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, cs15mtech01001, cs15resch01001)@iith.ac.in

Abstract

Major focus of software transaction memory systems (STMs) has been to felicitate 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...

1 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_begin* which begins a transaction, *t_read* which reads from a buffer, *t_write* which writes onto a buffer, *tryC* which validates the operations of the transaction and tries to commit. If validation is successful then it returns commit otherwise STMs export *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].

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We would like to propose an alternative model to achieve composability with greater concurrency for *STMs* by considering higher-level objects which milk the richer semantics of object level operations. We motivate this with an interesting example.

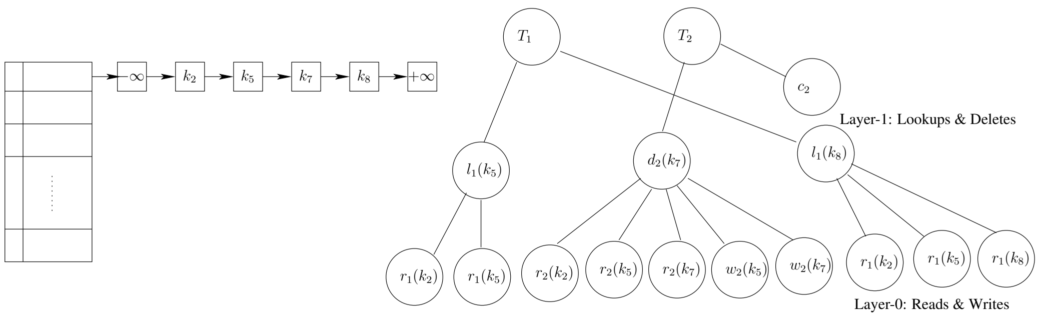
Consider an *OSTM* operating on the `hash-table` object exports the following methods: *t_begin* which begins a transaction (same as in *RWSTMs*), *t_insert* which inserts a value for a given key, *t_delete* which deletes the value associated with the given key, *t_lookup* which looks up the value associated with the given key and *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 [7] to implement the operations of the list denoted as: *list_insert*, *list_del* and *list_lookup* (referred as contains in [7]). Thus when a transaction invokes *t_insert*, *t_delete* and *t_lookup* methods, the STM internally invokes the *list_insert*, *list_del* and *list_lookup* methods respectively.

Consider an instance of list in which the nodes with keys $\langle k_2\ k_5\ k_7\ k_8\rangle$ are present in the `hash-table` as shown in Figure 1(i) and transactions T_1 and T_2 are concurrently executing *t_lookup*$_1(k_5)$, *t_delete*$_2(k_7)$ and *t_lookup*$_1(k_8)$ as shown in Figure 1(ii). In our representation, we abbreviate *t_insert* as i, *t_delete* as d and *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 *t_lookup* on the keys k_5, k_8. This would internally cause the *OSTM* to invoke *list_lookup* method on keys $\langle k_2, k_5\rangle$ and $\langle k_2, k_5, k_7, k_8\rangle$ respectively.

Concurrently, suppose transaction T_2 invokes the method *t_delete* on key k_7 between the two *t_lookup*s of T_1. This would cause, *OSTM* to invoke *list_del* method of list on k_7. Since, we are using lazy-list approach on the underlying list, *list_del* involves pointing the next field of element k_5 to k_8 and marking element k_7 as deleted. Thus *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 *t_lookup*, *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[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[2]).



(i) Underlying list (ii) H1: Transactional tree history

Figure 1 Motivational example for OSTMs

On the other hand consider the history $H1$ at layer-1 consisting of *t_lookup*, *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 $H1$ 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_insert*, *t_delete* and *t_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`, $ht1$ and $ht2$ such that a process p_1 need to delete k_5 from $ht1$ and insert it into $ht2$ 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 $ht1$ but has not inserted in the $ht2$) 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 $\langle k, v\rangle$. The keys and values are respectively from sets $\mathcal{K}$ and $\mathcal{V}$. The methods that a transaction T_i can invoke are:

(1) *init*$_i$, (2) *t_begin*$_i$, (3) *t_insert*$_i(ht, k, v)$, (4) *t_delete*$_i(ht, k, v)$, (5) *t_lookup*$_i(ht, k, v)$, (6) *tryC*$_i$ and (7) *tryA*$_i$. We assume that each method consists of a inv and rsp event.

Formally, we denote a method m by the tuple $\langle evts(m), <_m\rangle$. Here, $evts(m)$ are all the events invoked by m and the $<_m$ a total order among these events. We denote *t_insert* and *t_delete* as *update* methods (or upd_method) since both of these change the underlying data-structure. We denote *t_delete* and *t_lookup* as *return-value methods (or* rv_method*)* as these operations return values from ht. A method may return ok if successful or $\mathcal{A}$(abort) if it sees inconsistent state of ht.

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 T as the tuple $\langle evts(T), <_T\rangle$. Here $evts(T)$ are all the read/write events at *layer-0* of the transaction. $<_T$ 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 $T_i.firstEvt <_{T_i} rw <_{T_i} T_i.lastEvt$. 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 $<_m$ is a subset of $<_{T_i}$.

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 $\langle evts(H), <_H\rangle$ where all the events are totally ordered by $<_H$. 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 $<_T$ is a subset of $<_H$. Formally, $\langle \forall T \in txns(H) : (evts(T) \subseteq evts(H)) \wedge (<_T \subseteq <_H))\rangle$. 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 $\mathcal{A}$ response or after *tryC*(ok) method. We only consider *well-formed* histories for *OSTM*.

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*$_k(\mathcal{A})$ immediately after the last method of every transaction $T_k \in incomp(H)$. 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 $H.firstKeyMth(\langle ht, k\rangle, T_i)$, where m_{ij} stands for j^{th} method of i^{th} transaction. For a method $m_{ix}(ht, k, v)$ 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 $m_{ij}(ht, k, v) = H.prevKeyMth(m_{ix}, T_i)$.

Real-time Order & Serial Histories: Given a history H, $<_H$ orders all the events in H. For two complete methods m_{ij}, m_{pq} in $methods(H)$, we denote $m_{ij} \prec_H^{MR} m_{pq}$ if rsp$(m_{ij}) <_H$inv(m_{pq}). Here MR stands for method real-time order. It must be noted that all the methods of the same transaction are ordered.

Similarly, for two transactions T_i, T_p in $term(H)$, we denote $(T_i \prec_H^{TR} T_p)$ if $(T_i.lastEvt <_H T_p.firstEvt)$. Here TR stands for transactional real-time order.

We define a history H as *serial* [17] or *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, $\langle (H \text{ is serial}) \implies (\forall T_i \in txns(H) : (T_i \in term(H)) \wedge (\forall T_i, T_p \in txns(H) : (T_i \prec_H^{TR} T_p) \vee (T_p \prec_H^{TR} T_i))\rangle$. Since all the methods within a transaction are ordered, a serial history is also sequential. Refer Figure 15 in Appendix A to shows 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 $v \neq nil$) 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, $rvm_{ij} \neq H.firstKeyMth(\langle ht, k\rangle, T_i) \wedge (m_{ix}(ht, k, v') = H.prevKeyMth(\langle ht, k\rangle, T_i))$ (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. $t\_delete_{ix}(ht, k, v'/nil)$ then $v = nil$.

   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, $rvm_{ij}(ht, k, v) = H.firstKeyMth(\langle ht, k\rangle, T_i) \wedge (v \neq nil)$. Then,
   a. There is a *t_insert* method $t\_insert_{pq}(ht, k, v)$ in $methods(H)$ such that T_p committed before rvm_{ij}. Formally, $\langle \exists t\_insert_{pq}(ht, k, v) \in methods(H) : tryC_p \prec_H^{MR} rvm_{ij}\rangle$.
   b. There is no other update method up_{xy} of a transaction T_x operating on $\langle ht, k\rangle$ in $methods(H)$ such that T_x committed after T_p but before rvm_{ij}. Formally, $\langle \nexists up_{xy}(ht, k, v'') \in methods(H) : tryC_p \prec_H^{MR} tryC_x \prec_H^{MR} rvm_{ij}\rangle$.

   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, $rvm_{ij}(ht, k, v) = H.firstKeyMth(\langle ht, k\rangle, T_i) \wedge (v = nil)$. Then,
   a. There is *t_delete* method $t\_delete_{pq}(ht, k, v')$ in $methods(H)$ such that T_p (which could be T_0 as well) committed before rvm_{ij}. Formally, $\langle \exists t\_delete_{pq}(ht, k, v') \in methods(H) : tryC_p \prec_H^{MR} rvm_{ij}\rangle$. Here v' could be nil.
   b. There is no other update method up_{xy} of a transaction T_x operating on $\langle ht, k\rangle$ in $methods(H)$ such that T_x committed after T_p but before rvm_{ij}. Formally, $\langle \nexists up_{xy}(ht, k, v'') \in methods(H) : tryC_p \prec_H^{MR} tryC_x \prec_H^{MR} rvm_{ij}\rangle$.

   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 legality in Lemma 29 in Appendix D and then we finally show that *OSTM* histories are co-opaque[12] 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 $\overline{H}$, i.e. , $evts(\overline{H}) = evts(S)$ (2) S is legal and (3) S respects the transactional real-time order of H, i.e., $\prec_H^{TR} \subseteq \prec_S^{TR}$.

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., $(\langle ht, k\rangle \in updtSet(T_i)) \wedge (\langle ht, k\rangle \in updtSet(T_j))$, where $updtSet(T_i)$ is update set of T_i. (3) T_i's *tryC* completed before T_j's *tryC*, i.e., $tryC_i \prec_H^{MR} tryC_j$.
- **u-rv** conflict:(1) T_i is committed (2) T_i updates the key k of `hash-table`, ht. T_j invokes a rv_method rvm_{jy} on the key same k of `hash-table` ht which is the first method on $\langle ht, k\rangle$. Thus, $(\langle ht, k\rangle \in updtSet(T_i)) \wedge (rvm_{jy}(ht, k, v) \in rvSet(T_j)) \wedge (rvm_{jy}(ht, k, v) = H.firstKeyMth(\langle ht, k\rangle, T_j))$, where $rvSet(T_j)$ is return value set of T_j. (3) T_i's *tryC* completed before T_j's *rvm*, i.e., $tryC_i \prec_H^{MR} rvm_{jy}$.
- **rv-u** conflict:(1) T_j is committed (2) T_i invokes a rv_method on the key same k of `hash-table` ht which is the first method on $\langle ht, k\rangle$. T_j updates the key k of the `hash-table`, ht. Thus, $(rvm_{ix}(ht, k, v) \in rvSet(T_i)) \wedge (rvm_{ix}(ht, k, v) = H.firstKeyMth(\langle ht, k\rangle, T_i)) \wedge (\langle ht, k\rangle \in updtSet(T_j))$ (3) T_i's *rvm* completed before T_j's *tryC*, i.e., $rvm_{ix} \prec_H^{MR} tryC_j$.

▶ **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 $\overline{H}$, i.e. , $evts(\overline{H}) = evts(S)$ (2) S is legal and (3) S respects the transactional real-time order of H, i.e., $\prec_H^{TR} \subseteq \prec_S^{TR}$ and (4) S preserves conflicts (i.e. $\prec_H^{CO} \subseteq \prec_S^{CO}$) [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 *(V, E)* pair where $V \in txns(H)$ and E can be of following types:

- *conflict edges:* $\{(T_i, T_j) : (T_i, T_j) \in$ 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:* $\{(T_i, T_j) : T_i \prec_H^{TR} T_j\}$

The legality and conflict notion established here are used to prove that histories generated by the *OSTM* are correct or co-opaque[] in Section 5.

