Efficient means of Achieving Composability using

Transactional Memory

Ajay Singh

A Thesis Submitted to
Indian Institute of Technology Hyderabad
In Partial Fulfillment of the Requirements for
The Degree of Master of Technology

Department of Computer Science and Engineering

December 2017
Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

[Signature]

AJAY SINGH
(Ajay Singh)

CS15MTECH01001
(Roll No.)
Approval Sheet

This Thesis entitled Efficient means of Achieving Composability using Transactional Memory by Ajay Singh is approved for the degree of Master of Technology from IIT Hyderabad

(M. V. Panduranga Rao) Examiner
Dept. of Computer Science and Engineering
IITH

(Manish Singh) Examiner
Dept. of Computer Science and Engineering
IITH

(Sathya Peri) Adviser
Dept. of Computer Science and Engineering
IITH

Chairman
Dept. of Computer Science and Engineering
IITH
 Acknowledgements

First and foremost, all praise is due to Goddess Saraswati, the one who is source of all knowledge.

I would like to thank my advisor, Prof. Sathya Peri, for his endless help and support. In addition to his valuable technical help throughout my M.Tech. journey, he always provided me with sincere advice and encouragement, and never lost trust in me. I would also like to express my gratefulness to my committee members: Dr. M. V. Panduranga Rao, Dr. Subrahmanyam Kalyanasundaram and Dr. Manish Singh for the insightful comments and discussions. I would like to express my special thanks to Dr. Sparsh Mittal for providing useful reviews in writing my thesis.

I am so grateful to my colleagues in the Theory and Systems Research Group, who formed the best environment for an Mtech student to succeed. Working with such a team greatly benefited my academic career and personal life. I will never forget the technical and spiritual support provided by Archit Somani. I was really blessed by joining the lab at IIT Hyderabad at the same time Archit did.

I would like to thank my dear parents for being the main reason I am what I am. They worked hard in the past to raise me and prepare the way for my success, and then they suffered, but never complained, when they missed me for three years. May I be always the righteous son they wish. I am also grateful to everyone in my family, especially my sister, Anju and Pooja for their continuous love and support.

Last but not least, I am blessed with being at IIT Hyderabad and among its amazing community for about three years. I am grateful to everyone whom I met here and everyone who gave me any sort of help, support, and advice.
Dedication

To Dr. APJ Abdul Kalam, my parents and teachers.
Abstract

The major focus of software transaction memory systems (STMs) has been to facilitate the multiprocessor programming and provide parallel programmers with an abstraction for fast development of the concurrent and parallel applications. Thus, STMs allow the parallel programmers to focus on the logic of parallel programs rather than worrying about synchronization.

Heart of such applications is the underlying concurrent data-structure. The design of the underlying concurrent data-structure is the deciding factor whether the software application would be efficient, scalable and composable. However, achieving composition in concurrent data structures such that they are efficient as well as easy to program poses many consistency and design challenges.

We say a concurrent data structure compose when multiple operations from same or different object instances of the concurrent data structure can be glued together such that the new operation also behaves atomically. For example, assume we have a linked-list as the concurrent data structure with lookup, insert and delete as the atomic operations. Now, we want to implement the new move operation, which would delete a node from one position of the list and would insert into the another or same list. Such a move operation may not be atomic(transactional) as it may result in an execution where another process may access the inconsistent state of the linked-list where the node is deleted but not yet inserted into the list. Thus, this inability of composition in the concurrent data structures may hinder their practical use.

In this context, the property of compositionality provided by the transactions in STMs can be handy. STMs provide easy to program and compose transactional interface which can be used to develop concurrent data structures thus the parallel software applications. However, whether this can be achieved efficiently is a question we would try to answer in this thesis.

Most of the STMs proposed in the literature are based on read/write primitive operations(or methods) on memory buffers and hence denoted RWSTMs. These lower level read/write primitive operations do not provide any other useful information except that a write operation always needs to be ordered with any other read or write. Thus limiting the number of possible concurrent executions. In this thesis, we consider Object-based STMs or OSTMs which operate on higher level objects rather than read/write operations on memory locations. The main advantage of considering OSTMs is that with the greater semantic information provided by the methods of the object, the conflicts among the transactions can be reduced and as a result, the number of aborts will also be less. This allows for larger number of permissive concurrent executions leading to more concurrency. Hence, OSTMs could be an efficient means of achieving composable operation in the software applications using the concurrent data structures. This would allow parallel programmers to leverage underlying multi-core architecture.

To design the OSTM, we have adopted the transactional tree model developed for databases. We extend the traditional notion of conflicts and legality to higher level operations in STMs which allows efficient composability. Using these notions we define the standard STM correctness notion of Conflict-Opacity. The OSTM model can be easily extended to implement concurrent lists, sets, queues or other concurrent data structures.

We use the proposed theoretical OSTM model to design HT-OSTM - an OSTM with underlying hash table object. We noticed that major concurrency hot-spot is the chaining data structure within the hash table. So, we have used Lazyskip-list approach which is time efficient compared to normal lists in terms of traversal overhead. At the transactional level, we use timestamp ordering protocol to ensure that the executions are conflict-opaque. We provide a detailed handcrafted proof of correctness starting from operational level to the transactional level. At the operational level we show that HT-OSTM generates legal sequential history. At
transactional level we show that every such sequential history would be opaque thus co-opaque.

