Speculative Distributed Transaction Processing

P. Krishna Reddy and Masaru Kitsuregawa
Institute of Industrial Science
The University of Tokyo
7-22-1, Roppongi, Minato-ku
Tokyo 106, Japan
{reddy, kitsure}@tkl.iis.u-tokyo.ac.jp

Abstract

In this paper, we propose speculative distributed transaction processing (SDTP) strategy, in which, a transaction releases the locks on the data objects immediately after the completion of its execution and starts commit processing. By accessing both original values and updated values immediately after the transaction's execution, the waiting transaction speculatively carries out alternative executions, and starts commit processing. Before the end of commit processing, the transaction that has carried out speculative executions retains the appropriate execution based on the termination decisions of preceding transactions. Using SDTP approach conflicting transactions can be processed in parallel without violating the serializability criteria. This approach is free from cascading aborts. The speculative transaction processing approach does not require extra number of messages since every message is piggy backed with the messages of commit processing. However, it needs both extra processing power and main memory to support speculative executions. The proposed approach can be tuned according to the resources available in the system. Through simulation experiments, it has been shown that the proposed approach considerbly reduces the response time and increases the throughput in the case of higher resource conflicts and the longer transmission times.

1 Introduction

In distributed database systems (DDBSs), the processing of a transaction can be separated into two stages: the execution stage and the commit stage. During the execution stage, a transaction accesses the data values from various sites by following a concurrency control algorithm such as two-phase locking [6, 7] and updates these data values in a temporary storage of arrival site. During the commit stage, these updated data values are written in the stable storage at participant sites in an atomic manner. To commit a transaction, two-phase commit (2PC) [7] and three-phase commit (3PC) protocols [13] are proposed for DDBSs. The general aim of a transaction processing (concurrency control and commit) approach is to ensure the consistency of the database and the correct completion of each transaction initiated in the system. An obvious additional requirement is to improve the performance of a database system by minimizing the average response time. In order to minimize the response time, it is necessary to maximize parallelism of transactions' execution. The major factor in parallelism is the period of time for which the data values become unavailable to the waiting transactions during the commit processing of conflicting transactions. In the DDBSs, especially in the wide area network (WAN) environments, the time required to commit a transaction has a significant share as compared to its execution.Thus, the time spent by a transaction during its commit processing becomes a major factor in degrading the performance of the DDBSs. In this paper, we have proposed a speculative distributed transaction processing (SDTP) approach, in which, by accessing the data object values, which were read and written by the conflicting transaction immediately after its execution, the waiting transaction speculatively carries out alternative executions, and starts commit processing. Before the end of commit processing, the transaction that has carried out speculative executions, retains the appropriate execution based on the termination decisions of preceding transactions. Using SDTP approach, conflicting transactions can be processed in parallel without violating the serializability criteria. This approach is free from cascading aborts. The speculative transaction processing approach does not require extra number of messages since every message is piggy backed with the messages of commit processing. However, it needs both extra processing power and main memory to support speculative executions. By tuning the proposed approach according to the available resources in the system, considerable performance improvement can be achieved.

Because of its critical performance issue, the transaction processing problem has been well studied in the context of both centralized and DDBSs [4]. Recently commit processing has been attracted attention due to its effect on the performance on the transaction processing. It has been shown in [12] that the time to commit accounts for one third of transaction duration in a general purpose database. In [3] experimental studies have been reported on behavior of concurrency control and commit algorithms in the WAN environments. It has been shown that the time to commit can be as high as 80 per cent of the transaction time in the WAN (Internet) environments. A concurrency control approach for real time environment by employing redundant computations is proposed in [1]. In

[11], a modified optimistic protocol (OPT) is proposed, in which a transaction carries out single execution by reading the uncommitted values produced by the conflicting transaction, up to one level. A failure of a transaction may result in aborting of another transaction which has read uncommitted values of the failed transaction. The OPT is proposed for local area network based environments in which the probability of failure during the commit processing of a transaction is less as compared to the WAN environments.

The body of the paper is organized as follows. In the next section, we explain the 2PC protocol and define a system model. In the section 3 we explain the the basic idea of speculative transaction processing without involving the commit processing. In section 4, we explain the lock compatibility matrix, and present the SDTP approach. In section 5 we present the results of the performance evaluation for both unlimited as well as limited resources environments. The last section finally consists of summary and conclusions.