4 OSTMs Design

We design the OSTMs 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.

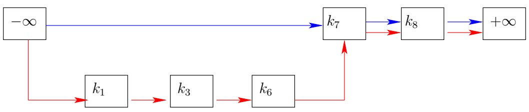


Figure 2 Searching k_8 over lazyskip-list

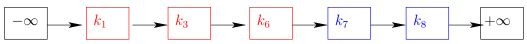


Figure 3 Searching k_8 over lazylist

In case search is invoked from rv_method, and node corresponding to the key is not present in BL and RL then the rv_method will create a node and insert it into underlying data structure as *dead* node. For example lookup wants to search key k_{10} in Figure 2, as key k_{10} is not present in the BL as well as RL then, lookup method will create a new node corresponding to the key k_{10} and insert it into RL (refer the Figure 4).

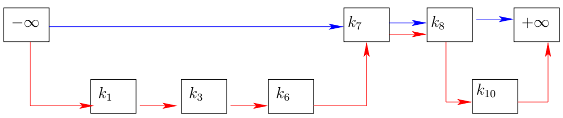


Figure 4 Execution under lazyskip-list

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[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[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 $lu_1(ht, k_1, Nil)$ to ensure opacity. Thus with help of information provided by the dead nodes we can ensure $H1$: T_1 followed by T_2 is the opaque history as depicted in the Figure 6. 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 *OSTM*.

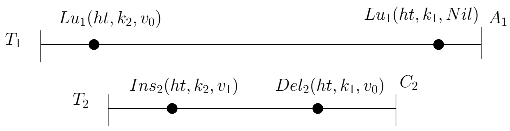


Figure 5 History H is not opaque

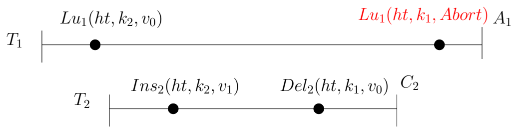


Figure 6 Opaque History H1

4.1 OSTM data-structure design

In proposed *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 $txlog$, which consists of *t_id* and 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.

```
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(*ll_list*) of meta information of each method a transaction executes in its life time. Each entry of the *ll_list* is of type *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 $preds\langle k_m, k_n\rangle$ (k_m is blue node reachable by BL and k_n is red node reachable by RL) and successor as $currs\langle k_p, k_q\rangle$ (k_p is red node reachable by RL and k_q is blue node reachable by blue node) respectively. Here, $\langle k_m, k_q\rangle$ are predecessor(pred[0]) and current(curr[0]) node for BL and $\langle k_n, k_p\rangle$ are predecessor(pred[1]) and current(curr[0]) node for RL. We use word location with $preds$ and $currs$ interchangeably in rest of the paper.

Class *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.

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

```
        getKey&Objid(); getValue();       getAptCurr();
        setValue();     setPreds&Currs(); setOpStatus();   setOpn();  };
enum OPERATION_NAME = {INSERT, DELETE, LOOKUP}
enum STATUS = {ABORT = 0, OK, FAIL, COMMIT}
```

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.

```
struct node{
        int key, value;         bool marked;    struct max_ts;
        lock;                   node rednext;   node bluenext;    };
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*.

```
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.

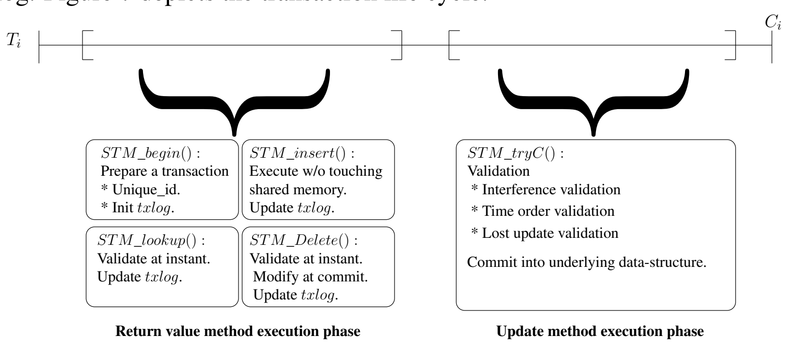


Figure 7 Transaction lifecycle of *OSTM*

Pseudocode convention: In each algorithm ↓ represents the input parameter and ↑ shows the output parameter (or return value) of the corresponding methods(such in and out variables are italicized). Instructions in *read()* and *write()* with in each method denote that they touch the shared memory. Color of preds & curs in algorithm depicts the red or blue node which are accessed by red or blue links respectively.

rv_method execution phase: Initially, in *rv_method execution* phase each transaction invokes $STM\_begin()$ of Algo 6 (in Appendix C) for getting unique transaction id and *local log*. Then transaction may encounter the upd_method or rv_method. *STM_insert()* of Algo 7 (in Appendix C), first looks for the node corresponding to the key into the ll_list(Line 2). If key is not found then it will create the ll_entry and store the value, operation name and status(Line 3 to Line 6) into it which would be validated and realized in shared memory in *STM_tryC()*.

STM_tryC() and rv_method of *OSTM* methods use *lslSearch()* to find the location at the lazyskip-list(thus the name) in lock free manner. Line 3 to Line 8 and Line 9 to Line 14 of Algo 1 find the location at lazyskip-list for BL and RL . This is motivated by the search in lazylist [8, chap 9](REF ALGO). The $preds$ and $currs$ thus identified are subjected to *interferenceValidation()* of Algo 2 and *toValidation()* of Algo 12(in Appendix C) after acquiring locks on the $preds$ and $currs$ (Line 15 to Line 18 of Algo 1). If the validation succeeds *lslSearch()* returns the correct location to the operation which invoked it, otherwise *lslSearch()* retries(if interference detected) or aborts(if time order violated) post releasing locks(Line 21 to Line 24) before finally returning.

Algorithm 1 lslSearch($obj\_id \downarrow, key \downarrow, preds[] \uparrow, currs[] \uparrow, value \uparrow, val\_type \downarrow$)

```
1:  procedure LSLSEARCH
2:      while (op_status = RETRY) do
3:          head ← getLslHead(obj_id ↓, key ↓);
4:          preds[0] ← read(head) ;
5:          currs[1] ← read(preds[0].BL) ;
6:          while (read(currs[1].key) < key) do
7:              preds[0] ← currs[1] ;
8:              currs[1] ← read(currs[1].BL) ;
9:          value ← read(currs[1].value) ;
10:         preds[1] ← preds[0] ;
11:         currs[0] ← preds[0].RL;
12:         while (read(currs[0].key) < key) do
13:             preds[1] ← currs[0] ;
14:             currs[0] ← read(currs[0].RL) ;
15:         preds[0].lock() ;
16:         preds[1].lock();
17:         currs[0].lock();
18:         currs[1].lock();
19:         op_status ← validation(key ↓, preds[] ↓, currs[] ↓,
    val_type ↓);
20:         if (op_status = RETRY) then
21:             preds[0].unlock();
22:             preds[1].unlock();
23:             currs[0].unlock();
24:             currs[1].unlock();
25:     return op_status ;
```

Algorithm 2 interferenceValidation($preds[] \downarrow, currs[] \downarrow$)

```
procedure INTERFERENCEVALIDATION
    if ((read(preds[0].marked)) & (read(currs[1].marked)) & ((read(preds[0].BL) ≠ currs[1])) & ((read(preds[1].RL) ≠
currs[0]))) then
        return RETRY ;
    else
        return OK ;
```

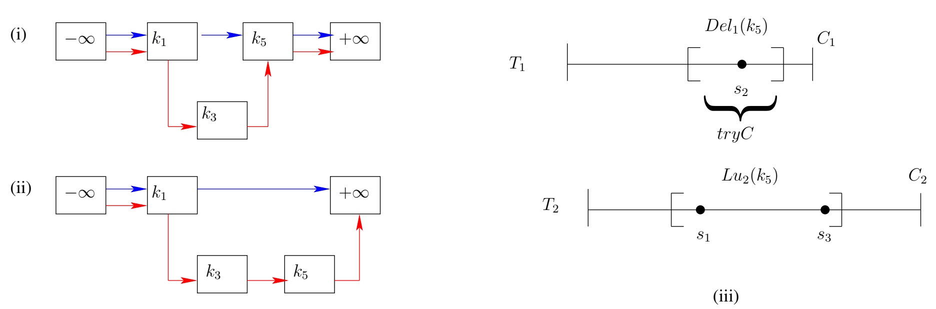


Figure 8 Interference Validation for conflicting concurrent methods on key k_5

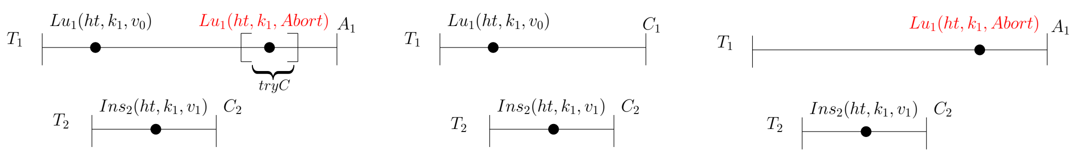
Interference validation helps detecting the execution where underlying data structure has been changed by second concurrent transaction while first was under execution without it realizing. This can be illustrated with Figure 8. Consider the history in Figure 8(iii) where two conflicting transactions

T_1 and T_2 are trying to access key k_5, here s_1, s_2 and s_3 represent the state of the lazyskip-list at that instant. Let at s_1 both the methods record the same $preds\langle k_1, k_3\rangle$ and $currs\langle k_5, k_5\rangle$ with the help of *lslSearch()* for key k_5 (refer Figure 8(i)). Now, let $Del_1(k_5)$ acquire the lock on the $preds$ and $currs$ before the $Lu_2(k_5)$ and delete the node corresponding to the key k_5 from BL leading to state s_2 (in Figure 8(iii)) and commit. Figure 8(ii) shows the state s_2 where key k_5 is the part of RL. Now, *interferenceValidation()* (in Algo 2) will identify that location of $Lu_2(k_5)$ is no more valid due to pred.$BL \neq$ curr at Line 2 of Algo 2. This strategy of validation is similar to [8, chap 9](ALGO REF). Thus, *lslSearch()* will retry to find the updated location for $Lu_2(k_5)$ at state s_3 (in Figure 8(iii)) and eventually T_2 will commit.

Consider $STM\_lookup_i(ht, k)$. 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 *STM_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 *STM_insert()* (or *STM_lookup()*) on same key then the subsequent *STM_lookup()* of the same transaction returns the previous value(Line 6) inserted (or observed) without accessing shared memory, and if the last operation was a *STM_delete()* then *STM_lookup()* returns the value NULL (Line 9). We denote this as *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 STM_lookup($obj\_id \downarrow, key \downarrow, value \uparrow, op\_status \uparrow$)

