J Supercomput (2015) 71:4095–4115
DOI 10.1007/s11227-015-1507-8

# A new concurrency control mechanism for multi-threaded environment using transactional memory

**Ammlan Ghosh[2] · Rituparna Chaki[3] · Nabendu Chaki[1,2]**

Published online: 7 September 2015


**Abstract** Software transactional memory (STM) is one of the techniques used towards achieving non-blocking process synchronization in multi-threaded computing environment. In spite of its high potential, one of the major limitations of transactional memory (TM) is that in order to ensure data consistency as well as progress condition, TM often forces transactions to abort. This paper proposes a new concurrency control mechanism. It starts with the existing TM implementations for obstruction freedom and eventually builds a new STM methodology. The primary objective is to reduce aborting of transactions in some typical scenarios. A programming model is described for a chain of update transactions that share the same data object among themselves. Using the proposed approach, any new update transaction appended in this chain need not wait for the earlier transactions to finish. The proposed STM allows wait-free, non-blocking implementation of a mix of read and multiple update transactions on the same shared data object with higher throughput.

This publication is an outcome of the R&D work undertaken in the ITRA project of Media Lab Asia entitled "Remote Health: A Framework for Healthcare Services using Mobile and Sensor-Cloud Technologies".

✉ Nabendu Chaki
nabendu@ieee.org

Ammlan Ghosh
ammlan.ghosh@gmail.com

Rituparna Chaki
rchaki@ieee.org

1 Faculty of Physics and Applied Computer Science, AGH University of Science and Technology, Kraków, Poland

2 Department of Computer Science and Engineering, University of Calcutta, Kolkata, India

3 A. K. Choudhury School of Information Technology, University of Calcutta, Kolkata, India

**Keywords** Concurrency control · Software transactional memory · Obstruction freedom

## 1 Introduction

Software transactional memory (STM) [20] is a promising technique to facilitate concurrent programming in modern multi-processor environment. A transaction in an STM executes series of reads and writes on shared data and then either commits or aborts. When two threads concurrently access the transactional data and at least one of these accesses is a write, conflict occurs.

The progress property of an STM demands that every transaction should eventually commit. The three different levels of progress guarantee for non-blocking process synchronization are wait freedom, lock freedom and obstruction freedom. The obstruction freedom [11] guarantees progress by ensuring that one thread makes progress if it executes in isolation. In presence of contention a transaction is allowed to abort the conflicting transaction or back off for arbitrary time interval to ensure progress [11,12]. Thus, one of the major challenges for STM-based solutions is concurrent abort-free execution of transactions maintaining progress condition, and data consistency.

One of the notable STM implementations is DSTM (Software Transactional Memory for Dynamic-sized Data Structures) [12]. It offers an abort-free non-blocking synchronization approach that guarantees progress when a thread executes in isolation. When a transaction faces contention with another, it consults with contention manager to decide which transaction to delay or abort and when to restart an aborting transaction.

There exist a few STMs [2,4–6,17] that have aimed to avoid spurious aborts. The propositions either use time stamp from a global clock [6], or maintain multiple versions [5,17], or use conflict serializability scheduling as in [2,4]. All these approaches are able to achieve abort-free execution for read-only transactions to some extent. However, none of them consider reducing abort for write transactions.

In very recent time, an obstruction-free non-blocking synchronization is proposed [8] that claims abort-free execution. However, the work in [8] is tailored for two concurrent transactions only. Moreover, the first transaction may be aborted by the second transaction under certain conditions. In this paper, we have proposed a new non-blocking, concurrency control approach for multi-threaded environment. The designing goal of the proposed algorithm is to allow multiple read and write transactions on the same data object. The proposed STM does not require aborting a transaction except towards handling a typical exception as detailed in procedure tryCommit (Step 50–58) of the algorithm proposed in Sect. 3. Thus, in this method every transaction with in a group is able to commit in a finite number of steps. The key idea of the algorithm is to create a chain of update transactions while accessing same data object concurrently. Every transaction in the chain shares the data value among them and always commits after satisfying certain conditions.

Although DSTM [12] is unable to provide desired progress guarantee, its implementation simplicity motivates us to build our solution using a data structure that is very similar to what is used for DSTM. Unlike DSTM, the proposed algorithm in Sect. 3 of this paper does not require any contention manager as the transactions are capable to resolve contention on their own.

The rest of the paper is organized as follows. Section 2 reviews the state of the art scenario for non-blocking process synchronization and explains some of its important terminologies and conceptions. In Sect. 3, the formal model of the proposed system is described. Section 4 presents the critical analysis of the proposed model. In Sect. 5, a comparative study of the proposed model is presented. The paper ends with concluding remarks in Sect. 6.

## 2 Review

The non-blocking synchronization technique in STM implementation ensures that at least some threads must commit while running concurrently. This property is known as progress condition. STM provides two levels of progress [10] i.e., transactional memory level (TM-level) progress and transaction level progress. At TM-level, progress means completion of the individual TM operation, whereas at transaction level, progress implies execution of a thread through a successful commit. At either of these two levels, non-blocking synchronization technique ensures that a thread pre-empted during its execution cannot prevent other transaction to make progress. Depending on the level of progress, three types of non-blocking progress guarantees are found [15]. Among these, the obstruction freedom guarantees that a thread makes progress if it executes in isolation. In obstruction free transactional memory (OFTM) [11], a transaction $T$ of a process $P$ may be forcefully aborted, if it concurrently executes with some process other than $P$ [9]. Thus, in presence of contention, choosing which transaction to abort and when to restart an aborting transaction is a crucial task. In order to cope up with the situation, OFTM takes help form contention manager. The contention management comprises of notification method for various events along with request methods that ask contention manager to make a decision. The notifications include beginning of a transaction, successful/unsuccessful commit, acquire of an object etc. The request method asks contention manager to decide whether to back off the transaction or to abort competing transactions [19].

The first OFTM that is implemented by Herlihy et al. [12], to manage dynamic set of data is known as DSTM (Software Transactional Memory for Dynamic-sized Data Structures). Since the inception of DSTM, several OFTM systems are implemented [7,14,16,21] that work upon the limitations of OFTM and propose a better solution. All of them include contention management policies to avoid conflicts among transactions. However, any contention management policy for obstruction freedom always eventually aborts a competing transaction to avoid deadlock [18]. There are different types of contention management policies those are evolved to work with a specific OFTM and to achieve better throughput. Thus, selecting a specific contention manager for a particular OFTM is a challenging task. In [19], the experimental evaluation shows that improper selection of contention manager deteriorates the throughput.

There are few works [2,5,6,17] on the conflict avoidance between transactions to reduce abort. These implementations either use time stamp from a global clock or maintain multiple versions or employ conflict serializability scheduling. In lazy Snapshot algorithm [6], every shared object gets a timestamp from a discrete logical global clock. The implementation retains multiple versions for each object and if sufficient versions available, then read-only transactions can commit without any back-off or abort. Multi-Version Permissive System (MV-Permissive) [17] also maintains multiple versions to avoid spurious aborts for read-only transactions. The SwissTM [5] combines global clock with hybrid conflict detection technique i.e., eager conflict detection for a write/write transactions and lazy conflict detection for read/write transactions. The approach gets best result when read transactions commit before writes. The garbage collection i.e., cleaning up of the older object versions is a challenging task for these algorithms. Moreover, they have focused on avoiding aborts for read-only transactions; how to reduce the number of aborts for write transaction has not been considered.

In [2], the conflict serializability model of database management system (DBMS) is introduced to reduce the rate of aborts. The system maintains a serializability order number for every transaction. In presence of contention, the transactions execute as per their order number without causing any abort. Construction of unique serializability order number is a crucial task as without this number the transaction cannot be serialized. Although this implementation is able to achieve a better throughput but cannot ensure that every transaction will commit in presence data accessing conflict.

