, and SL(unlimited) approaches at $trans\_time = 0$ msec. At rus = 1, the performance of these approaches is close because of the physical resource (disk) contention. As we increase rus to 10, physical resource contention is reduced and data contention influences the performance. At low MPL values, all the approaches behave similarly as the conflicts are rare. As we increase MPL, data contention increases accordingly. As a result, more transactions wait for the locks. In 2PL, at higher MPL values, the number of transactions that keep the system busy decreases due to the chained waiting. As a result, the performance of 2PL decreases. The abort policy of WDL allows the processing of more transactions as compared to 2PL as an abort of a transaction causes the release of already held locks, which causes multiple waiting transactions to proceed in parallel. However, this increased

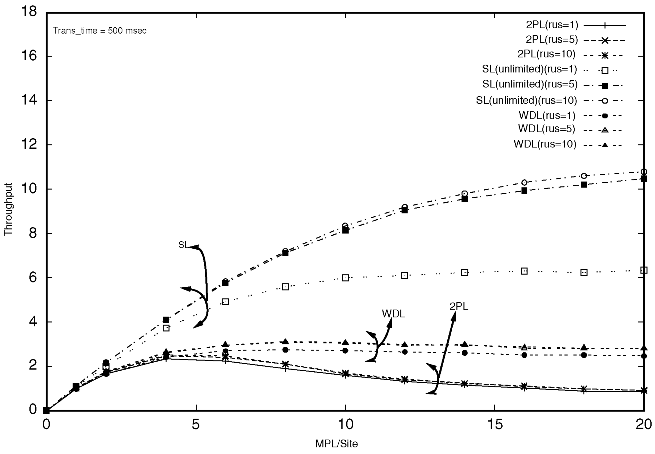

Fig. 8. MPL versus throughput at $trans\_time = 500$ msec.

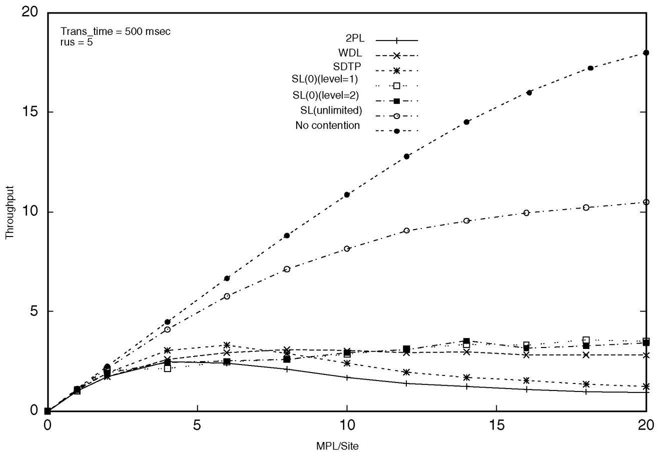

Fig. 9. MPL versus throughput: SL(0)(level=1), SL(0)(level=2), and SDTP.

parallelism is negated by the increased number of the aborts in WDL, since an abort is costly in DDBS. The performance of WDL saturates above 2PL. The performance of SL(unlimited) increases significantly due to the early release of locks.

Fig. 8 shows the performance results at $trans\_time = 500$ msec. In DDBSs, as $trans\_time$ increases, the transaction spends longer duration in execution and commit processing. In this figure, as we increase rus from 1 to 10, the performance of both 2PL and WDL is not increased accordingly. In case of 2PL, due to the long transmission time factor, transactions keep waiting for the data objects. So, the physical resources are underutilized due to the chained waiting. Therefore, the performance of 2PL saturates at $rus = 1$. WDL also fails to increase the performance due to a large number of aborts. As a result, the performance of WDL saturates at $rus = 1$. In case of SL(unlimited), even at $rus = 1$, the performance is increased considerably over WDL due to the reduced lock waiting time. As we increase rus from 1 to 5, the physical resource contention is further reduced. As a result, SL(unlimited) improves the performance significantly at the higher MPL values over other approaches. However, as $trans\_time$ increases, the performance of SL(unlimited) saturates at $rus = 5$. It can be observed that as we increase the rus value from 5 to 10, no significant improvement is observed in case of SL(unlimited).