2 The 2PC protocol and system model

2.1 Two-phase commit protocol

In DDBSs, the 2PC protocol extends the affects of local atomic commit actions to distributed transactions by insisting that all sites involved in the execution of a distributed transaction agree to commit the transaction before its effects are made permanent. A brief description of the 2PC protocol that does not consider failures is as follows. Initially, the coordinator (originating site of a transaction) writes a **begin-commit** record in the log, sends $PREPARE$ messages to all participating sites, and enters the *wait* state. When a participant receives $PREPARE$ message, it checks if it can commit the transaction. If so, the participant writes a **ready** record in the log, sends $VOTE_COMMIT$ message to the coordinator, and enters the *ready* state. Otherwise, the participant writes an **abort** record and sends a $VOTE_ABORT$ message to the coordinator. If the decision of the site is to abort, it can forget about that transaction. The coordinator aborts the transaction globally, even if the coordinator receives $VOTE_ABORT$ message from one participant. Then, it writes an **abort** record, sends a $GLOBAL_ABORT$ message to all participant sites, and enters the *abort* state; Otherwise, it writes a **commit** record, sends a $GLOBAL_COMMIT$ message to all participants, and enters the *commit* state. The participants either commit or abort the transaction according to the coordinators' instructions and sends back ACK (acknowledgment) message at which point the coordinator terminates the transaction by writing an **end-of-transaction** record in the log.

2.2 System model

The DDBS consists of a set of data objects. A data object is the smallest accessible unit of data. In this paper, $X, Y, \ldots, Z$ represent data objects. Each data object is stored at one site only. 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 data objects are stored at database sites connected by a computer network.

We define a transaction as a set of atomic operations on data objects. An operation is either read (returns the value of the data object), or a write (updates the object with a specified new value). For any T_i and data object 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. 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. A transaction that has been initiated but has not yet committed or aborted is said to be active. 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. The union of the RS and the WS of T_i constitute locking variables. The transactions T_i and T_j are said to have R(read)-W(write), W-R or W-W conflict, if $RS(T_i) \cap WS(T_j) \neq \phi$; $WS(T_i) \cap RS(T_j) \neq \phi$; or $WS(T_i) \cap WS(T_j) \neq \phi$, respectively. Also, T_i and T_j are said to be in conflict, if at least one of the above conflicts exists.

It is assumed that two-phase locking algorithm is employed for concurrency control and the 2PC protocol with centralized communication paradigm is employed for commit processing. Also, deadlock detection and removal [8] is carried out whenever a deadlock occurs in the system.

Figure 1: Processing of a T_i in the DDBSs

The processing of a T_i, in the DDBSs is depicted in Figure 1. In this Figure, s_i denotes the starting of execution and e_i denotes the completion of execution. The data values produced at e_i are termed as uncommitted (new) values. The completion of first and second phases of 2PC protocol are denoted by f_i and c_i, respectively.

In the proposed approach, a transaction starts its processing in the normal mode. During its execution, if a transaction reads at least one uncommitted value produced by another transaction, it enters into the speculative mode. In the speculative mode, a number of speculative executions are carried out on behalf of a T_i. We use notation T_{ij} or (i,j) to represent the jth ($j \geq 1$) execution of the T_i. The notation T_{i1} is used to represent the initial execution of T_i. The RS and WS of the T_{ij} are represented by RS_{ij} and WS_{ij} respectively. A data object may have more than one uncommitted versions. The notation X_q, ($q \geq 1$), is used to represent q^{th} version of X. The notation X_1 represents the initial value of X, when no transaction is accessing it.

3 Speculative transaction processing

In this section at first, we explain about the version tree of the data object, which is used to organize the uncommitted versions produced by speculative executions. In the section 3.2 we explain the basic idea of the speculative transaction processing without involving commit processing. In the section 3.3, we present the execution complexity of the speculative transaction processing.

3.1 Version tree of the data object

In the SDTP approach, we employ the tree data structure to organize the uncommitted versions of a data object produced by speculative executions. For a data object X, its version tree is denoted by $tree_X$. It is a tree with X_1 (committed version) as a root and its uncommitted versions as rest of the nodes. Initially, the data object X is represented as a tree with X_1 as the root node. When a transaction reads X, it accesses its $tree_X$. It is updated when a transaction that accesses X completes the execution. Consider that the T_i carries out 'n' $(T_{ik}, k = 1 \ldots n)$ number of executions, by accessing the set of objects, say $objects_i$. Let, $X \in objects_i$. When the T_i completes the execution, the trees of the elements of $objects_i$ are updated as follows. For each $X \in objects_i$, each new version of $X \in WS_{ik}$, $k = 1 \ldots n$, is included as a child to the corresponding read version node of $tree_X$.

3.2 Speculative transaction processing approach

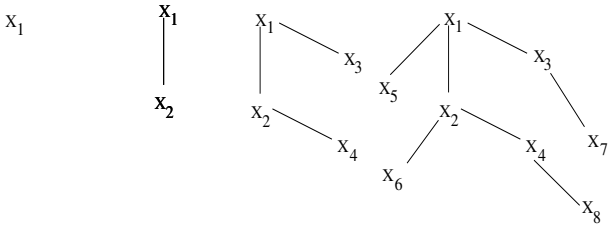
At first, we explain speculative transaction processing approach by considering simple case where multiple transactions access single data object. Next, we explain generalized case where a transaction accesses multiple data objects.

