Efficient means of Achieving Composability using Transactional Memory

Sathya Peri, Ajay Singh and Archit Somani

Department of Computer Science & Engineering, IIT Hyderabad, India
(sathya_p, cs15mtch01001, cs15resch01001)@iith.ac.in

Abstract

Major focus of software transaction memory systems (STMs) has been to facilitate the multiprocessor programming and provide parallel programmers an abstraction for speedy and efficient development of parallel applications. To this end different models for incorporating object/higher level semantics into STM have recently been proposed in transactional boosting, transactional data structure library, open nested transactions and abstract nested transactions.

We build an alternative object model STM (OSTM) by adopting the transactional tree model of Weikum et al. originally given for databases and extend the current work by proposing comprehensive legality definitions and conflict notion which allows efficient composability of OSTM. We first time show the proposed OSTM to be co-opaque.

We build OSTM using chained hash table data structure. Lazyskip-list is used to implement chaining using lazy approach. We notice that major concurrency hotspot is the chaining data structure within the hash table. Lazyskip-list is time efficient compared to lists in terms of traversal overhead by average case $O(\log(n))$. We optimise lookups as they are validated at the instant they execute and they are not validated again in commit phase. This allows lookup dominated transactions to be more efficient and at the same time co-opaque.

Keywords and phrases Software transactional memory, Lazyskip-list, Legality, Conflict-notion, Composability, Co-opacity, Opacity

Digital Object Identifier 10.4230/LIPIcs...

Introduction

Software Transaction Memory Systems (STMs) are a convenient programming interface for a programmer to access shared memory without worrying about concurrency issues [9, 18]. Concurrently executing transactions access shared memory through the interface provided by the STMs. Thus, the programmer can now focus on harnessing optimum parallelism from the application instead of worrying about the locking, races and deadlocks.

Most of the STMs proposed in the literature are specifically based on read/write primitive operations (or methods) on memory buffers (or memory registers). These STMs typically export the following methods: $t_{\text{begin}}$ which begins a transaction, $t_{\text{read}}$ which reads from a buffer, $t_{\text{write}}$ which writes onto a buffer, $\text{tryC}$ which validates the operations of the transaction and tries to commit. If validation is successful then it returns commit otherwise STMs export $\text{tryA}$ which returns abort. We refer to these as Read-Write STMs or RWSTMs. As a part of the validation, the STMs typically check for conflicts among the operations. Two operations are said to be conflicting if at least one of them is a write (or update) operation. Normally, the order of two conflicting operations can not be commutated. On the other hand, Object-based STM or OSTM operate on higher level objects rather than read & write operations on memory locations. They include more complicated operations such as enq/deq on queue objects, push/pop on stack objects etc.

It was shown in databases that object-level systems provide greater concurrency than read-write systems [21]. Chap 6]. Harris et al. [3] adopted this concept in STMs along with Herlihy et al. [16, 10].
We would like to propose an alternative model to achieve composability with greater concurrency for STM s by considering higher-level objects which milk the richer semantics of object level operations. We motivate this with an interesting example.

Consider an OSTM operating on the hash-table object exports the following methods: \texttt{t\_begin} which begins a transaction (same as in RWSTMs), \texttt{t\_insert} which inserts a value for a given key, \texttt{t\_delete} which deletes the value associated with the given key, \texttt{t\_lookup} which looks up the value associated with the given key and \texttt{tryC} which validates the operations of the transaction.

A simple way to implement the hash-table object is using a list where each element of the list stores the \{key, value\} pair. The elements of the list are sorted by their keys similar to the set implementations discussed in [8, Chap 9]. It can be seen that the underlying list is a concurrent data-structure (DS) manipulated by multiple transactions (and hence threads). So we have adopted the lazy-list approach \cite{7} to implement the operations of the list denoted as: \texttt{list\_insert}, \texttt{list\_del} and \texttt{list\_lookup} (referred as contains in \cite{7}). Thus when a transaction invokes \texttt{t\_insert}, \texttt{t\_delete} and \texttt{t\_lookup} methods, the STM internally invokes the \texttt{list\_insert}, \texttt{list\_del} and \texttt{list\_lookup} methods respectively.

Consider an instance of list in which the nodes with keys \(k_2, k_5, k_7, k_8\) are present in the hash-table as shown in Figure 1(i) and transactions \(T_1\) and \(T_2\) are concurrently executing \texttt{t\_lookup}(\(k_5\)), \texttt{t\_delete}(\(k_7\)) and \texttt{t\_lookup}(\(k_8\)) as shown in Figure 1(ii). In our representation, we abbreviate \texttt{t\_insert} as \(i\), \texttt{t\_delete} as \(d\) and \texttt{t\_lookup} as \(l\). For simplicity, we refer to nodes of the list by their keys. In this setting, suppose a transaction \(T_1\) of OSTM invokes methods \texttt{t\_lookup} on the keys \(k_5, k_8\). This would internally cause the OSTM to invoke \texttt{list\_lookup} method on keys \(k_2, k_5\) and \((k_2, k_5, k_7, k_8)\) respectively.

Concurrently, suppose transaction \(T_2\) invokes the method \texttt{t\_delete} on key \(k_7\) between the two \texttt{t\_lookups} of \(T_1\). This would cause, OSTM to invoke \texttt{list\_del} method of list on \(k_7\). Since, we are using lazy-list approach on the underlying list, \texttt{list\_del} involves pointing the next field of element \(k_5\) to \(k_8\) and marking element \(k_7\) as deleted. Thus \texttt{list\_del} of \(k_7\) would execute the following sequence of read/write level operations- \(r(k_2)r(k_5)r(k_7)w(k_5)w(k_7)\) where \(r(k_5), w(k_5)\) denote read & write on the element \(k_5\) with some value respectively. The execution of OSTM denoted as a history can be represented as a transactional forest as shown in Figure 1(ii). Here the execution of each transaction is a tree.

In this execution, we denote the read-write operations (leaves) as layer-0 and \texttt{t\_lookup}, \texttt{t\_delete} methods as layer-1. Consider the history (execution) at layer-0 (while ignoring higher-level operations), denoted as \(H0\). It can be verified this history is not opaque\cite{2}. This is because between the two reads of \(k_5\) by \(T_1, T_2\) writes to \(k_5\). It can be seen that if history \(H0\) is input to a RWSTMs one of the transactions among \(T_1\) or \(T_2\) would be aborted to ensure correctness (in this case opacity\cite{2}).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{motivational_example.png}
\caption{Motivational example for OSTMs}
\end{figure}

On the other hand consider the history \(H1\) at layer-1 consisting of \texttt{t\_lookup}, \texttt{t\_delete} methods while ignoring the underlying read/write operations. We ignore the underlying read & write operations.
since they do not overlap (referred to as pruning in [21, Chap 6]). Since these methods operate on different keys, they are not conflicting and can be re-ordered either way. Thus, we get that $H_1$ is opaque[2] with $T_1T_2$ (or $T_2T_1$) being an equivalent serial history.

The important idea in the above argument is ignoring lower-level operations since they do not overlap. Harris et al. referred to it as benign-conflicts[3]. This history clearly shows the advantage of considering STMs with higher level operations in this case they are $t\text{\_}insert$, $t\text{\_}delete$ and $t\text{\_}lookup$. With object level modeling of histories, we get a higher number of acceptable schedules than read/write model. This is because not all conflicts at the lower level matter at the higher level.

Now consider an application where we have two hash-tables, $ht_1$ and $ht_2$ such that a process $p_1$ need to delete $k_5$ from $ht_1$ and insert it into $ht_2$ and another process $p_2$ looks up $k_5$. Now if we do not have any synchronization mechanism for such an application these operations would not compose and would leave the application in incorrect state (i.e. if $p_2$ sees the intermediate state of the system where $p_1$ has deleted the $k_5$ from $ht_1$ but has not inserted in the $ht_2$) even though the individual operations are atomic. Our OSTM ensures that the sequence of operations compose powered by the legality and conflict notion and the correctness proofs of the histories generated. Following is the summary of our contribution:

- We build an alternative theoretical model for efficiently transactifying the concurrent data structures using their semantic information such that they are composable [4] too. We call it object software transactional memory system (OSTM).
- We propose legality definitions and the notion of conflicts for object histories generated by OSTM.
- We design the OSTM with hash-table where chaining is implemented via lazyskip-list and we show that the design approach saves traversal overhead for the operations and helps in optimized meta information management such that executions are guaranteed to be correct.
- We provide in-depth proof of correctness starting from layer-0 (operational level) to the layer-1 (transactional level) executions generated by the proposed OSTM. And first time we show that OSTM is guaranteed to be co-opaque[12].

Roadmap. We narrate our system model and legality of OSTM in Section 2. Section 3 depicts conflict notion and in Section 4, we present detailed data structure and algorithm design of OSTM. In Section 5, we outline correctness of OSTM. Section 6 explains related work and finally we conclude in Section 7.

2 Building System Model

In this paper, we assume that our system consists of finite set of $P$ processors, accessed by a finite number of $n$ threads that run in a completely asynchronous manner and communicate using shared objects. The threads communicate with each other by invoking higher-level methods on the shared objects and getting corresponding responses. Consequently, we make no assumption about the relative speeds of the threads. We also assume that none of these processors and threads fail or crash abruptly.

Events: We assume that the threads execute atomic events. Similar to Lev-Ari et. al.[14][15], we assume that these events by different threads are (1) read/write on shared/local memory objects, (2) method invocations (or $inv$) event & responses (or $rsp$) event on higher level shared-memory objects.

Global States: We define the global state or state of the system as the collection of local and shared variables across all the threads in the system. The system starts with an initial global state. We assume that all the events executed by different threads are totally ordered. Each update event transitions the global state of the system leading to a new global state.

Methods: Within a transaction, a process can invoke layer-1 (transactional) methods on a hash-table transaction object. A hash-table($ht$) consists of multiple key-value pairs of the form ($k,v$). The keys and values are respectively from sets $K$ and $V$. The methods that a transaction $T_i$ can invoke are:
Transactions: Following the notations used in database multi-level transactions [21], we model a transaction as a two-level tree. The layer-0 consist of read/write events and layer-1 of the tree consists of methods invoked by transaction. Having informally explained a transaction, we formally define a transaction as the tuple \( \langle evts(T), \prec_T \rangle \). Here \( evts(T) \) are all the read/write events at layer-0 of the transaction. \( \prec_T \) is a total order among all the events of the transaction.

We denote the first and last events of a transaction \( T_i \) as \( T_i.firstEvt \) and \( T_i.lastEvt \). Given any other read-write event \( rw \) in \( T_i \), we assume that \( T_i.firstEvt \prec_T rw \prec_T T_i.lastEvt \). All the methods of \( T_i \) denoted as \( methods(T_i) \). We assume that for any method \( m \) in \( methods(T_i) \), \( evts(m) \) is a subset of \( evts(T_i) \) and \( \prec_m \) is a subset of \( \prec_T \).

Histories: A history is a sequence of events belonging to different transactions. The collection of events is denoted as \( evts(H) \). Similar to a transaction, we denote a history \( H \) as tuple \( \langle evts(H), \prec_H \rangle \) where all the events are totally ordered by \( \prec_H \). The set of methods that are in \( H \) is denoted by \( methods(H) \). A method \( m \) is incomplete if \( inv(m) \) is in \( evts(H) \) but not its corresponding response event. Otherwise \( m \) is complete in \( H \).

Coming to transactions in \( H \), the set of transactions in \( H \) as \( txns(H) \). The set of committed (resp., aborted) transactions in \( H \) is denoted by \( committed(H) \) (resp., \( aborted(H) \)). The set of incomplete or live transactions in \( H \) is denoted by \( incomp(H) = live(H) = txns(H) - committed(H) - aborted(H) \). On the other hand, the set of terminated transactions are those which have either committed or aborted and is denoted by \( term(H) = committed(H) \cup aborted(H) \).

The relation between the events of transactions & histories is analogous to the relation between methods & transactions. We assume that for any transaction \( T \) in \( txns(H), evts(T) \) is a subset of \( evts(H) \) and \( \prec_T \) is a subset of \( \prec_H \). Formally, \( \forall T \in txns(H) : (evts(T) \subseteq evts(H)) \land (\prec_T \subseteq \prec_H) \). We denote two histories \( H_1, H_2 \) as equivalent if their events are the same, i.e., \( evts(H_1) = evts(H_2) \). A history \( H \) is qualified to be well-formed if: (1) all the methods of a transaction \( T_i \) in \( H \) are totally ordered, i.e. a transaction invokes a method only after it receives a response of the previous method invoked by it (2) \( T_i \) does not invoke any other method after it received an \( A \) response or after \( tryC(ok) \) method.

Sequential Histories: A history \( H \) is said to be sequential (term used in [12][13]) or linearized [11] if all the methods in it are complete and isolated. From now onwards, most of our discussion would relate to sequential histories.

Since in sequential histories all the methods are isolated, we treat each method as whole without referring to its \( inv \) and \( rsp \) events. For a sequential history \( H \), we construct the completion of \( H \), denoted \( \overline{H} \), by inserting \( tryA_k(A) \) immediately after the last method of every transaction \( T_i \in incomp(H) \). Since all the methods in a sequential history are complete, this definition only has to take care of completing transactions. Consider a sequential history \( H \). Let \( m_{ij}(ht, k, v/nil) \) be the first method of \( T_i \) in \( H \) operating on the key \( k \) as \( H.firstKeyMth\langle \langle ht, k \rangle, T_i \rangle \), where \( m_{ij} \) stands for \( j^{th} \) method of \( i^{th} \) transaction. For a method \( m_{kz}(ht, k, v) \) which is not the first method on \( \langle ht, k \rangle \) of \( T_i \) in \( H \), we denote its previous method on \( k \) of \( T_i \) as \( m_{ij}(ht, k, v) = H.prevKeyMth(m_{kz}, T_i) \).

Real-time Order & Serial Histories: Given a history \( H \), \( \prec_H \) orders all the events in \( H \). For two complete methods \( m_{ij}, m_{pq} \) in \( methods(H) \), we denote \( m_{ij} \prec_H m_{pq} \) if \( rsp(m_{ij}) \prec_H inv(m_{pq}) \). Here MR stands for method real-time order. It must be noted that all the methods of the same
We assume that when a transaction \( T_i \), \( T_p \) in \( \text{term}(H) \), we denote \((T_i, T_p) \) if \((T_i, last \text{Ev} < H)\) \( T_p, first \text{Ev} \)). Here TR stands for transactional real-time order.

We define a history \( H \) as serial \([17]\) or \( \text{t-sequential} \([13]\) if all the transactions in \( H \) have terminated and can be totally ordered w.r.t. \( \prec_{TR} \), i.e. all the transactions execute one after the other without any interleaving. Intuitively, a history \( H \) is serial if all its transactions can be isolated. Formally, \( \langle \{H \text{ is serial} \} \implies (\forall T_i \in \text{txn}(H) : (T_i \in \text{term}(H)) \land (\forall T_p \in \text{txn}(H) : (T_i \prec_{TR} T_p) \lor (T_p \prec_{TR} T_i)) \rangle \).

Since all the methods within a transaction are ordered, a serial history is also sequential. Refer Figure [15] in Appendix A to show a serial history.

**Legal Histories:** We define legality of rv_methods, \( t\_delete \) & \( t\_lookup \) on sequential histories.

Consider a sequential history \( H \) having a rv_method \( rvm_{ij}(ht, k, v) \) (with \( v \neq \text{nil} \)) belonging to transaction \( T_i \). We define this \( rvm \) method to be legal if:

1. If the \( rvm_{ij} \) is not first method of \( T_i \) to operate on \( \langle ht, k \rangle \) and \( m_{ix} \) is the previous method of \( T_i \) to operate on \( \langle ht, k \rangle \). Formally, \( rvm_{ij} \neq H.first \text{Mth}(\langle ht, k \rangle, T_i) \land (m_{ix}(ht, k, v') = H.prev \text{Mth}(\langle ht, k \rangle, T_i)) \) (where \( v' \) could be nil). Then,
   a. if \( m_{ix}(ht, k, v') \) is a \( t\_insert \) method i.e. \( t\_insert_{ix}(ht, k, v') \) then \( v = v' \).
   b. if \( m_{ix}(ht, k, v') \) is a \( t\_lookup \) method i.e. \( t\_lookup_{ix}(ht, k, v') \) then \( v = v' \).
   c. if \( m_{ix}(ht, k, v') \) is a \( t\_delete \) method i.e. \( t\_delete_{ix}(ht, k, v' / \text{nil}) \) then \( v = \text{nil} \).

   In this case, we denote \( m_{ix} \) as the last update method of \( rvm_{ij} \), i.e., \( m_{ix}(ht, k, v') = H.lastUpdt(rvm_{ij}(ht, k, v)) \).

2. If \( rvm_{ij} \) is the first method of \( T_i \) to operate on \( \langle ht, k \rangle \) and \( v \) is not nil. Formally, \( rvm_{ij}(ht, k, v) = H.first \text{Mth}(\langle ht, k \rangle, T_i) \land (v \neq \text{nil}) \). Then,
   a. There is a \( t\_insert \) method \( t\_insert_{pq}(ht, k, v) \) in \( \text{methods}(H) \) such that \( T_p \) committed before \( rvm_{ij} \). Formally, \( \langle \exists t\_insert_{pq}(ht, k, v) \in \text{methods}(H) : tryC_p \prec_{TR} rvm_{ij} \rangle \).
   b. There is no other update method \( up_{pq} \) of a transaction \( T_x \) on \( \langle ht, k \rangle \) in \( \text{methods}(H) \) such that \( T_x \) committed after \( T_p \) but before \( rvm_{ij} \). Formally, \( \langle \forall up_{pq} \langle ht, k, v'' \rangle \in \text{methods}(H) : tryC_p \prec_{TR} rvm_{ij} \rangle \).

   In this case, we denote \( tryC_p \) as the last update method of \( rvm_{ij} \), i.e., \( tryC_p(ht, k, v) = H.lastUpdt(rvm_{ij}(ht, k, v)) \).

3. If \( rvm_{ij} \) is the first method of \( T_i \) to operate on \( \langle ht, k \rangle \) and \( v \) is nil. Formally, \( rvm_{ij}(ht, k, v) = H.first \text{Mth}(\langle ht, k \rangle, T_i) \land (v = \text{nil}) \). Then,
   a. There is \( t\_delete \) method \( t\_delete_{pq}(ht, k, v') \) in \( \text{methods}(H) \) such that \( T_p \) (which could be \( T_0 \) as well) committed before \( rvm_{ij} \). Formally, \( \langle \exists t\_delete_{pq}(ht, k, v') \in \text{methods}(H) : tryC_p \prec_{TR} rvm_{ij} \rangle \). Here \( v' \) could be nil.
   b. There is no other update method \( up_{pq} \) of a transaction \( T_x \) on \( \langle ht, k \rangle \) in \( \text{methods}(H) \) such that \( T_x \) committed after \( T_p \) but before \( rvm_{ij} \). Formally, \( \langle \forall up_{pq} \langle ht, k, v'' \rangle \in \text{methods}(H) : tryC_p \prec_{TR} rvm_{ij} \rangle \).