Fig. 9 shows the performance results of both SL(0)(level=1) and SL(0)(level=2). At low MPL values, the performance of SL(0)(level=1) is approximately close to 2PL. The reason is that, even though SL(0)(level=1) reads after-images whenever a preceding transaction produces it, the increase in parallelism is negated by the cascading aborts. The main factor that affects the performance of SL(0) is cascading aborts. Also, it can be observed that the performance of SL(0)(level=2) is close to SL(0)(level=1). Owing to the fact that even though SL(0)(level=2) improves parallelism over SL(0)(level=1), the improvement is negated by the corresponding increase in cascading aborts. We have also conducted experiments at higher levels and found that performance of SL(0) further deteriorates with the number of levels. Fig. 9 also shows the performance under no contention. Under no contention, the performance grows linearly with MPL, and then saturates due to the physical resource contention. However, it can be observed that the performance of SL(unlimited) saturates at the lower throughput level much below the no contention line. This is due to the waiting involved in SL(unlimited) for the lock conversion and resolving commit dependencies.

Fig. 9 also shows the performance results of SDTP. At the lower MPL values, SDTP improves the performance over both 2PL and WDL. Because, at the lower MPL values, since the lock waiting time is less, the time required for the commit processing is comparable to the execution time. As data contention increases, transactions spend more time in the execution due to the increase in the lock waiting time. It can be observed that at higher MPL values, the performance improvement of SDTP over 2PL is not significant. Because, by releasing the locks after the execution, SDTP explores parallelism only during 2PC. However, as compared to 2PL, since the locks are being released before the completion of the commit processing, SDTP improves the throughput performance over 2PL. From the preceding experiments, it can be noted that SL(unlimited) improves performance significantly over 2PL, WDL, SL(0), and SDTP.

6.3 Simulation Results under the Limited Resource Environments

In this section, we discuss the experiment results about the impact of the main memory on the performance of SL(1), SL(2), and SL(n). We assume that a transaction (or its speculative) execution consumes one memory unit. In the following experiments, $mus/Site$ indicates the number of memory units available at that site, which are distributed equally among the active transactions at that site. So, we fix the EL (also VL) value equal to mus/MPL. When we fix EL for a transaction, we assume that appropriate processing power is added to the system to support speculative executions. So, in the simulation experiments we consider that speculative executions of a transaction are carried out in parallel. In the experiments, if a transaction conflicts with n transactions, the EL (also VL) value is fixed to 2^n, $n+1$, and $\sum n+1$ executions in case of SL(n), SL(1), and SL(2), respectively. The transaction is aborted if the number of speculative executions exceeds the EL value.

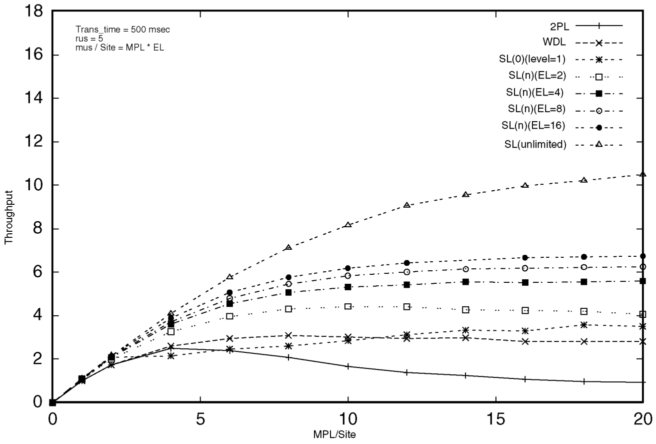

Fig. 10. MPL versus throughput: SL(n).

In the following experiments, we have not investigated the performance of SDTP, as the performance improvement is not significant under the limited resource environments.

Fig. 10 shows the performance results of SL(n) at $trans\_time = 500$ msec by varying MPL. It is assumed that, at a particular MPL/site value, mus equal to $(MPL/site) \times EL$ are available at that site. It can be observed that even at $EL = 2$, SL(n) exhibits improved performance over both SL(0)(level=1) and WDL. As we increase the EL value from 2 to 16, throughput of SL(n) increases significantly to wards SL(unlimited) due to the increased parallelism.

Fig. 11 shows the performance results of SL(1) at $trans\_time = 500$ msec by varying MPL. It can be observed that the performance of both SL(n) and SL(1) coincide at EL = 2. Also, the performance of SL(n) at EL = 8 coincides with the performance of SL(1) at EL = 4. This is due to the fact that at EL = 8, SL(n) increases concurrency up to 3-levels. The same performance can be achieved under SL(1) at EL = 4 because SL(1) explores more parallelism by carrying out few potential speculative executions for a transaction.