In [4], a wait-free non-blocking synchronization is designed to exploit parallelism between read and write transactions without involving contention manager. The algorithm maintains a list of instructions for each sharable data object. A scheduler places the transactions' instruction in the appropriate list. The list is chosen in such a way so that contention between transactions can be avoided. The major drawback of this implementation is that every transaction must know list of instructions in advance, which is a quite challenging task.

Attiya and Milani presented a BIMODAL transactional scheduler [1] in the context of read-dominated workload. The algorithm specially tailored for abort-free execution of read-only transactions without causing any delay to the early-write transaction most of the cases. The throughput of the algorithm is significantly deteriorated for late-write transactions, where updates are made at commit time.

The proposed work designs a new non-blocking algorithm to achieve concurrency control for multi-threaded environment. This non-blocking thread synchronization algorithm is an improvisation of OFTM that focuses on lowering transaction aborts for update-executions in presence of contention. The proposed method doesn't require to include any existing contention management policies [19] as the update transactions are able to resolve conflicts themselves while accessing the sharable data concurrently.

In case of read–write contention, there are several comprehensive works and tested approaches towards lowering the abort for read executions in STM [2,4,5,17]. This paper focuses on write–write contention and maintaining the read executions is beyond the scope of this paper.

## 3 Proposed method

### 3.1 Basic concept

Two transactions running concurrently may face contention with each other, if both try to acquire the same data object simultaneously and at least one of these is involved in a write operation. In typical OFTM implementation [12], the transaction has two options in such situation. Either abort one of the conflicting transactions, or back off for some arbitrary time interval. In the proposed method both the transactions are allowed to access the data object without causing any abort or delay.

Let's explain the scenario with an example. Suppose a transaction $T_k$ has opened a sharable object X for write and it is in active state. Now, another write transaction $T_x$ wants to access X. In this situation, in contrast to OFTM, the proposed method allows $T_x$ to access the data object from $T_k$ after forming a chain of transactions.

Figure 1 depicts a situation, where four write transactions simultaneously access the same data object by forming a chain of transactions. Let, $T_k$ be the first transaction in the chain, known as header, that owns the sharable object X. Transaction $T_x$ is the next one that reads X, while $T_m$ and $T_z$ appear next in that order. The header transaction, $T_k$ in this example, is also referred as owner transaction. A transaction is termed as *immediate-predecessor* transaction when it occurs immediately before a transaction. Thus in the example, $T_k$ is the immediate-predecessor transaction to $T_x$, which again is immediate-predecessor to $T_m$ and so on. The header can directly access the data object and commit without any dependencies. At commit point every transaction, other than header, ensures that its immediate-predecessor transaction is committed and the data value that it has read is consistent.

### 3.2 Data structure

The data Structure for the proposed model is similar but not identical to those used in [8] and DSTM [12] (Fig. 2). The Transactional Data Structure, Data Object and Locator are similar to those used in [8,12]. However, to match with the adaptation proposed in our algorithm, a new status called READY is incorporated. These revised data structures are briefly described here for the sake of completeness. The

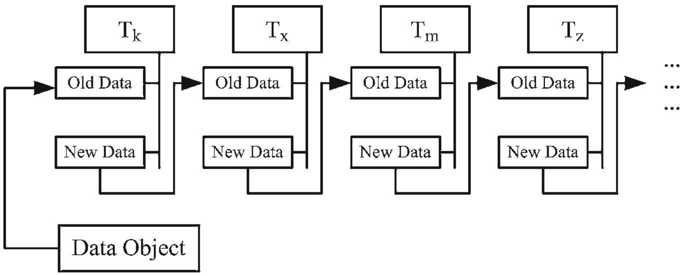



**Fig. 1** Chain of transactions sharing Data Object

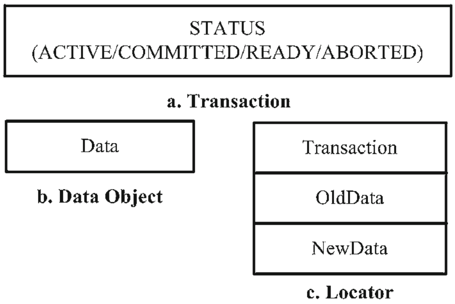



**Fig. 2** Transactional Data Structure with Data Object and Locator

transactional memory object (TMObject) in Sect. 3.2.3 is newly introduced in this paper.

### 3.2.1 Transactional data structure

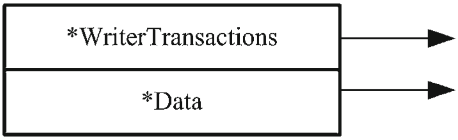
The **Transactional Data Structure** consists of a Status field with four states: ACTIVE, COMMITTED, READY and ABORTED states. These states are used to determine the current state of a transaction (Fig. 2a). ACTIVE status means that a transaction has began and in operation; COMMITTED means successfully completed all its tasks and READY means the transaction has completed its operations and waiting to commit.

It is important to mention here that the Transactional Data Structure also maintains the status ABORTED. This is apparently in conflict with the desired goal to achieve concurrent execution of transactions. Aborting of transactions is used in OFTM [11] to ensure that a new transaction is not blocked by an older transaction. The same technique is implemented in other reported citations [7,12,14,16,21]. However, indiscriminate use of such transaction aborting may seriously affect performance of a system. On the contrary, STM is used as an alternative of the conventional deadlock handling measures like mutual exclusion to avoid contention. However, a policy of never aborting the transactions may lead to a cyclic concurrency conflict situation where processes holding multiple shared resources may form a closed wait-for cycle. In order to handle such exceptions, a transaction $T_x$ may be allowed to abort its immediate-predecessor transaction $T_k$, which is owner of the data object and $T_x$ has waited for a very long time. This is expected to increase the throughput of overall system.

### 3.2.2 Data object and locator object

Figure 2b, c depicts **Data Object** and **Locator object** respectively. **Data Object** contains the last committed data. The **Transaction** field of the locator points to the transaction that creates the locator. In **OldData** field transaction copies the read data value and in **NewData** transaction stores last undated value at the time of execution. When transaction successfully commits, the stored value of **NewData** field is being saved into Data object.

**Fig. 3** TMObject sructure




### *3.2.3 Transactional memory object (TMObject)*

TMObject (Fig. 3) encapsulates a program object that can be accessed by the transaction. TMObject has two fields:

- ***WriterTransactions:** This field points to an array of pointers. Each element of this array points to the transaction locator opened in a cascading manner to access the data object. The first element of the array points to the transaction locator of the first initiated transaction in the chain that owns the sharable object. In the rest of the paper the first array element is termed as **header**. Rest of the transaction locators in the chain other than **header** are the pseudo owners of that sharable object.
- ***Data** This field points to the Data object to read the recent committed data.

### *3.2.4 Proposed concurrency control mechanism*

This section describes the proposed algorithm that aims to reduce the number of aborts for write execution while accessing common shared object. Before the new algorithm is described, let's state the assumption on how multiple write transactions forms a cascading chain. We assume that if a write transaction faces contention with other transaction(s) while accessing a sharable object then it includes itself as the last element in a chain of active transactions and reads the sharable object's value. Thus, in the chain of transactions, the header is the owner of the sharable object and all other transactions are the pseudo owner of that same sharable object. When the header transaction wants to commit, as it is the owner, it can commit directly.