The HT-OSTM exports STM\_insert, STM\_lookup and STM\_delete methods to the programmer along-with STM\_begin and STM\_trycommit. Using these higher level operations user may easily and efficiently program any parallel software application involving concurrent hash table. To demonstrate the efficiency of composition we build a test application which executes the number of hash-tab methods (generated with a given probability) atomically in a transaction. Finally, we evaluate HT-OSTM against ESTM based hash table of synchrobench and the hash-table designed for RWSTM based on basic time stamp ordering protocol. We observe that HT-OSTM outperforms ESTM by the average magnitude of $10^6$ transactions per second (throughput) for both lookup intensive and update intensive work load. HT-OSTM outperforms RWSTM by 3% & 3.4% update intensive and lookup intensive workload respectively.
Contents

Declaration .......................................................... ii
Approval Sheet ...................................................... iii
Acknowledgements ................................................... iv
Abstract ............................................................. vi

Nomenclature ........................................................ ix

1 Introduction 1
  1.1 Introduction to STM ............................................. 1
  1.2 Introduction to Concurrent Data Structure ..................... 4
  1.3 Motivation ...................................................... 4

2 Literature 7

3 Methodology 10
  3.1 Building System Model for OSTM ............................ 10
     3.1.1 Preliminary definitions & notations ..................... 10
     3.1.2 Legal Histories ........................................... 13
     3.1.3 Conflict Notion ............................................ 16
  3.2 HT-OSTM Design ............................................... 18
     3.2.1 HT-OSTM data-structure design ......................... 19
     3.2.2 HT-OSTM execution cycle ............................... 22
  3.3 HT-OSTM Pseudocode ......................................... 25
  3.4 Optimizations ................................................. 35

4 Proof Of Correctness 36
  4.1 Proof Sketch of OSTMs ....................................... 38
     4.1.1 Operational Level ........................................ 38
     4.1.2 Transactional Level ....................................... 55

5 Results 60
  5.1 Test application .............................................. 61
  5.2 Lookup intensive workload .................................... 62
     5.2.1 Experiment 1: 80% lookup ................................ 62
     5.2.2 Experiment 2: 50% lookup ................................ 63
  5.3 Update intensive workload .................................... 65
Chapter 1

Introduction

1.1 Introduction to STM

Growing ubiquity of multicores processors and onset of Moore’s law saturation and powerwall era has made parallel and concurrent programming inevitable and programmer must write parallel and concurrent programs to leverage underlying multi/many core architecture. Thus, focus on programming for multicore programing is need of the hour.

In words of Seymon Peyton Jones[1], “The free lunch is over. We have grown used to the idea that our programs will go faster when we buy a next-generation processor, but that time has passed. While the next generation chip will have more CPUs, each individual CPU will be no faster than the previous years model. If we want our programs to run faster, we must learn to write parallel programs.”

So, to exploit the parallel architecture, applications need to be parallelly programmed. Unfortunately, parallel programming is far more difficult to design, maintain and debug than sequential programming. Formulating algorithms and proving their correctness is even more difficult. The bugs are non-deterministic and parallel programs often give poor performance. Adding to the woes, reasoning about parallel programs does not come naturally to human mind. For instance, implementing a sequential queue data structure is very easy but implementing a queue that allows concurrent operation on both its ends is still an active area in research. Therefore, parallel programming, which untill now is the domain of a few high-performance computing experts, will now have to be mastered by common programmers. Multithreading is essential for full exploitation of the multi-core hardware and effective use of multiple processor systems. However, they do pose synchronization challenges, some of them being:

- Collaboration between threads which involves sharing of data in memory or on secondary storage.
- Uncontrolled writes can lead to inconsistent data values or race condition.
- Synchronized memory access is required since processors cannot modify shared memory locations atomically.
-Granularity of access to shared memory, which is a deciding factor for efficiency of the concurrent systems.
For instance, consider the classic banking example where two threads (transactions), $T_1$ and $T_2$ are trying to withdraw an amount from the account 'from' where 'balance' is shared objects. Now, if $T_1$ and $T_2$ are not synchronized, then balance $T_1$ may overwrite the withdraw by $T_2$. Thus, even though withdraw was done twice from the account 'from' but it might appear that withdraw was done only once. Let's take initial value of 'balance' = 100 and amount = 20. $T_1$ reads the balance into bal and later $T_2$ also reads the balance into bal. Now, assume $T_2$ is context switched. $T_1$ goes ahead and updates the balance to 80. Now, $T_2$ wakes up and since its local value bal is 100, it also updates the balance to 80. Please note that the final value of balance should have been 60 but it is 80. Hence, the system is inconsist.

\begin{verbatim}
  T1
  void withdraw ( int amount )
  {   
      bal = read (balance)  
      balance = write (bal - amount);  
  }

  T2
  void withdraw ( int amount )
  {   
      bal = read (balance)  
      balance = write (bal - amount);  
  }
\end{verbatim}

In response to these synchronization issues most popular technology used by the industry is to use locks for every read and write access, or to use semaphores or monitors to update the shared code sections or shared resources within a program. This ensures atomic update of different variables (shared resources) and avoids inconsistency. For example, the below code snippet represents a solution to the above race condition.

\begin{verbatim}
  T1
  void withdraw ( int amount )
  {   
      lock (balance);  
      bal = read (balance)  
      balance = write (bal - amount);  
      unlock (balance);  
  }

  T2
  void withdraw ( int amount )
  {   
      lock (balance);  
      bal = read (balance)  
      balance = write (bal - amount);  
      unlock (balance);  
  }
\end{verbatim}

However, using a single program wide lock (coarse grained locks) decreases system performance and have scalability issues. Hence, fine grained locking is required, but this too turns out to have engineering challenges, as locks have to be used in proper ordering for whole application. One missing lock acquisition may lead to race conditions which may cause program crashes and memory corruption. Besides this other synchronization issues like priority inversions, livelocks, convoying, starvation and deadlocks add to programmers nightmare of writing concurrent programs.