Fig. 12 shows the throughput results of SL(2) at EL = 16. This graph highlights the significant fact that SL(2) closely performs with SL(n). Approximately, same degree of concurrency is obtained under both approaches at EL = 16.

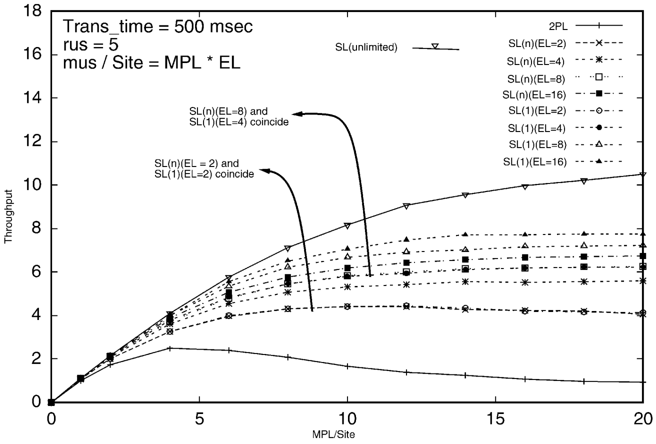

Fig. 11. MPL versus throughput: S(1).

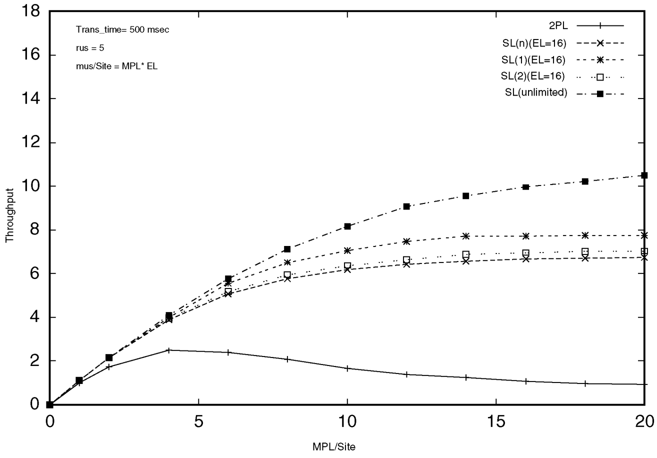

Fig. 12. MPL versus throughput: S(2).

However, due to reduced number of executions, SL(2) slightly improves the performance over SL(n). We have also carried out experiments by increasing the EL value and it was observed that SL(2) performs closely with SL(n).

At different $trans\_time$ values, Fig. 13 shows the performance results of SL(n). It can be observed that, as we increase EL, the performance keeps increasing. In DDBS, as $trans\_time$ increases, the transaction spends more time in the processing. As a result, the performance of all the approaches decreases (see the no contention line). However, the performance of SL(n) decreases slowly as compared to both WDL and SL(0)(level = 1). The reason is that as $trans\_time$ increases, the aborted transactions increase the wasted work in both WDL and SL(0)(level = 1). In case of 2PL, the throughput decreases due to the chained waiting. So as $trans\_time$ increases, SL(n) maintains high throughput as compared to in case of both SL(0)(level = 1) and WDL because of the increased concurrency. Also, as EL increases, SL(n) significantly improves the performance under all $trans\_time$ values over both SL(0)(level = 1) and WDL.

By varying $trans\_time$ values we have also conducted experiments for both SL(1) and SL(2) under the limited resource environments and observed that SL(1) improves the performance significantly over SL(2). Also, it was observed that the performance of SL(2) is close to SL(n).

Experiment on aborts. To study the impact of aborts, we conducted experiments by modeling a high abort situation. In this experiment, after completing the execution, a transaction is forcibly aborted with a certain probability. We conducted the experiment by fixing probability of transaction-abort at 5, 10, 15, 20, and 25 percent[2] In this experiment, we do not take into account the aborts caused by the nature of algorithm (such as the aborts due to deadlocks in case of 2PL, for instance).

Fig. 14 shows that as we increase the probability of abort, the performance of all the approaches decreases due to the wasted work. The performance of SL(0)(level=1) decreases at a faster rate due to the increase in cascading aborts. However, the performance of SL(1), SL(2), and SL(n)

[2] It has been reported [9] that in WAN environments, percentage of aborts vary up to 25 percent.