3.2.1 Simple case

Consider that the data object X is stored at S_1 and both T_1 and T_2 are arrived simultaneously into the system to update it. At first, the T_1 reads X_1 and produces the uncommitted value, X_2, and proceeds for commit processing. In order to explain the approach it is assumed that a long time is required to commit a transaction as compared to its execution. The processing of transactions using speculation as follows.

We employ nested parentheses notation to represent $tree_X$. Initially, $tree_X$ is (X_1). Consider that, the T_1 starts first execution, the T_{11} with $RS_{11} = \{X_1\}$. It completes the execution with $WS_{11} = \{X_2\}$. At the end of its execution, the X_2 is included in the $tree_X$ as $(X_1(X_2))$ as shown in the Figure 2(b). The T_1 proceeds with its commit processing. In the SDTP approach, the T_2 is allowed to access the $tree_X$ $(X_1(X_2))$. The T_2 carries out two speculative executions: one is T_{21} with $RS_{21} = \{X_1\}$ and the other is T_{22}, with $RS_{22} = \{X_2\}$. It completes the speculative executions, with $WS_{21} = \{X_3\}$, and $WS_{22} = \{X_4\}$ respectively. When the T_2 completes the execution, the $tree_X$ is updated by including the new versions X_3 and X_4 as $(X_1(X_2(X_4)), X_3)$ as shown in the Figure 2(c). If the T_1 terminates with successful commit, the execution T_{22} is retained. Otherwise, the T_{21} is retained.

Now consider that the T_3 enters into the system to access X. By reading the $tree_X$, the T_3 carries out four speculative executions. These are T_{31}, T_{32}, T_{33}, and T_{34} with $RS_{31} = \{X_1\}$, $RS_{32} = \{X_2\}$, $RS_{33} = \{X_3\}$, and $RS_{34} = \{X_4\}$, respectively. The T_3 completes the speculative executions, with $WS_{31} = \{X_5\}$, $WS_{32} = \{X_6\}$,$WS_{33} = \{X_7\}$, and $WS_{34} = \{X_8\}$, respectively. When, the T_3 completes the execution, the $tree_X$ is updated by including the new versions X_4, X_5, X_6, and X_7 as $(X_1(X_2(X_4(X_8)), X_6), X_3(X_7), X_5)$ as shown in the Figure 2(d). Before the end of its commit processing, the T_3 retains the appropriate execution depending on the commit/abort decisions of both T_1 and T_2.



(a)Initial state (b) After T_1's execution (c) After T_2's execution (d) After T_3's execution

Figure 2: State of $tree_X$

When a transaction terminates, the trees of corresponding data objects are updated. For example, if the T_1 commits, the $tree_X$ is updated with X_2 as the root. Otherwise, if the T_1 aborts, the subtree of X_2 (including the node X_2) is deleted from the $tree_X$.

3.2.2 Generalized case

Now, we explain the case when a transaction accesses multiple data objects. Before accessing any object, a transaction may be executing either in the normal mode or in the speculative mode.

A normal mode transaction has only one execution. When it accesses the data object X having 'n' versions in its $tree_X$, it carries out 'n' speculative executions and enters into speculative mode. Consider that T_i is executing in speculative mode with 'm' executions. Next, it accesses the data object X. Then, there exist two cases: the $tree_X$ contains only one version and the $tree_X$ contains 'n' versions. If the $tree_X$ contains only one version, each one of the $T_i's$ speculative executions accesses $tree_X$ and carries out its execution. Otherwise, if the $tree_X$ contains 'n' versions, the process is extended to all versions of the $tree_X$. That is, for each T_{iq}, (q=1 ... m), 'n' speculative executions are carried out (one for each version of $tree_X$).

3.3 Complexity of speculative transaction processing

In the SDTP approach, the number of speculative executions produced by T_i depends on the number of objects it accessed and the number of uncommitted versions in the tree of each object.

Theorem 1: Consider that T_i has completed execution by accessing 'm' data objects. Let, v_k be the number of uncommitted versions in the tree of k^{th} $(1 \leq k \leq m)$ data object and N_i be the total number of executions produced by T_i. Then, $N_i = \prod_{k=1}^{m} v_k$.
Proof: At first, the T_i accesses the first data object having v_1 nodes in its tree and carries out v_1 executions. When it accesses the second object having v_2 nodes in its tree, each one of the v_1 executions carries out v_2 executions. Following this, after accessing all 'm' objects, the N_i becomes : $\prod_{k=1}^{m} v_k$.

4 Speculative distributed transaction processing

In this section at first we explain the lock compatibility matrix employed in the SDTP approach. In Section 4.2, we briefly present the SDTP approach. In Section 4.3. we explain the processing of transactions employing SDTP approach. In Section 4.4 we present the ways to tune the SDTP approach according to the available resources of the system.