When a transaction, which is the pseudo owner of the data object, wants to commit, it checks the status of its immediate-predecessor transaction. If the immediate-predecessor transaction is in committed state then transaction checks the data consistency with the recently committed data value and re-executes its write operation if necessary. If the pseudo owner finds that its immediate-predecessor transaction is Ready/Active state then the transaction checks for the data consistency and re-executes it write operation if necessary. The pseudo owner transaction cannot commit until its immediate-predecessor transaction commits successfully.

Let us explain the commit process from the transaction's point of view. At commit point a transaction checks the status of its immediate-predecessor transaction. In the example (Fig. 4) $T_m$ will check the status of its immediate-predecessor transaction i.e., $T_x$.

- **If $T_x$'s status is Committed**; then $T_m$ checks for the data consistency with its old value and Data Object's value (as Data Object stores the recent committed data

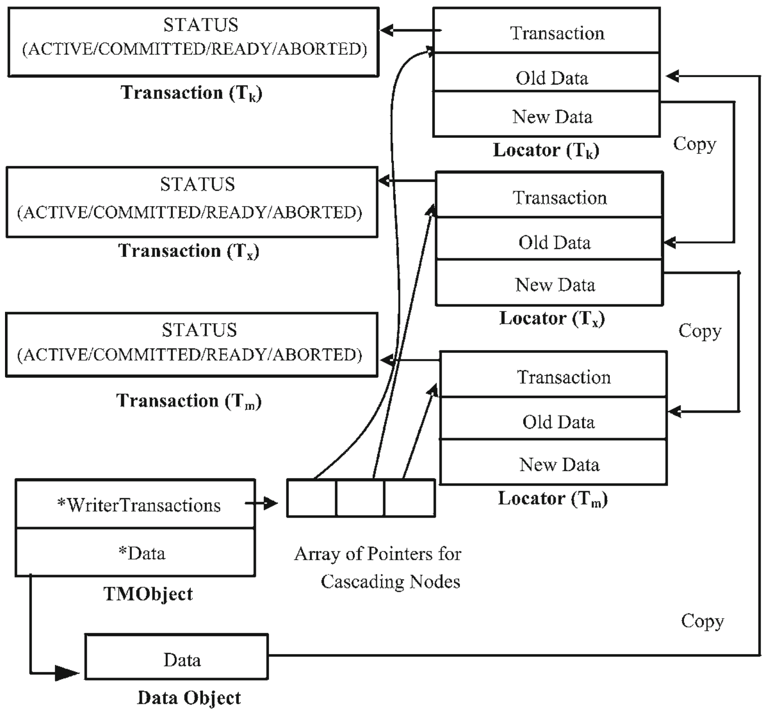



**Fig. 4** Concurrent write for transactions in presence of contention

updated by $T_x$). If the data is consistent then $T_m$ commits otherwise $T_m$ re-executes write operation after reading recent data value and then commits.

- **If $T_x$'s status is Ready**; then $T_m$ checks the data consistency with its OldField and $T_x$'s NewField value. If data value is consistent then $T_m$ backs off for some arbitrary time and if data is inconsistent then $T_m$ re-executes its write operation after reading data value from $T_x$'s NewField and backs off for a very small interval and retries to commit.
- **If $T_x$'s status is Active**; then $T_m$ follows the steps same as $T_x$ is in ready state and backs off for some arbitrary time to give chance to $T_x$ to commit.

The proposed solution is presented in Algorithm 1. The workflow of this algorithm is as follows: a transaction, $T$, tries to acquire an object X (Line 2). If $T$ finds that the sharable data object is not currently owned by any other transaction then $T$ becomes the owner of that data object. Otherwise, $T$ becomes the pseudo owner of the sharable object. Pseudo owner implies that, although transaction is accessing the sharable object, it may face inconsistency at commit time. In this process a chain of transactions is formed (Line 12–28), where the first transaction in the chain is the owner of the sharable object and rest of the transactions are pseudo owners. Each transaction in the chain points to their respective transaction locator to point old and new versions of the sharable object. In the execution process, transaction executes its update query (Line 3) and tries to commit (Line 4–10). If the transaction, say $T$, is the owner of the sharable object then it can commit immediately (Line 6, 29–34), otherwise $T$ executes tryCommit() (Line 8, 35–65) after an arbitrary time of back-off until it becomes the owner of the data object.

## 4 New concurrency control mechanism: a critical analysis

The proposed methodology is based on the foundations of OFTM, while it provides a completely new mechanism for write transactions to execute concurrently while sharing common data object.

Suppose, $T_1, T_2, T_3, \ldots, T_k$ are consecutive write transactions in the chain. These transactions have formed the chain in the order as these are mentioned. The statements of Lemmas 1, 2 and 3 are stated for these transactions in the chain.

**Lemma 1** *Transaction $T_i$ can commit only when $T_{i-1}$ is committed for $i \in [2..k]$.*

*Proof* Suppose $T_1$ and $T_2$ are consecutive transactions in a chain, appearing in the order in which these are mentioned. Transaction $T_k$ is executed by a process $P_i$ and let $T_k$ be the owner of the sharable object X. So, Committed $[T_k]$ is true for $k = 1$.
For $k = 2$, $T_1 \leftarrow T_2$
Thus $T_2$ will commit when Committed$[T_1]$ is true and $T_1$[X, New] = $T_2$[X, Old]. Now it is to be shown that $T_{m-1} \leftarrow T_m$ i.e., to commit $T_m$, $T_{m-1}$ must be committed.
Committed$[T_{m-1}]$ is true iff Committed$[T_{m-2}]$ is true and $T_{m-1}$[X, New] = $T_{m-2}$[X, Old].
Hence, Committed$[T_m]$ is true iff Committed$[T_{m-1}]$ is true and $T_{m-1}$[X, New] = $T_m$[X, Old].
So we can say, $T_i$ can commit only when Committed$[T_{i-1}]$ is true for $i = 2, 3, \ldots, k$. □

**Algorithm 1** Proposed Algorithm

```
▷ acq_st:Acquired State; either exclusive owner or pseudo owner.
▷ t_state: Transaction state; Committed, Active, Ready, Aborted.
▷ cmt_st: Commit status; true or false. Initial value is false.

1: upon write of sharable object x by T_k do
2:     acq_st = Acquire(T_k, x);
3:     executeUpdate(T_k, x);
4: repeat
5:     if acq_st= 'owner' then
6:         cmt_st = Commit(T_k)
7:     else if acq_st= 'pseudo_owner' then
8:             cmt_st = tryCommit()
9:     end if
10: until cmt_st = true
11: return ok
12: procedure Acquire (T_k, x)
13: if x is free then
14:     acq_st = 'owner';
15:     front = 1;
```