   In this case similar to step 2, we denote \( tryC_p \) as the last update method of \( rvm_{ij} \), i.e., \( tryC_p(ht, k, v) = H.lastUpdt(rvm_{ij}(ht, k, v)) \).

We assume that when a transaction \( T_i \) operates on key \( k \) of a hash-table \( ht \), the result of this method is stored in local logs of \( T_i \) for later methods to reuse. Thus, only the first \( rv \_method \) operating on \( \langle ht, k \rangle \) of \( T_i \) accesses the shared-memory. The other \( rv \_methods \) of \( T_i \) operating on \( \langle ht, k \rangle \) do not access the shared-memory and they see the effect of the previous method from the local logs. This idea is utilized in step [1] of legality. With reference to step 2 and step 3, it is possible that \( T_x \) could have aborted before \( rvm_{ij} \). For step 3, since we are assuming that transaction \( T_0 \) has invoked a \( t\_delete \) method on all the keys used of all hash-table objects, there exists at least one \( t\_delete \)
method for every rv_method on k of ht. For more details please refer Figure 16, Figure 17, Figure 18 and Figure 19 in Appendix A. We formally prove legability in Lemma 29 in Appendix D and then we finally show that OSTM histories are co-opaque as defined in Definition 1.

Coming to t_insert methods, since a t_insert method always returns ok as they overwrite the node if already present therefore they always take effect on the ht. Thus, we denote all t_insert methods as legal. We denote a sequential history H as legal if all its rvm methods are legal. While defining legality of a history, we are only concerned about rvm (t_lookup and t_delete) methods since all t_insert methods are by default legal.

**Correctness-Criteria & Opacity**: A correctness-criterion is a set of histories. A history H satisfying a correctness-criterion has some desirable properties. A popular correctness-criterion is opacity [2]. A sequential history H is opaque if there exists a serial history S such that: (1) S is equivalent to H, i.e., evts(H) = evts(S) (2) S is legal and (3) S respects the transactional real-time order of H, i.e., $\prec^R_H \subseteq \prec^R_S$.

### 3 Conflict Notion

**Motivation towards new conflict notion**: As we discussed in Figure 1(ii), some lower level conflicts can be ignored at the higher level. So, we defined following conflict notion for proving the correctness (opacity, to be precise co-opacity[12]) of higher level. We say two transactions $T_i, T_j$ of a sequential history H are in conflict if atleast one of the following conflicts holds:

- **u-u conflict**: (1) $T_i \& T_j$ are committed and (2) $T_i \& T_j$ update the same key k of the hash-table, ht, i.e., $(\langle ht, k \rangle \in \text{updtSet}(T_i)) \& (\langle ht, k \rangle \in \text{updtSet}(T_j))$, where updtSet($T_i$) is update set of $T_i$. (3) $T_i$'s tryC completed before $T_j$'s tryC, i.e., $\prec^C_H \prec^C_{\text{tryC}_j}$.

- **u-rv conflict**: (1) $T_i$ is committed (2) $T_j$ updates the key k of hash-table, ht. $T_j$ invokes a rv_method rvm$_{jy}$ on the same key k of hash-table ht which is the first method on $\langle ht, k \rangle$. Thus, $(\langle ht, k \rangle \in \text{updtSet}(T_i)) \& (\text{rvm}_{jy}(ht, k, v) \in \text{rvSet}(T_j)) \& (\text{rvm}_{jy}(ht, k, v) = H.i\text{firstKeyMethod}(\langle ht, k \rangle, T_j))$, where rvSet($T_j$) is return value set of $T_j$. (3) $T_j$'s tryC completed before $T_j$'s rvm, i.e., $\prec^M_H \prec^M_{\text{rvm}_{jy}}$.

- **rv-u conflict**: (1) $T_j$ is committed (2) $T_i$ invokes a rv_method on the same key k of hash-table ht which is the first method on $\langle ht, k \rangle$. $T_i$ updates the key k of hash-table, ht. Thus, $(\text{rvm}_{ix}(ht, k, v) \in \text{rvSet}(T_i)) \& (\text{rvm}_{ix}(ht, k, v) = H.i\text{firstKeyMethod}(\langle ht, k \rangle, T_i)) \& (\langle ht, k \rangle \in \text{updtSet}(T_j))$ (3) $T_j$'s rvm completed before $T_j$’s tryC, i.e., $\prec^M_H \prec^M_{\text{rvm}_{ix}}$.

**Definition 1.** Co-opacity : A sequential history H is conflict-opaque (or co-opaque) if there exists a serial history S such that: (1) S is equivalent to H, i.e., evts(H) = evts(S) (2) S is legal and (3) S respects the transactional real-time order of H, i.e., $\prec^R_H \subseteq \prec^R_S$ and (4) S preserves conflicts (i.e. $\prec^C_S \subseteq \prec^C_H$) [12].

A rv_method rvm$_{ij}$ conflicts with a tryC method only if rvm$_{ij}$ is the first method of $T_i$ that operates on hash-table with a given key. Thus the conflict notion is defined only by the methods that access the shared memory. (tryC$_i$.tryC$_j$, (tryC$_i$.t_lookup$_j$), (t_lookup$_i$.tryC$_j$), (tryC$_i$.t_delete$_j$) and (t_delete$_i$.tryC$_j$) can be the conflicting methods. Based on these conflicts we build a conflict graph as follows:

**Graph Characterization:** Let conflict graph (CG) be set of $\langle V, E \rangle$ pair where $V \in \text{txns}(H)$ and $E$ can be of following types:

- **conflict edges**: $\{ (T_i, T_j) : (T_i, T_j) \in \text{conflict}(H) \}$. Where, conflict(H) is an ordered pair of transactions such that the transactions have one of the above pair of conflicts.

- **real-time edge**: $\{ (T_i, T_j) : T_i \prec^R_H T_j \}$
The legality and conflict notion established here are used to prove that histories generated by the OSTM are correct or co-opaque\cite{2} in Section 5.

4 OSTM Design

We design the OSTM using hash-table where chaining is done using lazyskip-list. Here, major concurrency hot-spot is the chaining data-structure. Lazyskip-list based chain implementation assumes that there are head and tail nodes which are immutable. The value of key in head is \(-\infty\) and the value of key in tail is \(+\infty\). Lazyskip-list have two types of nodes 1) live node: represents the nodes which are not marked (not deleted) and 2) dead node: represent the nodes which are marked (i.e. logically deleted). Also, each node in lazyskip-list has two links namely, BL(blue links) and RL(red links) which can be thought of as it’s two levels. All live nodes are accessed via BL and all the nodes including dead nodes are accessed via RL from the head. Every node of lazyskip-list is in increasing order of it key.

We now explain the search mechanism over such a lazyskip-list. A node is always first probed in BL. If the node is present in BL then it will store location( found over the BL) of the node corresponding to the key in local log otherwise it will search through RL within the same location identified by traversing the BL. For example, let say we search \(k_5\) in Figure 2. We observe that \(k_5\) is not present in BL and we stop at location (\(-\infty\) and \(k_7\) the predecessor and successor respectively for \(k_5\)). Now we try to search the \(k_5\) over the RL between \(-\infty\) and \(k_7\) (because all nodes are in increasing order of their keys). This chaining data structure is our design choice because it has inherent advantage of being search efficient. To illustrate this, consider the example in Figure 2 for searching key \(k_8\) in lazyskip-list. Key \(k_8\) is present in BL so we do not need to traverse keys \(k_1, k_3\) and \(k_6\) which saves significant search time. Had it been a simple lazy list (Figure 3) searching \(k_8\) would have involved unnecessarily traversal over dead nodes represented by \(k_1, k_3\) and \(k_6\).

Why we need to maintain dead nodes? Dead nodes are either the deleted nodes or the nodes inserted by the rv_method over the course of their execution. We need the dead nodes to store the meta information which is used to satisfy opacity\cite{2} of the OSTM execution. We further explain this using example in Figure 5 and Figure 6.

History H shown in Figure 5 is not opaque because we can’t come up with any serial order between \(T_1\) and \(T_2\). In order to make it opaque \(lu_1(ht, k_1, Nil)\) needs to be aborted. And \(lu_1(ht, k_1, Nil)\) can only be aborted if OSTM scheduler knows that a conflicting operation \(del_2(ht, k_1, v_0)\) has already been scheduled violating the time-order\cite{21}. One way to have this information is that if the node represented by \(k_1\) records the time-stamp of the delete operation, so that the scheduler realizes the
violation and aborts \( \text{lu}_1(\text{ht}, k_1, \text{Nil}) \) to ensure opacity. Thus with help of information provided by the dead nodes we can ensure \( H_1 \): \( T_1 \) followed by \( T_2 \) is the opaque history as depicted in the Figure 5

These dead nodes can always be reused if any insert arrives later in the transaction. Next we discuss the data structure and algorithm which powers the \( \text{OSTM} \).

4.1 \( \text{OSTM} \) data-structure design

In proposed \( \text{OSTM} \), we use thread local DS which is private to each thread for logging the local execution and shared memory DS which is concurrently accessed by multiple transactions to communicate the meta information logged for validation of the methods.

4.1.1 Thread local DS

Each transaction \( T_i \) maintains local log of type \text{txlog}, which consists of \text{t_id} and \text{tx_status} of the transaction. Transactions can have live, commit or abort as their status signifying that transaction is executing, has successfully committed or has aborted due to some method failing the validation respectively.

```cpp
class txlog{
private :
    int t_id;
    STATUS tx_status;
    vector<ll_entry> ll_list;
public :
    txlog(); ~txlog(); findInLL();
    getLlList(); handleAbort();
};
```

The local log also maintains a list(\text{ll_list}) of meta information of each method a transaction executes in its life time. Each entry of the \text{ll_list} is of type \text{ll_entry} which logs 1) key and value a method operates on, 2)opn: name of the method, 3)op_status: method’s status (OK, FAIL) and 4) preds, currs: its location over the lazyskip-list.

We say a method identifies its location over the lazyskip-list when it finds the predecessor and successor nodes over the \( BL \) and \( RL \) respectively. We represent predecessor as \( \langle k_m, k_n \rangle \) (\( k_m \) is blue node reachable by \( BL \) and \( k_n \) is red node reachable by \( RL \)) and successor as \( \langle k_p, k_q \rangle \) (\( k_p \) is red node reachable by \( RL \) and \( k_q \) is blue node reachable by blue node) respectively. Here, \( \langle k_m, k_n \rangle \) are predecessor(pred[0]) and current(curr[0]) node for \( BL \) and \( \langle k_p, k_q \rangle \) are predecessor(pred[1]) and current(curr[0]) node for \( RL \). We use word location with \text{preds} and \text{currs} interchangeably in rest of the paper.

Class \text{ll_entry} also shows the getter and setter methods for each of the member variables which are self explanatory. Interested reader can find their description at table 1 in appendix.

```cpp
class ll_entry{
private :
    int obj_id, key, value;
    node* preds, currs;
    STATUS op_status;
    operation_name opn;
public :
    getOpn(); getPreds&Currs(); getOpStatus();
};
```
4.1.2 Shared memory DS:

**OSTM** shared memory is the chained hash-table where each node of the chain (lazyskip-list) is a key-value pairs of the form \( \langle k, v \rangle \). Most of the notations used here are derived from [20]. A node \( n \) when created is initialized as follows: (1) key and val is the key and val of the method that creates the node (2) rednext and bluenext are set to nil (3) marked is set to false (4) lock is null (5) \( max \_ts \) is initialized to 0.

```c
struct node{
    int key, value;  // key-value pair
    bool marked;    // for transaction validity
    lock;           // for synchronization
    node rednext;   // for atomicity
    node bluenext;  // for serializability
};
node* shared_ht[]; /*Each array index is a lazyskip list chain*/
vector <shared_ht*> object_list; //array index is obj_id
```

We adapt timestamp validation[21] to ensure schedules generated by proposed **OSTM** are serial. Therefore we maintain \( max \_ts \_lookup(ht, k) \), \( max \_ts \_insert(ht, k) \) and \( max \_ts \_delete(ht, k) \) that represents timestamp of last committed transaction which executed \( t \_lookup(ht, k) \), \( t \_insert(ht, k) \) and \( t \_delete(ht, k) \) respectively. \( max \_ts \), node and ll_entry form the part of the meta information for the **OSTM**.

```c
struct max_ts { lookup; insert; delete; }
```

4.2 Pseudocode

Through out its life an **OSTM** transaction may execute **STM_begin()**, **STM_insert()**, **STM_lookup()**, **STM_delete()** and **STM_tryC()** methods which are also exported to the user. Each transaction has a 1) rv_method execution phase: where upd_method & rv_method locally identify and logs the location to be worked upon and other meta information which would be needed for successful validation. Within rv_method execution phase rv_methods do lock free traversal and then validate while **STM_insert()** merely log their execution to be validated and updated during transaction commit. 2) upd_method execution phase: where it validates the upd_method executed during its lifetime and validates whether the transaction will commit and finally make changes in hash-table atomically or it will abort and flush its log. Figure [7] depicts the transaction life cycle.

```
Figure 7 Transaction lifecycle of **OSTM**
```
Algorithm 2 interferenceValidation(preds[], currs[])
T1 and T2 are trying to access key k5, here s1, s2 and s3 represent the state of the lazyskip-list at that instant. Let at s1 both the methods record the same \( \text{preds}(k_1, k_3) \) and \( \text{carrus}(k_3, k_5) \) with the help of \text{IIsSearch}() for key k5 (refer Figure 3(iii)). Now, let \( \text{Del}(k_5) \) acquire the lock on the \( \text{preds} \) and \( \text{carrus} \) before the \( \text{Lus}(k_5) \) and delete the node corresponding to the key k5 from BL leading to state s2 (in Figure 3(iii)) and commit. Figure 3(ii) shows the state s2 where key k5 is the part of RL. Now, \text{interferenceValidation}() (in Algo 2) will identify that location of \( \text{Lus}(k_5) \) is no more valid due to \( \text{pred}.BL \neq \text{curr} \) at Line 2 of Algo 2. This strategy of validation is similar to [8, chap 9](ALGO REF). Thus, \text{IIsSearch}() will retry to find the updated location for \( \text{Lus}(k_5) \) at state s3 (in Figure 3 iii) and eventually T2 will commit.

Consider \( \text{STM} \_\text{lookup}(ht, k) \). If this is the subsequent operation by a transaction \( T_i \) for a particular key k on hash-table ht i.e. an operation on k has already been scheduled with in the same transaction \( T_i \), then this \( \text{STM} \_\text{lookup}() \) return the value from the ll-list and does not access shared memory(Line 3 to Line 10). If the last operation was a \( \text{STM} \_\text{insert}() \) (or \( \text{STM} \_\text{lookup}() \)) on same key then the subsequent \( \text{STM} \_\text{lookup}() \) of the same transaction returns the previous value(Line 5 inserted (or observed) without accessing shared memory, and if the last operation was a \( \text{STM} \_\text{delete}() \) then \( \text{STM} \_\text{lookup}() \) returns the value NULL (Line 9). We denote this as \text{conflict-inheritance} as the methods within a transaction are bound to behave as per the previous methods on same key. Thus in this process subsequent methods also have same conflicts as the first method on same key within the same transaction.

Algorithm 3 \( \text{STM} \_\text{lookup}(obj\_id \downarrow, key \downarrow, value \uparrow, op\_status \uparrow) \)

1: procedure \( \text{STM} \_\text{lookup} \)
2: \( \text{op} \_\text{status} \leftarrow \text{RETRY} \);
3: if \((\text{txlog.findLL}(\text{obj\_id} \downarrow, \text{key} \downarrow))\) then
4: \( \text{opn} \leftarrow \text{llgetOpStatus}(\text{obj\_id} \downarrow, \text{key} \downarrow) \);
5: if \((\text{INSERT} = \text{opn})\) \((\text{LOOKUP} = \text{opn})\) then
6: \( \text{value} \leftarrow \text{llgetValue}(\text{obj\_id} \downarrow, \text{key} \downarrow) \);
7: \( \text{op} \_\text{status} \leftarrow \text{llgetOpStatus}(\text{obj\_id} \downarrow, \text{key} \downarrow) \);
8: else if \((\text{DELETE} = \text{opn})\) then
9: \( \text{value} \leftarrow \text{NULL} \);
10: \( \text{op} \_\text{status} \leftarrow \text{FAIL} \);
11: else
12: \( \text{op} \_\text{status} \leftarrow \text{llisSearch}(\text{obj\_id} \downarrow, \text{key} \downarrow, \text{preds} \uparrow) \);
13: if \((\text{op} \_\text{status} = \text{ABORT})\) then
14: \( \text{obj} \_\text{Abort}(\text{obj\_id} \downarrow) \);
15: else
16: if \((\text{read}(\text{carrus} \[1\]) \text{key} = \text{key})\) then
17: \( \text{op} \_\text{status} \leftarrow \text{OK} \);
18: \( \text{write}(\text{carrus} \[1\]) \_\text{max} \_\text{ts} \_\text{lookup}, \text{TS}(t) \);
19: \( \text{value} \leftarrow \text{value} \_\text{BL} \);
20: else if \((\text{read}(\text{carrus} \[0\]) \_\text{key} = \text{key})\) then
21: \( \text{op} \_\text{status} \leftarrow \text{FAIL} \);
22: \( \text{write}(\text{carrus} \[0\]) \_\text{max} \_\text{ts} \_\text{lookup}, \text{TS}(t) \);
23: \( \text{value} \leftarrow \text{NULL} \);
24: else
25: \( \text{llsetPredsAndCurrs}(\text{obj\_id} \downarrow, \text{key} \downarrow, \text{preds} \downarrow) \);
26: \( \text{llsetOpStatus}(\text{obj\_id} \downarrow, \text{key} \downarrow, \text{op} \_\text{status} \downarrow) \);
27: \( \text{return} \);

Figure 9 Advantages of lookup validated once

If \( \text{STM} \_\text{lookup}() \) is the first operation on a particular key then it has to do a wait free traversal (Line 12) with the help of \text{IIsSearch}() (Algo 1) to identify the target node(\text{preds} and \text{carrus}) to be logged in ll_list for subsequent methods in \text{rv_method execution} phase (discussed above for the case where \( \text{STM} \_\text{lookup}() \) is the subsequent method). If the node is present as blue(red) node then it updates the operation status as \text{OK} (\text{FAIL}) and returns the value respectively(Line 16 to Line 23). If node corresponding to the key is not found then it inserts that node(Line 24 to Line 28) corresponding to the key into RL of lazyskip-list. The inserted node can be accessed only via red links. Hence, it will not visible to any subsequent \( \text{STM} \_\text{lookup}() \). The node is inserted to take care of situations as
We prefer `STM_lookup()` to be validated instantly and is never validated again in `STM_tryC()` as the design choice to aid performance. Let’s consider `OSTM` history in Figure 9(ii), if we would have validated `L(h, k, v)` again during `tryC`, `T_1` would abort due to time order violation, but we can see that this history is acceptable where `T_1` can be serialized before `T_2` (Figure 9(ii)). Thus, `OSTM` prevents such unnecessary aborts. Another advantage for this design choice is that `T_1` doesn’t have to wait for `tryC` to know that the transaction is bound to abort as can be seen in Figure 9(iii). Here `L(h, k, Abort)` instantly aborts as soon as it realizes that time order is violated and schedule can no more be ensured to be correct saving significant computations of `T_1`. This gain becomes significant if the application is lookup intensive where it would be inefficient to wait till `STM_tryC()` to validate the `STM_lookup()` only to know that transaction has to abort.