4.1 Lock compatibility matrix

The lock compatibility matrix in the locking based approach is shown in the Figure 3(a).The S and X denotes shared (read) and exclusive (read-write) locks respectively. In the Figure, *yes* indicates that the corresponding locks are compatible and *no* indicates that the corresponding lock requests are not compatible.

Lock requested by T_j	**Lock held by T_i**	
	S	**X**
S	yes	no
X	no	no

(a)

Lock requested by T_j	**Lock held by T_i**			
	ES	**CS**	**EX**	**CX**
ES	yes	yes	no	yes
EX	no	yes	no	yes

(b)

Figure 3. Lock compatibility matrix (a) Locking (b) SDTP approach

In the SDTP approach, the S lock is partitioned into the execution-shared (ES) lock and the commit-shared (CS) lock. Also, the X lock is partitioned into the execution-exclusive (EX) lock and the commit-exclusive (CX) lock. The transactions request only ES and EX locks. On completion of a transaction's execution, after including the new data values in the respective trees of data objects, the ES/EX locks are changed to CS/CX locks. The lock compatibility matrix in the SDTP approach is shown in the Figure 3(b).

In the SDTP approach, multiple transactions can have CS/ CX locks on a data object at the same time. To ensure consistency, the SDTP approach follows the below rules.

1. If the T_i obtains the EX lock, while T_j is holding either the CS lock or the CX lock on a data object, the T_i is committed only after the termination of the T_j.

2. If the T_i obtains the ES lock, while T_j is holding the CX lock on a data object, the T_i is committed only after the termination of the T_j.

4.2 Description of SDTP approach

For each data object X, $tree_X$ is maintained. Initially, the transaction obtains ES/EX locks on the required data objects. On completion of the transaction's execution, after including the new values to the corresponding tree, the ES/EX locks are converted to CS/CX locks. (It can be noted that in DDBSs, if the data object resides at home site (arrival site of the transaction), the locks are converted on completion of transaction's execution. Otherwise, the locks are converted on receiving the PREPARE message of the 2PC protocol.) Next, it starts commit processing. The waiting transaction accesses the trees of required data objects and carries out speculative executions. It also, in turn, converts the ES/EX locks into CS/CX locks and starts commit processing. Before the end of the commit processing the transaction which has carried out speculative executions confirms the appropriate execution after getting the terminating decisions of preceding transactions. Also, on termination of a transaction, the trees of the locked data objects are updated.

4.3 Example using SDTP approach

We explain the processing of transactions in DDBSs by considering an example. Consider two transactions T_1 and T_2 that operate in a bank environment. Assume that the data objects (accounts) X, Y, and Z are stored at sites S_1, S_2 and S_3 respectively. Consider that the transactions T_1 and T_2 arrived at S_2 simultaneously to update X,Y and Y,Z respectively. They issue their lock requests to the concerned objects. Consider that, at first, the T_1 obtains locks on both X and Y. The T_2 obtains a lock on Z, but waits for a lock on Y, due to a conflict with T_1. Normally, the processing proceeds as follows. The T_1 completes the execution by producing the uncommitted data values of both X and Y. After the completion of 2PC processing, the T_1 releases the locks on both X and Y. Then, T_2 obtains the lock on Y. The execution is depicted in Figure 4(a).

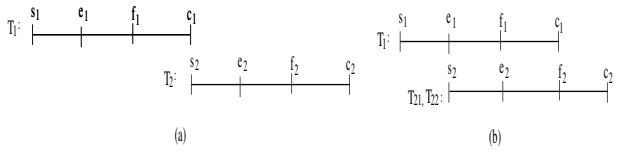


Figure 4. Processing: (a) without SDTP (b) with SDTP

Now, we explain the processing employing SDTP approach. Initially, $tree_X = (X_1)$, $tree_Y = (Y_1)$ and $tree_Z = (Z_1)$. At first, the T_1 obtains EX locks on both X and Y. It completes the execution, T_{11} with $RS_{11} = \{X_1, Y_1\}$ and $WS_{11} = \{X_2, Y_2\}$. On completion of T_{11}, after updating the $tree_X$ and the $tree_Y$ as $(X_1(X_2))$ and $(Y_1(Y_2))$ respectively, the EX lock on both X and Y are changed to CX lock. (It can be noted that the $tree_Y$ can be updated immediately on completion of T_1's execution as Y resides at T_1's home site. However, the $tree_X$ is updated on receiving the PREPARE message of 2PC protocol behalf of T_1.) Next, the T_2 starts two executions by taking the Y_1 and the Y_2 as inputs at e_1. The processing depicted in Figure 4(b). The first execution of T_2 is T_{21} with $RS_{21} = \{Y_1, Z_1\}$ and $WS_{21} = \{Y_3, Z_2\}$ and the second execution of T_2 is T_{22} with $RS_{22} = \{Y_2, Z_1\}$ and $WS_{22} = \{Y_4, Z_3\}$. After completion of execution, the $tree_Y$ and $tree_Z$ are updated as $tree_Y = (Y_1(Y_2, (Y_4)), Y_3)$ and $tree_Z = (Z_1(Z_2, Z_3))$ respectively.