Moreover, lock based programs are not modular[1, 2], scalable and they are difficult to debug and maintain. Hence, all these issues amount to parallel programming being difficult and less popular amongst programmers. The following bank transaction example demonstrates incorrect locking scenario: here $T_1$ may see the incorrect state of the from and to accounts because $T_2$ has debited money equivalent to amount but has not credited to account to. Thus, money equal to amount may appear missing to $T_1$.

\[\text{Please note that the code snippets are by no means the complete or correct programs. They are used here to show the problems with locking and motivate the advantage of the STM.}\]
Thus, we see that with lock based solution programmers would mostly be focusing on synchronisation issues rather than designing the logic for their applications. **Software Transactional Memory**[3] is one promising abstraction programming paradigm to efficiently and easily write the parallel programs such that programmers do not need to explicitly worry about the synchronization. STM exports its transactional interface i.e. methods like `tx_begin`, `tx_read`, `tx_write` and `tx_commit`. A programmer has to write its section of code that needs synchronisation using these constructs. And, STM takes over all the task of correctly and efficiently synchronising the application. Thus, making writing parallel programs easier. Lets, try writing the previous `withdraw` function of our banking example using STM.

Thus, we see that STM makes writing parallel programs easier by shifting synchronisation to itself, either in form of a library or compiler constructs, depending upon the way STM is implemented. TL2[4], SwissTM
1.2 Introduction to Concurrent Data Structure

Concurrent data structures are heart of the multithreaded software applications which enable extraction of maximum parallelism from the underlying multi core architecture. But, designing and and proving correctness of such concurrent data structures or applications based on them is non-trivial and it poses many design and consistency challenges.

One of them being composibility of operation of concurrent data structures. Often, individual operations of the concurrent data structures execute atomically. But practical use of such data structure very often requires these individually correct operation to glue together and appear to be happening atomically.

For instance, consider a concurrent hash-table object which exports insert, delete & lookup methods, these operation work correctly in multithreaded environment and appear to behave transactionally individually. But, real world application needs these operations to compose together for example move, which requires delete & insert to occur together in transactional manner. Please note that implementation of move requires that a delete and then insert from same or different hash-table object appear to happen together.

This inability of composition of operation in concurrent data structures hinders software reusability and as it can be used only in limited number of ways, thus raising question on their practical use[7].

Lock based solutions are very popular in industry, but they have their own problems as discussed in Section 1.1. STM again here proves to be a promising alternative to design composable and easily programmable concurrent data structures hence concurrent software applications[1].

1.3 Motivation

Software Transaction Memory Systems (STMs) are a convenient programming interface for a programmer to access shared memory without worrying about concurrency issues [3, 8]. 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. Moreover, the transactions provide atomicity implying operations executed within the transactions either take effect together or do not take effect at all. This prevents other transactions from observing the intermediate effects of other transactions. Thus, STMs are natural choice for achieving composability[9].

Most of the STMs [4, 5, 6] 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 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 mere read/writes.

It was shown in databases that object-level systems provide greater concurrency than read/write systems [10, Chap 6]. Harris et al.[11] and Herlihy et al.[12, 13] worked on the concept of Object-based STM. We
would like to propose an alternative model to achieve composability with greater concurrency for STMs by considering higher-level objects which leverage the richer semantics of object level operations. We motivate this with an interesting example.

Consider an OSTM operating on the hash-table object. Thus, we can call it HT-OSTM. Such an HT-OSTM exports the following methods: \( t_{\text{begin}} \) which begins a transaction (same as in RWSTMs), \( t_{\text{insert}} \) which inserts a value for a given key, \( t_{\text{delete}} \) which deletes the value associated with the given key, \( t_{\text{lookup}} \) which looks up the value associated with the given key and \( \text{tryC} \) 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 [14, Chap 9]. It can be seen that the underlying list is a concurrent data-structure manipulated by multiple transactions (and hence threads). So we have used the lazy-list based concurrent set [15] to implement the operations of the list denoted as: \( \text{list\_insert}, \text{list\_del} \) and \( \text{list\_lookup} \) (referred as contains in [15]). Thus, when a transaction invokes \( t_{\text{insert}}, t_{\text{delete}} \) and \( t_{\text{lookup}} \) methods, the STM internally invokes the \( \text{list\_insert}, \text{list\_del} \) and \( \text{list\_lookup} \) 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 1.1(i) and transactions \( T_1 \) and \( T_2 \) are concurrently executing \( t_{\text{lookup}}(k_5), t_{\text{delete}}(k_7) \) and \( t_{\text{lookup}}(k_8) \) as shown in Figure 1.1(ii). In our representation, we abbreviate \( t_{\text{delete}} \) as \( d \) and \( t_{\text{lookup}} \) as \( l \). For simplicity, we refer to nodes of the list by their keys. In this setting, suppose a transaction \( T_1 \) of HT-OSTM invokes methods \( t_{\text{lookup}} \) on the keys \( k_5, k_8 \). This would internally cause the HT-OSTM to invoke \( \text{list\_lookup} \) 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 \( t_{\text{delete}} \) on key \( k_7 \) between the two \( t_{\text{lookups}} \) of \( T_1 \). This would cause, HT-OSTM to invoke \( \text{list\_del} \) method of list on \( k_7 \). Since, we are using lazy-list approach on the underlying list, \( \text{list\_del} \) involves pointing the next field of element \( k_5 \) to \( k_8 \) and marking element \( k_7 \) as deleted. Thus \( \text{list\_del} \) 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 HT-OSTM denoted as a history can be represented as a transactional forest as shown in Figure 1.1(ii). Here the execution of each transaction is a tree.