```
1: procedure STM_LOOKUP
2:     op_status ← RETRY ;
3:     if (txlog.findInLL(obj_id ↓, key ↓)) then
4:         opn ← ll.getOpn(obj_id ↓, key ↓) ;
5:         if ((INSERT = opn )||( LOOKUP = opn)) then
6:             value ← ll.getValue(obj_id ↓, key ↓) ;
7:             op_status ← ll.getOpStatus(obj_id ↓, key ↓) ;
8:         else if (DELETE = opn) then
9:             value ← NULL ;
10:            op_status ← FAIL ;
11:    else
12:        op_status ← lslSearch(obj_id ↓, key ↓, preds[] ↑, currs[] ↑, value_BL ↑, rv ↓) ;
13:        if (op_status = ABORT) then
14:            tryAbort(obj_id ↓) ;
15:        else
16:            if (read(currs[1].key) = key) then
17:                op_status ← OK ;
18:                write(currs[1].max_ts.lookup, TS(t_i)) ;
19:                value ← value_BL ;
20:            else if (read(currs[0].key) = key) then
21:                op_status ← FAIL ;
22:                write(currs[0].max_ts.lookup, TS(t_i)) ;
23:                value ← NULL ;
24:            else
25:                lslIns(preds[] ↓, currs[] ↓, RL ↓) ;
26:                op_status ← FAIL ;
27:                write(node.max_ts.lookup, TS(t_i)) ;
28:                value ← NULL ;
29:            new ll_entry;                      ▷ log entry created if not exists
30:            ll.setPreds&Currs(obj_id ↓, key ↓, preds[] ↓, currs[] ↓) ;
31:            ll.setOpn((obj_id ↓, key ↓, LOOKUP ↓) ;
32:            preds[0].unlock() ;
33:            preds[1].unlock();
34:            currs[0].unlock();
35:            currs[1].unlock();
36:        ll.setOpStatus(obj_id ↓, key ↓, op_status ↓) ;
37:        return ;
```



(i) Invalid schedule of two time validation (ii) Valid schedule of one time validation (iii) Early detection of invalid schedule

Figure 9 Advantages of lookup validated once

If *STM_lookup()* is the first operation on a particular key then it has to do a wait free traversal (Line 12) with the help of *lslSearch()*(Algo 1) to identify the target node($preds$ and $currs$) to be logged in ll_list for subsequent methods in *rv_method execution* phase (discussed above for the case where *STM_lookup()* is the subsequent method). If the node is present as blue(red) node then it updates the operation status as OK(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 *STM_lookup()*. The node is inserted to take care of situations as illustrated in Figure 5 & Figure 6 . Finally, it updates the meta information in ll_list and releases the locks acquired inside *lslSearch()*(Line 12).

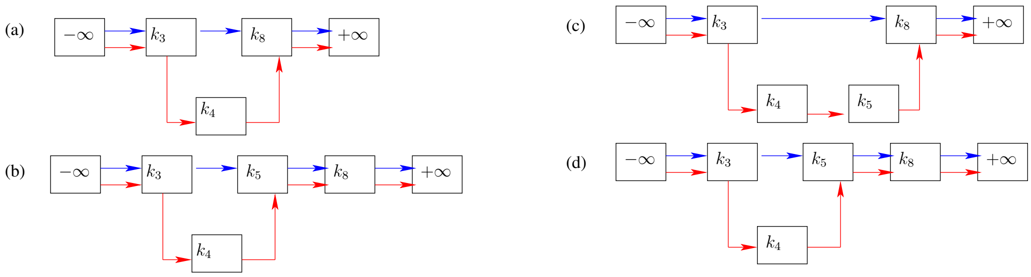
We prefer *STM_lookup()* to be validated instantly and is never validated again in *STM_tryC()* as the design choice to aid performance. Lets consider *OSTM* history in Figure 9(i), if we would have validated $Lu(ht, k_1, v_0)$ again during $tryC$, T_1 would abort due to time order violation[21], but 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 $Lu(ht, k_1, Abort)$ 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 Algo 8 (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[2]. 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 donot worry about it.

Algorithm 4 STM_tryC($txstatus$ ↑)

```
1:  procedure STM_TRYC
2:      t_i ← getTid() ;
3:      ll_list ← ll.get(t_id ↓) ;
4:      ordered_ll_list ← ll.sort (ll_list ↓) ;
5:      while (ll_entry_i ← next(ordered_ll_list)) do
6:          (key, obj_id) ← ll.getKey&Objid(ll_entry_i ↓) ;
7:          op_status ← lslSearch(obj_id ↓, key ↓, preds[] ↑, currs[] ↑, value_BL ↑, 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_i ← next(ordered_ll_list)) do
13:         (key, obj_id) ← ll.getKey&Objid(ll_entry_i ↓) ;
14:         opn ← ll_entry_i.opn ;
15:         lostUpdateValdation(ll_entry_i ↓, preds[] ↑, currs[] ↑) ;
16:         if (INSERT = opn) then
17:             if (read(currs[1].key) = key) then
18:                 value ← read(currs[1].value) ;
19:                 write(currs[1].value, value) ;
20:                 ll.setOpStatus(obj_id ↓, key ↓, OK ↓) ;
21:                 write(currs[1].max_ts.insert, TS(t_i)) ;
22:             else if (read(currs[0].key) = key) then
23:                 lslIns(preds[] ↓, currs[] ↓, RL_BL ↓) ;
24:                 ll.setOpStatus(obj_id ↓, key ↓, OK ↓) ;
25:                 write(currs[1].max_ts.insert, TS(t_i)) ;
26:             else
27:                 lslIns(preds[] ↓, currs[] ↓, BL ↓) ;
28:                 ll.setOpStatus(obj_id ↓, key ↓, OK ↓) ;
29:                 write(node.max_ts.insert, TS(t_i)) ;
30:         else if (DELETE = opn) then
31:             if (read(currs[1].key) = key) then
32:                 lslDel(preds[] ↓, currs[] ↓) ;
33:                 ll.setOpStatus(obj_id ↓, key ↓, OK ↓) ;
34:                 write(currs[1].max_ts.delete, TS(t_i)) ;
35:             else
36:                 ll.setOpStatus(obj_id ↓, key ↓, FAIL ↓) ;
37:                 write(currs[0].max_ts.delete, TS(t_i)) ;
38:     releaseOrderedLocks(ordered_ll_list ↓) ;
39:     txstatus ← OK ;
40:     txlog.setStatus(txstatus ↓, OK ↓) ;
41:     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 Algo 4) 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 34. 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 lazyskip-list at location $preds\langle k_3, k_4\rangle$ and $currs\langle k_8, k_8\rangle$. Figure 10(i)(a) is lazyskip-list before addition of k_5 and Figure 10(i)(b) is lazyskip-list state post addition. Similarly, Figure 10(ii) represents the case when k_5 is present in RL. It adds k_5 to lazyskip-list at location $pred\langle k_3, k_4\rangle$ and $curr\langle k_5, k_8\rangle$. Figure 10(i)(c) is lazyskip-list before addition of k_5 and Figure 10(i)(d) is lazyskip-list state post addition. In case of $del(k_5)$ from lazyskip-list when k_5 is present in BL Figure 11(i) represent the lazyskip-list state before k_5 is deleted at location $pred\langle k_1, k_3\rangle$ and $curr\langle k_5, k_5\rangle$ and Figure 11(ii) represents the lazyskip-list state after deletion.



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

(ii) When k_5 is present in RL

Figure 10 $Ins(k_5)$ using *lslIns()* in *STM_tryC()*

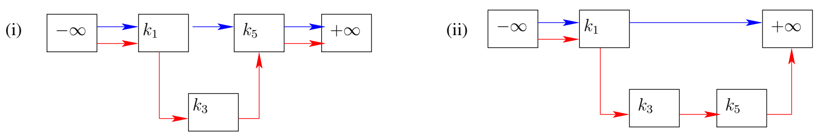


Figure 11 $Del(k_5)$ using *lslDel()* in *STM_tryC()*

While updating the methods of same transaction from its log, the $preds$ and $currs$ 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 *STM_tryC()* at state s where $ins(k_5)$ and $ins(k_7)$ are waiting to take effect over the lazyskip-list. The lazyskip-list at s is as in Figure 12(i) also $ins(k_5)$ and $ins(k_7)$ have $pred\langle k_3, k_4\rangle$ and $curr\langle k_8, k_8\rangle$ as their location. Now, Lets say $ins(k_5)$ 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 $preds$ and $currs$ of $ins(k_7)$ are still k_3 and k_8 thus it wrongly adds k_7 between k_3 and k_8 overwriting $ins(k_5)$ as shown in Figure 12(iii) with dotted links. We correct this through *lostUpdateValidation()* which is *lostUpdateValidation()* is invoked before every upd_method over the lazyskip-list in update phase of *STM_tryC()*(Line 12 to Line 37 of Algo 4). Figure 12 represents the functionality of *lostUpdateValidation()* of Algo 5. Here, If *lostUpdateValidation()* fails for any upd_method then as a corrective measure the $preds$ and $currs$ of the upd_method under execution will be updated using the previous upd_method's $preds$ and $currs$ with the help of its ll_entry.

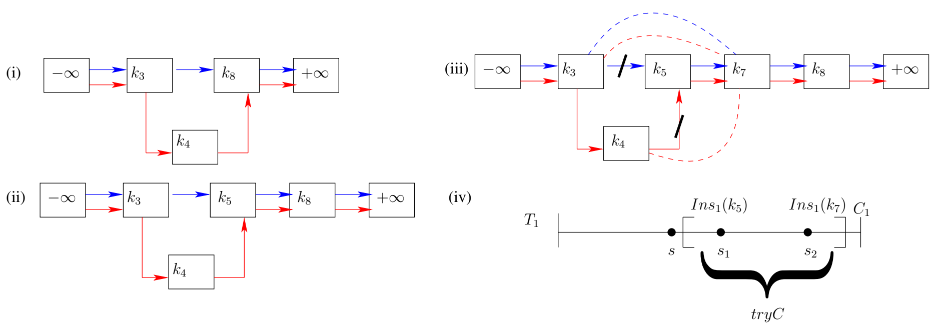


Figure 12 Problem in execution without lostUpdateValidation() ($ins(k_5)$ and $ins(k_7)$). (i) lazyskip-list at state s. (ii) lazyskip-list at state s_1. (iii)lazyskip-list at state s_2 (lost update problem).

Algorithm 5 lostUpdateValidation($ll\_entry_i \downarrow, preds[] \uparrow, currs[] \uparrow$)

```
procedure LOSTUPDATEVALIDATION
    ll.getAllPreds&Currs(ll_entry_i ↓, preds[] ↑, currs[] ↑) ;
    if ((read(preds[0].marked))|| (read( preds[0].BL) != currs[1])) then
        if ((ll_entry_{i-1}.opn) = INSERT) then
            preds[0] ← (ll_entry_{i-1}.key) ;
        else
            preds[0] ← (ll_entry_{i-1}.preds[0]) ;
    if (read( preds[1].BL) != currs[0]) then
        preds[1] ← (ll_entry_{i-1}.key) ;
```

5 Correctness of OSTMs

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 *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 *STM_insert()*, *STM_delete()*, *STM_lookup()* and *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 *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).

▶ **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 32) 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 lazylinked list[5].

Hassan[] uses C-SWC model to prove that OTB transactions compose. We on otherhand 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 composabale applications easier than its lock based counterparts[18, 4], not to be forgotten scalabilty 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[12] thus composable.

OSTM combines the scalable abstraction and ease of programming from *STM*s with our efficient mechanism of achieving composability using object level semantics. Our prototype implementation of *OSTM* shows significant performance gain over read/write STM for a simple SET application(in Appendix E). 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(transfer in SET, hash table etc.). We believe that *OSTM* would be a significant contribution for achieving the goal of efficienct, 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: `http://arxiv.org/abs/0910.0366`.

2 Rachid Guerraoui and Michal Kapalka. On the Correctness of Transactional Memory. In *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. `doi:http://doi.acm.org/10.1145/1345206.1345233`.

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 *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. `doi:http://doi.acm.org/10.1145/1065944.1065952`.

5 Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. On developing optimistic transactional lazy set. In *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, *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: `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. *Parallel Processing Letters*, 17(4):411–424, 2007. URL: `http://dx.doi.org/10.1142/S0129626407003125`, `doi:10.1142/S0129626407003125`.

8 M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier Science, 2012. URL: `https://books.google.co.in/books?id=pFSwuqtJgxYC`.