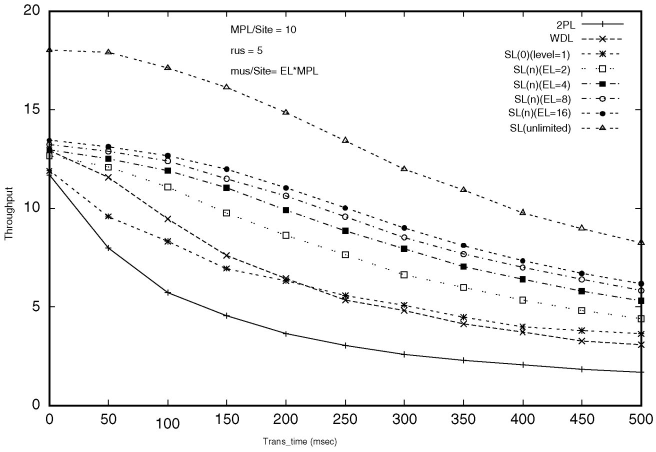

Fig. 13. $Trans\_time$ versus throughput: SL(n).

decreases much slowly as compared to SL(0)(level=1). It can be observed that under a wide range (up to 25 percent) of abort probabilities, SL(2) performs closely with SL(n). Also, as expected, performance crossover takes place between SL(1) and SL(2) as we increase the probability of abort. Note that the performance of SL(2) decreases gradually with the aborts as it is more robust against cascading aborts as compared to SL(1). In case of SL(2) also, a small number of cascading aborts do occur, but these have negligible effect on the performance. From this experiment, we conclude that SL(1) is appropriate for low abortive situations and SL(2) is appropriate for high abortive situations.

6.4 Simulation Results under the Skewed Environments

We report the experimental results conducted by considering a different access pattern and database size. It has been observed that data access pattern by transactions is nonuniform (skewed) [35] in real database applications. To model nonuniform access, some models have used $b-c$ access behavior [36], where $c = 100 - b$. It means that b% of transaction accesses are uniformly distributed over c% of the data objects and the remaining accesses are uniformly distributed over $(100 - c)\%$ of data. In the graphs, the improvement factor (IF) is also shown, which gives the throughput improvement over 2PL. For instance, IF of SL(n) is calculated as $\frac{SL(n)-2PL}{2PL}$ (these values indicate the corresponding throughput values).

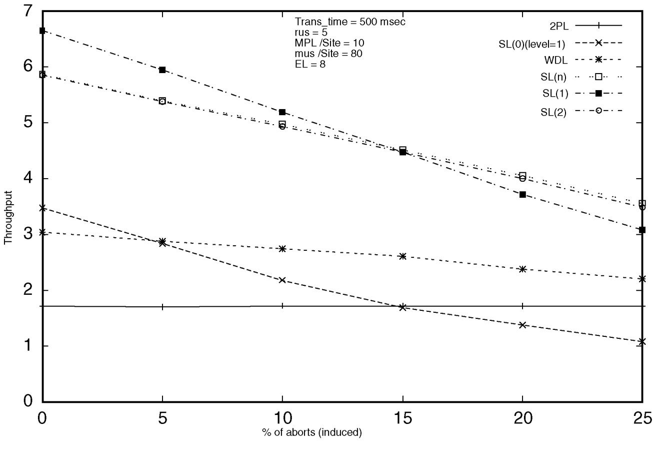

Fig. 14. Percentage of aborts versus throughput.

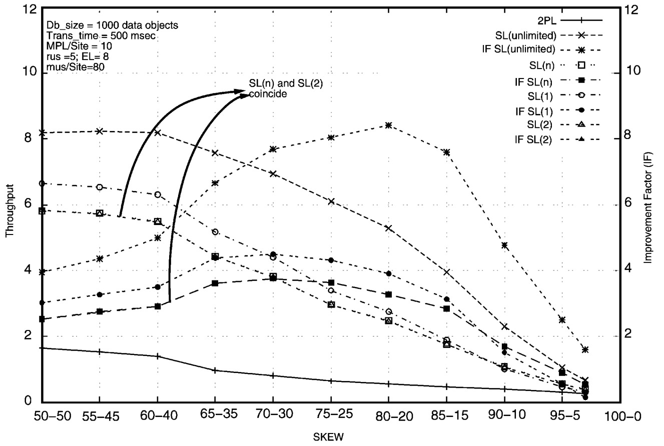

Fig. 15. Skew versus throughput, $db\_size = 1{,}000$.