In this execution, we denote the read/write operations (leaves) as layer-0 and \( t_{\text{lookup}}, t_{\text{delete}} \) methods as layer-1. Consider the history (execution) at layer-0 (while ignoring higher-level operations), denoted as \( H_0 \). It can be verified this history is not opaque[16]. 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 \( H_0 \) 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)[16]. On the other hand consider the history \( H_1 \) at layer-1 consisting of \( t_{\text{lookup}}, t_{\text{delete}} \) 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 [10, 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[16] 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[11]. This history clearly shows the advantage of considering STMs with higher level operations in this case they are $t_{\text{insert}}, t_{\text{delete}}$ and $t_{\text{lookup}}$. With object level modeling of histories, we get a higher number of acceptable schedules than read/write model. This is because of not all conflicts at the lower level matter at the higher level.

The atomic property of transactions helps to correctly glue together the individual operations and the concurrency in such STMs can be enhanced by considering the object level semantics of the underlying data structure. Thus, considering higher level semantics provides efficient means of achieving composability of operations of a concurrent data structure. Our OSTM model to design concurrent data structures ensures that the sequence of operations compose efficiently. The OSTM can be moulded to any specific data structure (in this work we show it for concurrent hash-table and we name it HT-OSTM). OSTM models includes detailed discussion of legality of transactions executing over single or multiple shared objects (or data structures) We also discuss conflict notion for the operations involved by characterizing them into rv_method and upd_method followed by the correctness proofs of the histories generated by OSTM. Following is the summary of our contribution:

- We build OSTM: an alternative theoretical model for efficiently transactifying the concurrent data structures using their semantic information such that they are composable [9] too. We implement OSTM with a concurrent hash table object named as HT-OSTM. The OSTM can also be implemented with other data structures like list, stack, queue or tree. It would be very natural to see that HT-OSTM can easily be adapted to implement a list-OSTM with list as an underlying data structure.

- We propose legality definitions and the notion of conflicts for object histories generated by HT-OSTM. This we achieve by formally categorizing HT-OSTM methods as rv_method and upd_method.

- HT-OSTM is designed with hash-table where chaining is implemented via lazyskip-list. We provide full implementation of the methods exported by the HT-OSTM such that every method composes correctly within HT-OSTM transactions.

- We provide in-depth proof of correctness starting from layer-0 (operational level) to the layer-1 (transactional level) executions generated by the proposed HT-OSTM. And first time we show that HT-OSTM is guaranteed to be co-opaque[17].

- We evaluate HT-OSTM against the concurrent hash-table of Synchrobench with ESTM as synchronization mechanism. We also evaluate HT-OSTM against hash-table developed using read/write STM with BTO as synchronization mechanism[18].
Chapter 2

Literature

Earliest work of using the semantics of concurrent data structures or using STMs for object level granularity include that of open nested transactions [12] and transaction boosting of Herlihy et al.[13]. Abstract nested transactions[11] is another STM that is motivated by the need to avoid aborts of transactions due to conflicts at a lower level (Harris refers to them as benign conflicts). Harris et al.[11] identify the transactions which are victims of benign conflicts and prevent 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.[20] 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[21].

Transactional boosting idea of Herlihy et. al[13] tries to utilize the object level semantics of linearizable datastructures. They assume datastructure to be blackbox and try to transactify the base object(underlying datastructure); We inturn, treat the physical layer(in terminology of open nested transactions) or layer-0 as well; This provides us the oppurtunity to customize the optimization of an underlying datastructure. Herlihy claims to differ from open nested transactions by providing a precise methodology and characterization of the mechanism. However, they maintain a log of each operation’s inverse, which needs to execute once a transaction aborts; this incurs additional computational and memory cost. Moreover, many datastructures donot provide reverse operations (for example, priority queue). The proposed OSTM do not need reverse operation as we follow deffered update augmented with optimism of time-order based validation.

Moreover, transactional boosting use abstract locks at semantic layer (abstract-layer) which is a pessimistic approach and distracts from the more general correctness criteria for TMs i.e opacity. Herlihy et. al. give a model to support the mechanism of transactional boosting based on serializabilty(strict or commit order serializability) of generated schedules as correctness critera. They briefly cover the sequential specification of underlying objects, while we give a more detailed sequential specification that can be adapted to most of the data structure having generic update and lookup operations(eg list has insert and delete as update operations; and lookup ). Herlihy’s model also has rollbacks which is obvious, given their pessimistic strategy. Our model is more optimistic in that sense and underlying data structure is updated only after there is a guarantee that there is no inconsistency due to concurrency. Thus, we donot need to do rollbacks. This also solves the problem of irrevocable operations being executed during a transaction which might abort later otherwise.

Our work is adaptation of Weikum and Vossens transactional tree model in databases. Herlihy’s Boosting
strategy or Hasan’s optimistic boosting sticks to 2-flat object histories, while our model is open to higher levels of abstractions as we directly adapt the transactional tree model with co-opacity [17] as correctness criterion a subclass of Opacity. Their main focus as underlying datastructure is a linked list. We use a hash table as underlying object which utilizes a lazyskip-list, which turns out to be more efficient in terms of space and time.