Next,without waiting for the completion of 2PC of T_1, the T_2 proceeds with the first phase of 2PC by sending the effect (RS and WS) of both T_{21} and T_{22}, to the participating sites as a part of $PREPARE$ message. When a participant site receives $PREPARE$ message, it checks if it can commit a transaction. If so, after writing the $PREPARE$ message in the stable storage, it sends $VOTE_COMMIT$ message to S_2. Otherwise, it sends $VOTE_ABORT$ message to S_2. If all the participant sites respond with $VOTE_COMMIT$ messages, the S_2 takes the decision to retain one of the two executions as follows. After the completion of first phase of 2PC of T_2, the coordinator checks for ABORT/COMMIT decision of T_1. If T_1 commits, the coordinator of T_2 takes the decision to retain the execution T_{21} as it follows the serial order $< T_1, T_2 >$. In case T_1 aborts, then the coordinator takes the decision to retain the execution T_{22} as it follows the serial order $< T_2, T_1 >$ (This is under the assumption that T_1 is resubmitted after its abort and that no other transaction enters into the system.) Next, the coordinator sends this decision as a part of the GLOBAL_COMMIT message to all participants. In this way, the T_1 and T_2 can be processed in parallel without violating the serializability criteria.

4.4 Tuning of SDTP approach

It can be observed that, the SDTP approach does not require extra number of messages as every message is sent with the messages of 2PC protocol. But, it needs extra processing power and the main memory to carry out speculative executions. As the current technology provides high speed parallel computers at low cost, the processing cost may not be considered as a considerable overhead. However, the size of main memory available in the system can put the limitation on the number of speculative executions. Because, in the SDTP approach, when the data contention increases, the number of speculative executions that a transaction has to carry out increases exponentially. However, the SDTP algorithm can be tuned in accordance with the amount of main memory available in the system. Consider that the amount of memory to carry out out single execution is one unit. Then, the N denotes the number of executions that can be executed by the system in parallel. To model the limited resources situation, we introduce two variables:*versions_limit* and *executions_limit*.

For any data object, the *versions_limit* variable limits the maximum number of versions allowed in its tree. If number of versions in the tree of a data object is more than the *versions_limit*, the lock request is put to wait. That is, the lock request waits for the termination of earlier transaction. On termination of the earlier transaction (which updates the tree) if the number of nodes falls within the *versions_limit*, the waiting lock request obtains the lock.

The *executions_limit* variable limits the maximum number of speculative executions that can be carried out by the system at any time.This can be decided based on the amount of the main memory available in the system. After getting the responses to the lock requests, if the number of executions crosses the *executions_limit*, the transaction is rejected. The rejected transaction is resubmitted immediately.

5 Performance study

5.1 Simulation settings

The meaning of each model parameter for simulation is given in Table 1. The communication network is simply modeled as a fully-connected network. In the simulation we employ static two phase locking approach, in which the transaction starts the execution only after receiving the locks on all the required data objects. The 2PC protocol is employed for the commit processing. The setting of res_io, res_cpu, db_size, trans_size, min_size, and max_size values are given in the Table 1 [2]. Accessing a data object requires res_cpu and res_io. Also, each message of the 2PC requires res_cpu and res_io both while sending and receiving. The local_to_total ratio for a transaction is fixed at 0.6. Thus, 60% of the data objects are randomly chosen from the local database and the 40% of the data objects are randomly chosen from the remaining database sites. A transaction writes all the data objects it reads. With these settings, by varying MPL values, sufficient variation in the data contention is realized.

In the simulation experiments, we have evaluated the following performance metrics. The throughput is the number of transactions completed per second. The response time is the time spent by the transaction in the system. This is the difference between when the transaction first submitted and when the transaction decides to commit. Also, in the limited resources environment we have measured the number of transactions that are rejected (resubmitted) due to crossing of *execution_limit*.

Parameter	Meaning	Value
db_size	Number of objects in the database	1000 objects
num_sites	number of sites in the system	5 sites
tran_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 the global requests	0.6
res_io	time to carry out i/o request	35 msec
res_cpu	time to carry out CPU request	15 msec
MPL	Multiprogramming level	Variable
trans_time	Transmission time between two sites	Variable

Table 1. Model parameters with settings

5.2 Unlimited resources

In this section, we assume that unlimited resources exist in the system. That is, we put no limit on both *versions_limit* and *executions_limit* variables.

At different MPL values, Figure 5 shows the throughput results and Figure 6 shows the response time results of both approaches. In the DDBSs, the increase in the MPL results higher data contention. As a result, more number of transactions wait for the data objects. Therefore, more number of transactions carry out their executions in the speculative mode. This results in increase in the parallelism which in turn results increase in the throughput values. Also, this increase in the parallelism reduces the response time of the transactions. As a result, the SDTP approach exhibits better performance.