9 Maurice Herlihy and J. Eliot B.Moss. Transactional memory: Architectural Support for Lock-Free Data Structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993. `doi:http://doi.acm.org/10.1145/173682.165164`.

10 Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *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. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. `doi:http://doi.acm.org/10.1145/78969.78972`.

12 Petr Kuznetsov and Sathya Peri. Non-interference and local correctness in transactional memory. *Theor. Comput. Sci.*, 688:103–116, 2017. 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 *OPODIS*, pages 112–127, 2011.

14 Kfir Lev-Ari, Gregory V. Chockler, and Idit Keidar. On correctness of data structures under reads-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`, `doi: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`, `doi: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.

A Appendix

Methods: The n processes access a collection of *transaction objects* via atomic *transactions* supported by a OSTMs. 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 $\langle k, v\rangle$. The keys and values are respectively from sets $\mathcal{K}$ and $\mathcal{V}$. The methods that a transaction T_i can invoke are: (1) $t\_insert_i(ht, k, v)$: this method inserts the pair $\langle k, v\rangle$ into object ht and return ok. If ht already has a pair $\langle k, v'\rangle$ then v' gets replaced with v. (2) $t\_delete_i(ht, k, v)$: if ht has a $\langle k, v\rangle$ pair then this operation deletes the pair and returns v. If no such $\langle k, v\rangle$ pair is present in ht, then the operation returns nil. (3) $t\_lookup_i(ht, k, v)$: if ht has a $\langle k, v\rangle$ pair then this operation returns v. If no such $\langle k, v\rangle$ pair is present in ht, then the method returns nil. It can be seen that *t_lookup* is similar to *t_delete*.

For simplicity, we assume that all the values inserted by transactions through *t_insert* method are unique. We denote *t_insert* and *t_delete* as *update* methods since both these change the underlying data-structure.We denote *t_delete* and *t_lookup* as *return-value methods or rv_methods* as these return values which are different from ok.

In addition to these return values, each of these methods can always return an abort value $\mathcal{A}$ which implies that the transaction T_i is aborted. A method m_i returns $\mathcal{A}$ if m_i along with all the methods of T_i executed so far are not consistent (w.r.t correctness-criterion which is formally defined later).

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

When any method of T_i returns $\mathcal{A}$, 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 $\mathcal{A}$ 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 a inv and rsp event. Specifically, the inv & rsp events of the methods of a transaction T_i are: (1) $t\_insert_i(ht, k, v)$: inv($t\_insert_i(ht, k, v)$) and rsp($t\_insert_i(ht, k, v, ok/\mathcal{A})$). (2) $t\_delete_i(ht, k, v)$: inv($t\_delete_i(ht, k)$) and rsp($t\_delete_i(h, k, v/nil/\mathcal{A})$). (3) $t\_lookup_i(h, k, v)$: inv($t\_lookup_i(h, k)$) and rsp($t\_lookup_i(h, k, v/nil/\mathcal{A})$). (4) $tryC_i$: inv($tryC_i()$) and rsp($tryC_i(ok/\mathcal{A})$). (5) $tryA_i$: inv($tryA_i()$) and rsp($tryA_i(\mathcal{A})$).

For clarity, we have included all the parameters of inv event in 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 inv and the last event is rsp.

Formally, we denote a method m by the tuple $\langle evts(m), <_m\rangle$. Here, $evts(m)$ are all the events invoked by m and the $<_m$ a total order among these events. For instance, the method $l_{11}(k_5)$ of Figure 13 is represented as: inv($l_{11}(h, k_5)$) $r_{111}(k_2, o_2) r_{112}(k_5, o_5)$ rsp($l_{11}(h, k_5, o_5)$). 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 rw of m, inv(m) $<_m rw <_m$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 e_{pqr} belonging to some other method m_{pq} (of transaction T_p) either e_{pqr} occurs before inv(m_{ij}) or after rsp(m_{ij}). Formally, $\langle m_{ij} \in methods(H)$: m_{ij} is isolated $\equiv (\forall m_{pq} \in methods(H), \forall e_{pqr} \in m_{pq} : e_{pqr} <_Hinv(m_{ij}) \vee$rsp$(m_{ij}) <_H e_{pqr})\rangle$. For instance in $H1$ shown in Figure 1(ii), $d_2(k_2)$ is isolated. In fact all the methods of $H1$ are isolated.

Consider history $H2$ shown in Figure 14. It can be seen that the all the three methods in $H2$,

(l_{11}, d_{21}, l_{12}) are not isolated.

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. Thus, it can be seen that $H1$ is sequential whereas $H2$ is not. 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 $tryA_k(\mathcal{A})$ immediately after the last method of every transaction $T_k \in incomp(H)$. 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. 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 $H.firstKeyMth(\langle ht, k\rangle, T_i)$. For a method $m_{ix}(ht, k, v)$ 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 $m_{ij}(ht, k, v) = H.prevKeyMth(m_{ix}, T_i)$.
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).

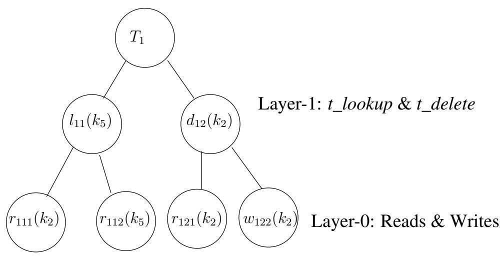


Figure 13 T1 : A sample transaction on lazyskip-list (of Figure 1(i)) representing a `hash-table` object.

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 $\langle evts(T), <_T\rangle$. Here $evts(T)$ 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: inv$(l_{11}(ht, k_5))\, r_{111}(k_2, o_2) r_{112}(k_5, o_5)$ rsp$(l_{11}(ht, k_5, o_5))$ inv$(d_{12}(ht, k_2))\, r_{121}(k_2, o_2)\, w_{122}(k_2, o_2)$ rsp$(d_{12}(ht, k_2, o_2))$. 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 $T_i.firstEvt$ and $T_i.lastEvt$. Given any other read-write event rw in T_i, we assume that $T_i.firstEvt <_{T_i} rw <_{T_i} T_i.lastEvt$.

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 $<_{T_i}$. Formally, $\langle \forall m \in methods(T_i) : evts(m) \subseteq evts(T_i) \wedge <_m \subseteq <_{T_i}\rangle$.

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 $<_{T_i}$. Formally, $\langle \forall m_p, m_q \in methods(T_i) : (m_p <_{T_i} m_q) \vee (m_q <_{T_i} m_p))\rangle$.

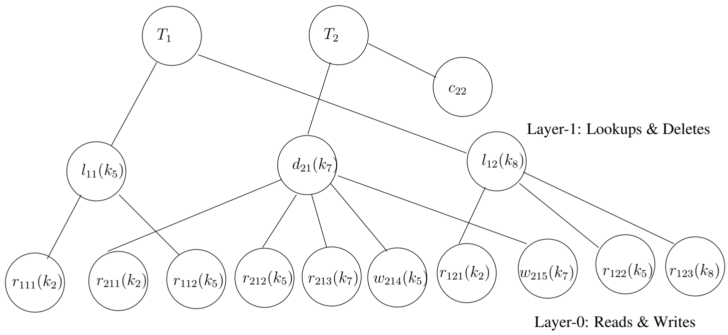


Figure 14 H2 : A non-sequential History.

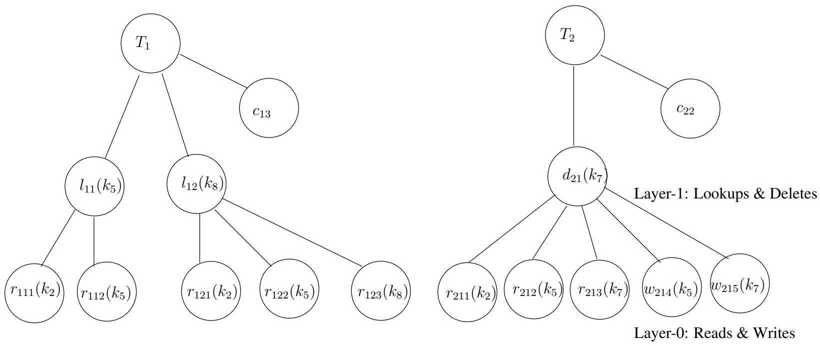


Figure 15 A serial History

Legal History: If 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 $Lu_{ij}(ht, k_5, v_5)$ of transaction T_i on same key k_5 is $Ins_{ix}(ht, k_5, v_5)$. So, $Lu_{ij}(ht, k_5, v_5)$ will return the same value which will be inserted by previous method $Ins_{ix}(ht, k_5, v_5)$. Same technique will be follow in Figure 16(ii) and Figure 16(iii).

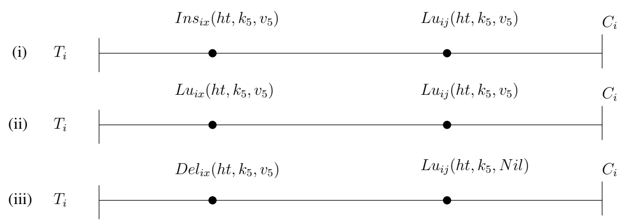


Figure 16 *STM_lookup()* is returning the same value as previous method of the same transaction on same key

If 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 $Lu_{ij}(ht, k, v_p)$ of transaction T_i on same key k is $Ins_{pq}(ht, k, v_p)$ of transaction T_p. So, $Lu_{ij}(ht, k, v_p)$ will return the same value which has been inserted by $Ins_{pq}(ht, k, v_p)$ and there can't be any other transaction upd_method working on the same key between T_p and T_i. Figure 18 represents, previous closest method of committed transaction T_p is $Del_{pq}(ht, k, v_p)$ on key k so $Lu_{ij}(ht, k, Nil)$ of transaction T_i returns nil for same key k.

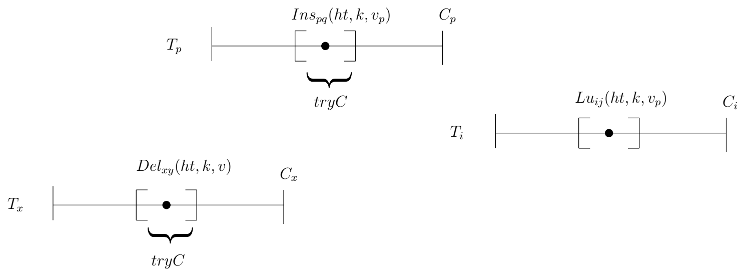


Figure 17 *STM_lookup()* is returning the same value as previous closest conflicting method of committed transaction

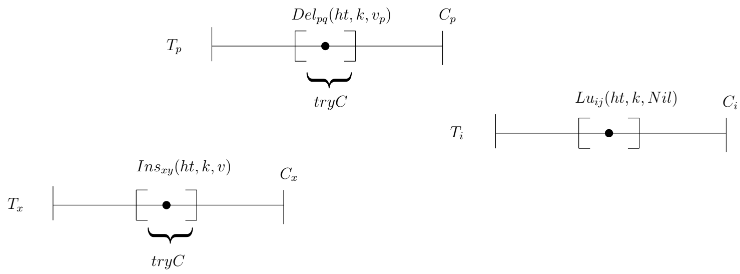


Figure 18 *STM_lookup()* is returning the same value as previous closest conflicting method of committed transaction

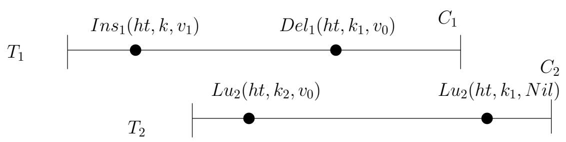


Figure 19 Legal History H2

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($obj\_id \downarrow, key \downarrow, value \downarrow$) : 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 $value$, opn 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($preds[] \downarrow$, $currs[] \downarrow$,) at the *upd_method execution* phase.