Hassan[22] uses C-SWC model to prove that OTB transactions compose. We on other hand propose alternate object model STMs where we lay down a detailed legality definition for the underlying data structures to be transactified and build a bottom up correctness proof starting from operational level to the transactional level showing that HT-OSTM ensures co-opacity[17] thus compose. OTB uses the 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 HT-OSTM is co-opaque. We also assume that there can be multiple operations on the 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 3.3. Moreover, unlike OTB, STM_lookup() is validated only once at the instant of their execution and unlike original boosting HT-OSTM do not need to rollback thus saving considerable logging overhead.

Spiegelman et al[19] believe Boosting is based on a semantic variant of two phase locking, in which the data structure operations are protected by a set of abstract locks. They transactify the CDSL and aim to build a TDSL. However, their major focus is mainly on transactifying concurrent datastructures we differ in focussing more on utilising the datastructure semantics by differentiating between abstract level and physical level access; Providing a methodology of transactional trees in context of STM with generic semantic specification of underlying objects and co-opacity as correctness criterion. TSDL work again is much of a mechanism rather than a methodology.

Open nested transactions[12] tries to exploit concurrency by differentiating between memory layer conflicts as physical layer and logical conflicts as abstract layer. They achieve so by using abstract locking at abstract-layer and claims the generated histories to be serializable. They too in their approach are pessimistic and rely on fallback mechanism i.e. once a transactions aborts they execute compensating operations that incurs significant memory and computational cost. We use time ordered optimistic mechanism to address synchrony at abstract level. Open nesting seems to be more of a mechanism while we give a detailed methodology and our model is well supported by hand crafted correctness proofs and generic specifications of the underlying objects. Our work is in C++.

Several researchers have established that STM makes the development of concurrent composable applications easier than its lock based counterparts[8, 9], not to be forgotten scalability issues in lock based solutions. Tim Harris et. al.[9] proposed an STM based solution to achieve composability and at the same time maintain the abstraction, such that internal details of the atomic methods are not required for the programmer to glue multiple operations together in concurrent Haskell. Zhang et al [23] 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 Tx_FastRead & Tx_Flush 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 HT-OSTM. Cederman & Tsigas[24] propose a methodology to implement the composable operation in lock free concurrent object. Their approach is restricted in application
to the objects which meet the criterion, named as move candidates and requires mechanical changes in the candidate data structure by the programmer to implement the composable operations.

Fraser et. al.[2] proposed OSTM which is based on shadow copy mechanism, which involves a level of indirection to access the shared objects through \textit{OSTMOpenForReading} and \textit{OSTMOpenForWriting}. These read/write methods are exported to the programmer. On the other the OSTM model proposed by us exports the higher object level methods like \textit{STM\_lookup()}, \textit{STM\_insert()} and \textit{STM\_delete()} while hiding the internal read and write lower level primitives. So, it seems that using the Fraser OSTM one can write the higher level methods transactionally. For example one may implement a \textit{lookup} on the underlying list object using transactions. We differ here because we allow such multiple higher level operations to be grouped together atomically without requiring user to implement them. The exported methods in Fraser et.al’s OSTM may allow \textit{OSTMOpenForReading} to see the inconsistent state of the shared objects but our OSTM model precludes this possibility by validating the access during execution of rv\_method(i.e. the methods which donot modify the underlying objects and only return some value by performing a search on them.)

Fraser’s OSTM uses the transaction descriptors which stores the previous and new copies of the shared objects increasing the memory requirement to maintain the meta data. We on the other hand, maintain single copy of the underlying shared object and the meta information is augmented within each shared object. For example in case of a list each node is a shared object. Here we augment each shared node with the meta data (in our case the time-stamp of access by the other transactions) along with an unique key and the value pair (value may store any complex data type of any type). Thus, we can say our motivation and implementation is different from Fraser OSTM[2] only the name happens to coincide.
Chapter 3

Methodology

3.1 Building System Model for OSTM

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. Please note that we have designed the model taking hash-table as underlying object and implemented the proposed techniques for efficiently composing the hash-table object, thus we call it HT-OSTM henceforth. The HT-OSTM model can easily be extended to any general underlying object, say linked-list, lazylist, queue etc and thus we may refer to the proposed model as OSTM while referring to general underlying objects.

3.1.1 Preliminary definitions & notations

Methods: The \( n \) processes access a collection of transaction objects via atomic transactions supported by the HT-OSTM. Each transaction has a unique identifier typically denoted as \( T_i \). Within a transaction, a process can invoke transactional methods on a hash-table transaction object. A hash-table(\( ht \)) consists of multiple key-value pairs of the form \( \langle k, v \rangle \). The keys and values are respectively from set of integers and any data type respectively. The methods that a transaction \( T_i \) can invoke are: (1) \( t_{\text{insert}}(ht, k, v) \): 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) \( t_{\text{delete}}(ht, k, v) \): 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) \( t_{\text{lookup}}(ht, k, v) \): 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 \( t_{\text{lookup}} \) is similar to \( t_{\text{delete}} \).