**Algorithm 1** (continued)

```
    *WritetrTransactions[front] = locator(T_k);
    rear = front;
    *WritetrTransactions[front].locator.TransactionStatus = 'Active';
    return acq_st;
else
    acq_st = 'pseudo_owner';
    rear = rear+1;
    *WriterTransactions[rear]=locator(T_k);
    *WriterTransactions[rear].locator(T_k).OldData = WriterTransaction[rear-1].locator(T_k).NewData;
    *WriterTransactions[rear].locator.TransactionStatus = 'Active';
    return acq_st;
end if
end procedure        ▷ The header of the transaction-chain is the exclusive owner and it can commit
  tryCommit(T) also calls this procedure
procedure Commit(T_k)
     *Data = *WriterTransactions[front].locaotr(T_k).NewData;
     *WriterTransactions[front].locaotr(T_k).TransactionStatus = 'Commited';
     front=front+1;
     return true;
end procedure        ▷ When a transaction tries to commit, either it can commit or re-execute or
  back off. When a transaction backs-off for several times it may abort its immediate-predecessor, if that
  transaction is the header in the chain.
procedure tryCommit(T_k)
     pos = findElementPosition(*WriterTransactions, locator(T_k))
if pos = front then
    Commit(T_k);
    return true;
else
    *WriterTransactions(pos).locaotor.TransactionStatus = 'Ready';
              t_state = *WriterTransactions(pos-1).locaotor.TransactionStatus;
    if t_state = 'Ready' or t_state = 'Active' then
        if *WriterTransactions(pos).locator.OldData = *WriterTransactions(pos-1).locator.NewData
  then
            *WriterTransactions(pos).locator.OldData = *WriterTransactions(pos-1).locator.NewData;
            *WriterTransactions(pos).locaotor.TransactionStatus = 'Active';
            Re-executeUpdate();
            return false;
        else
            back-off();
            if back-off_time >back-off_limit and pos-1=front then
                WriterTransactions(pos).t_state='Aborted';
                if *WriterTransactions(pos).locator.OldData = *Data then
                    WriterTransactions(pos).locator.OldData = *Data;
                    Re-executeUpdate();
                end if
                Commit(T_k)
                return true;
            end if
            return false;
        end if
        return false;
    end if
end if
end procedure
```

**Lemma 2** *$T_i$ will re-execute its write operation when Committed*$[T_{i-1}]$ *is true or Active*$[T_{i-1}] = $ *true and* $T_{i-1}$[X, New] $\neq T_i$[X, Old] *for* $i \in [2..k]$*, where k is the total number of transactions in the chain.*

*Proof* Suppose a transaction $T_1$ is executed by a process $P_i \cdot T_1$ is owner of the sharable object X.

Now, $T_2$ is another transaction where $T_1 \leftarrow T_2$. $T_2$ will re-execute its write operation when Committed[$T_1$] is true or Active[$T_1$] = true and $T_1$[X, New] $\neq T_2$[X, Old]. Now it is to be shown that $T_{m-1} \leftarrow T_m$ i.e., $T_m$ will re-execute its write operation iff $T_{m-1}$ writes data value for the sharable object X after $T_m$ read the value of X from $T_{m-1}$. Using **Lemma** 1 it can be proved that $T_i$ re-executes its write operation for the sharable object X when Committed[$T_{i-1}$] is true or Active[$T_{i-1}$] = true and $T_{i-1}$[X, New] $\neq T_i$[X, Old] for $i = 2, 3, \ldots, n$. □

**Lemma 3** *Proposed algorithm is step contention Free.*

*Proof* A transaction $T_k$ of a process $P_i$ encounters a step contention when some process other than $P_i$ executes a step between first event of $T_k$ and before commit/abort of $T_k$ [9]. In presence of step contention, generally, transaction may be forcefully aborted. In the proposed method, transaction $T_k$ (assumed as firstly initiated transaction) can only own the sharable object. All other $T_i$ transactions, for $i = 2, 3, 4, \ldots, n$, are pseudo owners and depend on the values owned by the $T_{i-1}$ transaction. Hence transactions are step contention free and thus no transaction is forcefully aborted while accessing sharable object concurrently.

Although the proposed algorithm claims to be step contention free, in only one scenario a transaction, say $T_x$ is allowed to forcefully abort its immediate-predecessor transaction, say $T_k$, if $T_k$ is the owner of the sharable object and $T_x$ has backed off more than a certain duration. This abort mechanism facilitates to overcome the infinite wait problem which otherwise may affect, cumulatively, the average execution time of other transactions in the chain. □

## 5 Performance evaluation

We have considered the efficiency of the proposed algorithm on the basis of transactions' start time, access time of the sharable object and the execution length. The data sets are considered and grouped to cover all possible classes of scenarios that may occur between transactions in terms of these parameters As for example, a typical scenario may consider that the second transaction occurs at a time when the first transaction has already accessed the shared resource, but could not commit itself as the first transaction is yet to commit. In another scenario, the second transaction may occur and then start accessing the shared resource even before the first transaction could access the resource although first transaction is initiated before the second transaction. For each and every scenario, data sets are taken with random values in the range of that particular group. The throughput of the proposed STM is evaluated and compared with conventional lock-based concurrency control algorithm [3,13] for each scenario.

The proposed algorithm has a single iterative step and it iterates exactly $n$ times, where $n$ is the number of transactions in the chain. If average time of execution for

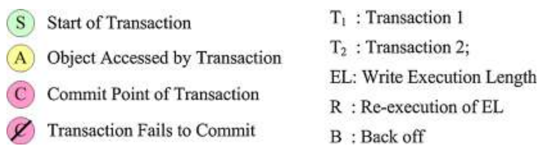



**Fig. 5** Symbols and abbreviations used in this section

each transaction is $\tau$, then the worst case performance is O $(n\tau)$, which is equivalent to serial execution of the transactions one after another. However, the actual turn-around time for a group of $n$ transactions would be much less than $n\tau$, due to the concurrent execution of transactions.

It is not quite realistic to make a best-case or even an average-case time estimation for the proposed algorithm as execution time would depend on the relative length of successive transactions as well as on the actual time of accessing the shared data object. Thus, in this section, the performance evaluation is done using an abstracted view. All possible scenarios are grouped into three distinct types of cases in Sects. 5.1, 5.2 and 5.3. At first the different scenarios in terms of the access times of the successive transactions are considered for two arbitrary transactions $T_1$ and $T_2$, where transaction $T_1$ is the immediate-predecessor transaction of $T_2$. Subsequently, to make the analysis true for multiple transactions, the result set of five transactions executing in cascading manner and sharing same sharable object has been considered. In this section, we have included various diagrams regarding the commit process of two transactions in different scenarios for a better understandability. The tables in this section list some representative cases to study the effectiveness of the proposed algorithm. Figure 5 shows different symbols and abbreviated forms used in the diagrams and tables. In figures and tables, the legend for the proposed new STM is termed as PSTM for brevity. Few symbols and abbreviations (e.g. S, A, C etc.) those are self-explanatory are not described. The other abbreviations used in the tables and figures are as follows:

- **SZ** is the size of each transaction in terms of clock cycle and hence a large value indicates more clock cycle to commit.
- **ST** is the initiation time of a transaction.
- Access Time **AC** is the number of cycles after which a transaction accesses a sharable object.
- **EL** is the write execution length in clock cycles.
- The term **CommitPoint** implies the commit point of the transaction in absence of contention.
- **R_B** states the number of re-execution of write process and/or number of back-offs. For example, R2B3 implies transaction has re-executed its write operations for two times and backed off three times before commit.
- $T_1$ and $T_2$ are consecutive transactions in the chain, appearing in the order in which these are mentioned.
- **PSTM**: The legend for the Proposed STM.
- **LOCK**: Lock-based concurrency control algorithm.
- **EET**: Effective Execution Time of Proposed Method over Lock-based Synchronization.
- $T_1$ and $T_2$: Are two update-transactions, where $T_1$ has initiated before $T_2$.
- **Access** $(T_2) > Access(T_1)$: transaction $T_2$ accesses the sharable object after $T_1$.

- **Access** $(T_2) < Access(T_1)$: transaction $T_2$ accesses the sharable object before $T_1$.
- **CommitPoint** $(T_2) > CommitPoint(T_1)$: transaction $T_2$ reaches its commit point after $T_1$.
- **CommitPoint** $(T_2) < CommitPoint(T_1)$: transaction $T_2$ reaches its commit point before $T_1$.