At different transmission time values, Figure 7 shows the throughput results and Figure 8 shows the response time results of both approaches. In the DDBSs, as the transmission time increases, the transaction spends longer duration in the commit processing. Consequently, more number of transactions wait for the data objects for longer duration. In the SDTP approach, the performance gain is achieved as the transactions are processed in parallel.

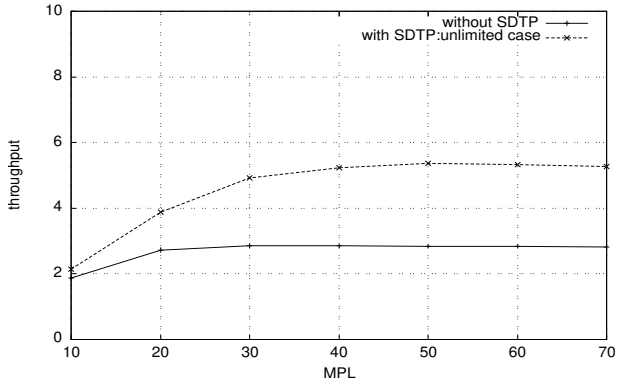


Figure 5. MPL versus throughput at transmission time=1000 msec

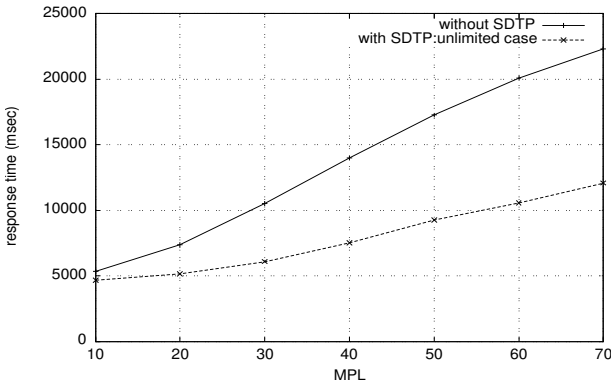


Figure 6. MPL versus response time at transmission time=1000 msec

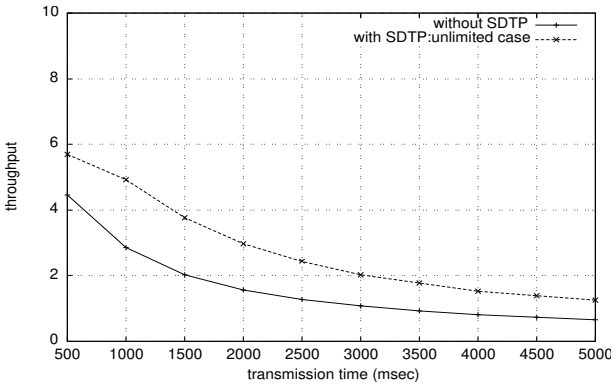


Figure 7. transmission time versus throughput at MPL=30

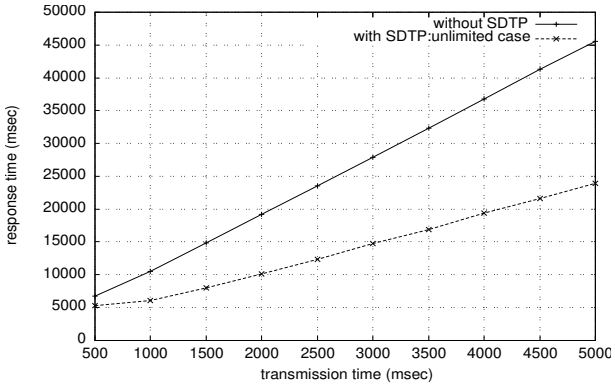


Figure 8. Transmission time versus response time at MPL=30

5.3 Limited resources situation

In the following experiments, we have fixed the *executions_limit* to 1000.

At different MPL values and *versions_limit* values, Figure 9 shows the throughput results and Figure 10 shows the response time results. Figure 11 shows the number of transactions rejected (resubmitted) due to crossing of *executions_limit*. From Figure 9, at beginning, by increasing *versions_limit* value the throughput keeps increasing. But, at higher *versions_limit* values the throughput reaches a plateau. The reason is that at higher

versions_limit values, more number of transactions are rejected (Figure 11).In the Figure 10, it can be observed that by increasing the *versions_limit* value, the response time is improved considerably towards the unlimited case.