For simplicity, we assume that all the values inserted by transactions through \( t_{\text{insert}} \) method are unique. We denote \( t_{\text{insert}} \) and \( t_{\text{delete}} \) as \( \text{upd methods} \) since both these change the underlying data-structure. We denote \( t_{\text{delete}} \) and \( t_{\text{lookup}} \) as \( \text{return-value methods or rv methods} \) 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 \( A \) which implies that the transaction \( T_i \) is aborted. A method \( m_i \) returns \( A \) if \( m_i \) along with all the methods of \( T_i \) executed so far are not consistent (w.r.t opacity, the correctness-criterion which is formally defined later in this section).
The HT-OSTM supports two other methods: (4) tryC: this method tries to validate all the operations of the T_i. HT-OSTM returns ok if T_i is successfully committed. Otherwise, HT-OSTM returns A implying abort. This method is invoked by a process after completing all its transactional operations. (5) tryA: this method returns A and HT-OSTM aborts T_i.

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

Events: Having described about methods of a transaction, we describe about the events invoked by these methods. We assume that each method consists of an 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/A)). (2) t_delete_i(ht, k, v): inv(t_delete_i(ht, k)) and rsp(t_delete_i(ht, k, v/nil/A)). (3) t_lookup_i(h, k, v): inv(t_lookup_i(h, k)) and rsp(t_lookup_i(h, k/nil/A)). (4) tryC_i: inv(tryC_i()) and rsp(tryC_i(ok/A)). (5) tryA_i: inv(tryA_i()) and rsp(tryA_i(A)). We assume that the threads execute atomic events. Similar to Lev-Ari et. al.[25, 26], 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, (3) lock/unlock events on the shared-memory objects.

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, represented as: r_i(x, v) implying that T_i reads value v for shared object x; w_i(x, v) implying that T_i writes value v onto the shared object 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 (evts(m), <_m). 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 3.1 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)) and the method d_12(k_2) is represented as: inv(d_12(h, k_2)) r_121(k_2, o_2) w_122(k_2, o_2) rsp(d_12(h, k_2, o_2)).

Please note that wlog, for convenience we shorten t_delete_i(ht, k, v) to d_i_j(k), t_insert_i(ht, k, v) to i_i_j(k) and t_lookup_i(ht, k, v) to l_i_j(k) respectively. Here, subscript i, j implies that it is the jth method of the ith transaction. Also, depending on the context we may omit the parameters. From our assumption, we get that for any read/write primitive rw of m, inv(m) <_m rw <_m rsp(m).

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. The events read/write on shared/local memory objects change the global state. The inv & rsp events on higher level shared-memory objects do not change the contents of the global state. Although we would denote the resulting state with a new label while establishing the correctness of HT-OSTM.

Transactions: Following the notations used in database multi-level transactions [10], we model a transaction as a two-level tree. Figure 3.1 shows a tree execution of a transaction T_i. 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 3.1, level-1 consists of t_lookup and t_delete methods operating on the lazy skip-list as also shown in Figure 1.1(i). Thus a transaction is a tree whose nodes are methods and leaves are events.
Having informally explained a transaction, we formally define a transaction $T$ as the tuple $(evts(T), <_T)$. 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 3.1 is: $\text{inv}(l_{11}(ht, k_5)) r_{111}(k_2, o_2) r_{112}(k_5, o_5) \text{rsp}(l_{11}(ht, k_5, o_5)) \text{inv}(d_{112}(ht, k_2)) r_{121}(k_2, o_2) w_{122}(k_2, o_2) \text{rsp}(d_{12}(ht, k_2, o_2))$. 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 $\text{methods}(T_i)$. We assume that for any method $m$ in $\text{methods}(T_i)$, $evts(m)$ is a subset of $evts(T_i)$ and $<_m$ is a subset of $<_T$. Formally, $(\forall m \in \text{methods}(T_i) : evts(m) \subseteq evts(T_i) \land m \subseteq \text{methods}(T_i))$.

We assume that if a transaction has invoked a method, then it does not invoke a new method until it gets the response of the previous one. Thus all the methods of a transaction can be ordered by $<_T$. Formally, $(\forall m_p, m_q \in \text{methods}(T_i) : (m_p <_T m_q) \lor (m_q <_T m_p))$.

Figures 3.1 and 3.2 illustrate these concepts. Figure 3.1: T1 : A sample transaction on lazyskip-list (of Figure 1.1(i)) representing a hash-table object.

**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 $(evts(H), <_H)$ where all the events are totally ordered by $<_H$. The set of methods that are in $H$ is denoted by $\text{methods}(H)$. A method $m$ is incomplete if $\text{inv}(m)$ 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 $H$ are denoted as $\text{txns}(H)$. The set of committed (resp., aborted) transactions in $H$ is denoted by $\text{committed}(H)$ (resp., $\text{aborted}(H)$). The set of incomplete or live transactions in $H$ is denoted by $\text{incomp}(H) = \text{live}(H) = \text{txns}(H) - \text{committed}(H) - \text{aborted}(H)$. On the other hand, the set of terminated transactions are those which have either committed or aborted and is denoted by $\text{term}(H) = \text{committed}(H) \cup \text{aborted}(H)$.
The relation between the events of transactions & histories is analogous to the relation between methods & transactions. We assume that for any transaction \( T \) in \( \text{txns}(H) \), \( \text{evts}(T) \) is a subset of \( \text{evts}(H) \) and \( <_T \) is a subset of \( <_H \). Formally, \( \langle \forall T \in \text{txns}(H) : (\text{evts}(T) \subseteq \text{evts}(H)) \land (<_T \subseteq <_H) \rangle \). We denote two histories \( H_1, H_2 \) as equivalent if their events are the same, i.e., \( \text{evts}(H_1) = \text{evts}(H_2) \). 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 \( r \) response or after \( \text{tryC}(ok) \) method. We only consider well-formed histories for HT-OSTM.