### 5.1 Case I: transaction $T_2$ accesses sharable object after transaction $T_1$

In this case transaction $T_2$ accesses the sharable object after $T_1$. So, hopefully, at the commit point $T_2$ will find $T_1$ in committed state with consistent value of the data object that $T_2$ has read. In such case, $T_2$ can commit without any back-off. Figure 6 shows this scenario and Table 1 analyzes the result set. In Table 1, in all five cases, $T_2$ has accessed the object after $T_1$ and the size of $T_2$ and/or update execution length is greater than $T_1$. Thus $T_2$ is expected to commit after $T_1$. The result set shows that the proposed algorithms perform better or at par in comparison with conventional lock-based commit protocol.

In the next scenario, transaction $T_2$ accesses the object after $T_1$, as in earlier case, but $T_2$ reaches the commit point when $T_1$ is in active state due to $T_2$'s shorter size and/or update execution length. Thus $T_2$ will back off for certain time to give the chance to $T_1$ to commit. $T_2$ will retry to commit after back-off time period. In the proposed algorithm this back-off time is decided to make same as transaction's write execution time to avoid the intervention of contention manager and its overheads.

Figure 7 shows that $T_2$ has to back off two times before it can commit. Result set in Table 2 shows that $T_2$ requires back off one or more time but re-execution is not necessary until $T_2$ gets an inconsistent data value at commit time.

It is important to mention here that the number of back-offs is dependent on the size and/or execution length (EL) of the second transaction; lesser size/EL implies higher

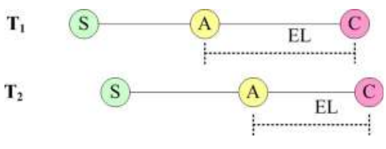



**Fig. 6** Access$(T_2) > Access(T_1)$ and CommitPoint$(T_2) > CommitPoint(T_1)$

**Table 1** Efficiency: Access$(T_2) > Access(T_1)$ and CommitPoint$(T_2) > CommitPoint(T_1)$

| – | Transaction 1 ($T_1$) | | | | Transaction 2 ($T_2$) | | | | | Commit time | | – |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SL | SZ | ST | AC | EL | SZ | ST | AC | EL | R_B | PSTM | LOCK | EET (%) |
| 1 | 60 | 1 | 7 | 53 | 65 | 10 | 10 | 55 | R0B0 | 74 | 115 | 64.30 |
| 2 | 90 | 1 | 25 | 65 | 85 | 15 | 30 | 55 | R0B0 | 99 | 145 | 68.30 |
| 3 | 33 | 1 | 15 | 35 | 38 | 15 | 5 | 33 | R0B0 | 52 | 66 | 78.79 |
| 4 | 27 | 22 | 10 | 17 | 29 | 25 | 12 | 17 | R0B0 | 53 | 65 | 81.50 |
| 5 | 20 | 6 | 13 | 7 | 20 | 14 | 15 | 5 | R0B0 | 33 | 33 | 100.00 |

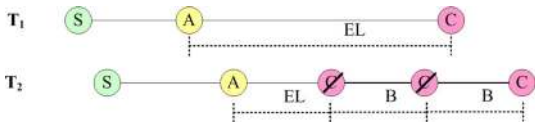



**Fig. 7** Access($T_2$) $>$ $Access(T_1)$ and CommitPoint($T_2$) $<$ $CommitPoint(T_1)$

**Table 2** Efficency: Access($T_2$) $>$ $Access(T_1)$ and CommitPoint($T_2$) $<$ $CommitPoint(T_1)$

| – | Transaction 1 ($T_1$) | | | | Transaction 2 ($T_2$) | | | | | Commit time | | – |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SL | SZ | ST | AC | EL | SZ | ST | AC | EL | R_B | PSTM | LOCK | EET (%) |
| 1 | 50 | 5 | 15 | 35 | 30 | 10 | 12 | 18 | R0B1 | 56 | 72 | 77.78 |
| 2 | 89 | 57 | 51 | 38 | 74 | 61 | 59 | 15 | R0B1 | 148 | 160 | 92.50 |
| 3 | 99 | 1 | 41 | 58 | 91 | 3 | 58 | 33 | R0B1 | 125 | 132 | 94.70 |
| 4 | 20 | 1 | 10 | 10 | 12 | 5 | 8 | 4 | R0B2 | 22 | 24 | 91.67 |
| 5 | 86 | 49 | 33 | 53 | 76 | 59 | 54 | 22 | R0B1 | 155 | 156 | 99.36 |

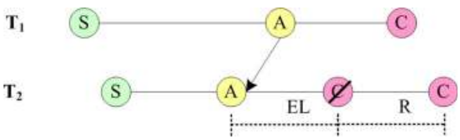



**Fig. 8** Access($T_2$) $<$ $Access(T_1)$; CommitPoint($T_2$) $<$ $CommitPoint(T_1)$ and EL($T_2$) $<$ $EL(T_1)$

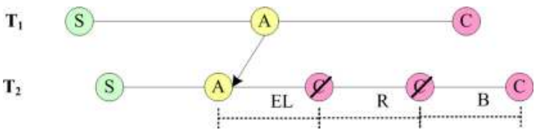



**Fig. 9** Access($T_2$) $<$ $Access(T_1)$; CommitPoint($T_2$) $\ll$ $CommitPoint(T_1)$ and EL($T_2$) $\ll$ $EL(T_1)$

number of back-offs. Although second transaction may back off for several times, still it produces better throughput than lock-based method.

### 5.2 Case II: transaction $T_2$ accesses sharable object before transaction $T_1$

When $T_2$ accesses the sharable object before $T_1$, it is obvious that at commit time $T_2$ will get an inconsistent data value, and thus, $T_2$ must re-execute its write operation (i.e., same as its execution length, EL). At new commit point $T_2$ checks for data consistency again, if inconsistent then $T_2$ re-executes its write operation, otherwise checks for the status of $T_1$. If $T_1$ is in active state then $T_2$ backs off, otherwise commit. Figures 8 and 9 depict this scenario. In Fig. 8, EL of $T_2$ is less than EL of $T_1$ and in Fig. 9 EL of $T_2$ is much lesser than EL of $T_1$ [EL($T_2$) $\ll$ EL($T_1$)]. Record of row 4 in Table 3 shows a special case where second transaction requires to re-execute and back off for several times (i.e., R5B4) due to its lesser execution length but still proposed method shows a better efficiency than lock based.

**Table 3** Efficency: Access($T_2$) > *Access*($T_1$) and CommitPoint($T_2$) < *CommitPoint*($T_1$)

| – | Transaction 1 ($T_1$) | | | | Transaction 2 ($T_2$) | | | | | Commit Time | | – |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SL | SZ | ST | AC | EL | SZ | ST | AC | EL | R_B | PSTM | LOCK | EET (%) |
| 1 | 20 | 1 | 10 | 10 | 15 | 2 | 7 | 8 | R1B0 | 23 | 28 | 82.14 |
| 2 | 70 | 1 | 53 | 17 | 40 | 18 | 25 | 15 | R1B0 | 71 | 85 | 83.53 |
| 3 | 51 | 15 | 10 | 65 | 30 | 10 | 10 | 20 | R1B1 | 77 | 85 | 90.59 |
| 4 | 20 | 10 | 10 | 10 | 10 | 3 | 8 | 2 | R5B4 | 30 | 31 | 96.80 |
| 5 | 20 | 1 | 10 | 10 | 10 | 2 | 7 | 3 | R1B3 | 23 | 23 | 100.00 |