`STM_delete()` of Alg 5 in Appendix C behaves as `STM_lookup()` (during local execution) but it is validated twice. First, in local execution similar to `STM_lookup()` and secondly in validation-commit (of `STM_tryC()`)) to ensure opacity. We adopt lazy delete approach for `STM_delete()` method. Thus, nodes are marked for deletion and not physically deleted for `STM_delete()` method. In the current work we assume that a garbage collection mechanism is present and we don’t worry about it.

**Algorithm 4**

```
procedure STM_tryC(txstatus ↑)
1. t, v ← getTid();
2. ll_list ← ll.get(t_id ↓);
3. ordered_LL_list ← ll.sort(ll_list ↓);
4. while (ll_entry, ← next(ordered_LL_list)) do
5. (key, obj_id) ← ll.getKey&ObjId(ll_entry ↓);
6. op_status ← lslSearch(obj_id ↓, key ↓, preds[] ↑, tels[] ↑);
7. currs[] ↑, valueRL ↑, COMMIT ↓;
8. if (op_status = ABORT) then
9. tryAbort(obj_id ↓);
10. return;
11. ll.setPreds&Currs(obj_id ↓, key ↓, preds[] ↓, currs[] ↓);
12. while ((ll_entry, ← next(ordered_LL_list)) do
13. (key, obj_id) ← ll.getKey&ObjId(ll_entry ↓);
14. op ← ll_entry, opn;
15. lostUpdateValidation(ll_entry ↓, preds[] ↑, currs[] ↑);
16. if (INSERT = opn) then
17. if (read(curs[] ↑, key) = key) then
18. value ← read(curs[] ↑, value);
19. write(curs[] ↑, value);
20. ll.setOpStatus(obj_id ↓, key ↓, OK ↓);
21. write(curs[] ↑, max_ts.insert, TS(t, 0));
22. else if (read(curs[] ↑, key) = key) then
23. lslInsLL(key ↓, RL_BL ↓);
24. ll.setOpStatus(obj_id ↓, key ↓, OK ↓);
25. write(curs[] ↑, max_ts.insert, TS(t, 0));
26. else
27. lslDel(key ↓, RL ↓, OK ↓);
28. ll.setOpStatus(obj_id ↓, key ↓, OK ↓);
29. write(node.max_ts.insert, TS(t, 0));
30. else if (DELETE = opn) then
31. if (read(curs[] ↑, key) = key) then
32. lslDelPreds[] ↓, curs[] ↓, RL_BL ↓;
33. ll.setOpStatus(obj_id ↓, key ↓, OK ↓);
34. write(curs[] ↑, max_ts.delete, TS(t, 0));
35. else
36. ll.setOpStatus(obj_id ↓, key ↓, OK ↓);
37. write(curs[] ↑, max_ts.delete, TS(t, 0));
38. return;
```

**upd_method execution phase:** Finally a transaction after executing the designated operations reaches the `upd_method execution` phase executed by the `STM_tryC()` method. It starts with modifying the log to ordered LL_list which contains the log entries in sorted order of the keys (so that locks can be acquired in an order; refer Line 4 of Alg 5 and contains only the upd_method (because we do not validate the lookup again for the reasons explained above). From Line 5 to Line 10 we re-validate the modified log operation to ensure that the location for the operations has not changed since the point they were logged during `rv_method execution` phase. If the location for an operation has changed this block ensures that they are updated. Now, `STM_tryC()` enters the phase where it updates the shared memory using logs from Line 11 to Line 24. Figure 10 & Figure 11 explain the execution of insert and delete in update phase of `STM_tryC()` using `lslIns() and lslDel()` respectively. Figure 10(i) represents the case when `k_5` is neither present in `BL` and nor in `RL`. It adds `k_5` to lazy-skips-list at location `preds(k_5, k_6)` and `curs(k_5, k_6)`. Figure 10(ii) represents the case when `k_5` is present in `RL`. It adds `k_5` to lazy-skips-list at location `pred(k_5, k_4)` and `curs(k_5, k_6)`. Figure 10(iii) is lazyskip-list before addition of `k_5` and Figure 10(iv) is lazyskip-list state post addition. Similarly, Figure 10(v) represents the case when `k_5` is present in `RL`. It adds `k_5` to lazy-skips-list at location `pred(k_5, k_4)` and `curs(k_5, k_6)`. Figure 10(vi) is lazyskip-list before addition of `k_5` and Figure 10(vii) is lazyskip-list state post addition. In case of deletion from lazy-skips-list when `k_5` is present in `BL` Figure 11(i) represent the
lazyskip-list state before \( k_5 \) is deleted at location \( \text{pred}(k_1, k_3) \) and \( \text{curr}(k_5, k_5) \) and Figure 11(ii) represents the lazyskip-list state after deletion.

(a) \( -\infty \rightarrow k_3 \rightarrow k_5 \rightarrow +\infty \)
(b) \( -\infty \rightarrow k_3 \rightarrow k_1 \rightarrow k_5 \rightarrow +\infty \)

(i) When \( k_5 \) is not present in BL and RL

(c) \( -\infty \rightarrow k_3 \rightarrow k_5 \rightarrow +\infty \)
(d) \( -\infty \rightarrow k_3 \rightarrow k_5 \rightarrow +\infty \)

(ii) When \( k_5 \) is present in RL

Figure 10 \( \text{Ins}(k_5) \) using \( \text{lslIns}() \) in \( \text{STM\_tryC}() \)

Figure 11 \( \text{Del}(k_5) \) using \( \text{lslDel}() \) in \( \text{STM\_tryC}() \)

While updating the methods of same transaction from its log, the \( \text{preds} \) and \( \text{currs} \) might change for two consecutive updates over the lazyskip-list causing the later update to overwrite the former (lost update). Figure 12 explains this lucidly. Suppose, \( T_1 \) is in update phase of \( \text{STM\_tryC}() \) at state \( s \) where \( \text{ins}(k_5) \) and \( \text{ins}(k_7) \) are waiting to take effect over the lazyskip-list. The lazyskip-list at \( s \) is as in Figure 12(i) also \( \text{ins}(k_5) \) and \( \text{ins}(k_7) \) have \( \text{pred}(k_5, k_4) \) and \( \text{curr}(k_8, k_8) \) as their location. Now, Lets say \( \text{ins}(k_5) \) adds \( k_5 \) between \( k_3 \) and \( k_8 \) and changes lazyskip-list (as in Figure 12(ii)) at state \( s_1 \) in Figure 12(iv). But, at \( s_1 \) BL \( \text{preds} \) and \( \text{currs} \) of \( \text{ins}(k_7) \) are still \( k_3 \) and \( k_8 \) thus it wrongly adds \( k_7 \) between \( k_3 \) and \( k_8 \) overwriting \( \text{ins}(k_5) \) as shown in Figure 12(iii) with dotted links. We correct this through \( \text{lostUpdateValidation}() \) which is \( \text{lostUpdateValidation}() \) is invoked before every \( \text{upd\_method} \) over the lazyskip-list in update phase of \( \text{STM\_tryC}() \)(Line 12 to Line 57 of Algo 4). Figure 12 represents the functionality of \( \text{lostUpdateValidation}() \) of Algo 5. Here, If \( \text{lostUpdateValidation}() \) fails for any \( \text{upd\_method} \) then as a corrective measure the \( \text{preds} \) and \( \text{currs} \) of the \( \text{upd\_method} \) under execution will be updated using the previous \( \text{upd\_method} \)'s \( \text{preds} \) and \( \text{currs} \) with the help of its \( \text{ll\_entry} \).

Figure 12 Problem in execution without \( \text{lostUpdateValidation}() \) (\( \text{ins}(k_5) \) and \( \text{ins}(k_7) \)). (i) lazyskip-list at state \( s \). (ii) lazyskip-list at state \( s_1 \), (iii) lazyskip-list at state \( s_2 \) (lost update problem).
Algorithm 5 \texttt{lostUpdateValidation}(ll_{entry},\downarrow \texttt{preds}[\uparrow \texttt{currs}][\uparrow])

1: procedure \texttt{lostUpdateValidation} \\
2: \hspace{1em} ll.getAllPreds&Currs(ll_{entry},\downarrow \texttt{preds}[\uparrow \texttt{currs}][\uparrow]) ; \\
3: \hspace{1em} if ((\texttt{read}(\texttt{preds}[0].\texttt{marked})) || (\texttt{read}(\texttt{preds}[0].\texttt{BL}) \neq \texttt{currs}[1])) then \\
4: \hspace{2em} if ((ll_{entry} - 1.\texttt{opn} = \texttt{INSERT}) then \\
5: \hspace{3em} \texttt{preds}[0] \leftarrow (ll_{entry} - 1.\texttt{key}) ; \\
6: \hspace{3em} else \texttt{preds}[0] \leftarrow (ll_{entry} - 1.\texttt{preds}[0]) ; \\
7: \hspace{1em} if (\texttt{read}(\texttt{preds}[1].\texttt{BL}) \neq \texttt{currs}[0]) then \\
8: \hspace{2em} \texttt{preds}[1] \leftarrow (ll_{entry} - 1.\texttt{key}) ; \\
5

5 \textbf{Correctness of OSTM}s

Methods in Read/Write STMs are atomic read/write methods. Proving that such methods can be partially ordered or linearized is a complex task. In OSTM where methods are intervals which also overlap with methods of different transactions exacerbates this task. We need to establish that all methods can be linearized at operational level before arguing about the co-opacity of OSTM history at transaction level. We present the proof sketch in this section.

OSTM design ensures representational invariants that 1) every node in hash-table represents an unique key(Corollary 11), 2) head and tail nodes represent minimum and maximum keys and are immutable, 3) all nodes of lazyskip-list are always in increasing order of their keys(Lemma 14), 4) all updates to shared object are done by acquiring locks. Also, all unmarked nodes are reachable by BL and every node (marked or unmarked) is reachable by RL. From code it can be observed \texttt{lslSearch()} is guaranteed to return correct location for a method.

Operational level correctness: Here we establish the above OSTM invariants (using observations directly from code or formulating them as lemma) and subsequently prove that \texttt{STM_insert()}, \texttt{STM_delete()}, \texttt{STM_lookup()} and \texttt{STM_tryC()} ensure that the invariants are adhered and the OSTM history is equivalent to the execution in which all the methods are linearized. This we achieve by identifying the linearization points (first unlock point of each successful OSTM method) such that each method execution leads the object from one correct state to the another (refer Lemma 20, Lemma 21 and Lemma 22 in appendix) and the 2PL locking mechanism [21] as observed in Observation 26 and Observation 27. We prove that lost update validation is not violated by subsequent updates in \texttt{STM_tryC()} in Lemma 18.

Transactional level correctness: Operational level correctness gives us a linearizable history which needs to be shown co-opaque by obtaining a sequential order of the involved transactions. We consider sequential (linearized) history generated by the OSTM. We then show that it is co-opaque[12] by showing its conflict graph is acyclic. Since our algorithm uses time-order validation[21], we show that conflict graph is acyclic by showing that all the edge follow timestamp order as proved in Lemma 45, Lemma 46. Finally, using the fact that OSTM generates legal histories whose conflict graph is acyclic. We show that OSTM histories are co-opaque [12] as stated below(proved in Theorem 48).

\textbf{Theorem 2.} A legal history \( H \) is co-opaque iff \( CG(H) \) is acyclic.

Deadlock freedom of OSTM: The algorithm is guaranteed to be deadlock free due to the locking invariant maintained throughout the transaction life cycle. The locking invariant holds that locks are always acquired and released in increasing order of the keys.

Safety of OSTM: We formally say that OSTM generates linearizable history at operational level (Observation 12) and the conflict graph generated by OSTM history is acyclic (Theorem 47). For complete proof of all the above lemmas and theorem please refer the Appendix D. Above discussion gives enough intuition to believe that OSTM will indeed be co-opaque[12] hence opaque [2]. Moreover, depending upon the lock implementation OSTM can be starvation free(if locks provide starvation free mutual exclusion).
6 Related Work

Earliest work of using semantics of concurrent data structures or using STMs for object level granularity include that of open nested transactions [16] and transaction boosting of Herlihy et al. [10]. Abstract nested transactions [3] is another STM that is motivated by the need to avoid aborts of transactions due to conflicts at lower level (Harris refers to them as benign conflicts). Harris et al. [3] identify the transactions which are victims of benign conflicts and preventing such unnecessary aborts by re-executing the transaction. Spiegelman et al. [19] try to build a transactional data structure library from existing concurrent data structure library. Their work is much of a mechanism than a methodology. Hassan et al. [6] have recently proposed Optimistic Transactional Boosting (OTB) that extends original transactional boosting methodology by optimizing and making it more adaptable to STMs. They further have implemented OTB on set data structure using lazy linked list [5].

Hassan [6] uses C-SWC model to prove that OTB transactions compose. We on other hand propose alternate object model STMs where we laydown a detailed legality definition for the underlying data structures to be transanctified and build a bottom up correctness proof starting from operational level to the transactional level showing that OSTM ensures co-opacity [12] thus compose. OTB uses notion of semantic read set and write set to log the methods locally and their conflicts are based on classic read-write conflict notion. Given the complexity at object level we believe that the classic conflict notion alone is not enough to capture the correctness of such STMs. We propose conflicts notion that helps to prove that OSTM is co-opaque. We also assume that their can be multiple operations on same shared object and during the execution of a transaction only the last update method which executed on a shared object needs to be validated. This avoids unnecessary validation time spent in upd_method execution phase, we achieve this by notion of conflict inheritance as discussed in Section 4.2.

Moreover unlike OTB, STM_lookup() is validated only once at the instant of their execution and unlike original boosting OSTM do not need to rollback thus saving considerable logging overhead.

Several researchers have established that STM makes development of concurrent composable applications easier than its lock based counterparts [18, 4], not to be forgotten scalability issues in lock based solutions. Tim Harris et. al. [4] proposed a STM based solution to achieve composability and at the same time maintain the abstraction, such that internal details of the atomic methods is not required for the programmer to glue multiple operations together in concurrent Haskell. Zhang et al. [22] identify composability loop holes in implementing optimized transactions which allow direct access to the shared memory to gain performance. To this end they propose replacing direct read calls to the shared memory by the encapsulated TxFastRead & TxFlush method which allows efficient composability. Thus, they achieve optimized transaction such that ensuring composability is easier. They however leave ensuring correctness to the programmer. We have laid down full theoretical correctness model for OSTM. Cederman & Tsigas propose a methodology to implement composable operation in lock free concurrent object. Their approach is restricted in application to the objects which meet the criterion, named as move candidates [1] and requires mechanical changes in the candidate data structure by the programmer to implement the composable operations.

7 Conclusion and Future Work

In this paper we build an alternative theoretical model for building highly concurrent and composable data structures with object level transactions called as OSTM. We show that higher concurrency can be obtained by using OSTM as compared to traditional RWSTMs by milking richer object-level semantics. We propose conflict notion and legality semantics for such a system keeping in mind that multiple operations can be glued together to achieve composability. Finally, using these semantics we design an efficient & composable closed addressed hash-table where chaining is done via
lazyskip-list. We prove OSTM to be co-opaque\cite{12} thus composable.

OSTM combines the scalable abstraction and ease of programming from STMs with our efficient mechanism of achieving compositability using object level semantics. Our prototype implementation of OSTM shows significant performance gain over read/write STM for a simple SET application\cite{1}. We tested it only for validating the performance gain of object level transaction over read/write transactions only. We would implement the OSTM with its full functionality to evaluate it with several applications\cite{transfer in SET, hash table etc.}. We believe that OSTM would be a significant contribution for achieving the goal of efficient, scalable and composable concurrent application.

References

1 Daniel Cederman and Philippas Tsigas. Supporting lock-free composition of concurrent data objects. CoRR, abs/0910.0366, 2009. URL: \url{http://arxiv.org/abs/0910.0366}

2 Rachid Guerraoui and Michal Kapalka. On the Correctness of Transactional Memory. In \textit{PPoPP ’08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming}, pages 175–184, New York, NY, USA, 2008. ACM. \url{doi:http://doi.acm.org/10.1145/1345205.1345233}

3 Tim Harris and et al. Abstract nested transactions, 2007.

4 Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In \textit{PPoPP ’05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming}, pages 48–60, New York, NY, USA, 2005. ACM. \url{doi:http://doi.acm.org/10.1145/1065944.1065952}

5 Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. On developing optimistic transactional lazy set. In \textit{International Conference on Principles of Distributed Systems}, pages 437–452. Springer, 2014.

6 Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Optimistic transactional boosting. In José E. Moreira and James R. Larus, editors, \textit{ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’14, Orlando, FL, USA, February 15-19, 2014}, pages 387–388. ACM, 2014. URL: \url{http://doi.acm.org/10.1145/2555243.2555283 doi:10.1145/2555243.2555283}

7 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. \textit{Parallel Processing Letters}, 17(4):411–424, 2007. URL: \url{http://dx.doi.org/10.1142/S0129626407003125 doi:10.1142/S0129626407003125}

8 M. Herlihy and N. Shavit. \textit{The Art of Multiprocessor Programming}. Elsevier Science, 2012. URL: \url{https://books.google.co.in/books?id=pFSwug7JgxYC}

9 Maurice Herlihy and J. Eliot B.Moss. Transactional memory: Architectural Support for Lock-Free Data Structures. \textit{SIGARCH Comput. Archit. News}, 21(2):289–300, 1993. \url{doi:http://doi.acm.org/10.1145/133682.165164}

10 Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In \textit{Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming}, pages 207–216. ACM, 2008.

11 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. \textit{ACM Trans. Program. Lang. Syst.}, 12(3):463–492, 1990. \url{doi:http://doi.acm.org/10.1145/78969.78972}

12 Petr Kuznetsov and Sathya Peri. Non-interference and local correctness in transactional memory. \textit{Theor. Comput. Sci.}, 688:103–116, 2017. URL: \url{https://doi.org/10.1016/j.tcs.2016.06.021 doi:10.1016/j.tcs.2016.06.021}
13 Petr Kuznetsov and Srivatsan Ravi. On the cost of concurrency in transactional memory. In *OPDIS*, pages 112–127, 2011.

14 Kfir Lev-Ari, Gregory V. Chockler, and Idit Keidar. On correctness of data structures under read-write concurrency. In Fabian Kuhn, editor, *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, volume 8784 of *Lecture Notes in Computer Science*, pages 273–287. Springer, 2014. URL: https://doi.org/10.1007/978-3-662-45174-8_19.