**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 \( e_{pq} \) belonging to some other method \( m_{pq} \) (of transaction \( T_p \)) either \( e_{pq} \) occurs before \( \text{inv}(m_{ij}) \) or after \( \text{rsp}(m_{ij}) \). Formally, \( \langle \forall m_{ij} \in \text{methods}(H) : m_{ij} \text{ is isolated} \equiv (\forall m_{pq} \in \text{methods}(H), \forall e_{pq} \in m_{pq} : e_{pq} <_H \text{inv}(m_{ij}) \lor \text{rsp}(m_{ij}) <_H e_{pq}) \rangle \). For instance in \( H1 \) shown in Figure 1.1(ii), \( d_2(k_2) \) is isolated. In fact all the methods of \( H1 \) are isolated. Consider history \( H2 \) shown in Figure 3.3a. 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 [17, 27]) or linearized [28] 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 \( \text{tryA}_k(A) \) immediately after the last method of every transaction \( T_k \in \text{incomp}(H) \). Since all the methods in a sequential history are complete, this definition only has to take care of completing transactions.

Consider a sequential history \( H \). Let \( m_{ij}(ht,k,v/\text{nil}) \) be the first method of \( T_i \) in \( H \) operating on the key \( k \). Since all the methods of a transaction are sequential and ordered, we can clearly identify the first method of \( T_i \) on key \( k \). Then, we denote \( m_{ij}(ht,k,v) \) as \( H.\text{firstMth}(\langle ht,k,T_i \rangle) \). 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 \( m_{ij}(ht,k,v) = H.\text{prevKeyMth}(m_{ix},T_i) \).

**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 \( \text{methods}(H) \), we denote \( m_{ij} <_H^M m_{pq} \) if \( \text{rsp}(m_{ij}) <_H \text{inv}(m_{pq}) \). 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 \( \text{term}(H) \), we denote \( T_i <_H^{TR} T_p \) if \( (T_i, \text{lastEvt} <_H T_p, \text{firstEvt}) \). Here TR stands for transactional real-time order.

Thus, \( <_H \) partially orders all the methods and transactions in \( H \). It can be seen that if \( H \) is sequential, then \( <_H^M \) totally orders all the methods in \( H \). Formally, \( \langle (H \text{ is sequential}) \implies (\forall m_{ij}, m_{pq} \in \text{methods}(H) : (m_{ij} <_H^M m_{pq}) \lor (m_{pq} <_H^M m_{ij})) \rangle \).

We define a history \( H \) as serial [29] or \( t \)-sequential [27] if all the transactions in \( H \) have terminated and can be totally ordered w.r.t \( <_H^{TR} \), i.e. all the transactions execute one after the other without any interleaving. Intuitively, a history \( H \) is serial if all its transactions can be isolated. Formally, \( \langle (H \text{ is serial}) \implies (\forall T_i \in \text{txns}(H) : (T_i \in \text{term}(H)) \land (\forall T_i, T_p \in \text{txns}(H) : (T_i <_H^{TR} T_p) \lor (T_p <_H^{TR} T_i)) \rangle \). Since all the methods within a transaction are ordered, a serial history is also sequential. Figure 3.3b shows a serial history.

### 3.1.2 Legal Histories

We define legality of \( \text{rv}_m \text{methods (} \text{r}_\text{delete} \text{ & } \text{r}_\text{lookup} \) on sequential histories. Consider a sequential history \( H \) having a \( \text{rv}_m \text{method} \text{rv}_m_{ij}(ht,k,v) \) (with \( v \neq \text{nil} \)) belonging to transaction \( T_i \). We define this \( \text{rv}_m \text{method to} \ldots \)
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, \( rvm_{ij} \neq H.firstKeyMth(\langle ht, k \rangle, T_i) \land (m_{ix}(ht, k, v') = H.prevKeyMth(\langle ht, k \rangle, T_i)) \) (where \( v' \) could be nil). Then,
   
   (a) if \( m_{ix}(ht, k, v') \) is a \( t\_\text{insert} \) method i.e. \( t\_\text{insert}_ix(ht, k, v') \) then \( v = v' \).
   
   (b) if \( m_{ix}(ht, k, v') \) is a \( t\_\text{lookup} \) method i.e. \( t\_\text{lookup}_ix(ht, k, v') \) then \( v = v' \).
   
   (c) if \( m_{ix}(ht, k, v') \) is a \( t\_\text{delete} \) method i.e. \( t\_\text{delete}_ix(ht, k, v'/nil) \) then \( v = \text{nil} \).

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

2. If \( rvm_{ij} \) is the first method of \( T_i \) to operate on \( \langle ht, k \rangle \) and \( v \) is not nil. Formally, \( rvm_{ij}(ht, k, v) = H.firstKeyMth(\langle ht, k \rangle, T_i) \land (v \neq \text{nil}) \). Then,

   (a) There is a \( t\_\text{insert} \) method \( t\_\text{insert}_{pq}(ht, k, v) \) in \( \text{methods}(H) \) such that \( T_p \) committed before \( rvm_{ij} \). Formally, \( \langle \exists t\_\text{insert}_{pq}(ht, k, v) \in \text{methods}(H) : \text{try}_C_p \prec_H^{mr} rvm_{ij} \rangle \).

   (b) There is no other update method \( up_{xy} \) of a transaction \( T_x \) operating on \( \langle ht, k \rangle \) in \( \text{methods}(H) \) such that \( T_x \) committed after \( T_p \) but before \( rvm_{ij} \). Formally, \( \langle \nexists up_{xy}(ht, k, v') \in \text{methods}(H) : \text{try}_C_p \prec_H^{mr} \text{try}_C_x \prec_H^{mr} rvm_{ij} \rangle \).