Algorithm 8 STM_delete($obj\_id \downarrow, key \downarrow, value \uparrow, op\_status \uparrow$) : If the transaction has locally done an operation on the same key then returns apt value and status. Else do the *lslSearch()* 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().

```
procedure STM_DELETE
    op_status ← RETRY ;
    if (txlog.findInLL(obj_id ↓, key ↓)) then
        opn ← ll.getOpn(obj_id ↓, key ↓) ;
        if (INSERT = opn) then
            value ← ll.getValue(obj_id ↓, key ↓) ;
            ll.setValue(obj_id ↓, key ↓, NULL ↓) ;
            ll.setOpn((obj_id ↓, key ↓, DELETE ↓) ;
            op_status ← OK ;
        else if (DELETE = opn) then
            ll.setValue(obj_id ↓, key ↓, NULL ↓) ;
            value ← NULL ;
            op_status ← FAIL ;
        else
            value ← ll.getValue(obj_id ↓, key ↓) ;
            ll.setValue(obj_id ↓, key ↓, NULL ↓) ;
            ll.setOpn((obj_id ↓, key ↓, DELETE ↓) ;
            op_status ← ll.getOpStatus(obj_id ↓, key ↓) ;
    else
        op_status ← lslSearch(obj_id ↓, key ↓, preds[] ↑, curr s[] ↑, value_BL ↑, rv ↓) ;
        if (op_status = ABORT) then
            tryAbort(obj_id ↓) ;
        else
            if (read(currs[1].key) = key) then
                op_status ← OK ;
                write(currs[1].max_ts.lookup, TS(t_i)) ;
                value ← value_BL ;
            else if (read(currs[0].key) = key) then
                op_status ← FAIL ;
                write(currs[0].max_ts.lookup, TS(t_i)) ;
                value ← NULL ;
            else
                lslIns(preds[] ↓, currs[] ↓, RL ↓) ;
                op_status ← FAIL ;
                write(node.max_ts.lookup, TS(t_i)) ;
                value ← NULL ;
            create ll_entry ;
            ll.setValue(obj_id ↓, key ↓, NULL ↓) ;
            ll.setPreds&Currs(obj_id ↓, key ↓, preds[] ↓, currs[] ↓) ;
            ll.setOpn((obj_id ↓, key ↓, DELETE ↓) ;
            preds[0].unlock();
            preds[1].unlock();
            currs[0].unlock();
            currs[1].unlock() ;
    ll.setOpStatus(obj_id ↓, key ↓, op_status ↓) ;
    return ;
```

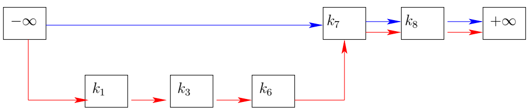


Figure 20 k_{10} is not present in BL as well as RL

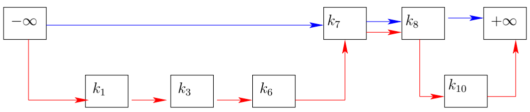


Figure 21 Adding k_{10} into RL

Algorithm 9 lslIns($preds[] \downarrow, currs[] \downarrow, list\_type \downarrow$) : Inserts or overwrites a node in underlying hash table at location corresponding to $preds$ & $currs$.

```
procedure LSLINS
    if ((list_type) = (RL_BL)) then
        write(currs[0].marked, false) ;
        write(currs[0].BL, currs[1]) ;
        write(preds[0].BL, currs[0]) ;
    else if ((list_type) = RL) then
        node = new node() ;
        write(node.marked, True) ;
        write(node.RL, currs[0]) ;
        write(preds[1].RL, node) ;
    else
        node = new node() ;
        write(node.RL, currs[0]) ;
        write(node.BL, currs[1]) ;
        write(preds[1].RL, node ) ;
        write(preds[0].BL, node) ;
```

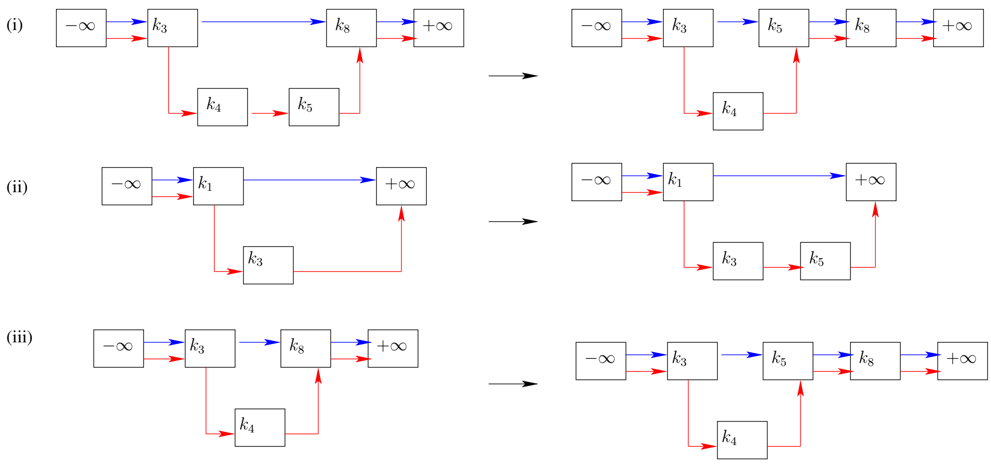


Figure 22 Execution of *lslIns()*: (i) key k_5 is present in RL and adding it into BL, (ii) key k_5 is not present in RL as well as BL and adding it into RL, (iii) key k_5 is not present in RL as well as BL and adding it into RL as well as BL

lslIns($preds$[] ↓, $curr s$[] ↓,) (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($preds$[] ↓, $curr s$[] ↓,) 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 curr s[0]mark field and update the BL to the curr s[1] and preds[0] BL to curr s[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 curr s[0] and RL field of the 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($preds$[] ↓, $curr s$[] ↓) : Deletes a node from blue link in underlying hash table at location corresponding to $preds$ & $curr s$.

1: **procedure** LSLDEL
2: *write*($curr s$[1].marked, True) ;
3: *write*($preds$[0].BL, $curr s$[1].BL) ;

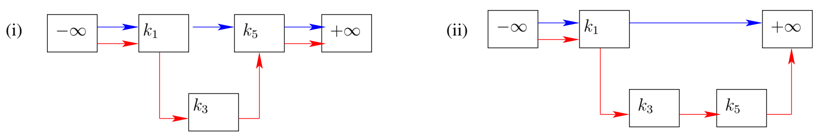


Figure 23 Execution of *lslDel()*: (i) lazyskip-list before k_5 is deleted, (ii) lazyskip-list after k_5 is deleted from BL

lslDel($preds$[] ↓, $curr s$[] ↓) 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(curr s[1]) and changes the BL of preds[1] to curr s[0] as shown in Line 2 and Line 3 of

Algo 10 respectively. Figure 23 shows the deletion of node corresponding to k_5.

Algorithm 11 findInLL($obj\_id \downarrow, key \downarrow$) : Checks whether any operation corresponding to $\langle obj\_id, key \rangle$ is present in ll_list.

```
1: procedure FINDINLL
2:     t_i ← getTid() ;
3:     ll_list ← txlog.getLlList(t_i ↓) ;
4:     while (ll_entry_i ← next(ll_list)) do
5:         if ((ll_entry_i.first.first = obj_id)&(ll_entry_i.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 $\langle obj\_id, key \rangle$ pair(refer Line 5).

Algorithm 12 toValidation($key \downarrow, currs[] \downarrow, val\_type \downarrow$) : Time-order validation for each transaction.

```
procedure TOVALIDATION
    t_i ← getTid() ;
    op_status ← OK ;
    curr ← ll.getAptCurr(currs[] ↓, key ↓) ;
    if ((curr ≠ NULL) ∧ ((curr.key) = key)) then
        if ((val_type = rv) ∧ (TS(t_i) < (read(curr.max_ts.insert(k))) ||
                (TS(t_i) < (read(curr.max_ts.delete(k))))) then
            op_status ← ABORT ;
        else if ((TS(t_i) < (read(curr.max_ts.insert(k))) || TS(t_i) < (read(curr.max_ts.delete(k))) ||
                TS(t_i) < (read(curr.max_ts.lookup(k)))) then
            op_status ← ABORT ;
    return op_status ;
```

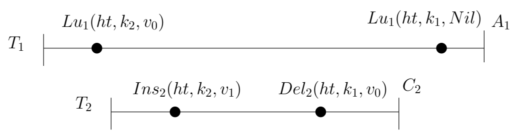


Figure 24 Not maintaining time-stamp: history H is not opaque

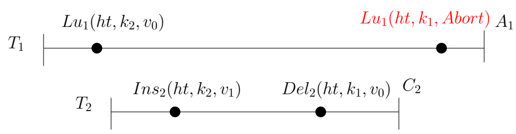


Figure 25 Maintaining time-stamp: opaque history H1

Algorithm 13 validation($key \downarrow, preds[] \downarrow, currs[] \downarrow, val\_type \downarrow$) : Double validation.

```
procedure VALIDATION
    op_status ← (interferenceValidation(preds[] ↓, currs[] ↓));
    if (RETRY ≠ op_status) then
        op_status ← toValidation(key ↓, currs[] ↓, val_type ↓) ;
    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.

Algorithm 14 get_aptcurr($curr s[] \downarrow, key \downarrow$) : Returns a curr node from underlying DS which corresponds to the key of ll_entry_i.

```
1: procedure GET_APTCURR
2:     if (curr s[1].key = key) then
3:         curr ← curr s[1] ;
4:     else if (curr s[0].key = key) then
5:         curr ← curr s[0] ;
6:     return curr ;
```

While executing the *toValidation()* the time-stamp field of the corresponding $node$ has to be updated. Such a node can be either the marked(dead or curr s[0]) or the unmarked(live curr s[1]). *get_aptcurr* is the utility method which returns the appropriate $node$ corresponding to the key.

Algorithm 15 release_ordered_locks($ordered\_ll\_list \downarrow$) : Release all locks taken during *lslSearch()*.

```
1: procedure RELEASE_ORDERED_LOCKS
2:     while (ll_entry_i ← next(ordered_ll_list)) do
3:         ll_entry_i.preds[0].unlock() ;
4:         ll_etry_i.preds[1].unlock() ;
5:         ll_entry_i.currs[0].unlock() ;
6:         ll_entry_i.currs[1].unlock() ;
```

release_ordered_locks is an utility method to release the locks in order.

D Proof Sketch of OSTMs

D.1 Operational Level

For a global state, S, we denote $evts(S)$ as all the events that has lead the system to global state S. We denote a state S' to be in future of S if $evts(S) \subset evts(S')$. In this case, we denote $S \sqsubset S'$. We have the following definitions and lemmas:

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

▶ **Definition 4.** *Abstract List (Abs):* At any global abstract state S, $S.Abs$ 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, $\langle S.Abs = S.Abs.RL \bigcup S.Abs.BL \rangle$, where,
$S.Abs.RL := \{\forall n | (n \in S.PublicNodes) \wedge (S.Head \rightarrow^*_{RL} S.n)\}$.
$S.Abs.BL = \{\forall n | (n \in S.PublicNodes) \wedge (\neg S.n.marked) \wedge (S.Head \rightarrow^*_{BL} S.n)\}$

▶ 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, $\langle \forall S, S' : (n \in S.nodes) \wedge (S \sqsubset S') \Rightarrow (n \in S'.nodes) \rangle$.