At different transmission time values and *versions_limit* values, Figure 12 shows the throughput results and Figure 13 shows the response time results. The corresponding rejections are shown in Figure 14. From the Figure 12, it can be observed that, at the beginning, as we increase the *versions_limit* value, the throughput keeps increasing. But, at higher *versions_limit* values the throughput stops increasing. From the Figure 13, it can be observed that as we increase the *versions_limit* value the the response time decreases correspondingly. But, at higher *versions_limit* values the response time stops decreasing. From Figure 14, it can be observed that the increase in the *versions_limit* value increases the number of rejections. However, at the fixed *versions_limit*, the number of rejections in the SDTP case is same at different transmission times. This is because the increase in the transmission time increases the waiting for the data objects but the contention for the data objects is same.

From above experiments it can be observed that by selecting the appropriate *versions_limit* and *executions_limit* values we can achieve considerable improvement in both throughput and response time metrics.

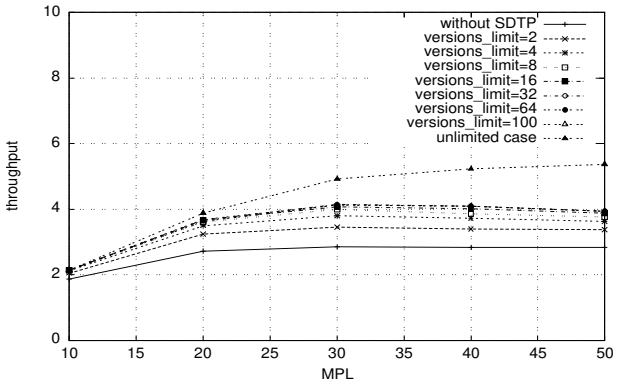


Figure 9. MPL versus throughput: transmission time=1000 msec, *executions_limit*=1000

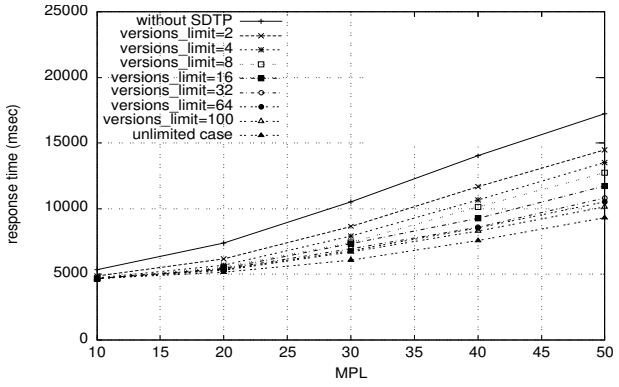


Figure 10. MPL versus response time: transmission time = 1000 msec, *executions_limit*=1000

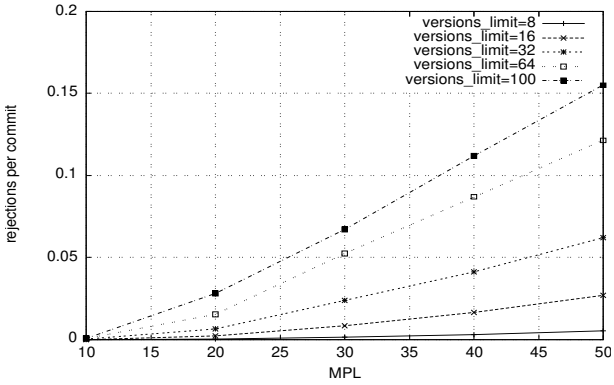


Figure 11. MPL versus rejections: transmission time=1000 msec *executions_limit*=1000

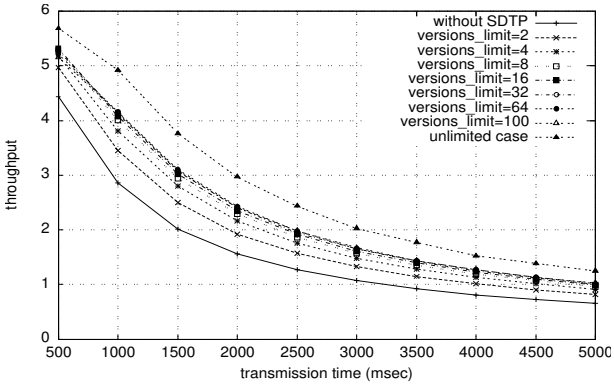


Figure 12. transmission time versus throughput: MPL=30, *executions_limit*=1000

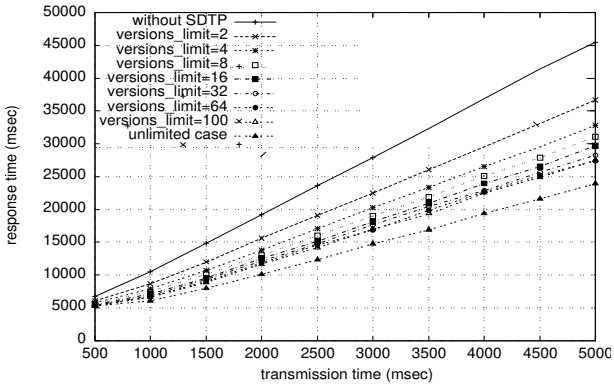


Figure 13. transmission time versus response time: MPL=30, *executions_limit*=1000