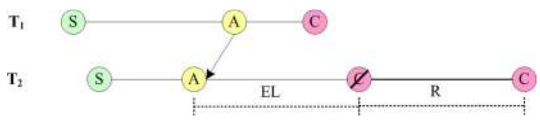



**Fig. 10** Access($T_2$) < *Access*($T_1$); CommitPoint($T_2$) > *CommitPoint*($T_1$) and EL($T_2$) > *EL*($T_1$)

**Table 4** Access($T_2$) < *Access*($T_1$); CommitPoint($T_2$) > *CommitPoint*($T_1$) and EL($T_2$) > *EL*($T_1$)

| – | Transaction 1 ($T_1$) | | | | Transaction 2 ($T_2$) | | | | | Commit time | | – |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SL | SZ | ST | AC | EL | SZ | ST | AC | EL | R_B | PSTM | LOCK | EET (%) |
| 1 | 85 | 1 | 60 | 25 | 65 | 22 | 35 | 30 | R1B0 | 115 | 115 | 100.00 |
| 2 | 71 | 1 | 53 | 18 | 69 | 4 | 49 | 20 | R1B0 | 91 | 91 | 100.00 |
| 3 | 71 | 1 | 53 | 18 | 71 | 3 | 50 | 21 | R1B0 | 93 | 92 | 101.09 |
| 4 | 85 | 1 | 60 | 25 | 75 | 15 | 40 | 35 | R1B0 | 123 | 120 | 102.50 |
| 5 | 71 | 1 | 53 | 18 | 71 | 5 | 45 | 26 | R1B0 | 100 | 97 | 103.09 |

In the next scenario, the performance of proposed method deteriorates due to higher EL of second transaction. It means whenever EL($T_2$) is larger than EL($T_1$), the update re-execution process takes long time to complete. Hence, $T_2$ requires long time to commit. Figure 10 explains this case, where $T_2$ has to re-execute its write operation as it finds a data inconsistency at the commit time. As the execution length of $T_2$ is larger, it takes long time to re-execute and hence requires longer time to commit. The result set (Table 4) shows that proposed algorithm has same or worse performance than Lock-based approach.

### 5.3 Case III: transaction $T_2$ accesses sharable object after commit of transaction $T_1$

In this case, transaction $T_2$ accesses the sharable object after commit of transaction $T_1$ (Fig. 11). Thus, at commit time, $T_2$ does not face any contention with $T_1$ and commits without any re-execution or back-off. Results in Table 5 show that this condition has the same performance result for the proposed algorithm and lock-based commit algorithm.

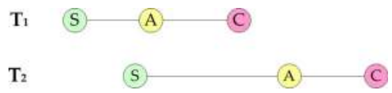



**Fig. 11** Access($T_2$) > $Commit(T_1)$

**Table 5** Access($T_2$) > $Commit(T_1)$

| – | Transaction 1 ($T_1$) | | | | Transaction 2 ($T_2$) | | | | | Commit time | | – |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SL | SZ | ST | AC | EL | SZ | ST | AC | EL | R_B | PSTM | LOCK | EET (%) |
| 1 | 70 | 1 | 60 | 10 | 65 | 40 | 35 | 30 | R0B0 | 104 | 104 | 100.00 |
| 2 | 40 | 5 | 20 | 20 | 35 | 30 | 25 | 10 | R0B0 | 64 | 64 | 100.00 |
| 3 | 35 | 1 | 12 | 23 | 30 | 28 | 9 | 21 | R0B0 | 57 | 57 | 100.00 |
| 4 | 25 | 1 | 10 | 15 | 23 | 20 | 17 | 6 | R0B0 | 42 | 42 | 100.00 |
| 5 | 20 | 1 | 5 | 15 | 20 | 10 | 12 | 8 | R0B0 | 29 | 29 | 100.00 |

### 5.4 Performance of PSTM over Loack based for a chain of transactions

The Figs. 12 and 13 along with Table 6 show the throughput comparison between the proposed algorithm and the conventional lock-based commit protocol. In this comparison, throughput is tested where five write transactions are accessing the same sharable object in a cascading manner by forming a chain (Table 6). Figure 12 shows the result set from row 1 to 5 and row 6 to 10 of Table 6. Result set shows that PSTM is able to achieve a better throughput than the Lock-based algorithm. Figure 13 shows same or a deteriorated performance of the proposed STM in some cases. This is due to larger execution length (and/or size) of the transactions in comparison to their immediate-predecessor transaction (Table 6, Row 11–20). It is worthwhile to mention here that the deteriorated commit time affects other transactions in the chain i.e., the delay in commit time for transaction $T_1$, in the above example, will affect commit time for transaction $T_2$, $T_3$ and so on. The abbreviations used in the Table 6 have already been described in Sect. 5. Due to the scarcity of space in Table 6, the commit time of PSTM and LOCK are written as $P$ and $L$ respectively. The column $P_1$ and $L_1$ shows the commit time for the first write transaction, whereas $P_2$ and $L_2$ show the commit time for the second write transactions in case of proposed STM and lock based and so on.

## 6 Concluding remarks

In this paper, a new non-blocking concurrency control mechanism for multi-threaded environment is proposed. The proposed STM aims to avoid aborting transactions excepting typical scenarios that may otherwise lead to a long denial of access to shared object. In this algorithm, when a write transaction faces contention with other write transactions, it neither aborts the conflicting transaction nor backs off. Instead, the transaction adds itself as an element in the chain of write transactions and reads the data object. Thus, multiple write transactions are allowed to execute concurrently