Figs. 15 and 16 show the impact of skew on the performance when db_size = 1,000 and 10,000, respectively. When the skew is low, the SL protocols exhibit steady performance. As the skew is increased, the performance of all the approaches gradually decreases due to the increased contention. It can be observed that both SL(1) and SL(2) improve the performance significantly over 2PL under high skew. However, if we increase the skew beyond certain limit, the performance of the SL protocols falls sharply due to long waiting and finally settles above the 2PL's performance. The reason for such behavior can be explained as follows: In a database system, the average number of lock requests waiting for a data object is approximately equal to $((MPL * trans\_size)/(db\_size))$. So, the number of lock requests waiting per data object is inversely proportional to db_size. When skew = $50-50$, data accesses are uniform. As we increase the skew, the effective db_size value decreases accordingly. Due to the inverse law, the number of lock requests waiting for a data object increases sharply. As a result, the performance of both SL(1) and SL(2) falls sharply. The sharp fall of the performance of the SL protocols with skew can be more clearly noticed in Fig. 16.

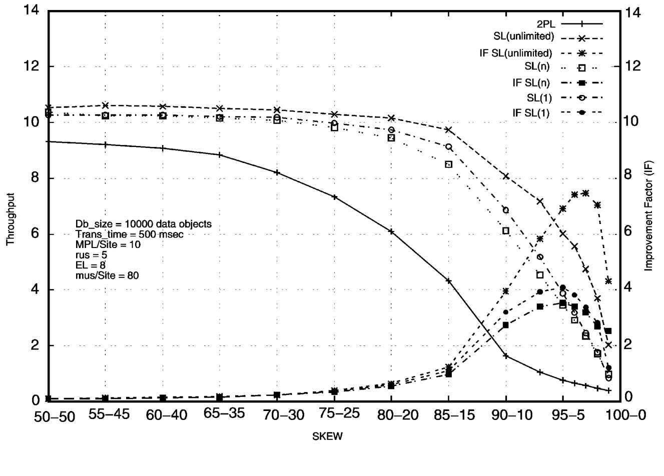

Fig. 16. Skew versus throughput, $db\_size = 10{,}000$.

From the preceding experiments, we can conclude that both SL(1) and SL(2) improve the performance significantly over 2PL under a wide variety of transaction workloads.

7 Performance and Implementation Issues

In this section, we discuss the performance and implementation issues concerning SL.

7.1 Performance Issues

Here, we discuss about disk I/O, logging, and message overhead issues under SL.

- **No additional disk I/O:** Under SL, even though the transaction produces multiple data object versions during its execution, it force-writes only one version for each data object. Therefore, SL involves no extra disk write cost.
- **No additional logging overhead:** Under SL, the logging process is delayed until the transaction confirms the single version. All the data object versions are kept in the main memory. However, due to the delay in logging process, the already completed work may be wasted in case of a failure. We believe that the effect of such failures on the performance is negligible.
- **Message overheads:** During execution, the after-images of a data object are sent to the object sites whenever the transaction produces them. The number of messages can be reduced in two ways. First, the after-images of the remote data objects piggy-back the PREPARE messages of 2PC. As a result, the size of the message increases, but this will have negligible effect on the performance. Second, after-images are transferred immediately only for the remote hot-spot data objects. Thus, with extra messages, performance can be improved by processing the waiting lock requests. The number of messages during the commit processing is not increased.

7.2 Implementation Issues

The architecture of a transaction processing system is discussed in [37]. Concurrency control is performed by the resource manager through a software module called *lock manager* that controls the accesses to the data objects using a locking approach. The lock requests are stored in a data object queue and are processed in a FIFO manner. As per the lock compatibility matrix, the locks are granted to the waiting lock requests. The recovery part is managed through the logging mechanism. In DDBS, after the completion of execution, 2PC is followed for commit processing. We now explain the possible modifications required to a typical transaction processing system to implement SL.