With Observation 5 , we assume that nodes once created do not get deleted (ignoring garbage collection for now).

▶ 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, $\langle \forall S, S' : (n \in S.nodes) \wedge (S \sqsubset S') \Rightarrow (n \in S'.nodes) \wedge (S.n.key = S'.n.key) \rangle$.

▶ 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 ($preds[0]$, $preds[1]$, $currs[0]$, $currs[1]$). Then in the state S, we have,

7.1 $(preds[0] \wedge preds[1] \wedge currs[0] \wedge currs[1]) \in S.PublicNodes$
7.2 $(S.preds[0].locked) \wedge (S.preds[1].locked) \wedge (S.currs[0].locked) \wedge (S.currs[1].locked)$
7.3 $(\neg S.preds[0].marked) \wedge (\neg S.currs[1].marked) \wedge (S.preds[0].\text{BL} = S.currs[1]) \wedge (S.preds[1].\text{RL} = S.currs[0])$

In Observation 7, *lslSearch()* method returns only if validation succeed at Line 19.

▶ **Lemma 8.** *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* ($preds[0]$, $preds[1]$, $curRs[0]$, $currs[1]$)*. Then in the state S, we have,*

8.1 $((S.preds[0].key) < key \leq (S.currs[1].key))$.
8.2 $((S.preds[1].key) < key \leq (S.currs[0].key))$.

Proof. 8.1 $(S.preds[0].key < key \leq S.currs[1].key)$ **:**
Line 4 of *lslSearch()* method of Algo 1 initializes $S.preds[0]$ to point head node. Also, ($S.currs[1]$ = $S.preds[0].BL$) by line 5. As in penultimate execution of line 6 ($S.currs[1].key < key$) and at line 7 ($S.preds[0] = S.currs[1]$) this implies,

$$(S.preds[0].key < key) \tag{1}$$

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

$$(key \leq S.currs[1].key) \tag{2}$$

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

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

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 *lslSearch()* method of Algo 1 returns.

8.2 $(S.preds[1].key < key \leq S.currs[0].key)$ **:**

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

$$(S.preds[1].key < key) \tag{4}$$

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

$$(key \leq S.currs[0].key) \tag{5}$$

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

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

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 *lslSearch()* method of Algo 1 returns.

◀

▶ **Lemma 9.** *For a node n in any global state S, we have that,*$\langle \forall n \in S.nodes : (S.n.key < S.n.RL.key) \rangle$.

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 9 . It can be seen by observing the code that *lslDel()* method of Algo 10 do not have any update events of RL.

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

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

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 Algo 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 $(S.node.RL = S.currs[0])$. 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 Algo 1 returns. From Lemma 8.2, we know that when *lslSearch()* method of Algo 1 returns then,

$$(S.preds[1].key) < key \leq (S.currs[0].key) \tag{7}$$

To reach line 10 of *lslIns()* method, line 32 of *STM_delete()* method of Algo 8 or line 24 of *STM_lookup()* method of Algo 3 should ensure that,

$$(S.currs[0].key \neq key) \xrightarrow{eq(7)} (S.preds[1].key) < key < (S.currs[0].key) \tag{8}$$

From Observation 7.3, we know that,

$$(S.preds[1].RL = S.currs[0]) \tag{9}$$

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

$$\begin{aligned}(S.preds[1].RL = node) &\xrightarrow{eq(8)} (S.preds[1].key < node.key)\\ &\implies (S.preds[1].key < S.preds[1].RL.key)\end{aligned} \tag{10}$$

Where $(S.node.key = key)$. Since $(preds[1], node) \in S.nodes$ 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 Algo 1 returns. Line 12 of the *lslIns()* method creates a new node, $node$ with key. Line 13 then sets $(S.node.RL = S.currs[0])$. 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.
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()* Algo 1 method returns. From Lemma 8.2, we know that when *lslSearch()* method of Algo 1 returns then,

$$(S.preds[1].key) < key \leq (S.currs[0].key) \tag{11}$$

To reach line 15 of *lslIns()* method, line 26 of *STM_tryC()* method of Algo 4 should ensure that,

$$(S.currs[0].key \neq key) \xRightarrow{eq(11)} (S.preds[1].key) < key < (S.currs[0].key) \quad (12)$$

From Observation 7.3, we know that,

$$(S.preds[1].RL = S.currs[0]) \quad (13)$$

Also, the atomic event at line 15 of *lslIns()* sets,

$$(S.preds[1].RL = node) \xRightarrow{eq(12)} (S.preds[1].key < node.key) \implies (S.preds[1].key < S.preds[1].RL.key) \quad (14)$$

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

◀

▶ **Lemma 10.** *In a global state S, any public node n is reachable from $Head$ via red links. Formally,* $\langle \forall S, n : n \in S.PublicNodes \implies S.Head \rightarrow^*_{RL} S.n \rangle$.

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 9 . It can be seen by observing the code that *lslDel()* 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 $(head, tail) \in S.PublicNodes \wedge \neg(S.head.marked) \wedge \neg(S.tail.marked) \wedge (S.head \rightarrow^*_{RL} S.tail)$.

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 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 $(S.node.RL = S.currs[0])$. 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 Algo 1 returns. From line 9 & 10 of *lslIns()* method, $(S.node.RL = S.currs[0]) \wedge (S.preds[1].RL = S.node) \wedge (node \in S.PublicNodes) \wedge (S.node.marked = true)$ (because inserting the node only into the redlink). It is to be noted that (from Observation 7.2), $(preds[0], preds[1], curr s[0], curr s[1])$ are locked, hence no other thread can change marked field of $S.preds[1]$ and $S.curr s[0]$ simultaneously. Also, from Observation 6, 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, $(S.Head \rightarrow^*_{RL} S.preds[1]) \wedge (S.preds[1] \rightarrow^*_{RL} S.node) \Rightarrow (S.Head \rightarrow^*_{RL} S.node)$.

3. ***Line 13 of* lslIns() *method:*** Line 12 of the *lslIns()* method creates a new node, $node$ with key. Line 13 then sets ($S.node.RL$ = $S.currs[0]$). 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.
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, $(S.node.RL = S.currs[0]) \wedge (S.preds[1].RL = S.node) \wedge (node \in S.PublicNodes) \wedge (node.marked = false)$ (because new node is created by default with unmarked field). It is to be noted that (from Observation 7.2), $(preds[0], preds[1], curs[0], curs[1])$ are locked, hence no other thread can change marked field of $S.preds[1]$ and $S.curs[0]$ simultaneously. Also, from Observation 6, a node's key field does not change after initialization. Before executing line 15, $preds[1]$ is reachable from head by RL (from induction hypothesis). After line 15, we know that from $preds[1]$, public unmarked node, $node$ is also reachable. Thus, we know that $node$ is also reachable from head. Formally, $(S.Head \rightarrow^*_{RL} S.preds[1]) \wedge (S.preds[1] \rightarrow^*_{RL} S.node) \Rightarrow (S.Head \rightarrow^*_{RL} S.node)$.

◀

▶ **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.key and strictly smaller than $n.RL$.key, then the node corresponding to the key does not belong to S.Abs. Formally, $\langle \forall S, n, key : S.PublicNodes \wedge (S.n.key < key < S.n.RL.key) \implies node(key) \notin S.Abs\rangle$.*

▶ 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, $\langle \forall S, S' : (n \in S.nodes) \wedge (S \sqsubset S') \wedge (S.head \rightarrow^*_{RL} S.n) \Rightarrow (n \in S'.nodes) \wedge (S'.head \rightarrow^*_{RL} S'.n)\rangle$.

Proof. From Observation 5, we have that for any node n, $n \in S.nodes \Rightarrow n \in S'.nodes$. 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,$\langle \forall n \in S.nodes : (S.n.key < S.n.BL.key)\rangle$.*

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, we know the underlying lazyskip-list has immutable $S.head$ and $S.tail$ nodes with $(S.head.BL = S.tail)$ and $(S.head.RL = S.tail)$. The relation between their keys is $(S.head.key < S.tail.key) \wedge (head, tail) \in S.nodes$.
Induction Hypothesis: Say, upto k events that change the BL field of any node, $(\forall n \in S.nodes : (S.n.key < S.n.BL.key))$.
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,

$$((S.preds[0].key) < key \leq (S.currs[1].key)) \wedge ((S.preds[1].key) < key \leq (S.currs[0].key)) \tag{15}$$

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

$$(S.currs[1].key \neq key) \wedge (S.currs[0].key = key) \xRightarrow{eq(15)} ((S.preds[0].key) < key < (S.currs[1].key)) \wedge((S.preds[1].key) < (key = S.currs[0].key)) \tag{16}$$

From Observation 7.3, we know that,

$$(S.preds[0].BL = S.currs[1]) \wedge (S.preds[1].RL = S.currs[0]) \tag{17}$$

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

$$(S.currs[0].BL = S.currs[1]) \xRightarrow[Lemma\ 9]{eq(16), Lemma\ 10} (S.currs[0].key) < (S.currs[1].key) \Longrightarrow (S.currs[0].key) < (S.currs[0].BL.key) \tag{18}$$

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

$$(S.preds[0].BL = S.curr[0]) \xRightarrow{eq(16)} (S.preds[0].key) < (S.curr[0].key) \Longrightarrow (S.preds[0].key) < (S.preds[0].BL.key). \tag{19}$$

Where $(S.curr[0].key = key)$. Since $(preds[0], curr[0]) \in S.nodes$ 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 $(S.node.BL = S.curr[1])$. Since this event doest 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,

$$(S.preds[0].key) < key \leq (S.curr[1].key) \wedge (S.preds[1].key) < key \leq (S.curr[0].key) \tag{20}$$

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

$$(S.curr[0].key \neq key) \wedge (S.curr[1].key \neq key) \xRightarrow{eq(20)} (S.preds[0].key) < key < (S.curr[1].key) \wedge(S.preds[1].key) < key < (S.curr[0].key) \tag{21}$$

From Observation 7.3, we know that,

$$(S.preds[0].BL = S.currs[1]) \tag{22}$$

Also, the atomic event at line 16 of *lslIns()* sets,

$$(S.preds[0].BL = S.node) \xrightarrow{eq(21)} (S.preds[0].key < S.node.key) \implies (S.preds[0].key < S.preds[0].BL.key) \tag{23}$$

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,

$$(S.preds[0].key) < key \leq (S.currs[1].key) \tag{24}$$

To reach line 3 of *lslDel()* method, line 31 of *STM_tryC()* method of Algo 4 should ensure that,

$$(S.currs[1].key = key) \xrightarrow{eq(24)} (S.preds[0].key) < (key = S.currs[1].key) \tag{25}$$

From Observation 7.3, we know that,

$$(S.preds[0].BL = S.currs[1]) \tag{26}$$

We know from Induction hypothesis,

$$(curs[1].key < curs[1].BL.key) \tag{27}$$

Also, the atomic event at line 3 of *lslDel()* sets,

$$(S.preds[0].BL = S.curs[1].BL) \xrightarrow{eq(25), eq(27)} (S.preds[0].key < S.curs[1].BL.key) \implies (S.preds[0].key < S.preds[0].BL.key) \tag{28}$$

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, $\langle \forall S, n : (S.PublicNodes) \wedge (\neg S.n.marked) \implies (S.Head \rightarrow^*_{BL} S.n)\rangle$.*

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 $(head, tail) \in S.PublicNodes \wedge \neg(S.head.marked) \wedge \neg(S.tail.marked) \wedge (S.head \rightarrow^*_{BL} S.tail)$.