15 Kfir Lev-Ari, Gregory V Chockler, and Idit Keidar. A Constructive Approach for Proving Data Structures’ Linearizability. In Yoram Moses, editor, *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015. Proceedings*, volume 9363 of *Lecture Notes in Computer Science*, pages 356–370. Springer, 2015. URL: https://doi.org/10.1007/978-3-662-48653-5_24.

16 Yang Ni, Vijay S Menon, Ali-Reza Adl-Tabatabai, Antony L Hosking, Richard L Hudson, J Eliot B Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 68–78. ACM, 2007.

17 Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979. doi:http://doi.acm.org/10.1145/322154.322158.

18 Nir Shavit and Dan Touitou. Software Transactional Memory. In *PODC ’95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM. doi:http://doi.acm.org/10.1145/224964.224987.

19 Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. Transactional data structure libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 682–696. ACM, 2016.

20 Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP ’06, pages 129–136, New York, NY, USA, 2006. ACM. URL: http://doi.acm.org/10.1145/1122971.1122992 doi:10.1145/1122971.1122992.

21 Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

22 Rui Zhang, Zoran Budimlić, and William N Scherer III. Composability for application-specific transactional optimizations. Technical report, Department of Computer Science, Rice University, 2010.
Appendix

Methods: The $n$ processes access a collection of transaction objects via atomic transactions supported by a OSTM. Each transaction has a unique identifier typically denoted as $T_i$. Within a transaction, a process can invoke transactional methods on a hash-table transaction object. A hash-table($ht$) consists of multiple key-value pairs of the form $(k, v)$. The keys and values are respectively from sets $K$ and $V$. The methods that a transaction $T_i$ can invoke are: (1) $t\_insert(ht, k, v)$: this method inserts the pair $(k, v)$ into object $ht$ and return ok. If $ht$ already has a pair $(k, v')$ then $v'$ gets replaced with $v$. (2) $t\_delete(ht, k, v)$: if $ht$ has a $(k, v)$ pair then this operation deletes the pair and returns v. If no such $(k, v)$ pair is present in $ht$, then the operation returns nil. (3) $t\_lookup(ht, k, v)$: if $ht$ has a $(k, v)$ pair then this operation returns $v$. If no such $(k, v)$ pair is present in $ht$, then the method returns nil. It can be seen that $t\_lookup$ is similar to $t\_delete$.

Formally, we denote a method $m$, a process $p$, a history $h$, a transaction $T$, a total order among these events. For instance, the method $m_i$ executed so far are not consistent (w.r.t correctness-criterion which is formally defined later).

The OSTM supports two other methods: (4) $t\_tryc$: this method tries to validate all the operations of the $T_i$. OSTM returns ok if $T_i$ is successfully committed. Otherwise, OSTM returns $A$ implying abort. This method is invoked by a process after completing all its transactional operations. (5) $t\_trya$: this method returns $A$ and OSTM aborts $T_i$.

When any method of $T_i$ returns $A$, we denote that method as well as $T_i$ as aborted. We assume that a process does not invoke any other operations of a transaction $T_i$ once it has been aborted. We denote a method which does not return $A$ as unaborted.

Having described about methods of a transaction, we describe about the events invoked by these methods. We assume that each method consists of $\text{inv}$ and $\text{rsp}$ event. Specifically, the $\text{inv}$ & $\text{rsp}$ events of the methods of a transaction $T_i$ are: (1) $t\_insert(ht, k, v)$: $\text{inv}(t\_insert(ht, k, v))$ and $\text{rsp}(t\_insert(ht, k, v, ok/A))$. (2) $t\_delete(ht, k, v)$: $\text{inv}(t\_delete(ht, k))$ and $\text{rsp}(t\_delete(ht, k, v/nil/A))$. (3) $t\_lookup(ht, k, v)$: $\text{inv}(t\_lookup(ht, k))$ and $\text{rsp}(t\_lookup(ht, k, v-nil/A))$. (4) $t\_tryc$: $\text{inv}(t\_tryc())$ and $\text{rsp}(t\_tryc(ok/A))$. (5) $t\_trya$: $\text{inv}(t\_trya())$ and $\text{rsp}(t\_trya(A))$.

For clarity, we have included all the parameters of $\text{inv}$ event in $\text{rsp}$ event as well. In addition to these, each method invokes read-write primitives (operations) of $T_i$ are represented as: $r_i(x, v)$ implying that $T_i$ reads value $v$ for $x$; $w_i(x, v)$ implying that $T_i$ writes value $v$ onto $x$. Depending on the context, we ignore some of the parameters of the transactional methods and read/write primitives.

We assume that the first event of a method is $\text{inv}$ and the last event is $\text{rsp}$.

Formally, we denote a method $m$ by the tuple $(\text{evts}(m), <_m)$. Here, $\text{evts}(m)$ are all the events invoked by method $m$ and the $<_m$ a total order among these events. For instance, the method $l_{11}(k_3)$ of Figure 13 is represented as: $\text{inv}(l_{11}(h, k_3)) r_{111}(k_2, a_2) r_{112}(k_5, o_5) \text{rsp}(l_{11}(h, k_5, a_5))$. In our representation, we abbreviate $t\_insert$ as $i$, $t\_delete$ as $d$ and $t\_lookup$ as $l$. From our assumption, we get that for any read-write primitive $\text{rw}$ of $m$, $\text{inv}(m) <_m \text{rw} <_m \text{rsp}(m)$.

Sequential Histories: A method $m_{ij}$ of a transaction $T_i$ in a history $H$ is said to be isolated if for any other event $\text{e}_{pq}$ belonging to some other method $m_{pq}$ (of transaction $T_p$) either $\text{e}_{pq}$ occurs before $\text{inv}(m_{ij})$ or after $\text{rsp}(m_{ij})$. Formally, $(m_{ij} \in \text{methods}(H) : m_{ij} \text{ is isolated } \equiv \forall (m_{pq} \in \text{methods}(H), \forall \text{e}_{pq} \in m_{pq} : \text{e}_{pq} <_H \text{inv}(m_{ij}) \lor \text{rsp}(m_{ij}) <_H \text{e}_{pq})$. For instance in $H_1$ shown in Figure 11, $d_2(k_2)$ is isolated. In fact all the methods of $H_1$ are isolated.

Consider history $H_2$ shown in Figure 14. It can be seen that the all the three methods in $H_2$, ...
Following the notations used in database multi-level transactions [21], we model a transaction. Formally, \( \langle \text{inv}(m), \text{rsp}(m) \rangle \) denotes the first and last events of a transaction \( T \). We denote the first and last events of a transaction without referring to its inv and rsp events. For a sequential history \( H \), we construct the completion of \( H \), denoted \( \overline{H} \), by inserting \( \text{tryA}_k(A) \) immediately after the last method of every transaction \( T_k \in \text{incomp}(H) \). Since all the methods in a sequential history are complete, this definition only has to take care of completing transactions.

Consider a sequential history \( H \). Let \( m_{ij}(ht, k, v/\text{nil}) \) be the first method of \( T_i \) in \( H \) operating on the key \( k \). Since all the methods of a transaction are sequential and ordered, we can clearly identify the first method of \( T_i \) on key \( k \). Then, we denote \( m_{ij}(ht, k, v) \) as \( H.\text{firstKeyMth}(\langle ht, k \rangle, T_i) \). For a method \( m_{ij}(ht, k, v) \) which is not the first method on \( \langle ht, k \rangle \) of \( T_i \) in \( H \), we denote its previous method on \( k \) of \( T_i \) as \( m_{ij}(ht, k, v) = H.\text{prevKeyMth}(m_{ij}, T_i) \).

**Transactions:** Following the notations used in database multi-level transactions [21], we model a transaction as a two-level tree. Figure 13 shows a tree execution of a transaction \( T_1 \). The leaves of the tree denoted as layer-0 consist of read, write primitives on atomic objects. Hence, they are atomic. For simplicity, we have ignored the inv & rsp events in level-0 of the tree. Level-1 of the tree consists of methods invoked by transaction. In the transaction shown in Figure 13, level-1 consists of \( t\_lookup \) and \( t\_delete \) methods operating on the lazyskip-list as also shown in Figure 1(i).

Thus a transaction is a tree whose nodes are methods and leaves are events. Having informally explained a transaction, we formally define a transaction \( T \) as the tuple \( \langle \text{evts}(T), <_T \rangle \). Here \( \text{evts}(T) \) are all the read-write events (primitives) at level-0 of the transaction. \( <_T \) is a total order among all the events of the transaction. For instance, the transaction \( T_1 \) of Figure 13 is:

\[
\text{inv(l11(ht, k3)) r}111(k2, o2) r_{122}(k5, o5) \text{ rsp(l11(ht, k5, o5)) inv(d12(ht, k2)) r}_{121}(k2, o2) w_{122}(k2, o2) \text{ rsp(d12(ht, k2, o2))}
\]

Given all level-0 events, it can be seen that the level-1 methods and the transaction tree can be constructed.

We denote the first and last events of a transaction \( T_i \) as \( T_i.\text{firstEvt} \) and \( T_i.\text{lastEvt} \). Given any other read-write event \( rw \) in \( T_i \), we assume that \( T_i.\text{firstEvt} <_T \text{rw} <_T T_i.\text{lastEvt} \).

All the methods of \( T_i \) are denoted as methods\( (T_i) \). We assume that for any method \( m \) in methods\( (T_i) \), evts\( (m) \) is a subset of evts\( (T_i) \) and \( <_m \) is a subset of \( <_T \). Formally, \( \forall m \in \text{methods}(T_i) : \text{evts}(m) \subseteq \text{evts}(T_i) \land <_m \subseteq <_T \).

We assume that if a transaction has invoked a method, then it does not invoke a new method until it gets the response of the previous one. Thus all the methods of a transaction can be ordered by \( <_T \). Formally, \( \forall m_p, m_q \in \text{methods}(T_i) : (m_p <_T m_q) \lor (m_q <_T m_p)) \).
**Legal History:** If \texttt{rv\_method} is not the first method of a transaction on any key then it will return the same value as the previous method of the same transaction on the same key. In Figure 16(i), previous method for \texttt{Lu}_{ij}(ht, k_5, v_5) of transaction \texttt{T}_i on same key \(k_5\) is \texttt{Ins}_{t_{ij}}(ht, k_5, v_5). So, \(Lu_{ij}(ht, k_5, v_5)\) will return the same value which will be inserted by previous method \texttt{Ins}_{t_{ij}}(ht, k_5, v_5). Same technique will be follow in Figure 16(ii) and Figure 16(iii).

If \texttt{rv\_method} is the first method of a transaction on any key and value is not null then the previous closest method of committed transaction should be insert on the same key. In Figure 17 previous closest method for \texttt{Lu}_{ij}(ht, k, v_p) of transaction \texttt{T}_j on same key \(k\) is \texttt{Ins}_{pq}(ht, k, v_p) of transaction \texttt{T}_p. So, \texttt{Lu}_{ij}(ht, k, v_p) will return the same value which has been inserted by \texttt{Ins}_{pq}(ht, k, v_p) and there can’t be any other transaction \texttt{upd\_method} working on the same key between \texttt{T}_p and \texttt{T}_i. Figure 18 represents, previous closest method of committed transaction \texttt{T}_p is \texttt{Del}_{pq}(ht, k, v_p) on key \(k\) so \texttt{Lu}_{ij}(ht, k, Nil) of transaction \texttt{T}_i returns nil for same key \(k\).
Figure 17: STM_lookup() is returning the same value as previous closest conflicting method of committed transaction.

Figure 18: STM_lookup() is returning the same value as previous closest conflicting method of committed transaction.

Figure 19: Legal History H2

History $H_2$ in Figure 19 is legal because both the lookup of transaction $T_2$ are reading from a previously closest committed transaction.

| Functions     | Description                                                                                      |
|---------------|-------------------------------------------------------------------------------------------------|
| setOpn()      | store method name into ll_list of the txlog                                                     |
| setValue()    | store value of the key into ll_list of the txlog                                                |
| setOpStatus() | store status of method into ll_list of the txlog                                                |
| setPreds&Currs() | store location of preds and currs according to the node corresponding to the key into ll_list of the txlog |
| getOpn()      | give operation name from ll_list of the txlog                                                    |
| getValue()    | give value of the key from ll_list of the txlog                                                 |
| getOpStatus() | give status of the method from ll_list of the txlog                                             |
| getKey&Objid() | give key and obj_id corresponding to the method from ll_list of the txlog                      |
| getAptCurr()  | give the red or blue curr node from the log corresponding to the key of the txlog                |
| getPreds&Currs() | give location of preds and currs according to the node corresponding to the key from ll_list of the txlog |

Table 1: User-level functions accessed by methods
B  Optimizations

1. If $STM\_delete()$ returns FAIL in $rv\_method\ execution$ phase then no need to validate it in $STM\_tryC()$ ($upd\_method\ execution$ phase).

2. In case of insert method if node corresponding to the key $k$ is part of $BL$ then no need to identify the $preds$ and $currs$ for same key into $RL$. Thus we can reduce the number of locks in the case of insert method (for increasing the concurrency).

3. If node corresponding to the key is part of underlying data structure and $interferenceValidation()$ is unsuccessful (return retry) then optimistically we can check $toValidation()$,

   a. If $toValidation()$ is successful then we can retry else

   b. No need to find new $preds$ and $currs$ for node corresponding to the key, return $Abort$.

C  Pseudocode

Algorithm 6  STM\_begin : Allocates unique transaction ID from global\_cntr, initializes transaction log.

1: procedure STM\_BEGIN
2:    txlog ← new txlog() ;
3:    txlog.t_id ← global\_cntr++ ;

$STM\_begin$ is the first function a transaction executes in its life cycle. It initiates the $txlog$ (local log) for the transaction (Line 2) and provides an unique id to the transaction (Line 3).

Algorithm 7  STM\_insert($obj\_id$, $key$, $value$) : Optimistically defers the insert operation till the $tryC()$, stores the operational info in local log.

1: procedure STM\_INSERT
2:    if (!txlog.findInLL($obj\_id$, $key$)) then
3:        create ll\_entry ;
4:        ll.setValue($obj\_id$, $key$, $value$) ;
5:        ll.setOpn($obj\_id$, $key$, $INSERT$) ;
6:        ll.setOpStatus($obj\_id$, $key$, $OK$) ;

$STM\_insert()$ method in $rv\_method\ execution$ phase simply checks if their is a previous method that executed on the same $key$. If their is already a previous method that has executed within the same transaction it simply updates the new $value$, $opm$ as insert and $op\_status$ to $OK$ (Line 4, Line 5, and Line 6). In case the $STM\_insert()$ is the first method on $key$ it creates a new log entry for the $ll\_list$ of $txlog$. Finally the $STM\_insert()$ gets to modify the underlying $hash\_table$ using $lslIns(preds[]$, $currs[]$) at the $upd\_method\ execution$ phase.
Algorithm 8  \textit{STM\textunderscore delete(obj\_id ↓, key ↓, value ↑, op\_status ↑)}: If the transaction has locally done an operation on the same key then returns apt value and status. Else do the \textit{IslSearch()} to find the correct location of the key and validate it after that locally logs the method information to be revalidated and written in underlying data-structure during tryC().

```plaintext
1: procedure STM\_DELETE  
2: op\_status ← RETRY ;  
3: if (txlog\_findInLL(obj\_id ↓, key ↓)) then  
4: opn ← ll\_getOpn(obj\_id ↓, key ↓) ; 
5: if (INSERT = opn) then  
6: value ← ll\_getValue(obj\_id ↓, key ↓) ;  
7: ll\_setValue(obj\_id ↓, key ↓, NULL) ;  
8: ll\_setOpn(obj\_id ↓, key ↓, DELETE) ;  
9: op\_status ← OK ;  
10: else if (DELETE = opn) then  
11: ll\_setValue(obj\_id ↓, key ↓, NULL) ;  
12: value ← NULL ;  
13: op\_status ← FAIL ;  
14: else  
15: value ← ll\_getValue(obj\_id ↓, key ↓) ;  
16: ll\_setValue(obj\_id ↓, key ↓, NULL) ;  
17: ll\_setOpn(obj\_id ↓, key ↓, DELETE) ;  
18: op\_status ← ll\_getOpStatus(obj\_id ↓, key ↓) ;  
19: else  
20: op\_status ← islSearch(obj\_id ↓, key ↓, preds[] ↑, 
21: currs[] ↑, value, ll ↑, rv ↑) ;  
22: if (op\_status = ABORT) then  
23: tryAbort(obj\_id ↓) ;  
24: else  
25: if (read(currs[1].key) = key) then  
26: op\_status ← OK ;  
27: write(currs[1].max\_ts.lookup, TS(t)) ;  
28: value ← value \_RL ;  
29: else if (read(currs[0].key) = key) then  
30: op\_status ← FAIL ;  
31: write(currs[0].max\_ts.lookup, TS(t)) ;  
32: value ← NULL ;  
33: else  
34: llIns(preds[] ↓, currs[] ↓, RL ↓) ;  
35: op\_status ← FAIL ;  
36: write(node\_max\_ts.lookup, TS(t)) ;  
37: value ← NULL ;  
38: create ll\_entry ;  
39: ll\_setValue(obj\_id ↓, key ↓, NULL) ;  
40: ll\_setPredsCurrs(obj\_id ↓, key ↓, preds[] ↓, 
41: currs[] ↓) ;  
42: ll\_setOpn(obj\_id ↓, key ↓, DELETE) ;  
43: preds[0].unlock() ;  
44: preds[1].unlock() ;  
45: currs[0].unlock() ;  
46: currs[1].unlock() ;  
47: ll\_setOpStatus(obj\_id ↓, key ↓, op\_status ↓) ;  
48: return ;
```

**Figure 20** \(k_{10}\) is not present in BL as well as RL

**Figure 21** Adding \(k_{10}\) into RL

Algorithm 9  \textit{llIns(preds[] ↓, currs[] ↓, list\_type ↓)}: Inserts or overwrites a node in underlying hash table at location corresponding to \(preds\ & currs\).

```plaintext
1: procedure ISL\_INS  
2: if ([list\_type] = (RL \_BL)) then  
3: write(currs[0].marked, false) ;  
4: write(currs[0].BL, currs[1]) ;  
5: write(preds[0].BL, currs[0]) ;  
6: else if ([list\_type] = (RL)) then  
7: node = new node() ;  
8: write(node\_marked, True) ;  
9: write(node\_RL, currs[0]) ;  
10: write(preds[1].RL, node) ;  
11: else  
12: node = new node() ;  
13: write(node\_RL, currs[0]) ;  
14: write(node\_BL, currs[1]) ;  
15: write(preds[1].BL, node) ;  
16: write(preds[0].BL, node) ;
```
Algorithm 10 lslDel($\text{preds}[]$, $\text{currs}[]$) : Deletes a node from blue link in underlying hash table at location corresponding to $\text{preds} \& \text{currs}$.