- **Precompiling:** To implement the locks under SL, the transaction processing system should know when the transaction completes the work with the data object. Therefore, before the starting of its execution, the transaction has to be scanned by a software module that puts a lock conversion marker for each data object. During execution, when the lock conversion marker is encountered, the EW-lock on the data object should be converted into the SPW-lock. Since the transactions are stored procedures, we believe that it is not difficult to put the lock conversion markers by analyzing the stored procedures.
- **Lock management:** The lock manager under 2PL manages two kinds of locks: R and W. However, the lock manager under SL should be modified to manage three kinds of locks: R, EW, and SPW. During the execution of a transaction, the request to change the EW-lock to the SPW-lock is sent to the lock manager whenever a lock conversion marker is encountered. The lock manager then changes the EW-lock on the corresponding data object to the SPW-lock. Also, whenever a the lock manager grants the EW-lock to the lock request, it maintains the corresponding *depend_set*. We believe that these changes can be easily incorporated.
- **Object tree and speculative execution management:** Two additional things are to be managed under SL over 2PL: object trees and speculative executions. Regarding the management of object trees, it can be noted that each active data object under the 2PL implementation is replaced by the corresponding tree under SL. So, the exiting transaction processing systems should be extended to manage the object trees under SL. And regarding the management of speculative executions, it can be noted that single execution of the transaction under 2PL is replaced by multiple executions under SL. Similar to the case of data object trees, the existing implementation should be extended to manage speculative executions. Especially, recently speculative execution mechanism on multithreaded processor attracts a strong attention and a lot of research on this topics are already being done. We believe that the SL protocols effectively utilize such environments and improve the performance.
- **Index modification:** Under 2PL, when the transaction requests a data object value, its existence is first checked in the main memory (database buffer). Otherwise, it is retrieved from the disk using B-tree index [37]. Concurrency control protocols such as ARIES/IM [38] are employed to preserve the consistency of the B-tree against concurrent operations. The existing B-tree index can be adopted to access the data objects under SL as follows: Under SL, an active data object is represented by a tree which contains the uncommitted versions produced by the transactions. It can be observed that, under SL, the transaction accesses either all the versions in the tree or not. Logically this is equal to accessing the single version. So, the change to the existing index implementations is minimal.

It can be noted that current database systems also provide options to create and maintain secondary and value based indexes to improve the query performance. The detailed analysis how such indexes can be extended under SL is beyond the scope of this work.

- **Commit processing:** Under SL, there exists two options to implement 2PC. Firstly, 2PC is started after the transaction's execution without waiting for the termination of the transactions in its *depend_set*. When a participant receives a PREPARE message, it will respond only after the termination of all the transactions in the *depend_set*s of the corresponding transaction's lock requests. To prevent unnecessary aborts, the time-out period in existing 2PC implementations should be dynamically set based on the number of transactions in the *depend_set* of the transaction. Second, the transaction starts 2PC only after the termination of all the transactions in the *depend_set*. The waiting period can be fixed based on the number of transactions in the *depend_set*. In this option, when the transaction terminates, all those transactions that have formed the commit dependency with the aborted transaction have to be informed which increases the communication cost. With this option, however, the existing 2PC implementation can be adopted under 2PL without modifications.

8 Summary and Conclusions

In this paper, we have proposed the SL protocols to improve the performance of DDBSs by trading extra processing resources. In SL, the waiting transaction is allowed to access the locked data object whenever the lock-holding transaction produces the corresponding after-image during execution. By exploiting the fact that a submitted transaction is more likely to commit than abort, we proposed SL variants, SL(1) and SL(2), that process transactions efficiently by significantly reducing the number of speculative executions. Our simulation study under a wide variety of transaction workloads indicate that even with manageable extra resources both SL(1) and SL(2) significantly improve the performance over 2PL in the DDBS environments where a transaction spends longer time in processing and transaction-aborts occur frequently.

As a part of future work, we will investigate the issues regarding modifications required to the existing transaction processing systems to implement SL. These issues include the design of a precompiler routine to put the lock conversion markers in the transaction code, efficient management of object trees and speculative executions, and changes to existing indexes and commit processing routines. We will also investigate efficient strategies to distribute extra processing resources among the active transactions by considering the transaction-specific properties such as priority and length.

Acknowledgments

This work is supported by "Research for the Future" (in Japanese Mirai Kaitaku) under the program of the Japan Society for the Promotion of Science, Japan. The authors are thankful to the anonymous referees for their useful comments. This work was done while P. Krishna Reddy was at Institute of Industrial Science, University of Tokyo.

References

[1] K.R. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Comm. ACM*, vol. 19, no. 11, pp. 624-633, 1976.

[2] J.N. Gray, "Notes on Database Operating Systems: In Operating Systems an Advanced Course," *Lecture Notes in Computer Science*, vol. 60, pp. 393-481, 1978.