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

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 (BL field changing event) can be executed only after the *lslSearch()* method of Algo 1 returns. It is to be noted that (from Observation 7.2), $(preds[0], preds[1], curr s[0], curr s[1])$ are locked, hence no other thread can change $S.preds[0].marked$ and $S.curr s[1].marked$ simultaneously. Also, from Observation 6, a node's key field does not change after initialization. Before executing line 4, from Observation 7.3 ,
$$(S.preds[0].marked = false) \wedge (S.curr s[1].marked = false) \tag{29}$$
And from Lemma 10 and induction hypothesis,
$$(S.Head \rightarrow^*_{RL} S.curr s[0]) \wedge (S.Head \rightarrow^*_{BL} S.curr s[1]) \tag{30}$$
After line 4, we know that from $curr s[0]$, public unmarked node, $curr s[1]$ is also reachable, implies that,
$$(S.curr s[0] \rightarrow^*_{BL} S.curr s[1]) \tag{31}$$
Also, before executing line 5, from induction hypothesis and Lemma 10 ,
$$(S.Head \rightarrow^*_{BL} S.preds[0]) \wedge (S.Head \rightarrow^*_{RL} S.curr s[0]) \tag{32}$$
After line 5, we know that from $preds[0]$, public unmarked node (from line 3 of *lslIns()* method), $curr s[0]$ is also reachable via BL, implies that,
$$(S.preds[0] \rightarrow^*_{BL} S.curr s[0]) \wedge (S.curr s[0].marked = false) \tag{33}$$
From eq(31) and eq(33),
$$(S.preds[0] \rightarrow^*_{BL} S.curr s[0]) \wedge (S.curr s[0] \rightarrow^*_{BL} S.curr s[1]) \wedge \\ (S.curr s[0].marked = false) \tag{34}$$
Since $(preds[0], curr s[0]) \in S.PublicNode$ and hence, $(S.Head \rightarrow^*_{BL} S.preds[0]) \wedge (S.preds[0] \rightarrow^*_{BL} S.curr s[0]) \wedge (S.curr s[0].marked = false) \Rightarrow (S.Head \rightarrow^*_{BL} S.curr s[0])$.
2. ***Line 14 of* lslIns() *method:*** Line 12 of the *lslIns()* method creates a new node, $node$ with key. Line 14 then sets ($S.node.BL$ = $S.curr s[1]$). Since this event doest 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. It is to be noted that (from Observation 7.2), $(preds[0], preds[1], curr s[0], curr s[1])$ are locked, hence no other thread can change $S.preds[0].marked$ and $S.curr s[1].marked$ simultaneously. Also, from Observation 6, a node's key field does not change after initialization. Before executing line 14, from Observation 7.3 ,
$$(S.preds[0].marked = false) \wedge (S.curr s[1].marked = false) \tag{35}$$
And from induction hypothesis,
$$(S.Head \rightarrow^*_{BL} S.curr s[1]) \tag{36}$$
After line 14, we know that from $node$, public unmarked node, $curr s[1]$ is also reachable via BL, implies that,
$$(S.node \rightarrow^*_{BL} S.curr s[1]) \tag{37}$$

Also, before executing line 16, from induction hypothesis,

$$(S.Head \rightarrow^*_{BL} S.preds[0]) \tag{38}$$

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

$$(S.preds[0] \rightarrow^*_{BL} S.node) \wedge (S.node.marked = false) \tag{39}$$

From eq(37) and eq(39),

$$(S.preds[0] \rightarrow^*_{BL} S.node) \wedge (S.node \rightarrow^*_{BL} S.curs[1]) \wedge (S.node.marked = false) \tag{40}$$

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

◀

▶ **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, $\langle \forall S, n : (n \in S.nodes) \wedge (S.head \rightarrow^*_{BL} S.n) \subseteq (S.head \rightarrow^*_{RL} S.n) \rangle$.*

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.

◀

▶ **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 max_ts of that node should be equal to method transaction time-stamp. Formally, $\langle (node(key) \in ([E^H.Post(m_i.LP)].Abs.RL)) \wedge (node.max\_ts = TS(T_i)) \rangle$.*

Proof. 1. ***For rv_method method:*** By observing the code, each rv_method first invokes *lslSearch()* method of Algo 1 (line 12, line 20 of *STM_lookup()* method of Algo 3 & *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 *lslSearch()* is invoked from a method, it returns correct location $(preds[0], preds[1], currs[0], currs[1])$ 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).
In the pre-state of LP event of rv_method , if $(node.key \in S.Abs.RL)$, means key is already there in RL and time-stamp of that node is less then the rv_method transactions time-stamp, from *toValidation()* method of Algo 12 , then in the post-state of LP event of rv_method, $node.key$ should be the part of RL from Observation 13 and key can't be change from Observation 6 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 rv_method , if $(node.key \notin S.Abs.RL)$, means key is not there in RL then, in the post-state of LP event of rv_method, insert the $node$ corresponding to the key into RL by using *lslIns()* method of Algo 9 and update the max_ts field for corresponding node key by method transaction time-stamp. Since, $node.key$ should be the part of RL from Observation 13 and key can't be change from Observation 6 , in post-state of LP event of rv_method.

2. ***For upd_method method:*** 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 $(preds[0], preds[1], curs[0], curs[1])$ 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).
   a. ***If upd_method is insert:*** In the pre-state of LP event of upd_method, if $(node.key \in S.Abs.RL)$, 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 $(node.key \notin S.Abs.RL)$, 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 9 at line 30 of *STM_tryC()* 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 $(node.key \in S.Abs.RL)$, 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_tryC()* 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, $(node.key \notin S.Abs.RL)$ 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_tryC()* method of Algo 4 . Thus, $(node.key \in S.Abs.RL)$ 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 $(preds[0], preds[1], curs[0], curs[1])$, in this case the previous upd_method effect might be overwritten if the next upd_method preds and currs 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_tryC()*, 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$(preds[0], preds[1], curs[0], curs[1])$ then effect of first upd_method will be overwritten by the next upd_method.

By observing the code at line 15 of *STM_tryC()* method of Algo 4, current upd_method will go for *lostUpdateValidation()* and at line 3 of *lostUpdateValidation()* method of Algo 5 , current upd_method will validate its $(preds[0].marked)$ and $(preds[0].BL! = curs[1])$. If any condition is true then, at line 4 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 curs 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, E^H, 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,*

20.1 If upd_method is insert, then node corresponding to the key should be part of BL and node.val should be equal to v. Formally, $\langle(node(key) \in ([E^H.Post(m_i.LP)].Abs.BL) \wedge (node.val = v)\rangle$.

20.2 If upd_method is delete, then node corresponding to the key should not be part of BL. Formally, $\langle(node(key) \notin ([E^H.Post(m_i.LP)].Abs.BL)\rangle$.

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 ($preds[0], preds[1], currs[0], currs[1]$) 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).

20.1 ***If upd_method is insert:*** In the pre-state of LP event of upd_method at Line 17, 22 of *STM_tryC()* method of Algo 4, if $(node.key \in S.Abs.RL)$, 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 BL and it will update the $value$ as v.
In the pre-state of LP event of upd_method at Line 26 of *STM_tryC()* method of Algo 4 , if $(node.key \notin S.Abs.RL)$, 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 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 modified from Observation 6.

20.2 ***If 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 $(node.key \in S.Abs.RL)$, 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 not be the part of BL, from *lslDel()* method of Algo 10 at line 31 of *STM_tryC()* method of Algo 4 .
In the pre-state of LP event of upd_method, $(node.key \notin S.Abs.RL)$ 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_tryC()* method of Algo 4 .

◀

▶ **Lemma 21.** *Consider a concurrent history, E^H, where S be the pre-state of LP event of successful rvm method, in that, if node corresponding to the key is the part of BL and node.val is equal to v then, rv_method return OK and value v. Formally,* $\langle(node(key) \in ([E^H.Pre(m_i.LP)].Abs.BL)) \wedge (S.node.val = v) \implies \text{rvm}(key, OK, v)\rangle$.

Proof. Let the rv_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 $(preds[0], preds[1], curds[0], curds[1] \in S.PublicNodes)$ and are locked(from Observation 7.1 & Observation 7.2) until *STM_lookup()* method of Algo 3 return. Also, from Lemma 8.1 ,

$$(S.preds[0].key < key \leq S.currs[1].key) \tag{41}$$

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_method. And after observing code, at line 16 of *STM_lookup()* method of Algo 3,

$$(S.currs[1].key = key) \xRightarrow{eq(41)} (S.preds[0].key < (key = S.currs[1].key)) \tag{42}$$

Also, from Observation 7.3 ,

$$(S.preds[0].BL = S.currs[1]) \tag{43}$$

And $(currs[1] \in S.nodes)$, we know $(currs[1] \in S.Abs.BL)$ 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 $(node(key) \in ([E^H.Pre(m_i.LP)].Abs.BL) \wedge (S.node.val = v))$.

`*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_method, in that, if node corresponding to the key is not the part of BL then, rv_method return $FAIL$. Formally,* $\langle(node(key) \notin ([E^H.Pre(m_i.LP)].Abs.BL)) \implies \text{rvm}(key, FAIL)\rangle$.

Proof. Let the rv_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 $(preds[0], preds[1], currs[0], currs[1] \in S.PublicNodes)$ and are locked(from Observation 7.1 & Observation 7.2) until *STM_lookup()* method of Algo 3 return. Also, from Lemma 8.2 ,

   $$(S.preds[1].key < key \leq S.currs[0].key) \tag{44}$$

   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_method. And after observing code, at line 20 of *STM_lookup()* method of Algo 3 ,

   $$(S.currs[0].key = key) \xRightarrow{eq(44)} (S.preds[1].key < (key = S.currs[0].key)) \tag{45}$$

   Also, from Observation 7.3 ,

   $$(S.preds[1].RL = S.currs[0]) \tag{46}$$

And $(curr s[0] \in S.nodes)$, we know $(curr s[0] \in S.Abs.RL)$ where S is the pre-state of the LP event of the method and $(S.curr s[0].marked = true)$. Thus, $(curr s[0] \notin S.Abs.BL)$ from Definition 4 . Hence $(node(key) \notin ([E^H.Pre(m_i.LP)].Abs.BL)$

2. From line 12 of *STM_lookup()* method of Algo 3, when *lslSearch()* method of Algo 1 returns we have $(preds[0], preds[1], curr s[0], curr s[1] \in S.PublicNodes)$ and are locked(from Observation 7.1 & Observation 7.2) until *STM_lookup()* method of Algo 3 return. Also, from Lemma 8.2 ,

$$(S.preds[1].key < key \leq S.curr s[0].key) \tag{47}$$

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

$$(S.curr s[1].key \neq key) \wedge (S.curr s[0].key \neq key) \xRightarrow{eq(47)} (S.preds[1].key < key < S.curr s[0].key) \tag{48}$$

Also, from Observation 7.3 ,

$$(S.preds[1].RL = S.curr s[0]) \tag{49}$$

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

`*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, $\langle (ul(m_i(k)) \prec l(m_j(k))) \Rightarrow (LP(m_i) \prec LP(m_j)) \rangle$

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. $S_1 \sqsubset S_2$ and $S_1.BL.value(k) \neq S_2.BL.value(k)$ then there exist S' s.t. $S' \sqsubset S_2$ and S' contain the* STM_tryC() *method on the same key k. Formally, $\langle (S_1.BL.value(k) \neq (S_2.BL.value(k)) \Rightarrow \exists (S' s.t., S_1.BL \prec S'.LP(\text{tryC}) \prec S_2.BL) \rangle$. 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,*

29.1 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.*

29.2 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.*

Proof29.1 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.