1: procedure lslDel
2: write($\text{currs}[1].\text{marked}$, True);
3: write($\text{preds}[0].\text{BL}$, $\text{currs}[1].\text{BL}$);

Figure 22 Execution of lslIns(): (i) key $k_5$ is present in $RL$ and adding it into $BL$, (ii) key $k_5$ is not present in $RL$ as well as $BL$ and adding it into $RL$, (iii) key $k_5$ is not present in $RL$ as well as $BL$ and adding it into $RL$ as well as $BL$.

lslIns($\text{preds}[]$, $\text{currs}[]$) (Algo 9) adds a new node to the lazyskip-list in the hash-table. There can be following cases: if node is present in $RL$ and has to be inserted to $BL$: such a case implies that the lslIns($\text{preds}[]$, $\text{currs}[]$) is invoked in upd_method execution phase for the corresponding STM_insert() in local log represented by the block from Line 2 to Line 5. Here we first reset the $\text{currs}[0].\text{mark}$ field and update the $BL$ to the $\text{currs}[1]$ and $\text{preds}[0].\text{BL}$ to $\text{currs}[0]$. Thus the node is now reachable by $BL$ also. if node is meant to be inserted only in $RL$: This implies that the node is not present at all in the lazyskip-list and is to be inserted for the first time. Such a case can be invoked from $rv\_method$ of $rv\_method$ execution phase, if $rv\_method$ is the first method of its transaction. Line 6 to Line 10 depict such a case where a new node is created and its marked field is set, depicting that its a dead node meant to be reachable only via $RL$. In Line 9 and Line 10 the $RL$ field of the node is updated to $\text{currs}[0]$ and $\text{preds}[1]$ field of the $\text{preds}[1]$ is modified to point to the node respectively. if node is meant to be inserted in $BL$: In such a case it may happen that the node is already present in the $RL$ (already covered by Line 2 to Line 5) or the node is not present at all. The later case is depicted in Line 11 to Line 16 which creates a new node and add the node in both $RL$ and $BL$. note that order of insertion is important as the lazyskip-list can be concurrently accessed by other transactions since traversal is lock free. Figure 22(i), Figure 22(ii) and Figure 22(iii) represent the cases in order.

Algorithm 10 lslDel($\text{preds}[]$, $\text{currs}[]$) : Deletes a node from blue link in underlying hash table at location corresponding to $\text{preds} \& \text{currs}$.

1: procedure lslDel
2: write($\text{currs}[1].\text{marked}$, True);
3: write($\text{preds}[0].\text{BL}$, $\text{currs}[1].\text{BL}$);

Figure 23 Execution of lslDel(): (i) lazyskip-list before $k_5$ is deleted, (ii) lazyskip-list after $k_5$ is deleted from $BL$

lslDel($\text{preds}[]$, $\text{currs}[]$) removes a node from $BL$. It can be invoked from upd_method execution phase for corresponding STM_delete() in txlog. It simply sets the marked field of the node to be deleted($\text{currs}[1]$) and changes the $BL$ of $\text{preds}[1]$ to $\text{currs}[0]$ as shown in Line 2 and Line 3 of
Algorithm 11: findInLL(obj_id ↓ , key ↓ ) : Checks whether any operation corresponding to ⟨obj_id, key⟩ is present in ll_list.

1. procedure findInLL
2. t_i ← getTid();
3. ll_list ← txlog.getllList(1);
4. while (ll_entry, ← next(ll_list)) do
5. if ((ll_entry.first.first = obj_id) ∧ (ll_entry.first.sec = Key)) then
6. return true;
7. return false;

findInLL is an utility method that returns true to the method that has invoked it, if the calling method is not the first method of the transaction on the key. This is done by linearly traversing the log and finding an entry corresponding to the key. If the calling method is the first method of the transaction for the key then findInLL return true as it would not find any entry in the log of the transaction corresponding to the key.

Since we consider that their can be multiple objects (hash-table) so we need to find unique ⟨obj_id, key⟩ pair (refer Line 5).

Algorithm 12: toValidation(key ↓ , currs[] ↓ , val_type ↓ ) : Time-order validation for each transaction.

1. procedure toValidation
2. t_i ← getTid();
3. op_status ← OK;
4. curr ← getAptCur(currs[] ↓ , key ↓ );
5. if ((curr ≠ NULL) ∧ (curr.key = key)) then
6. if ((val_type = rv) ∧ (TS(t_i) < (read(curr.max_ts.insert(k))))) then
7. op_status ← ABORT;
8. else if ((TS(t_i) < (read(curr.max_ts.insert(k)))) ∨ TS(t_i) < (read(curr.max_ts.delete(k)))) then
9. op_status ← ABORT;
10. return op_status;

rv_method and upd_method do the validation in rv_method execution phase and upd_method execution phase respectively. validation invokes interferenceValidation() and then does the toValidation() in the mentioned order. interferenceValidation() is the property of the method and toValidation() is the property of the transaction, thus first validating the method intuitively make sense than validating the time order of the transaction first.
With Observation 5, we assume that nodes once created do not get deleted (ignoring garbage collection for now). Formally, \(\forall n\, (\text{in PublicNodes}) \wedge (n \in S.\text{nodes}) \Rightarrow (n \in S'.\text{nodes})\). With Observation 5, we assume that nodes once created do not get deleted (ignoring garbage collection for now).

**Definition 3.** *PublicNodes:* Which is having a incoming RL, except head node.

**Definition 4.** *Abstract List (Abs):* At any global abstract state \(S\), \(S.\text{Abs}\) can be defined as set of all public nodes that are accessible from head via red links union of set of all unmarked public nodes that are accessible from head via blue links. Formally, \(\langle S.\text{Abs} = S.\text{Abs}.RL \cup S.\text{Abs}.BL \rangle\), where, \(S.\text{Abs}.RL := \{\forall n | (n \in S.\text{PublicNodes}) \wedge (S.\text{Head} \rightarrow_{RL} S.n)\}\), \(S.\text{Abs}.BL = \{\forall n | (n \in S.\text{PublicNodes}) \wedge (\lnot n.\text{marked}) \wedge (S.\text{Head} \rightarrow_{BL} S.n)\}\).

**Observation 5.** Consider a global state \(S\) which has a node \(n\). Then in any future state \(S'\) of \(S\), \(n\) is a node in \(S'\) as well. Formally, \(\forall S, S' : (n \in S.\text{nodes}) \wedge (S \subseteq S') \Rightarrow (n \in S'.\text{nodes})\).

**Observation 6.** Consider a global state \(S\) which has a node \(n\), initialized with key \(k\). Then in any future state \(S'\) the key of \(n\) does not change. Formally, \(\forall S, S' : (n \in S.\text{nodes}) \wedge (S \subseteq S') \Rightarrow (n \in S'.\text{nodes}) \wedge (S.n.key = S'.n.key)\).

**Observation 7.** Consider a global state \(S\) which is the post-state of return event of the function \(lslSearch()\) invoked in the \(STM_delete()\) or \(STM_tryC()\) or \(STM_lookup()\) methods. Suppose the \(lslSearch()\) method returns \((\text{preds}[0], \text{preds}[1], \text{currs}[0], \text{currs}[1])\). Then in the state \(S\), we have:

1. \((\text{preds}[0] \wedge \text{preds}[1] \wedge \text{currs}[0] \wedge \text{currs}[1]) \in S.\text{PublicNodes}\)
2. \((\text{S.preds}[0].\text{locked}) \wedge (\text{S.preds}[1].\text{locked}) \wedge (\text{currs}[0].\text{locked}) \wedge (\text{currs}[1].\text{locked})\)
3. \((\lnot \text{S.preds}[0].\text{marked}) \wedge (\lnot \text{S.currs}[1].\text{marked}) \wedge (\text{S.preds}[0].\text{BL} = \text{S.currs}[1]) \wedge (\text{S.preds}[1].\text{RL} = \text{S.currs}[0])\)
In Observation 7, \texttt{lsSearch()} method returns only if validation succeed at Line 19.

\begin{lemma}
Consider a global state $S$ which is the post-state of return event of the function \texttt{lsSearch()} invoked in the \texttt{STM_delete()} or \texttt{STM_tryC()} or \texttt{STM_lookup()} methods. Suppose the \texttt{lsSearch()} method returns \((\texttt{preds}[0], \texttt{preds}[1], \texttt{currs}[0], \texttt{currs}[1])\). Then in the state $S$, we have,

\begin{enumerate}
\item \((S.\texttt{preds}[0].\texttt{key}) < \textit{key} \leq (S.\texttt{currs}[1].\texttt{key})\).
\item \((S.\texttt{preds}[1].\texttt{key}) < \textit{key} \leq (S.\texttt{currs}[0].\texttt{key})\).
\end{enumerate}

\textbf{Proof.}\n\begin{enumerate}
\item \((S.\texttt{preds}[0].\texttt{key}) < \textit{key} \leq (S.\texttt{currs}[1].\texttt{key})\) :

Line 4 of \texttt{lsSearch()} method of Algo 1 initializes \(S.\texttt{preds}[0]\) to point head node. Also, \((S.\texttt{currs}[1] = S.\texttt{preds}[0].BL)\) by line 5. As in penultimate execution of line 6 \((S.\texttt{currs}[1].\texttt{key} < \textit{key})\) and at line 7 \((S.\texttt{preds}[0] = S.\texttt{currs}[1])\) this implies,

\[(S.\texttt{preds}[0].\texttt{key}) < \textit{key}\] (1)

The node key doesn’t change as known by Observation 6. So, before executing of line 9 we know that,

\[(\textit{key} \leq S.\texttt{currs}[1].\texttt{key})\] (2)

From eq(1) and eq(2), we get,

\[(S.\texttt{preds}[0].\texttt{key}) < \textit{key} \leq S.\texttt{currs}[1].\texttt{key}\] (3)

From Observation 7.2 and Observation 7.3 we know that these nodes are locked and from Observation 6 we have that key is not changed for a node, so the lemma holds even when \texttt{lsSearch()} method of Algo 1 returns.

\item \((S.\texttt{preds}[1].\texttt{key}) < \textit{key} \leq (S.\texttt{currs}[0].\texttt{key})\) :

Line 10 of \texttt{lsSearch()} method of Algo 1 initializes \(S.\texttt{preds}[1]\) to point \(S.\texttt{preds}[0]\). Also, \((S.\texttt{currs}[0] = S.\texttt{preds}[0].RL)\) by line 11. As in penultimate execution of line 12 \((S.\texttt{currs}[0].\texttt{key} < \textit{key})\) and at line 13 \((S.\texttt{preds}[1] = S.\texttt{currs}[0])\) this implies,

\[(S.\texttt{preds}[1].\texttt{key}) < \textit{key}\] (4)

The node key doesn’t change as known by Observation 6. So, before executing of line 15 we know that

\[(\textit{key} \leq S.\texttt{currs}[0].\texttt{key})\] (5)

From eq(4) and eq(5), we get,

\[(S.\texttt{preds}[1].\texttt{key}) < \textit{key} \leq S.\texttt{currs}[0].\texttt{key}\] (6)

From Observation 7.2 and Observation 7.3 we know that these nodes are locked and from Observation 6 we have that key is not changed for a node, so the lemma holds even when \texttt{lsSearch()} method of Algo 1 returns.

\end{enumerate}

\begin{lemma}
For a node $n$ in any global state $S$, we have that, \(\forall n \in S.\texttt{nodes} : (S.n.\texttt{key} < S.n.\texttt{RL}.\texttt{key})\).
\end{lemma}
Proof. We prove by Induction on events that change the $RL$ field of the node (as these affect reachability), which are Line 9, 10, 13 & 15 of $lslIns()$ method of Alg 9. It can be seen by observing the code that $lslDel()$ method of Alg 10 do not have any update events of $RL$.

**Base condition:** Initially, before the first event that changes the $RL$ field of the node (as these affect reachability), which are Line 9, 10, 13 & 15 of $lslIns()$ method of Alg 9. It can be seen by observing the code that $lslDel()$ method of Alg 10 do not have any update events of $RL$.

**Induction Hypothesis:** Say, upto $k$ events that change the $RL$ field of any node, $(\forall n \in S.nodess : S.n.key < S.n.RL.key)$.

**Induction Step:** So, as seen from the code, the $(k+1)^{th}$ event which can change the $RL$ field be only one of the following:

1. **Line 9 of $lslIns()$ method:** By observing the code, we notice that Line 9 ($RL$ field changing event) can be executed only after the $lslSearch()$ method of Alg 1 returns. Line 7 of the $lslIns()$ method creates a new node, node with key and at line 8 set the $(S.node.marked = true)$ (because inserting the node only into the redlink). Line 9 then sets $(S.node.RL = S.curs[0])$. Since this event doest not change the $RL$ field of any node reachable from the head of the list (because node / $\notin S.PublicNodes$), the lemma is not violated.

2. **Line 10 of $lslIns()$ method:** By observing the code, we notice that Line 10 ($RL$ field changing event) can be executed only after the $lslSearch()$ method of Alg 1 returns. From Lemma 8.2, we know that when $lslSearch()$ method of Alg 1 returns then,

   $$(S.preds[1].key) < key \leq (S.curs[0].key)$$  \hspace{1cm} (7)$$

   To reach line 10 of $lslIns()$ method, line 32 of $STM_delete()$ method of Alg 5 or line 24 of $STM_lookup()$ method of Alg 3 should ensure that,

   $$(S.curs[0].key \neq key) \iff (S.preds[1].key) < key < (S.curs[0].key)$$  \hspace{1cm} (8)$$

   From Observation 7.3, we know that,

   $$(S.preds[1].RL = S.curs[0])$$  \hspace{1cm} (9)$$

   Also, the atomic event at line 10 of $lslIns()$ sets,

   $$(S.preds[1].RL = node) \iff (S.preds[1].key < node.key) \implies (S.preds[1].key < S.preds[1].RL.key)$$  \hspace{1cm} (10)$$

   Where $(S.node.key = key)$. Since $(preds[1], node) \in S.nodess$ and hence, $(S.preds[1].key < S.preds[1].RL.key)$.

3. **Line 13 of $lslIns()$ method:** By observing the code, we notice that Line 13 ($RL$ field changing event) can be executed only after the $lslSearch()$ method of Alg 1 returns. Line 12 of the $lslIns()$ method creates a new node, node with key. Line 13 then sets $(S.node.RL = S.curs[0])$. Since this event does not change the $RL$ field of any node reachable from the head of the list (because node / $\notin S.PublicNodes$), the lemma is not violated.

4. **Line 15 of $lslIns()$ method:** By observing the code, we notice that Line 15 ($RL$ field changing event) can be executed only after the $lslSearch()$ method of Alg 1 method returns. From Lemma 8.2, we know that when $lslSearch()$ method of Alg 1 returns then,

   $$(S.preds[1].key) < key \leq (S.curs[0].key)$$  \hspace{1cm} (11)$$
To reach line 15 of `lslIns()` method, line 26 of `STM_tryC()` method of Algo 4 should ensure that,
\[(S.currs[0].key \neq key) \implies (S.preds[1].key < key < S.currs[0].key)\] (12)

From Observation 7.3, we know that,
\[(S.preds[1].RL = S.currs[0])\] (13)

Also, the atomic event at line 15 of `lslIns()` sets,
\[(S.preds[1].RL = node) \implies (S.preds[1].key < node.key) \implies (S.preds[1].key < S.preds[1].RL.key)\] (14)

where \((S.node.key = key)\). Since \((preds[1], node) \in S.nodes\) and hence, \((S.preds[1].key < S.preds[1].RL.key)\).

\[\blacktriangleright\]

**Lemma 10.** In a global state \(S\), any public node \(n\) is reachable from \(Head\) via red links. Formally, \(\forall S, n : n \in S.PublicNodes \implies S.Head \rightarrow_{RL}^* S.n\).

**Proof.** We prove by induction on events that change the RL field of the node (as these affect reachability), which are Line 9, 10, 13 & 15 of `lslIns()` method of Algo 4. It can be seen by observing the code that `lsDel()` method of Algo 10 do not have any update events of RL.

**Base condition:** Initially, before the first event that changes the RL field of any node, we know that \((head, tail) \in S.PublicNodes \land \neg(S.head.marked) \land \neg(S.tail.marked) \land (S.head \rightarrow_{RL} S.tail)\).

**Induction Hypothesis:** Say, upto \(k\) events that change the next field of any node, \(\forall n \in S.PublicNodes, (S.head \rightarrow_{RL}^* S.n)\).

**Induction Step:** So, as seen from the code, the \((k + 1)th\) event which can change the RL field be only one of the following:

1. **Line 9 of `lslIns()` method:** Line 9 of the `lslIns()` method creates a new node, `node` with `key` and at line 8 set the \((S.node.marked = true)\) (because inserting the node only into the redlink). Line 9 then sets \((S.node.RL = S.currs[0])\). Since this event does not change the RL field of any node reachable from the head of the list (because `node` \(\notin S.PublicNodes\), the lemma is not violated.

2. **Line 10 of `lslIns()` method:** By observing the code, we notice that Line 10 (RL field changing event) can be executed only after the `lsSearch()` method of Algo 1 returns. From line 9 & 10 of `lslIns()` method, \((S.node.RL = S.currs[0]) \land (S.preds[1].RL = S.node) \land (node \in S.PublicNodes) \land (S.node.marked = true)\) (because inserting the node only into the redlink).

It is to be noted that (from Observation 7.2), \((preds[0], preds[1], currs[0], currs[1])\) are locked, hence no other thread can change marked field of `preds[1]` and `currs[0]` simultaneously. Also, from Observation 5 a node’s key field does not change after initialization. Before executing line 10, `preds[1]` is reachable from head by RL (from induction hypothesis). After line 10 we know that from `preds[1]`, public marked node, `node` is also reachable. Thus, we know that `node` is also reachable from head. Formally, \((S.Head \rightarrow_{RL} S.preds[1]) \land (S.preds[1] \rightarrow_{RL} S.node) \implies (S.Head \rightarrow_{RL} S.node)\).
3. **Line 13 of lslIns() method**: Line 12 of the lslIns() method creates a new node, node with key. Line 13 then sets \((S,\text{node}.RL = S\text{.currs}[0])\). Since this event does not change the RL field of any node reachable from the head of the list (because \(\text{node} \notin S PublicNodes\)), the lemma is not violated.

4. **Line 15 of lslIns() method**: By observing the code, we notice that Line 15 (RL field changing event) can be executed only after the lslSearch() method of Algo 1 returns. From line 13 & 15 of lslIns() method, \((S,\text{node}.RL = S\text{.currs}[0]) \wedge (S,\text{preds}[1].RL = S\text{.node}) \wedge (\text{node} \in S PublicNodes) \wedge (\text{node}.marked = false)\) (because new node is created by default with unmarked field). It is to be noted that (from Observation 7), \((\text{preds}[0], \text{preds}[1], \text{currs}[0], \text{currs}[1])\) are locked, hence no other thread can change marked field of \(S\text{.preds}[1]\) and \(S\text{.currs}[0]\) simultaneously. Also, from Observation 5, a node's key field does not change after initialization. Before executing line 15 \(\text{preds}[1]\) is reachable from head by RL (from induction hypothesis). After line 15 we know that from \(\text{preds}[1]\), public unmarked node, node is also reachable. Thus, we know that node is also reachable from head. Formally, \((S,\text{Head} \to_{RL} S\text{.preds}[1]) \wedge (S,\text{preds}[1] \to_{RL} S\text{.node}) \Rightarrow (S,\text{Head} \to_{RL} S\text{.node})\).