[3] C. Mohan, B. Lindsay, and R. Obermark, "Transaction Management in the R^* Distributed Database Management System," *ACM Trans. Database Systems*, vol. 11, no. 4, pp. 378-396, 1986.

[4] A. Bestavros, "Speculative Algorithms for Concurrency Control in Responsive Databases," *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems*, pp. 143-165, Kluwer Academic, 1994.

[5] A. Bestavros and S. Braoudakis, "Value-Cognizant Speculative Concurrency Control," *Proc. 21th Very Large Databases Conf.*, pp. 122-133, 1995.

[6] B.W. Abeysundara and A.E. Kamal, "High-Speed Local Area Networks and their Performance," *ACM Computing Surveys*, vol. 23, no. 2, pp. 221-264, June 1991.

[7] R. Golding and D.E. Long, "Accessing Replicated Data in an Internetwork," *Int'l J. Computer Simulation*, vol. 1, no. 4, pp. 347-372, Dec. 1991.

[8] P. Spiro, A. Joshi, and T.K. Rangarajan, "Designing an Optimized Transaction Commit Protocol," *Digital Technical J.*, vol. 3, no. 1, pp. 23-35, 1991.

[9] B. Bhargava, Y. Zhang, and S. Goel, "A Study of Distributed Transaction Processing in an Internetwork," *Lecture Notes in Computer Science*, vol. 1006, pp. 135-152, 1995.

[10] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[11] V. Kumar, *Concurrency Control Mechanisms in Centralized Database Systems.* Prentice-Hall, 1996.

[12] K.-L. Wu, P.S. Yu, and C. Pu, "Divergence Control for Epsilon-Serializability," *Proc. Eighth Int'l Conf. Data Eng.*, pp. 506-515, 1992.

[13] P.A. Bernstein and N. Goodman, "Multi-Version Concurrency Control, Theory and Algorithms," *ACM Trans. Database Systems*, vol. 8, no. 4, pp. 465-483, Dec. 1983.

[14] B.R. Badrinath and K. Ramamritam, "Semantics Based Concurrency Control: Beyond Commutativity," *ACM Trans. Database Systems*, vol. 17, no. 1, pp. 163-199, Mar. 1992.

[15] D. Agrawal, A. El Abbadi, and A.E. Lang, "The Performance of Protocols Based on Locks with Ordered Sharing," *IEEE Trans. Knowledge and Data Eng.*, vol. 6, no. 5, pp. 805-818, Oct. 1994.

[16] K. Salem, H. Garciamolina, and J. Shands, "Altruistic Locking," *ACM Trans. Database Systems*, vol. 19, no. 1, pp. 117-165, Mar. 1994.

[17] R. Gupta, J. Haritsa, and K. Ramamritam, "Revisiting Commit Processing in Distributed Database Systems," *Proc. ACM SIGMOD*, pp. 486-497, 1997.

[18] H.V. Jagadish and O. Shmueli, "A Proclamation-Based Model for Cooperation Transactions," *Proc. 18th Very Large Databases Conf.*, pp. 265-276, 1992.

[19] A. Burger and P. Thanisch, "Branching Transactions: A Transaction Model for Parallel Database Systems," *Lecture Notes in Computer Science*, vol. 826, pp. 121-136, 1994.

[20] H.T. Kung and J.T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. Database Systems*, vol. 6, no. 2, pp. 213-226, June 1981.

[21] P. Krishna Reddy and M. Kitsuregawa, "Improving Performance in Distributed Database Systems using Speculative Transaction Processing," *Proc. Second European Parallel and Distributed Systems Conf.*, pp. 275-285, 1998.

[22] P. Krishna Reddy and M. Kitsuregawa, "Speculative Lock Management to Increase Concurrency in Mobile Environments," *Proc. First Int'l Conf. Mobile Data Access*, vol. 1748, pp. 81-95, Dec. 1999.

[23] P. Krishna Reddy and M. Kitsuregawa, "Speculation Based nested Locking Protocol to Increase the Concurrency of Nested Transactions," *Proc. Int'l Database Applications and Eng. Symp.*, pp. 296-305, Sept. 2000.

[24] D. Skeen, "Nonblocking Commit Protocols," *Proc. ACM SIGMOD*, pp. 133-142, June 1981.

[25] A.K. Elmagarmid, "A Survey of Distributed Deadlock Detection Algorithms," *ACM SIGMOD RECORDS*, vol. 15, no. 3, pp. 37-45, Sept. 1986.