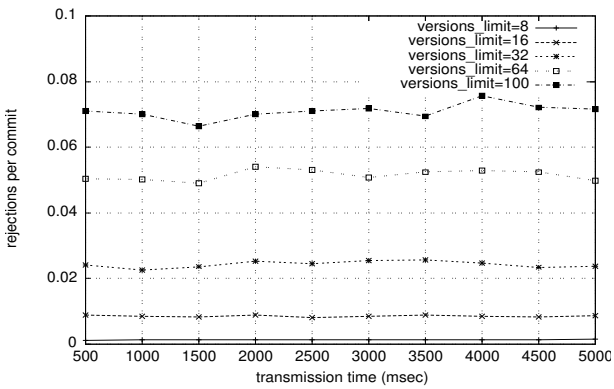


Figure 14. transmission time versus rejections:MPL=30, *executions_limit*=1000

6 Summary and conclusions

In this paper, we have proposed a speculative transaction processing approach in which a transaction releases the locks on the data objects immediately after the completion of its execution. By this, the waiting transaction accesses the required data objects and carries out speculative executions by reading the data object values read and written by preceding conflicting transactions. Before the end of commit processing it retains the appropriate execution based on the abort/commit decisions of preceding conflicting transactions. Using this approach, conflicting transactions can be processed in parallel without violating the serializability criteria. This increases parallelism which results in minimizing the response time. This approach is free from cascading aborts. The SDTP approach requires extra resources to carry out speculative executions. Through the simulation study, we have evaluated the performance of the SDTP approach. It has been found out that even with the limited resources the SDTP approach considerbly minimizes the response time and increases the throughput in the case of higher resource conflicts and the longer transmission times. The SDTP approach suits best for the WAN environments where the message transmission time dominates the processing time.

Recent advances in communication technology improves the transmission bandwidth significantly. However, we are still suffering from message latency and it will be very difficult to resolve this problem in near future. The SDTP plays an important role under such latency sensitive database applications. In this paper, we have presented the SDTP approach by employing the 2PC protocol. In case of 3PC protocol, the SDTP approach exhibits an improved performance as an extra round of message transmission is involved.

Acknowledgments The authors are thankful to the Japan Society for the Promotion of Science for its supports.

References

[1] Azer Bestavros and Spyridon Braoudakis, Value-cognizant speculative concurrency control, proc. of the 21th VLDB Conference, pp.122-133, 1995.

[2] R.Agrawal, M.J.Carey and M.Livny. Concurrency control performance modeling: alternatives and implications, ACM Transactions on Database Systems, vol.12, no.4, pp.609-654, December 1987.

[3] B.Bhargava, Y.Zhang, S. Goel, A study of distributed transaction processing in an internetwork, "Information Systems and Data Management", Volume 1006 of Lecture Notes in Computer Science, pp. 135-152, Springer-Verlag, 1995.

[4] P.A.Bernstein, V.Hadzilacos and N.Goodman, Concurrency control and recovery in database systems, Addison-Wesley, 1987.

[5] S.Ceri and P.Pelagatti, Distributed databases: principles and systems, New York:McGraw-Hill, 1984.

[6] K.R.Eswaran, J.N.Gray, R.A.Lorie, and I.L. Traiger, The notions of consistency and predicate locks in a database system, Communications of ACM, November 1976.

[7] J.N.Gray, Notes on database operating systems: in operating systems an advanced course, Volume 60 of Lecture Notes in Computer Science, pp.393-481, 1978.

[8] E.Knapp, Deadlock detection in distributed databases. ACM Computing Surveys, vol.19, no.4, pp.303-328, December 1987.

[9] W.H.Kohler, A survey of techniques for synchronization and recovery in decentralized computer systems, ACM Computing Surveys, pp.149-183, June 1981.

[10] H.T.Kung, and J.T.Robinson, On optimistic methods for concurrency control, ACM Transactions on Database Systems, vol.6, no.2, pp. 213-226, June 1981.

[11] Ramesh Gupta, Jayant Haritsa and Kirti Ramamritam, Revisiting commit processing in distributed database systems, ACM SIGMOD, pp. 486-497, 1997.

[12] P.Spiro, A.Joshi, and T.K.Rangarajan, Designing an optimized transaction commit protocol, Digital Technical Journal, (3) 1, Winter 1991.

[13] D.Skeen, Nonblocking commit protocols, proc. of ACM SIGMOD, June 1981.

[14] D.Skeen and M.Stonebraker, A formal model of crash recovery in a distributed system, IEEE Transactions on Software Engineering, vol.SE-9, no.3, pp. 219-227, 1983.

[15] M.Tamer Ozsu and Patrick Valduriez, Principles of distributed database systems, Prentice-Hall, 1991.