**Corollary 11.** Each node is associated with an unique key, i.e. at any given state \(S\), their cannot be two nodes with same key.

As every node is reachable by redlinks and has a strict ordering and from Observation 5 and Observation 6 we get this.

**Corollary 12.** Consider the global state \(S\) such that for any public node \(n\), if there exists a key strictly greater than \(n\text{.key}\) and strictly smaller than \(n\text{.RL.key}\), then the node corresponding to the key does not belong to \(S\text{.Abs}\). Formally, \(\forall S, n, key : S\text{.PublicNodes} \wedge (S,\text{node}.key < key < S,\text{node}.RL.key) \Rightarrow node\text{.key} \notin S\text{.Abs}\).

**Observation 13.** Consider a global state \(S\) which has a node \(n\) is reachable from head via RL. Then in any future state \(S'\) of \(S\), node \(n\) is also reachable from head via RL in \(S'\) as well. Formally, \(\forall S, S' : (n \in S\text{.nodes}) \wedge (S \subseteq S') \wedge (S,\text{head} \to_{RL} S,\text{node}) \Rightarrow (n \in S',\text{nodes}) \wedge (S',\text{head} \to_{RL} S',\text{node})\).

**Proof.** From Observation 5 we have that for any node \(n\), \(n \in S\text{.nodes} \Rightarrow n \in S'\text{.nodes}\). Also, we have that in absence of garbage collection no node is deleted from memory and the redlinks are preserved during delete update events (refer lslDel() method of Algo 10).

**Lemma 14.** For a node \(n\) in any global state \(S\), we have that, \(\forall n \in S\text{.nodes} : (S,\text{node}.key < S,\text{node}.BL.key)\).

**Proof.** We prove by induction on events that change the BL field of the node (as these affect reachability), which are Line 14, 15, & 16 of lslIns() method of Algo 9 and Line 9 of lslDel() method of Algo 10.

**Base condition**: Initially, before the first event that changes the BL field, we know the underlying lazyskip-list has immutable \(S,\text{head}\) and \(S,\text{tail}\) nodes with \((S,\text{head}.BL = S\text{.tail})\) and \((S,\text{head}.RL = S\text{.tail})\). The relation between their keys is \((S,\text{head}.key < S,\text{tail}.key) \wedge (\text{head}, \text{tail}) \in S\text{.nodes}\).

**Induction Hypothesis**: Say, upto \(k\) events that change the BL field of any node, \(\forall n \in S\text{.nodes} : (S,\text{node}.key < S,\text{node}.BL.key)\).

**Induction Step**: So, as seen from the code, the \((k + 1)^{th}\) event which can change the BL field be only one of the following:
1. **Line 4 & 5 of `lslIns()` method:** By observing the code, we notice that Line 4 & 5 (`BL` field changing event) can be executed only after the `lslSearch()` method of Algo 1 returns. From Lemma 8.1 and Lemma 8.2, we know that when `lslSearch()` method of Algo 1 returns then,

\[(S.preds[0].key) < key \leq (S.curs[1].key)) \land ((S.preds[1].key) < key \leq (S.curs[0].key))\]  

(15)

To reach line 4 of `lslIns()` method, line 22 of `STM_tryC()` method of Algo 4 should ensure that,

\[(S.curs[1].key) \neq key \land (S.curs[0].key = key) \implies ((S.preds[0].key) < key < (S.curs[1].key)) \land ((S.preds[1].key) < (key = S.curs[0].key))\]  

(16)

From Observation 7.3, we know that,

\[(S.preds[0].BL = S.curs[1]) \land (S.preds[1].RL = S.curs[0])\]  

(17)

The atomic event at line 4 of `lslIns()` sets,

\[(S.curs[0].BL = S.curs[1]) \iff (S.preds[0].key) < (S.curs[1].key) \implies (S.curs[0].key) < (S.curs[0].BL.key)\]  

(18)

Also, the atomic event at line 5 of `lslIns()` sets,

\[(S.preds[0].BL = S.curs[0]) \iff (S.preds[0].key) < (S.curs[0].key) \implies (S.preds[0].key) < (S.preds[0].BL.key)\]  

(19)

Where \((S.curs[0].key) = key\). Since \((preds[0], curs[0]) \in S.nodes\) and hence, \((S.preds[0].key < S.preds[0].BL.key)\).

2. **Line 14 of `lslIns()` method:** By observing the code, we notice that Line 14 (`BL` field changing event) can be executed only after the `lslSearch()` method of Algo 1 returns. Line 12 of the `lslIns()` method creates a new node, `node` with key. Line 14 then sets \((S.node.BL = S.curs[1])\). Since this event does not change the `BL` field of any node reachable from the head of the list (because `node \notin S.PublicNodes`), the lemma is not violated.

3. **Line 16 of `lslIns()` method:** By observing the code, we notice that Line 16 (`BL` field changing event) can be executed only after the `lslSearch()` method of Algo 1 returns. From Lemma 8.1 and Lemma 8.2, we know that when `lslSearch()` method of Algo 1 returns then,

\[(S.preds[0].key) < key \leq (S.curs[1].key) \land (S.preds[1].key) < key \leq (S.curs[0].key)\]  

(20)

To reach line 16 of `lslIns()` method, line 26 of `STM_entryC()` method of Algo 4 should ensure that,

\[(S.curs[0].key) \neq key \land (S.curs[1].key \neq key) \implies (S.preds[0].key) < key < (S.curs[1].key) \land (S.preds[1].key) < key < (S.curs[0].key)\]  

(21)
From Observation 7.3, we know that,
\[(S.preds[0].BL = S.curs[1])\] (22)

Also, the atomic event at line 16 of lslIns() sets,
\[(S.preds[0].BL = S.node) \equiv (S.preds[0].key < S.node.key) \implies (S.preds[0].key < S.preds[0].BL.key)\] (23)

Where \((S.node.key = key)\). Since \((preds[0], node) \in S.nodes\) and hence, \((S.preds[0].key < S.preds[0].BL.key)\).

4. **Line 3 of lslDel() method:** By observing the code, we notice that Line 3 (BL field changing event) can be executed only after the lslSearch() method of Algo 1 returns. From Lemma 8.1, we know that when lslSearch() method of Algo 1 returns then,
\[(S.preds[0].key < key < (S.curs[1].key))\] (24)

To reach line 3 of lslDel() method, line 31 of STM_tryC() method of Algo 4 should ensure that,
\[(S.curs[1].key = key) \equiv (S.preds[0].key < (key = S.curs[1].key))\] (25)

From Observation 7.3, we know that,
\[(S.preds[0].BL = S.curs[1])\] (26)

We know from Induction hypothesis,
\[(curs[1].key < curs[1].BL.key)\] (27)

Also, the atomic event at line 3 of lslDel() sets,
\[(S.preds[0].BL = S.curs[1].BL) \equiv (S.preds[0].key < S.curs[1].BL.key) \implies (S.preds[0].key < S.preds[0].BL.key)\] (28)

Where \((S.curs[1].key = key)\). Since \((preds[0], curs[1]) \in S.nodes\) and hence, \((S.preds[0].key < S.preds[0].BL.key)\)

▶ **Lemma 15.** In a global state \(S\), any unmarked public node \(n\) is reachable from Head via blue links. Formally, \(\forall S, n : (S.PublicNodes) \land (\neg S.n.marked) \implies (S.Head \rightarrow_{BL} S.n)).\)

**Proof.** We prove by Induction on events that change the BL field of the node (as these affect reachability), which are Line 4, 5, 14 & 16 of lslIns() method of Algo 9 and line 3 of lslDel() method of Algo 10.

**Base condition:** Initially, before the first event that changes the BL field of any node, we know that \((head, tail) \in S.PublicNodes \land \neg (S.head.marked) \land \neg (S.tail.marked) \land (S.head \rightarrow_{BL} S.tail)\).

**Induction Hypothesis:** Say, upto \(k\) events that change the next field of any node, \(\forall n \in S.PublicNodes, (\neg S.n.marked) \land (S.head \rightarrow_{BL} S.n)\).

**Induction Step:** So, as seen from the code, the \((k + 1)^{th}\) event which can change the BL field be only one of the following:
1. **Line 4 & 5 of lslIns() method:** By observing the code, we notice that Line 4 & 5 of the `lslIns()` method can be executed only after the `lslSearch()` method of Algo 1 returns. It is to be noted that (from Observation 7.2), `preds[0]`, `preds[1]`, `currs[0]`, `currs[1]` are locked, hence no other thread can change `S.preds[0].marked` and `S.currs[1].marked` simultaneously. Also, from Observation 6, a node’s key field does not change after initialization. Before executing line 4, from Observation 7.3,

\[(S.preds[0].marked = false) \land (S.currs[1].marked = false)\]  

(29)

And from Lemma 10 and induction hypothesis,

\[(S.Head \rightarrow_{RL} S.currs[0]) \land (S.Head \rightarrow_{BL} S.currs[1])\]  

(30)

After line 4, we know that from `currs[0]`, public unmarked node, `currs[1]` is also reachable, implies that,

\[(S.currs[0] \rightarrow_{BL} S.currs[1])\]  

(31)

Also, before executing line 5 from induction hypothesis and Lemma 10,

\[(S.Head \rightarrow_{RL} S.preds[0]) \land (S.Head \rightarrow_{BL} S.currs[0])\]  

(32)

After line 5, we know that from `preds[0]`, public unmarked node (from line 3 of `lslIns()` method), `currs[0]` is also reachable via `BL`, implies that,

\[(S.preds[0] \rightarrow_{BL} S.currs[0]) \land (S.currs[0].marked = false)\]  

(33)

From eq(31) and eq(33),

\[(S.preds[0] \rightarrow_{BL} S.currs[0]) \land (S.currs[0] \rightarrow_{BL} S.currs[1]) \land (S.currs[0].marked = false)\]  

(34)

Since \((preds[0], currs[0]) \in S.PublicNode\) and hence, \((S.Head \rightarrow_{RL} S.preds[0]) \land (S.preds[0] \rightarrow_{BL} S.currs[0]) \land (S.currs[0].marked = false) \Rightarrow (S.Head \rightarrow_{BL} S.currs[0])\).

2. **Line 14 of lslIns() method:** Line 12 of the `lslIns()` method creates a new node, `node` with key. Line 14 then sets `S.node.BL = S.currs[1]`. Since this event does not change the `BL` field of any node reachable from the head of the list (because `node \notin S.PublicNodes`), the lemma is not violated.

3. **Line 16 of lslIns() method:** By observing the code, we notice that Line 16 of the `lslIns()` method can be executed only after the `lslSearch()` method of Algo 1 returns. It is to be noted that (from Observation 7.2), `preds[0]`, `preds[1]`, `currs[0]`, `currs[1]` are locked, hence no other thread can change `S.preds[0].marked` and `S.currs[1].marked` simultaneously. Also, from Observation 6, a node’s key field does not change after initialization. Before executing line 14 from Observation 7.3,

\[(S.preds[0].marked = false) \land (S.currs[1].marked = false)\]  

(35)

And from induction hypothesis,

\[(S.Head \rightarrow_{BL} S.currs[1])\]  

(36)

After line 14, we know that from `node`, public unmarked node, `currs[1]` is also reachable via `BL`, implies that,

\[(S.node \rightarrow_{BL} S.currs[1])\]  

(37)
Also, before executing line 16 from induction hypothesis,

\[ (S.\text{Head} \rightarrow^{*}_{\text{BL}} S.\text{preds}[0]) \]  

(38)

After line 16 we know that from \text{preds}[0], public unmarked node (because new node is created by default with unmarked field), \text{node} is also reachable via BL, implies that,

\[ (S.\text{preds}[0] \rightarrow^{*}_{\text{BL}} S.\text{node}) \land (S.\text{node} . \text{marked} = \text{false}) \]  

(39)

From eq (37) and eq (39),

\[ (S.\text{preds}[0] \rightarrow^{*}_{\text{BL}} S.\text{node}) \land (S.\text{node} \rightarrow^{*}_{\text{BL}} S.\text{currs}[1]) \land (S.\text{node} . \text{marked} = \text{false}) \]  

(40)

Since \( (\text{preds}[0],\text{node}) \in S.\text{PublicNode} \) and hence, \( (S.\text{Head} \rightarrow^{*}_{\text{BL}} S.\text{preds}[0]) \land (S.\text{preds}[0] \rightarrow^{*}_{\text{BL}} S.\text{node}) \land (S.\text{node} . \text{marked} = \text{false}) \Rightarrow (S.\text{Head} \rightarrow^{*}_{\text{BL}} S.\text{node}). \)

\begin{itemize}
  \item **Corollary 16.** All public node \( n \), is reachable from head via bluelist is subset of all public node \( n \), is reachable from head via redlist. Formally, \( \langle \forall S, n : (n \in S.\text{nodes}) \land (S.\text{head} \rightarrow^{*}_{\text{BL}} S.n) \subseteq (S.\text{head} \rightarrow^{*}_{\text{RL}} S.n) \rangle. \)

  \begin{itemize}
    \item **Proof.** From Lemma 10, we know that all public nodes either marked or unmarked are reachable from head by RL, also from Lemma 15 we have that all unmarked public nodes are reachable by BL. Unmarked public nodes are subset of all public nodes thus the corollary.
  \end{itemize}
\end{itemize}

\begin{itemize}
  \item **Lemma 17.** Consider a concurrent history, \( E^{H} \), for any successful method which is call by transaction \( T_{i} \), after the post-state of LP event of the method, node corresponding to the key should be part of RL and \text{max}_\text{ts} of that node should be equal to method transaction time-stamp. Formally, \( \langle (\text{node}(\text{key}) \in (\{E^{H}.\text{Post}(m_{i}.LP)\}.\text{Abs.RL})) \land (\text{node}.\text{max}_\text{ts} = \text{TS}(T_{i})) \rangle. \)

  \begin{itemize}
    \item **Proof.** 1. **For \text{rv\_method} method:** By observing the code, each \text{rv\_method} first invokes \text{lslSearch()} method of Algo 1 (line 12) and \text{STM\_lookup()} method of Algo 3 & \text{STM\_delete()} method of Algo 8 respectively. From Lemma 9 & Lemma 14 we have that the nodes in the underlying data-structure are in increasing order of their keys, thus the key on which the method is working has a unique location in underlying data-structure from Corollary 11. So, when the \text{lslSearch()} is invoked from a method, it returns correct location \( (\text{preds}[0], \text{preds}[1], \text{currs}[0], \text{currs}[1]) \) of corresponding \text{key} as observed from Observation 7 & Lemma 8 and all are locked, hence no other thread can change simultaneously (from Observation 2). In the pre-state of LP event of \text{rv\_method}, if \( (\text{node}.\text{key} \in S.\text{Abs.RL}) \), means \text{key} is already there in RL and time-stamp of that node is less then the \text{rv\_method} transactions time-stamp, from \text{toValidation()} method of Algo 12, then in the post-state of LP event of \text{rv\_method}, \text{node}.\text{key} should be the part of RL from Observation 13 and \text{key} can’t be change from Observation 6 and it just update the \text{max}_\text{ts} field for corresponding node \text{key} by method transaction time-stamp else abort.

    In the pre-state of LP event of \text{rv\_method}, if \( (\text{node}.\text{key} \notin S.\text{Abs.RL}) \), means \text{key} is not there in RL then, in the post-state of LP event of \text{rv\_method}, insert the node corresponding to the \text{key} into RL by using \text{lslIns()} method of Algo 9 and update the \text{max}_\text{ts} field for corresponding node \text{key} by method transaction time-stamp. Since, \text{node}.\text{key} should be the part of RL from Observation 13 and \text{key} can’t be change from Observation 6, in post-state of LP event of \text{rv\_method}. 
  \end{itemize}
\end{itemize}
2. **For upd_method method:** By observing the code, each upd_method also first invokes lslSearch() method of Algo[1](line 7 of STM_commit() method of Algo[2]). From Lemma 9 & Lemma 14 we have that the nodes in the underlying data-structure are in increasing order of their keys, thus the key on which the method is working has a unique location in underlying data-structure from Corollary 11. So, when the lslSearch() is invoked from a method, it returns correct location (preds[0], preds[1], curr[0], curr[1]) of corresponding key as observed from Observation 7 & Lemma 8 and all are locked, hence no other thread can change simultaneously (from Observation 7).

   a. **If upd_method is insert:** In the pre-state of LP event of upd_method, if (node.key ∈ S.Abs.RL), means key is already there in RL and time-stamp of that node is less then the upd_method transactions time-stamp, from toValidation() method of Algo[12], then in the post-state of LP event of upd_method, node.key should be the part of RL and it just update the max_ts field for corresponding node key by method transaction time-stamp else abort. In the pre-state of LP event of upd_method, if (node.key ∉ S.Abs.RL), means key is not there in RL then in the post-state of LP event of upd_method, it will insert the node corresponding to the key into the RL as well as BL, from lslIns() method of Algo[8] at line 30 of STM_commit() method of Algo[4] and update the max_ts field for corresponding node key by method transaction time-stamp. Once a node is created it will never get deleted from Observation 13 and node corresponding to a key can’t be modified from Observation 6.

   b. **If upd_method is delete:** In the pre-state of LP event of upd_method, if (node.key ∈ S.Abs.RL), means key is already there in RL and time-stamp of that node is less then the upd_method transactions time-stamp, from toValidation() method of Algo[12], then in the post-state of LP event of upd_method, node.key should be the part of RL, from lslDel() method of Algo[10] at line 35 of STM_commit() method of Algo[4] and it just update the max_ts field for corresponding node key by method transaction time-stamp else abort.

   In the pre-state of LP event of upd_method, (node.key ∉ S.Abs.RL) this should not be happen because execution of STM_delete() method of Algo[8] must have already inserted a node in the underlying data-structure prior to STM_commit() method of Algo[4]. Thus, (node.key ∈ S.Abs.RL) and update the max_ts field for corresponding node key by method transaction time-stamp else abort.

   ▶

   In OSTM we have a upd_method execution phase where all buffered upd_method take effect together after successful validation of each of them. Following problem may arise if two upd_method within same transaction have at least one shared node amongst its recorded (preds[0], preds[1], curr[0], curr[1]), in this case the previous upd_method effect might be overwritten if the next upd_method preds and curr are not updated according to the updates done by the previous upd_method. Thus program order might get violated. Thus to solve this we have lost update validation after each upd_method in STM_commit(), during upd_method execution phase.

   ▶ **Lemma 18.** lostUpdateValidation() preserve the program order within a transaction.

   **Proof.** We are taking contradiction that lostUpdateValidation() is not preserving program order means two consecutive upd_method of same transaction which are having at least one shared node amongst its recorded (preds[0], preds[1], curr[0], curr[1]) then effect of first upd_method will be overwritten by the next upd_method.