[26] E. Knapp, "Deadlock Detection in Distributed Databases," *ACM Computing Surveys*, vol. 19, no. 4, pp. 303-328, Dec. 1987.

[27] *The Benchmark Handbook for Database and Transaction Processing Systems.* J. Gray, ed. Morgan Kaufmann, 1991.

[28] *TPC: Transaction Processing Performance Council,* http://www.tpc.org/, June 2001.

[29] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez, "Transaction Chopping: Algorithms and Performance Studies," *ACM Trans. Database Systems,* vol. 20, no. 3, pp. 325-363, Sept. 1995.

[30] R. Agrawal, M.J. Carey, and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Trans. Database Systems,* vol. 12, no. 4, pp. 609-654, Dec. 1987.

[31] A.N. Choudhary, "Cost of Distributed Deadlock Detection: A Performance Study," *Proc. IEEE Conf. Data Eng.,* pp. 174-181, 1990.

[32] Y.C. Tay, *Locking Performance in Centralized Databases.* Academic Press, 1987.

[33] M. Carey and M. Linvy, "Parallelism and Concurrency Control Performance in Distributed Database Machines," *Proc. ACM SIGMOD,* pp. 122-133, June 1989.

[34] P.A. Franaszek, J.T. Robinson, and A. Thomasian, "Concurrency Control for High Contention Environments," *ACM Trans. Database Systems,* vol. 17, no. 2, pp. 304-345, June 1992.

[35] V. Singhal and A.J. Smith, "Analysis of Locking Behavior in Three Real Database Systems," *The Very Large Databases J.* vol. 6, pp. 40-52, 1997.

[36] A. Dan, D.M. Dias, and P.S. Yu, "Buffer Analysis for a Data Sharing Environment with Skewed Data Access," *IEEE Trans. Knowledge and Data Eng.,* vol. 6, no. 2, pp. 331-337, 1994.

[37] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[38] C. Mohan and F. Levine, "ARIES/IM: An Efficient and High Concurrency Index Management using Write-Ahead Logging," *Proc. ACM SIGMOD,* pp. 371-381, 1992.

P. Krishna Reddy received his graduation in electronics engineering in 1988. He received the MTech and PhD degrees from the School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi, in 1991 and 1994, respectively. From 1994 to 1996, he was on the faculty at the Division of Computer Engineering, Netaji Subhas (formerly Delhi) Institute of Technology, Delhi. From 1996 to 1997, he was on the faculty at the Department of Computer Science, Philadelphia University, Jordan. From 1997 to 2002, he was a research associate at the Center for Conceptual Information Processing Research, Institute of Industrial Science, University of Tokyo. He is currently an associate professor at the International Institute of Information Technology (IIIT), Hyderabad, India. His research interests include transaction management, distributed computing systems, information systems, performance evaluation, Internet data management, data mining, and Web mining. He has published more than 25 refereed papers. He is a member of the IEEE Computer Society, ACM, and Computer Society of India.

Masaru Kitsuregawa received the PhD degreefrom University of Tokyo in 1983. He is currently a full professor and a director of the center for conceptual information processing research at the Institute of Industrial Science, the University of Tokyo in Japan. His current research interests cover database engineering, Web mining, parallel computer architecture, parallel database processing/data mining, storage system architecture, digital earth, transaction processing, etc. He is/was an associate editor of several international journals, such as the *Very Large Databases Journal*, *DAPD (Distributed and Parallel Database) Journal*, *New Generation Computing Journal*, and *IEEE Transactions on Knowledge and Data Engineering*. He served as cochair of IEEE International Conference on Data Engineering 1999, and served as cogeneral chair of PAKDD 2000 and WAIM 2002. He is appointed to serve as general chair of ICDE 2005 at Tokyo. He is currently also an officer of several professional society groups such as an Asian Coordinator of the IEEE Technical Committee on Data Engineering, a member of steering committee of PAKDD, and WAIM, and had been a trustee of the Very Large Databases Endowment. In Japan, he was the chair of data engineering special interest group of Institute of Electronic, Information, Communication, Engineering, Japan (IEICE) and also served the editor of transaction of IEICE. He is currently serving ACM SIGMOD Japan Chapter Chair and a director of the Information Processing Society Japan and a director of the Data Base Society of Japan. He is a member of the IEEE and the IEEE Computer Society.