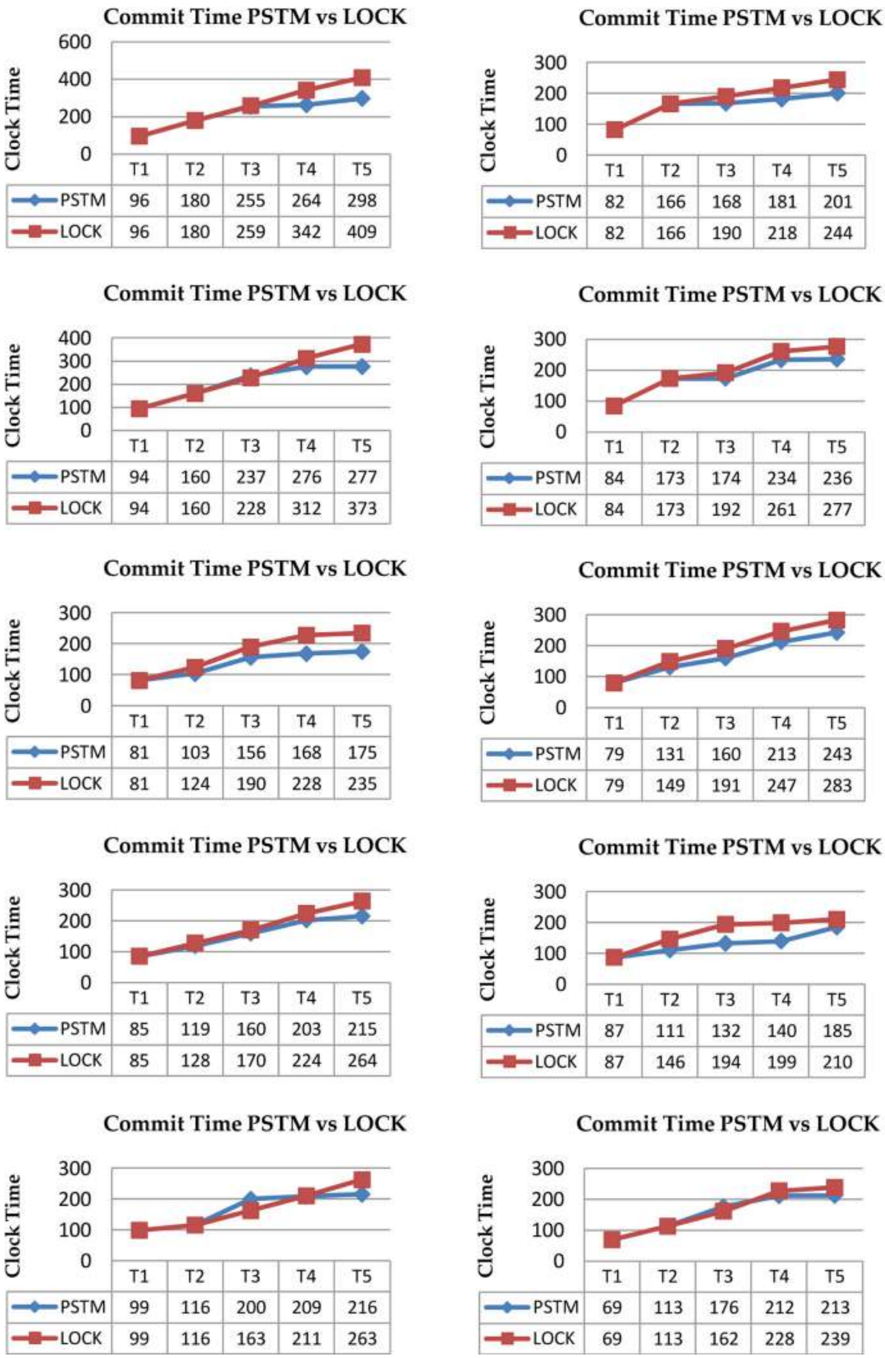



**Fig. 12** Performance Analysis of PSTM vs. lock-based protocol (Table 6, Row 1–5 and Row 6–10)

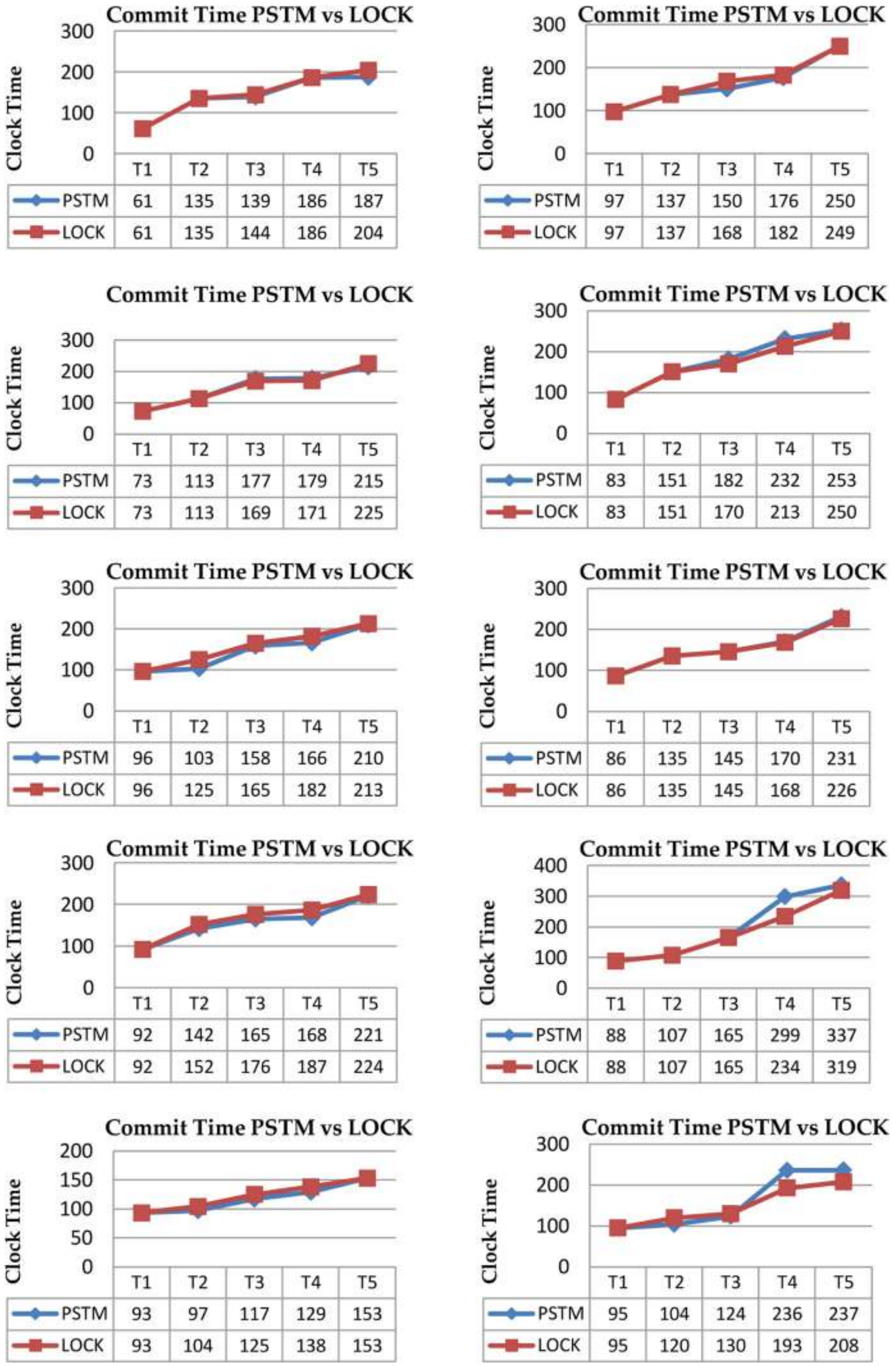



**Fig. 13** Performance analysis of PSTM vs. lock-based commit protocol (Table 6, Row 11–15 and Row 16–20)

**Table 6** Perfomance of PSTM over Lock-based commit for a chain of five transactions with random size and execution length