   By observing the code at line 13 of STM_commit() method of Algo[4] current upd_method will go for lostUpdateValidation() and at line 12 of lostUpdateValidation() method of Algo[5], current upd_method will validate its (preds[0].marked) and (preds[0].BL = curr[1]). If any condition is true then, at line 12 of lostUpdateValidation() method of Algo[5] will check for previous
upd_method. If the previous upd_method is insert then the current upd_method update its preds[0]
to previous upd_method, node.key else set current upd_method preds[0] to previous upd_method
preds[0].

After that at line 8 of lostUpdateValidation() method of Algo 5, current upd_method validate
its (preds[1], RL! = currs[0]). If condition is true then current upd_method set its preds[1] to
previous upd_method, node.key.

If we will not update the current method preds and currs using lostUpdateValidation() then effect
of first upd_method will be overwritten by the next upd_method.

Observation 19. For any global state S, the lostUpdateValidation() in STM_tryC() preserves the
properties of lslSearch() as proved in Observation 7 & Lemma 8.

Lemma 20. Consider a concurrent history, EH, after the post-state of LP event of successful
STM_tryC() method, where each key belonging to the last upd_method of that transaction, then,

\[\begin{align*}
20.1 & \text{If } \text{upd_method is insert, then node corresponding to the key should be part of BL and node.val} \\
& \text{should be equal to } v. \text{ Formally, } (\langle \text{node.key} \rangle \in (EH.\text{Post}(m_i, \text{LP})).\text{Abs.BL}) \wedge (\text{node.val} = v).
\end{align*}\]

\[\begin{align*}
20.2 & \text{If } \text{upd_method is delete, then node corresponding to the key should not be part of BL. Formally,} \\
& (\langle \text{node.key} \rangle \notin (EH.\text{Post}(m_i, \text{LP})).\text{Abs.BL}).
\end{align*}\]

Proof. By observing the code, each upd_method also first invokes lslSearch() method of Algo 1
(line 7 of STM_tryC() method of Algo 4). From Lemma 9 & Lemma 14 we have that the nodes
in the underlying data-structure are in increasing order of their keys, thus the key on which the
method is working has a unique location in underlying data-structure from Corollary 11. So,
when the lslSearch() is invoked from a method, it returns correct location \((\text{preds}[0], \text{preds}[1], \text{currs}[0], \text{currs}[1])\) of corresponding key as observed from Observation 7 & Lemma 8 and all are
locked, hence no other thread can change simultaneously (from Observation 7.2).

\[\begin{align*}
20.1 & \text{If } \text{upd_method is insert: In the pre-state of LP event of upd_method at Line 17 of} \text{STM_tryC()} \text{ method of Algo 4, if } (\text{node.key} \in S.\text{Abs.RL}), \text{means key is already there in} \text{RL} \\
& \text{and time-stamp of that node is less then the upd_method transactions time-stamp, from} \text{toValidation()} \text{method of Algo 4 then in the post-state of LP event of upd_method, node.key} \\
& \text{should be the part of BL and it will update the value as v.} \\
& \text{In the pre-state of LP event of upd_method at Line 26 of STM_tryC() method of Algo 4, if} \text{node.key} \notin S.\text{Abs.RL}, \text{means key is not there in} \text{RL} \text{then in the post-state of LP event of upd_method, it will insert the node corresponding to the key into the BL, from lslIns() method of Algo 9 at line 27 of STM_tryC() method of Algo 4 and update the value as v. Once a node is created it will never get deleted from Observation 13 and node corresponding to a key can’t be} \\
& \text{modified from Observation 6.}
\end{align*}\]

\[\begin{align*}
20.2 & \text{If } \text{upd_method is delete: In the pre-state of LP event of upd_method at Line 31 of STM_tryC() method of Algo 4, if} \text{node.key} \notin S.\text{Abs.RL}, \text{means key is already there in RL and time-stamp of that node is less then the} \text{upd_method transactions time-stamp, from toValidation()} \text{method of Algo 12, then in the post-state of LP event of upd_method, node.key should not be the part of} \text{BL, from lslDel() method of Algo 10 at line 31 of STM_tryC() method of Algo 4.} \\
& \text{In the pre-state of LP event of upd_method, (node.key} \notin S.\text{Abs.RL}) \text{this should not be happen because execution of} \text{STM_delete() method of Algo 8 must have already inserted a node in the underlying data-structure prior to STM_tryC() method of Algo 4.}
\end{align*}\]
Lemma 21. Consider a concurrent history, $E^H$, where $S$ be the pre-state of LP event of successful $rv_m$ method, in that, if node corresponding to the key is the part of $BL$ and node.val is equal to $v$ then, $rv_m$ method return $OK$ and value $v$. Formally, $((\text{node}(key)) \in (E^H.\text{Pre}(m_i, LP)].\text{Abs.BL}) \land (S.\text{node.val} = v) \implies rvm(key, OK, v))$.

Proof. Let the $rv_m$ method is $STM\_lookup()$ method of Algo 3 and it is the first key method of the transaction, we ignore the abort case for simplicity. From line 12 of $STM\_lookup()$ method of Algo 3, when $lslSearch()$ method of Algo 1 returns we have $(preds[0], preds[1], currs[0], currs[1]) \in S.\text{PublicNodes}$ and are locked (from Observation 7.1 & Observation 7.2) until $STM\_lookup()$ method of Algo 3 return. Also, from Lemma 8.1, $(S.preds[1].key < key \leq S.currs[0].key)$ (41)

To return OK, $S.currs[1]$ should be reachable from the head via bluelist from Definition 4, in the pre-state of LP of $rv_m$ method. And after observing code, at line 16 of $STM\_lookup()$ method of Algo 3, $(S.currs[1].key = key) \implies (S.preds[0].key < (key = S.currs[1].key))$ (42)

Also, from Observation 7.3, $(S.preds[0].BL = S.currs[1])$ (43)

And $(currs[1]) \in S.\text{nodes}$, we know $(currs[1]) \in S.\text{Abs.BL}$ where $S$ is the pre-state of the LP event of the method. From Lemma 20.1, there should be a prior $upd\_method$ which have to be insert and $currs[1].val$ is equal to $v$. Since Observation 6 tells, no node changes its key value after initialization. Hence $(\text{node}(key)) \in (E^H.\text{Pre}(m_i, LP)].\text{Abs.BL}) \land (S.\text{node.val} = v)$.

*Same argument can be extended to $STM\_delete()$ method.

Lemma 22. Consider a concurrent history, $E^H$, where $S$ be the pre-state of LP event of successful $rv_m$ method, in that, if node corresponding to the key is not the part of $BL$ then, $rv_m$ method return $FAIL$. Formally, $((\text{node}(key)) \notin (E^H.\text{Pre}(m_i, LP)].\text{Abs.BL})) \implies rvm(key, FAIL))$.

Proof. Let the $rv_m$ method is $STM\_lookup()$ method of Algo 3 and it is the first key method of the transaction, we ignore the abort case for simplicity.

1. From line 12 of $STM\_lookup()$ method of Algo 3, when $lslSearch()$ method of Algo 1 returns we have $(preds[0], preds[1], currs[0], currs[1]) \in S.\text{PublicNodes}$ and are locked (from Observation 7.1 & Observation 7.2) until $STM\_lookup()$ method of Algo 3 return. Also, from Lemma 8.2, $(S.preds[1].key < key \leq S.currs[0].key)$ (44)

To return FAIL, $S.currs[0]$ should not be reachable from the head via bluelist from Definition 4, in the pre-state of LP of $rv_m$ method. And after observing code, at line 20 of $STM\_lookup()$ method of Algo 3, $(S.currs[0].key = key) \implies (S.preds[1].key < (key = S.currs[0].key))$ (45)

Also, from Observation 7.3, $(S.preds[1].RL = S.currs[0])$ (46)
From line 12 of STM_lookup() method of Algo 3, when lslSearch() method of Algo 4 returns we have (preds[0], preds[1], curs[0], curs[1] ∈ S.PublicNodes) and are locked (from Observation 7.1 & Observation 7.2) until STM_lookup() method of Algo 3 return. Also, from Lemma 8.2,

\[
(S.preds[1].key < key ≤ S.curs[0].key)
\]

And after observing code, at line 24 of STM_lookup() method of Algo 3,

\[
(S.curs[1].key ≠ key) \land (S.curs[0].key ≠ key) \Rightarrow \text{false}
\]

\[
(S.preds[1].key < key < S.curs[0].key)
\]

Also, from Observation 7.3,

\[
(S.preds[1].RL = S.curs[0])
\]

From eq. (48), we can say that, \(\text{node}(\text{key}) \not\in S.Abs\) and from Corollary 12, we conclude that \(\text{node}(\text{key})\) not in the state after lslSearch() returns. Since Observation 6 tells, no node changes its key value after initialization. Hence \(\text{node}(\text{key}) \not\in (|E.H.| \text{Pre}(m_i, LP), Abs.BL)\).

*Same argument can be extended to STM_delete() method.

Observation 23. Only the successful STM_tryC() method working on the key k can update the Abs.BL.

By observing the code, only the successful STM_tryC() method of Algo 4 is changing the BL. There is no line which is changing the BL in STM_delete() method of Algo 8 and STM_lookup() method of Algo 3. Such that rv_method is not changing the BL.

Observation 24. If STM_tryC() and rv_method wants to update Abs on the key k, then first it has to acquire the lock on the node corresponding to the key k.

If node corresponding to the key k is not the part of Abs then STM_tryC() and rv_method have to create the node corresponding to the key k and before adding it into the shared memory(Abs), it has to acquire the lock on the particular node corresponding to the key k.

Definition 25. First unlocking point of each successful method is the LP.

Observation 26. Two concurrent conflicting methods of different transaction can’t acquire the lock on the same node corresponding to the key k simultaneously.

Observation 27. Consider two concurrent conflicting method of different transactions say \(m_i\) of \(T_i\) and \(m_j\) of \(T_j\) working on the same key k, then, if \(ul(m_i(k))\) happen before the \(l(m_j(k))\) then \(LP(m_i)\) happen before \(LP(m_j)\). Formally, \((ul(m_i(k)) < l(m_j(k))) \Rightarrow (LP(m_i) < LP(m_j))\)

If two concurrent conflicting methods are working on the same key k and want to update Abs then they have to acquire the lock on the node corresponding to the key k from Observation 24 and one of them succeed from Observation 26. If \(ul(m_i(k))\) happen before the \(l(m_j(k))\) then from Definition 25, \(LP(m_i)\) happen before the \(LP(m_j)\).
Lemma 28. Consider two state, $S_1, S_2$ s.t. $S_1 \sqsubseteq S_2$ and $S_1.BL.value(k) \neq S_2.BL.value(k)$ then there exist $S'$ s.t. $S' \sqsubseteq S_2$ and $S'$ contain the STM\_tryC() method on the same key $k$. Formally, \[ (S_1.BL.value(k) \neq (S_2.BL.value(k)) \Rightarrow \exists(S' s.t., S_1.BL \prec S'.LP(tryC) \prec S_2.BL) \]. Where $S_1$ is the post-state of LP event of STM\_tryC() method and $S_2$ is the pre-state of LP event of rv\_method.

Proof. In the state $S_1$ and $S_2$, if the value corresponding to the key $k$ is not same then from Observation 23, we know that only the successful STM\_tryC() method working on the same key $k$ can update the Abs. $BL$. For updating the Abs on the key $k$ it has to acquire the lock on the node corresponding to the key $k$ from Observation 24. Such that, $l(tryC(k))$ happen before the $l(S_2(k))$ from Observation 26, then, $ul(tryC(k))$ happen before the $l(S_2(k))$ then $LP(tryC)$ happen before the $LP(S_2)$ from Observation 27.

Lemma 29. Consider a successful STM\_tryC() method of a transaction $T_i$, which is performing last upd\_method on a key $k$ and a successful rv\_method of a transaction $T_j$, which is also working on the same key $k$, then,

1. If the pre-state of rv\_method, node corresponding to the key $k$ is the part of BL and value as $v$ then previous closest successful tryC method should having the last upd\_method as insert on the same key $k$ and value as $v$.
2. If the pre-state of rv\_method, node corresponding to the key $k$ is not the part of BL then previous closest successful tryC method should having the last upd\_method as delete on the same key $k$.

Proof. For proving this we are taking a contradiction that in the pre-state of rv\_method, node corresponding to the key $k$ is the part of BL and value as $v$, for that, there exist a previous closest successful tryC method should having the last upd\_method as insert on the same key $k$ from Corollary 11, node corresponding to the key $k$ is unique and value is $v'$. If the value of the node corresponding to the key $k$ is different for both the methods then from Lemma 28, there should be some other transaction tryC method working on the same key $k$ and its LP should lies in between these two methods LP. Therefore that intermediate tryC should be the previous closest method for the rv\_method and it will return the same value as previous closest method inserted.

For proving this we are taking contradiction that previous closest successful tryC method should having the last upd\_method as insert on the same key $k$. If the last upd\_method is insert on the same key $k$ then after the post-state of successful tryC method, node corresponding to the key $k$ should be the part of BL from Lemma 20. But we know that in the pre-state of rv\_method, node corresponding to the key $k$ is not the part of BL. Such that previous closest successful tryC method should not having last upd\_method as insert on the same key $k$. Hence contradiction.

Construction of sequential history based on the LP of concurrent methods of a concurrent history, $E^H$, and execute them in their LP order for returning the same return value.

Lemma 30. Consider a sequential history, $E^S$, for any successful method which is call by transaction $T_i$, after the post-state of the method, node corresponding to the key should be part of RL and max\_ts of that node should be equal to method transaction time-stamp. Formally, \[ \{(\text{node}(key) \in (P.Abs.RL)) \wedge (P.node.max\_ts = TS(T_i))\} \]. Where $P$ is the post-state of the method.

Proof. 1. For rv\_method method: By observing the code, each rv\_method first invokes lslSearch() method of Algo 1 and line 12 of STM\_lookup() method of Algo 3 & STM\_delete() method of Algo 8 respectively. From Lemma 6 & Lemma 14 we have that the nodes in the underlying
data-structure are in increasing order of their keys, thus the key on which the method is working has a unique location in underlying data-structure from Corollary\(^1\). So, when the \texttt{lslSearch()}

is invoked from a method, it returns correct location \((\text{preds}[0], \text{preds}[1], \text{currs}[0], \text{currs}[1])\) of corresponding \texttt{key} as observed from Observation\(^7\) & Lemma\(^8\) and all are locked, hence no other thread can change simultaneously (from Observation\(^7\))

In the pre-state of \texttt{rv_method}, if \((\text{node.key} \in S.Abs.RL)\), means \texttt{key} is already there in 

\texttt{RL} and time-stamp of that node is less then the \texttt{rv_method} transactions time-stamp, from 

toValidation()\ method of Algo\([12]\). then in the post-state of \texttt{rv_method}, \texttt{node.key} should be the part of \texttt{RL} from Observation\(^13\) and \texttt{key} can’t be change from Observation\(^6\) and it just update the \texttt{max_ts} field for corresponding \texttt{node.key} by method transaction time-stamp else abort.

In the pre-state of \texttt{rv_method}, if \((\text{node.key} \notin S.Abs.RL)\), means \texttt{key} is not there in \texttt{RL} then, in the post-state of \texttt{rv_method}, insert the \texttt{node} corresponding to the \texttt{key} into \texttt{RL} by using \texttt{lslIns()} \(\) \texttt{method of Algo\([0]\), and update the \texttt{max_ts} field for corresponding \texttt{node.key} by method transaction time-stamp. Since, \texttt{node.key} should be the part of \texttt{RL} from Observation\(^13\) and \texttt{key} can’t be change from Observation\(^6\), in post-state of \texttt{rv_method}.

2. **For \texttt{upd_method} method:** By observing the code, each \texttt{upd_method} also first invokes \texttt{lslSearch()} \texttt{method of Algo\([1]\) line\([7]\) of \texttt{STM_tryC()} \texttt{method of Algo\([4]\)}. From Lemma\(^9\) & Lemma\(^14\) we have that the nodes in the underlying data-structure are in increasing order of their keys, thus the key on which the method is working has a unique location in underlying data-structure from Corollary\(^11\).

So, when the \texttt{lslSearch()} is invoked from a method, it returns correct location \((\text{preds}[0], \text{preds}[1], \text{currs}[0], \text{currs}[1])\) of corresponding \texttt{key} as observed from Observation\(^7\) & Lemma\(^8\) and all are locked, hence no other thread can change simultaneously (from Observation\(^7\)).

   a. **If \texttt{upd_method is insert}**: In the pre-state of \texttt{upd_method}, if \((\text{node.key} \in S.Abs.RL)\), means \texttt{key} is already there in \texttt{RL} and time-stamp of that node is less then the \texttt{upd_method} transactions time-stamp, from toValidation()\ method of Algo\([12]\), then in the post-state of \texttt{upd_method}, \texttt{node.key} should be the part of \texttt{RL} and it just update the \texttt{max_ts} field for corresponding \texttt{node.key} by method transaction time-stamp else abort.

   In the pre-state of \texttt{upd_method}, if \((\text{node.key} \notin S.Abs.RL)\), means \texttt{key} is not there in \texttt{RL} then in the post-state of \texttt{upd_method}, it will insert the \texttt{node} corresponding to the \texttt{key} into the \texttt{RL} as well as \texttt{BL}, from \texttt{lslIns()} \texttt{method of Algo\([9]\) at line\([25]\) of \texttt{STM_tryC()} \texttt{method of Algo\([4]\)} and update the \texttt{max_ts} field for corresponding \texttt{node.key} by method transaction time-stamp. Once a node is created it will never get deleted from Observation\(^13\) and node corresponding to a key can’t be modified from Observation\(^6\).

   b. **If \texttt{upd_method is delete}**: In the pre-state of \texttt{upd_method}, if \((\text{node.key} \in S.Abs.RL)\), means \texttt{key} is already there in \texttt{RL} and time-stamp of that node is less then the \texttt{upd_method} transactions time-stamp, from toValidation()\ method of Algo\([12]\), then in the post-state of \texttt{upd_method}, \texttt{node.key} should be the part of \texttt{RL}, from \texttt{lslDel()} \texttt{method of Algo\([10]\) at line\([34]\) of \texttt{STM_tryC()} \texttt{method of Algo\([4]\)} and it just update the \texttt{max_ts} field for corresponding \texttt{node.key} by method transaction time-stamp else abort.

   In the pre-state of \texttt{upd_method}, \((\text{node.key} \notin S.Abs.RL)\) this should not be happen because execution of \texttt{STM_delete()} \texttt{method of Algo\([8]\) must have already inserted a node in the underlying data-structure prior to \texttt{STM_tryC()} \texttt{method of Algo\([4]\). Thus, \((\text{node.key} \in S.Abs.RL)\) and update the \texttt{max_ts} field for corresponding \texttt{node.key} by method transaction time-stamp else abort.
D.2 Transactional Level

From Section D.1, we are guaranteed to have a sequential history or in other terms we have a linearizable history. Now we shall prove that such linearizable history obtained from OSTM is opaque.

Observation 32. $H$ is a sequential history obtained from OSTM, as shown at operational level using LP.

Definition 33. $CG(H)$ is a conflict graph of $H$.