29.2 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 20.1 . 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, $\langle (node(key) \in (P.Abs.RL)) \wedge (P.node.max\_ts = TS(T_i)) \rangle$. 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 (line 12, line 20 of *STM_lookup()* method of Algo 3 & *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 *lslSearch()* is invoked from a method, it returns correct location $(preds[0], preds[1], curs[0], curs[1])$ 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).

In the pre-state of rv_method , if $(node.key \in S.Abs.RL)$, means key is already there in RL and time-stamp of that node is less then the rv_method transactions time-stamp, from *toValidation()* method of Algo 12 , then in the post-state of rv_method, $node.key$ should be the part of RL from Observation 13 and key can't be change from Observation 6 and it just update the max_ts field for corresponding node key by method transaction time-stamp else abort.

In the pre-state of rv_method , if $(node.key \notin S.Abs.RL)$, means key is not there in RL then, in the post-state of rv_method, insert the $node$ corresponding to the key into RL by using *lslIns()* method of Algo 9 and update the max_ts field for corresponding node key by method transaction time-stamp. Since, $node.key$ should be the part of RL from Observation 13 and key can't be change from Observation 6 , in post-state of rv_method.

2. ***For upd_method method:*** 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 $(preds[0], preds[1], curs[0], curs[1])$ 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).

   a. ***If upd_method is insert:*** In the pre-state of upd_method, if $(node.key \in S.Abs.RL)$, 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 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 upd_method, if $(node.key \notin S.Abs.RL)$, means key is not there in RL then in the post-state of upd_method, it will insert the $node$ corresponding to the key into the RL as well as BL, from *lslIns()* method of Algo 9 at line 29 of *STM_tryC()* 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 upd_method, if $(node.key \in S.Abs.RL)$, 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 upd_method, $node.key$ should be the part of RL, from *lslDel()* method of Algo 10 at line 34 of *STM_tryC()* 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 upd_method, $(node.key \notin S.Abs.RL)$ 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_tryC()* method of Algo 4 . Thus, $(node.key \in S.Abs.RL)$ and update the max_ts field for corresponding node key by method transaction time-stamp else abort.

◀

▶ **Corollary 31.** *After the post-state of any successful method on a key ensures that underlying RL contains a unique node corresponding to the key and max_ts field is updated by methods transactions* *time-stamp.*

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,

$$((T_1.lastEvt \prec_H T_k.firstEvt) \wedge (T_k.lastEvt \prec_H T_1.firstEvt)) \Rightarrow (T_1.lastEvt \prec_H T_1.firstEvt) \tag{50}$$

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 $m_i \prec_H^{MR} m_j$. And it is represented as $\prec_H^{CO}$.

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

38.1 *H_1 is equivalent to $H_2 \Rightarrow (methods(H1) = methods(H2))$.*
38.2 *$\prec_{H1}^{CO} \subseteq \prec_{H2}^{CO}$. 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) $\in \prec_{H1}^{CO} \Rightarrow$ ((T_i, T_j) $\in \prec_{H2}^{CO}$).

Lets assume that their exists a conflict (T_i, T_j) in $\prec_{H1}^{CO}$ but not in $\prec_{H2}^{CO}$. But, from Observation 35 & Observation 36 we know that $(T_i, T_j) \in \prec_{H2}^{CO}$. Thus, $\prec_{H1}^{CO} \subseteq \prec_{H2}^{CO}$.

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 $\prec_{H_1}^{CO} \subseteq \prec_{H_2}^{CO}$. Then, H_1 is legal $\implies H_2$ is legal.*

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

$$up_p(ht, k, v_p) \prec_{H_1}^{MR} rv_j(ht, k, v) \tag{51}$$

Let us assume H_2 is not legal. Since, H_1 is equivalent to H_2 from Lemma 38.1 such that ($rv_j(ht,k,v)$ $\in$ methods(H_2)). Since H_2 is not legal, there exist a ($up_r(ht,k,v_r) \in$ methods(H_2)) such that ($up_r(ht,k,v_r) = H_2.lastUpdt(rv_j(ht,k,v))$). So conflict($H_2$) has,

$$up_r(ht,k,v_r) \prec^{MR}_{H_2} rv_j(ht,k,v) \tag{52}$$

We know, ($\prec^{CO}_{H_1} \subseteq \prec^{CO}_{H_2}$) so,

$$up_p(ht,k,v_p) \prec^{MR}_{H_2} rv_j(ht,k,v) \tag{53}$$

From Lemma 38.1 ($up_r(ht,k,v_r) \in$ methods(H_1)). Since H_1 is legal $up_r(ht,k,v_r)$ can occur only in one of following *conflicts*,

$$up_r(ht,k,v_r) \prec^{MR}_{H_1} up_p(ht,k,v_p) \tag{54}$$

or

$$rv_j(ht,k,v) \prec^{MR}_{H_1} up_r(ht,k,v_r) \tag{55}$$

In H_1 eq(55) is not possible, because if (eq(55) $\in$ conflict(H_1)) implies (eq(55) $\in$ conflict(H_2)) from ($\prec^{CO}_{H_1} \subseteq \prec^{CO}_{H_2}$) 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,

$$up_r(ht,k,v_r) \prec^{MR}_{H_2} up_p(ht,k,v_p) \tag{56}$$

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

$$up_r(ht,k,v_r) \prec^{MR}_{H_2} up_p(ht,k,v_p) \prec^{MR}_{H_2} rv_j(ht,k,v)$$

This contradicts that H_2 is not legal. Thus if H_1 is legal $\longrightarrow H_2$ 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, $\langle \forall S, S' : (n \in S.Abs) \wedge (S \sqsubset S') \Rightarrow (n \in S'.Abs) \wedge (S.n.max\_ts \leq S'.n.max\_ts)\rangle$.*

Proof. We prove by Induction on events that change the max_ts field of a node associated with a key, which are Line 26, 30 & 35 of *STM_delete()* method of Algo 8, Line 18, 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 lazyskip-list has immutable $S.head$ and $S.tail$ nodes with ($S.head.BL = S.tail$) and ($S.head.RL = S.tail$).

Lets 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, upto 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 *lslSearch()* method of Algo 1 for finding the node corresponding to the key. Inside the *lslSearch()* method of Algo 1 , 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 *lslSearch()* method of Algo 1 for finding the node corresponding to the key. Inside the *lslSearch()* method of Algo 1 , 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 21, 25, 29, 34 & 37 of* STM_tryC() *method of Algo 4 :*** By observing the code, line 7 of *STM_tryC()* method of Algo 4 first invokes *lslSearch()* method of Algo 1 for finding the node corresponding to the key. Inside the *lslSearch()* method of Algo 1 , 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.

◀

▶ **Corollary 43.** *Every successful methods update the max_ts field of a node associated with a key always in increasing TS values.*

▶ **Lemma 44.** *If* STM_begin(T_i) *occurs before* STM_begin(T_j) *then* $TS(T_i)$ *preceds* $TS(T_j)$. *Formally,* $\langle \forall T \in H : (\text{STM\_begin}(T_i) \prec \text{STM\_begin}(T_j)) \Leftrightarrow (TS(T_i) < TS(T_j)) \rangle$.

Proof. $(Only\ if)$ If $(\textit{STM\_begin}(T_i) \prec \textit{STM\_begin}(T_j))$ then $(TS(T_i) < TS(T_j))$. Lets assume $(TS(T_j) < TS(T_i))$. From Observation 40 ,

$$\textit{STM\_begin}(T_j) \prec_H \textit{STM\_begin}(T_i) \tag{57}$$

but we know that,

$$\textit{STM\_begin}(T_j) \succ_H \textit{STM\_begin}(T_i) \tag{58}$$

Which is a contradiction thus, $(TS(T_i) < TS(T_j))$.

(if) If $(TS(T_i) < TS(T_j))$ then $(\mathit{STM\_begin}(T_i) \prec \mathit{STM\_begin}(T_j))$. Let us assume $(\mathit{STM\_begin}(T_j) \prec \mathit{STM\_begin}(T_i))$. From Observation 40 ,

$$TS(T_j) < TS(T_i) \tag{59}$$

but we know that,

$$TS(T_j) > TS(T_i) \tag{60}$$

Again, a contradiction. ◀

▶ **Lemma 45.** *If $(T_i, T_j) \in$ conflict(H) $\Rightarrow$ TS(T_i) $<$ TS(T_j).*

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

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

$$T_i.lastEvt \prec_H T_j.firstEvt \tag{61}$$

And from program order of T_i,

$$T_i.firstEvt \prec_H T_i.lastEvt \Rightarrow \mathit{STM\_begin}(T_i) \prec_H T_i.lastEvt \tag{62}$$

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

$$T_i.firstEvt \prec_H T_j.firstEvt \Rightarrow \mathit{STM\_begin}(T_i) \prec_H \mathit{STM\_begin}(T_j) \xrightarrow{Lemma\ 44} TS(T_i) < TS(T_j) \tag{63}$$

2. ***If (T_i, T_j) is a conflict edge:*** We prove this case by contradiction, lets assume $(T_i, T_j) \in$ conflict(H) & TS(T_j) $<$ TS(T_i). Given that $(T_i, T_j) \in$ conflict(H) and from Definition 37 we get, $m_i \prec_H^{MR} m_j$.
m_i can be $rv\_methods$ 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 of T_i, $node$ corresponding to the key should be there in RL (from Corollary 31 & Definition 4) 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 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 operation is working has a unique location in underlying data-structure from Corollary 11 . So, when the *lslSearch()* is invoked from a method m_j of T_j, it returns correct location $(preds[0], preds[1], currs[0], currs[1])$ 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 6 & 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, If $(T_i, T_j) \in$ 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,

$$TS(T_1) < TS(T_2) < \cdots < TS(T_{n-1}) \tag{64}$$

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,

$$TS(T_{n-1}) < TS(T_n) \xrightarrow{eq(64)} (TS(T_1) < TS(T_2) < \cdots < TS(T_n)) \tag{65}$$

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 ,

$$TS(T_1) < TS(T_2) \cdots < TS(T_n) < TS(T_1) \Rightarrow TS(T_1) < TS(T_1) \tag{66}$$

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 $\prec_H^{RT}$ and $\prec_H^{CO}$ (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 $\prec_H^{RT} = \prec_{\bar{H}}^{RT}$ and $\prec_H^{CO} = \prec_{\bar{H}}^{CO}$.

Since every two operations related by the conflict relation in S are also related by $\prec_{\bar{H}}^{CO}$, we obtain $\prec_{\bar{H}}^{CO} \subseteq \prec_S^{CO}$. 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.

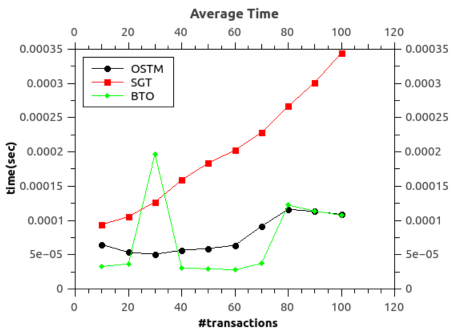


Figure 26 Average time taken by RWSTMs v/s OSTM

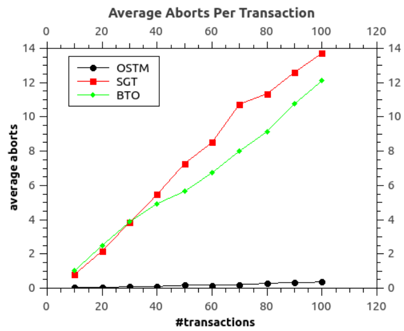


Figure 27 Average aborts per transaction by RWSTMs v/s OSTM