| – | Transaction ($T_1$) | | | | Transaction ($T_2$) | | | | | Transaction ($T_3$) | | | | | Transaction ($T_4$) | | | | | Transaction ($T_5$) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SL | SZ | ST | AC | $P1/L1$ | SZ | ST | AC | $P2$ | $L2$ | SZ | ST | AC | $P3$ | $L3$ | SZ | ST | AC | $P4$ | $L4$ | SZ | ST | AC | $P5$ | $L5$ |
| 1 | 96 | 1 | 18 | 96 | 85 | 96 | 49 | 180 | 180 | 81 | 96 | 2 | 255 | 259 | 86 | 96 | 3 | 264 | 342 | 69 | 96 | 2 | 298 | 409 |
| 2 | 94 | 1 | 57 | 94 | 74 | 87 | 40 | 160 | 160 | 78 | 92 | 10 | 237 | 228 | 99 | 94 | 15 | 276 | 312 | 62 | 94 | 1 | 277 | 373 |
| 3 | 81 | 1 | 15 | 81 | 65 | 39 | 22 | 103 | 124 | 81 | 76 | 15 | 156 | 190 | 93 | 76 | 55 | 168 | 228 | 65 | 76 | 58 | 175 | 235 |
| 4 | 85 | 1 | 17 | 85 | 61 | 16 | 18 | 119 | 128 | 87 | 74 | 45 | 160 | 170 | 71 | 79 | 17 | 203 | 224 | 95 | 81 | 55 | 215 | 264 |
| 5 | 99 | 1 | 89 | 99 | 78 | 39 | 72 | 116 | 116 | 60 | 47 | 13 | 200 | 163 | 63 | 99 | 15 | 209 | 211 | 66 | 99 | 14 | 216 | 263 |
| 6 | 82 | 1 | 71 | 82 | 97 | 70 | 69 | 166 | 166 | 95 | 74 | 71 | 168 | 190 | 79 | 75 | 51 | 181 | 218 | 69 | 81 | 43 | 201 | 244 |
| 7 | 84 | 1 | 41 | 84 | 91 | 83 | 16 | 173 | 173 | 72 | 84 | 53 | 174 | 192 | 82 | 84 | 13 | 234 | 261 | 73 | 84 | 57 | 236 | 277 |
| 8 | 79 | 1 | 51 | 79 | 78 | 54 | 8 | 131 | 149 | 62 | 57 | 20 | 160 | 191 | 89 | 69 | 33 | 213 | 247 | 99 | 73 | 63 | 243 | 283 |
| 9 | 87 | 1 | 19 | 87 | 64 | 48 | 5 | 111 | 146 | 60 | 73 | 12 | 132 | 194 | 68 | 73 | 63 | 140 | 199 | 99 | 87 | 88 | 185 | 210 |
| 10 | 69 | 1 | 56 | 69 | 97 | 17 | 84 | 113 | 113 | 75 | 53 | 26 | 176 | 162 | 91 | 56 | 25 | 212 | 228 | 70 | 56 | 59 | 213 | 239 |
| 11 | 61 | 1 | 21 | 61 | 80 | 56 | 13 | 135 | 135 | 72 | 59 | 63 | 139 | 144 | 84 | 61 | 42 | 186 | 186 | 91 | 61 | 73 | 187 | 204 |
| 12 | 73 | 1 | 63 | 73 | 83 | 31 | 59 | 113 | 113 | 82 | 40 | 26 | 177 | 169 | 77 | 43 | 75 | 179 | 171 | 60 | 48 | 6 | 215 | 225 |
| 13 | 96 | 1 | 61 | 96 | 73 | 31 | 44 | 103 | 125 | 77 | 82 | 37 | 158 | 165 | 72 | 95 | 55 | 166 | 182 | 85 | 95 | 54 | 210 | 213 |
| 14 | 92 | 1 | 12 | 92 | 88 | 55 | 28 | 142 | 152 | 75 | 67 | 51 | 165 | 176 | 98 | 71 | 87 | 168 | 187 | 93 | 92 | 56 | 221 | 224 |
| 15 | 93 | 1 | 78 | 93 | 91 | 7 | 80 | 97 | 104 | 63 | 34 | 42 | 117 | 125 | 60 | 70 | 47 | 129 | 138 | 62 | 92 | 51 | 153 | 153 |
| 16 | 97 | 1 | 34 | 97 | 60 | 78 | 33 | 137 | 137 | 62 | 89 | 31 | 150 | 168 | 86 | 91 | 72 | 176 | 182 | 90 | 94 | 23 | 250 | 249 |
| 17 | 83 | 1 | 39 | 83 | 81 | 71 | 77 | 151 | 151 | 93 | 71 | 74 | 182 | 170 | 70 | 77 | 27 | 232 | 213 | 63 | 80 | 26 | 253 | 250 |
| 18 | 86 | 1 | 64 | 86 | 62 | 74 | 38 | 135 | 135 | 71 | 75 | 68 | 145 | 145 | 66 | 82 | 43 | 170 | 168 | 92 | 82 | 34 | 231 | 226 |
| 19 | 88 | 1 | 56 | 88 | 87 | 21 | 79 | 107 | 107 | 95 | 71 | 93 | 165 | 165 | 88 | 74 | 19 | 299 | 234 | 90 | 78 | 5 | 337 | 319 |
| 20 | 95 | 1 | 33 | 95 | 91 | 14 | 66 | 104 | 120 | 68 | 57 | 58 | 124 | 130 | 86 | 88 | 23 | 236 | 193 | 74 | 89 | 59 | 237 | 208 |

by forming a cascading chain of transactions without causing any immediate abort or back-off to any transactions. Moreover, the proposed method doesn't include any additional contention manager as the transactions are able to resolve conflicts on their won. The uniqueness of the proposed implementation in this paper is in achieving reduced number of aborts for write transaction on top of obstruction-free non-blocking architecture. No such similar approach is found in the existing literature.



## References

1. Attiya H, Milani A (2012) Transactional scheduling for read-dominated workloads. J Parallel Distrib Comput 72(10):1386–1396
2. Aydonat U, Abdelrahman TS (2012) Relaxed concurrency control in software transactional memory. Parallel Distrib Syst IEEE Trans 23(7):1312–1325
3. Bernstein PA, Newcomer E (2009) Principles of transaction processing, 2nd edn., chap. 8. Morgan Kaufmann, San Francisco, California
4. Dolev S, Fatourou P, Kosmas E (2013) Abort free semantictm by dependency aware scheduling of transactional instructions. In: 8th ACM SIGPLAN workshop on transactional computing (Transact). Citeseer, Houston
5. Dragojevi CA, Guerraoui R, Kapalka M (2009) Stretching transactional memory. In: ACM sigplan notices, vol 44. ACM, pp 155–165
6. Felber P, Fetzer C, Marlier P, Riegel T (2010) Time-based software transactional memory. Parallel Distrib Syst IEEE Trans 21(12):1793–1807
7. Frank J, Chun R (2008) Adaptive software transactional memory: a dynamic approach to contention management. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA, pp 40–46
8. Ghosh A, Chaki N (2013) Design of a new oftm algorithm towards abort-free execution. Distributed computing and internet technology. Springer, Berlin, pp 255–266
9. Guerraoui R, Kapalka M (2008) On obstruction-free transactions. In: Proceedings of the twentieth annual symposium on parallelism in algorithms and architectures. ACM, pp 304–313
10. Harris T, Larus J, Rajwar R (2010) Transactional memory, 2nd edn., chap. 2. Morgan & Claypool Series, San Rafael, pp 46–47
11. Herlihy M, Luchangco V, Moir M (2003) Obstruction-free synchronization: double-ended queues as an example. In: Distributed computing systems, 2003. Proceedings 23rd international conference on. IEEE, pp 522–529
12. Herlihy M, Luchangco V, Moir M, Scherer III WN (2003) Software transactional memory for dynamic-sized data structures. In: Proceedings of the twenty-second annual symposium on principles of distributed computing. ACM, pp 92–101
13. Levy E, Korth HF, Silberschatz A (1991) An optimistic commit protocol for distributed transaction management, vol 20. ACM
14. Marathe VJ, Scherer WN III, Scott ML (2005) Adaptive software transactional memory. Distributed Computing. Springer, Berlin, pp 354–368
15. Marathe VJ, Scott M (2004) A qualitative survey of modern software transactional memory systems. University of Rochester Computer Science Dept., Tech. Rep
16. Marathe VJ, Spear MF, Heriot C, Acharya A, Eisenstat D, Scherer III WN, Scott ML (2015) The rochester software transactional memory runtime. http://www.cs.rochester.edu/research/synchronization/rstm
17. Perelman D, Fan R, Keidar I (2010) On maintaining multiple versions in stm. In: Proceedings of the 29th ACM SIGACT-SIGOPS symposium on principles of distributed computing. ACM, pp 16–25

18. Scherer III WN, Scott ML (2004) Contention management in dynamic software transactional memory. In: PODC workshop on concurrency and synchronization in java programs, pp 70–79
19. Scherer III WN, Scott ML (2005) Advanced contention management for dynamic software transactional memory. In: Proceedings of the twenty-fourth annual ACM symposium on principles of distributed computing. ACM, pp 240–248
20. Shavit N, Touitou D (1997) Software transactional memory. Distrib Comput 10(2):99–116
21. Tabba F, Moir M, Goodman JR, Hay AW, Wang C (2009) Nztm: nonblocking zero-indirection transactional memory. In: Proceedings of the twenty-first annual symposium on parallelism in algorithms and architectures. ACM, pp 204–213