Lemma 34. Conflict graph of a serial history is acyclic.

Proof. If conflict graph of serial history contains an conflict edge $(T_1, T_2)$, then $T_1.lastEvt \prec_H T_2.firstEvt$. Now, assume that conflict graph of a serial history is cyclic, then their exist a cycle path in the form $(T_1, T_2 \cdots T_k, T_1)$, $(k \geq 1)$. So, transitively,

$$((T_1.lastEvt \prec_H T_k.firstEvt) \wedge (T_k.lastEvt \prec_H T_1.firstEvt)) \Rightarrow (T_1.lastEvt \prec_H T_1.firstEvt)$$

This contradict our assumption as eq(50) is impossible, from definition of program order of a transaction. Thus, cycle is not possible in serial history.

Observation 35. $H_2$ is an history generated by applying topological sort on $CG(H_1)$.

Observation 36. Topological sort maintains conflict-order and real-time order of the original history $H_1$.

Definition 37. $conflict(H)$ is a set of ordered pair $(T_i, T_j)$, such that their exists conflicting methods $m_i, m_j$ in $T_i$ & $T_j$ respectively, such that $m_i \prec^{MR}_H m_j$. And it is represented as $\prec^{CO}_H$.

Lemma 38. $H_1$ is legal & $CG(H_1)$ is acyclic. then,

1. $H_1$ is equivalent to $H_2 \Rightarrow (methods(H_1) = methods(H_2))$.
2. $\prec^{CO}_{H_1} \subseteq \prec^{CO}_{H_2}$. i.e. $H_1$ preserves the conflicts of $H_2$

Proof. Lemma 38.2

We should show that $\forall (T_i, T_j)$, such that $(T_i, T_j) \in \prec^{CO}_{H_1} \Rightarrow (T_i, T_j) \in \prec^{CO}_{H_2}$.

Let’s assume that their exists a conflict $(T_i, T_j)$ in $\prec^{CO}_{H_1}$ but not in $\prec^{CO}_{H_2}$. But, from Observation 35 & Observation 36, we know that $(T_i, T_j) \in \prec^{CO}_{H_2}$. Thus, $\prec^{CO}_{H_1} \subseteq \prec^{CO}_{H_2}$.

The relation is of improper subset because topological sort may introduce new real-time orders in $H_2$ which might not be present in $H_1$.

Lemma 39. Let $H_1$ and $H_2$ be equivalent histories such that $\prec^{CO}_{H_1} \subseteq \prec^{CO}_{H_2}$. Then, $H_1$ is legal $\Rightarrow H_2$ is legal.

Proof. We know $H_1$ is legal, wlog let us say $(rv_j(ht, k, v) \in methods(H_1))$, such that $(up_p(ht, k, v_p) = H_1.lastUptd(rv_j(ht, k, v)))$ where, $(v = v_p \neq \text{nill})$, if $(up_p(ht, k, v_p) = t\_insert_p(ht, k, v_p))$ or $(v = \text{nill})$, if $(up_p(ht, k, v_p) = t\_delete_p(ht, k, v_p))$. From the conflict-notion conflict($H_1$) has,

$$up_p(ht, k, v_p) \prec^{MR}_{H_1} rv_j(ht, k, v)$$

(51)
Let us assume $H_2$ is not legal. Since, $H_1$ is equivalent to $H_2$ from Lemma 38 such that $(rv_j(ht, k, v) \in \text{methods}(H_2))$. Since $H_2$ is not legal, there exist a $(up_r(ht, k, v_r) \in \text{methods}(H_2))$ such that $(up_r(ht, k, v_r) = H_2., \text{lastU} \text{pdt}(rv_j(ht, k, v)))$. So conflict($H_2$) has,

$$up_r(ht, k, v_r) \prec_{H_2} rv_j(ht, k, v)$$

We know, $(\prec_{CO} \subseteq \prec_{H_2})$ so,

$$up_p(ht, k, v_p) \prec_{H_2} rv_j(ht, k, v)$$

From Lemma 38 $(up_r(ht, k, v_r) \in \text{methods}(H_1))$. Since $H_1$ is legal $up_r(ht, k, v_r)$ can occur only in one of following conflicts,

$$up_r(ht, k, v_r) \prec_{H_1} up_p(ht, k, v_p)$$

or

$$rv_j(ht, k, v) \prec_{H_1} up_r(ht, k, v_r)$$

In $H_1$ eq(55) is not possible, because if (eq(55) $\in \text{conflict}(H_1)$) implies (eq(55) $\in \text{conflict}(H_2)$) from $(\prec_{CO} \subseteq \prec_{H_2})$ and in $H_2$ eq(52) and eq(55) cannot occur together. Thus only possible way $up_r(ht, k, v_r)$ can occur in $H_1$ is via eq(54). From eq(54) we have,

$$up_r(ht, k, v_r) \prec_{H_1} up_p(ht, k, v_p)$$

From eq(52), eq(53) and eq(56) we have,

$$up_r(ht, k, v_r) \prec_{H_2} up_p(ht, k, v_p) \prec_{H_2} rv_j(ht, k, v)$$

This contradicts that $H_2$ is not legal. Thus if $H_1$ is legal $\rightarrow H_2$ is legal.

Observation 40. Each transaction is assigned a unique time-stamp in $STM\_begin()$ method using a shared counter which always increases atomically.

Observation 41. Each successful method of a transaction is assigned the time-stamp of its own transaction.

Lemma 42. Consider a global state $S$ which has a node $n$, initialized with max.ts. Then in any future state $S'$ the max.ts of $n$ should be greater then or equal to $S$. Formally, $(\forall S, S' : (n \in S\. \text{Abs}) \land (S \subseteq S') \Rightarrow (n \in S'.\text{Abs}) \land (S.n\. \text{max.ts} \leq S'.n\. \text{max.ts}))$.

Proof. We prove by Induction on events that change the max.ts field of a node associated with a key, which are Line 26, 28, 30 of $STM\_delete()$ method of Algo 8 Line 22 & 27 of $STM\_lookup()$ method of Algo 3 and Line 21, 25, 29, 34 & 37 of $STM\_tryC()$ method of Algo 4.

Base condition: Initially, before the first event that changes the max.ts field of a node associated with a key, we know the underlying lазyskip-list has immutable $S\. \text{head}$ and $S\. \text{tail}$ nodes with ($S\. \text{head}.BL = S\. \text{tail}$) and ($S\. \text{head}.RL = S\. \text{tail}$).

Let assume, a node corresponding to the key is already the part of underlying $RL$ which is having a time-stamp of $m_1$ as $T_1$ from Observation 41. Let say $m_2$ of $T_2$ wants to perform on that node, by observing the code at line 6 of $toValidation()$ method of Algo 12, if TS($T_2$) < curr.max.ts.$m_1()$, $T_2$ will return abort, else to succeed, TS($T_2$) > curr.max.ts.$m_1()$ should evaluate to true. Thus, for successful completion of $m_2$ of $T_2$, TS($T_2$) should be greater then the TS($T_1$). Hence, node corresponding to the key, max.ts field should be updated in increasing order of TS values.

Induction Hypothesis: Say, up to k events that change the max.ts field of a node associated with a key always in increasing TS value.

Induction Step: So, as seen from the code, the $(k + 1)^{th}$ event which can change the max.ts field be only one of the following:
1. **Line 26, 30 & 35 of STM_delete() method of Algo 8**: By observing the code, line 18 of STM_delete() method of Algo 8 first invokes IslSearch() method of Algo 4, for finding the node corresponding to the key. Inside the IslSearch() method of Algo 4, it will do the toValidation() method of Algo 12, if (curr.key = key).

   From induction hypothesis, node corresponding to the key is already the part of underlying RL which is having a time-stamp of $m_k$ of $T_k$ from Observation 41. Let say $m_{k+1}$ of $T_{k+1}$ wants to perform on that node, by observing the code at line 6 of toValidation() method of Algo 12, if $TS(T_{k+1}) < curr.max_ts.m_k()$, $T_{k+1}$ will return abort, else to succeed, $TS(T_{k+1}) > curr.max_ts.m_k()$ should evaluate to true. Thus, for successful completion of $m_{k+1}$ of $T_{k+1}$, $TS(T_{k+1})$ should be greater then the $TS(T_k)$. Hence, node corresponding to the key, $max_ts$ field should be updated in increasing order of TS values.

2. **Line 18, 22 & 27 of STM_lookup() method of Algo 3**: By observing the code, line 12 of STM_lookup() method of Algo 3 first invokes IslSearch() method of Algo 4, for finding the node corresponding to the key. Inside the IslSearch() method of Algo 4, it will do the toValidation() method of Algo 12, if (curr.key = key).

   From induction hypothesis, node corresponding to the key is already the part of underlying RL which is having a time-stamp of $m_k$ as $T_k$ from Observation 41. Let say $m_{k+1}$ of $T_{k+1}$ wants to perform on that node, by observing the code at line 6 of toValidation() method of Algo 12, if $TS(T_{k+1}) < curr.max_ts.m_k()$, $T_{k+1}$ will return abort, else to succeed, $TS(T_{k+1}) > curr.max_ts.m_k()$ should evaluate to true. Thus, for successful completion of $m_{k+1}$ of $T_{k+1}$, $TS(T_{k+1})$ should be greater then the $TS(T_k)$. Hence, node corresponding to the key, $max_ts$ field should be updated in increasing order of TS values.

3. **Line 26, 30 & 35 of STM_tryC() method of Algo 4**: By observing the code, line 7 of STM_tryC() method of Algo 4 first invokes IslSearch() method of Algo 4, for finding the node corresponding to the key. Inside the IslSearch() method of Algo 4, it will do the toValidation() method of Algo 12, if (curr.key = key).

   From induction hypothesis, node corresponding to the key is already the part of underlying RL which is having a time-stamp of $m_k$ as $T_k$ from Observation 41. Let say $m_{k+1}$ of $T_{k+1}$ wants to perform on that node, by observing the code at line 6 of toValidation() method of Algo 12, if $TS(T_{k+1}) < curr.max_ts.m_k()$, $T_{k+1}$ will return abort, else to succeed, $TS(T_{k+1}) > curr.max_ts.m_k()$ should evaluate to true. Thus, for successful completion of $m_{k+1}$ of $T_{k+1}$, $TS(T_{k+1})$ should be greater then the $TS(T_k)$. Hence, node corresponding to the key, $max_ts$ field should be updated in increasing order of TS values.

\[\text{Corollary 43. Every successful methods update the max_ts field of a node associated with a key always in increasing TS values.}\]

\[\text{Lemma 44. If STM_begin(T_i) occurs before STM_begin(T_j) then TS(T_i) preceds TS(T_j).} \]

\[\text{Formally, } (\forall T_i \in H : (\text{STM_begin}(T_i) < \text{STM_begin}(T_j)) \Rightarrow (TS(T_i) < TS(T_j))). \]

\[\text{Proof. (Only if)} \text{ If (STM_begin}(T_i) < \text{STM_begin}(T_j)) \text{ then (TS}(T_i) < TS(T_j)). \text{ Lets assume (TS}(T_j) < TS(T_i). \text{ From Observation 40,} \]

\[\text{STM_begin}(T_j) \prec_H \text{STM_begin}(T_i) \quad (57) \]

\[\text{but we know that,} \]

\[\text{STM_begin}(T_j) \succ_H \text{STM_begin}(T_i) \quad (58) \]
Which is a contradiction thus, \((TS(T_i) < TS(T_j))\).

\((i f\) \( (TS(T_i) < TS(T_j)) \) then \((STM\text{-}begin(T_i) < STM\text{-}begin(T_j))\). Let us assume \((STM\text{-}begin(T_j) < STM\text{-}begin(T_i))\). From Observation 40.

\[
TS(T_j) < TS(T_i)
\]

but we know that,

\[
TS(T_j) > TS(T_i)
\]

Again, a contradiction.

\textbf{Lemma 45.} \( (T_i, T_j) \in conflict(H) \Rightarrow TS(T_i) < TS(T_j) \).

\textbf{Proof.} \((T_i, T_j)\) can have two kinds of conflicts from our conflict notion.

1. \textbf{If} \((T_i, T_j)\) \textbf{is an real-time edge:} Since, \(T_i\) \& \(T_j\) are real time ordered. Therefore,

\[
T_i\text{-}lastEvt \prec_H T_j\text{-}firstEvt
\]

And from program order of \(T_i\),

\[
T_i\text{-}firstEvt \prec_H T_i\text{-}lastEvt \Rightarrow STM\text{-}begin(T_i) \prec_H T_i\text{-}lastEvt
\]

From eq(61) and eq(62) implies that,

\[
T_i\text{-}firstEvt \prec_H T_j\text{-}firstEvt \Rightarrow STM\text{-}begin(T_i) \prec_H STM\text{-}begin(T_j)
\]

\[
\text{Lemma 43}
\]

\[
TS(T_i) < TS(T_j)
\]

2. \textbf{If} \((T_i, T_j)\) \textbf{is a conflict edge:} We prove this case by contradiction, lets assume \((T_i, T_j)\) \in conflict(H) \& TS(T_i) < TS(T_j). Given that \((T_i, T_j)\) \in conflict(H) and from Definition 37 we get, \(m_i \prec_H m_j\).

\(m_i\) can be 
\(r_u\_methods\) or 
\(upd\_methods\) (which are taking the effects in 
\(STM\_tryC()\) method of 
\(Algo\_4\)) and we know that after the LP of \(m_i\), \(node\) corresponding to the key should be there in 
\(RL\) (from Corollary 31 \& Definition 9) and the time-stamp of that \(node\) corresponding to key should be equal to time-stamp of this method transaction-time-stamp from Corollary 31 \& Observation 41.

From Lemma 7 \& Lemma 14 we have that the nodes in the underlying data-structure are in increasing order of their keys, thus the key on which the operation is working has a unique location in underlying data-structure from Corollary 11. So, when the 
\(IsSearch()\) is invoked from a method \(m_j\) of \(T_j\), it returns correct location \((preds[0], preds[1], curr\_s[0], curr\_s[1])\) of corresponding key as observed from Observation 7 \& Lemma 8.

Now, \(m_j\) similar to \(m_i\) take effect on the same node represented by key \(k\) (from Observation 9 \& Corollary 11) \& from Observation 13 we know that the \(node\) corresponding to the key \(k\) is still reachable via 
\(RL\). Thus, we know that \(T_i\) \& \(T_j\) will work on same node with key \(k\).

By observing the code at line 6 \& 9 of 
\(toValidation()\) method of 
\(Algo\_12\), we know since, \(TS(T_j) < curr\_max\_ts.m_i()\), \(T_j\) will return abort from Corollary 43. In Algo 12 for 
\(toValidation()\) to succeed, \(TS(T_j) > curr\_max\_ts.m_i()\) should evaluate to true from Corollary 43. Thus, \(TS(T_j) < TS(T_i)\), a contradiction. Hence, \( (T_i, T_j) \in conflict(H) \Rightarrow TS(T_i) < TS(T_j) \).
Lemma 46. If \((T_1, T_2 \cdots T_n)\) is a path in \(CG(H)\), this implies that \((TS(T_1) < TS(T_2) < \cdots < TS(T_n))\).

Proof. The proof goes by induction on length of a path in \(CG(H)\).

**Base Step:** Assume \((T_1, T_2)\) be a path of length 1. Then, from Lemma 45 \((TS(T_1) < TS(T_2))\).

**Induction Hypothesis:** The claim holds for a path of length \((n-1)\). That is, \(TS(T_1) < TS(T_2) < \cdots < TS(T_{n-1})\) (64)

**Induction Step:** Let \(T_n\) is a transaction in a path of length \(n\). Then, \((T_{n-1}, T_n)\) is path in \(CG(H)\). Thus, it follows from Lemma 45 that,

\[ TS(T_{n-1}) < TS(T_n) \overset{eq}(64) \Rightarrow (TS(T_1) < TS(T_2) < \cdots < TS(T_n)) \] (65)

Hence, the lemma.

Theorem 47. \(CG(H)\) is acyclic.

Proof. Assume that \(CG(H)\) is cyclic, then their exist a cycle say of form \((T_1, T_2 \cdots T_n, T_1)\), for all \((n \geq 1)\). From Lemma 46,

\[ TS(T_1) < TS(T_2) < \cdots < TS(T_n) \Rightarrow TS(T_1) < TS(T_1) \] (66)

But, this is impossible as each transaction has unique time-stamp, refer Observation 40. Hence the theorem.

Theorem 48. A legal history \(H\) is co-opaque iff \(CG(H)\) is acyclic.

Proof. (Only if) If \(H\) is co-opaque and legal, then \(CG(H)\) is acyclic: Since \(H\) is co-opaque, there exists a legal t-sequential history \(S\) equivalent to \(\bar{H}\) and \(S\) respects \(\preceq_{RT}^{H}\) and \(\preceq_{CO}^{H}\) (from Definition 1). Thus from the conflict graph construction we have that \((CG(\bar{H})=CG(H))\) is a sub graph of \(CG(S)\). Since \(S\) is sequential, it can be inferred that \(CG(S)\) is acyclic using Lemma 34. Any sub graph of an acyclic graph is also acyclic. Hence \(CG(H)\) is also acyclic.

(if) If \(H\) is legal and \(CG(H)\) is acyclic then \(H\) is co-opaque: Suppose that \(CG(H) = CG(\bar{H})\) is acyclic. Thus we can perform a topological sort on the vertices of the graph and obtain a sequential order. Using this order, we can obtain a sequential schedule \(S\) that is equivalent to \(\bar{H}\). Moreover, by construction, \(S\) respects \(\preceq_{RT}^{H} = \preceq_{RT}^{\bar{H}}\) and \(\preceq_{CO}^{H} = \preceq_{CO}^{\bar{H}}\).

Since every two operations related by the conflict relation in \(S\) are also related by \(\preceq_{CO}^{\bar{H}}\), we obtain \(\preceq_{CO}^{H} \subseteq \preceq_{CO}^{\bar{H}}\). Since \(H\) is legal, \(\bar{H}\) is also legal. Combining this with Lemma 39 we get that \(S\) is also legal. This satisfies all the conditions necessary for \(H\) to be co-opaque.

E Preliminary results of OSTM

We build initial version of OSTM where each method \(STM_{insert}\), \(STM_{delete}\) and \(STM_{lookup}\) is a single transaction. And to compare against we take a read/write STM with its two implementations one with Basic time stamp protocol and another with Serialization graph testing. We evaluate the SET application which has \(add\), \(remove\) and \(find\) methods. The evaluation is done with a setup where 40% of the operations are \(find\), 40% are \(remove\) and 20% are \(add\).

setup: ram cpu blah blah..... The evaluation is done on following two criteria:

1. Average time taken per execution (Figure 26).
2. Average number aborts per execution (Figure 27).

As evident from the plots $OSTM$ takes lesser time also the number of aborts are reduced in comparison to the average time and aborts for read/write STM with underlying BTO and SGT protocols.

**Figure 26** Average time taken by RWSTMs v/s OSTM

**Figure 27** Average aborts per transaction by RWSTMs v/s OSTM