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

3. If \( rvm_{ij} \) is the first method of \( T_i \) to operate on \( \langle ht, k \rangle \) and \( v \) is nil. Formally, \( rvm_{ij}(ht, k, v) = H.firstKeyMth(\langle ht, k \rangle, T_i) \land (v = \text{nil}) \). Then,
(a) There is $t_{\text{delete}}$ method $t_{\text{delete}}_{pq}(ht, k, v')$ in $\text{methods}(H)$ such that $T_p$ (which could be $T_0$ as well) committed before $rvm_{ij}$. Formally, $(\exists t_{\text{delete}}_{pq}(ht, k, v') \in \text{methods}(H) : \text{try}_C \prec^M_{H} \text{rvm}_{ij})$. 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 $\text{methods}(H)$ such that $T_x$ committed after $T_p$ but before $rvm_{ij}$. Formally, $(\exists up_{xy}(ht, k, v'') \in \text{methods}(H) : \text{try}_C \prec^M_{H} \text{rvm}_{ij})$.

In this case similar to step 2, we denote $\text{try}_C$ as the last update method of $rvm_{ij}$, i.e., $\text{try}_C(\langle ht, k, v \rangle) = H\.lastUpdt(\text{rvm}_{ij}(\langle ht, k, v \rangle))$.

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_method 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 we also call conflict inheritance as the conflict of the later method of $T_i$ operating on $\langle ht, k \rangle$ can be found using the conflicts of the first method of $T_i$. 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 $t_{\text{delete}}$ method on all the keys used of all hash-table objects, there exists at least one $t_{\text{delete}}$ method for every rv_method on $k$ of $ht$. We formally prove legality in Lemma 28 in Section 4.1 and then we finally show that HT-OSTM histories are co-opaque[17] as defined in Definition 2.

Coming to $t_{\text{insert}}$ methods, since a $t_{\text{insert}}$ method always returns ok as they overwrite the node if already present therefore they always take effect on the $ht$. Thus, we denote all $t_{\text{insert}}$ 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 $rvm$ ($t_{\text{lookup}}$ and $t_{\text{delete}}$) methods since all $t_{\text{insert}}$ methods are by default legal.

**Intuitive examples for Legality** If $rvm$ 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 3.4(i), previous method for $lu_{ij}(\langle ht, k_5, v_5 \rangle)$ of transaction $T_i$ on same key $k_5$ is $ins_{is}(\langle ht, k_5, v_5 \rangle)$. So, $lu_{ij}(\langle ht, k_5, v_5 \rangle)$ will return the same value which will be inserted by previous method $ins_{is}(\langle ht, k_5, v_5 \rangle)$. Same mechanism will follow in Figure 3.4(ii) and Figure 3.4(iii).

![Figure 3.4: STM_lookup() is not the first method of its transaction](image)

If $rvm$ 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 3.5, previous closest method for $lu_{ij}(\langle ht, k, v_p \rangle)$ of transaction $T_i$ on same key $k$ is $ins_{pq}(\langle ht, k, v_p \rangle)$ of transaction $T_p$. So, $lu_{ij}(\langle ht, k, v_p \rangle)$ will return the same value which has been inserted by $ins_{pq}(\langle ht, k, v_p \rangle)$ and there can’t be any other transaction $upd$ method working on the same key between $T_p$ and $T_i$. Figure 3.6 represents, previous closest method of committed transaction $T_p$ is $del_{pq}(\langle ht, k, v_p \rangle)$ on key $k$ so $lu_{ij}(\langle ht, k, \text{Nil} \rangle)$ of transaction $T_i$ returns nil for same key $k$. 


**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 [16].

**Definition 1.** A sequential history \( H \) is opaque if there exists a serial history \( S \) such that: (1) \( S \) is equivalent to \( H \), i.e., \( \text{evts}(H) = \text{evts}(S) \) (2) \( S \) is legal and (3) \( S \) respects the transactional real-time order of \( H \), i.e., \( T_R^H \subseteq T_R^S \).

In this definition, we are restricting only to sequential histories. It can be seen that this definition of opacity is very similar to the definition given in [17] with methods on read-write objects. But the definition of legality is very different which takes care of the object model case.

### 3.1.3 Conflict Notion

In order to show that any concurrent history of \( HT-OSTM \) is linearizable we need to know which methods can be ordered and in what order. Thus, establishing the conflict relation between all the methods of an concurrent object (in this case \( \text{hash-table} \)) is important. As we discussed in Figure 1.1(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[17]) of higher level history. We use this conflict notion to show...
that HT-OSTM histories are co-opaque. We say two transactions $T_i, T_j$ of a sequential history $H$ are in conflict if at least 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., $(ht, k) \in updtSet(T_i) \cap (ht, k) \in updtSet(T_j))$, 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 $rvm_{ij}$ on the key same $k$ of hash-table $ht$ which is the first method on $(ht, k)$. Thus, $(ht, k) \in updtSet(T_i) \cap (rvm_{ij}((ht, k, v) \in rvSet(T_j)) \cap (rvm_{ij}(ht, k, v) = H.firstKeyMth((ht, k), T_j))))$, where $rvSet(T_j)$ is return value set of $T_j$. (3) $T_i$’s $tryC$ completed before $T_j$’s $rv$, i.e., $rvm_{ij} \prec_H^{MR} rvm_{ij}$.

- **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 $(ht, k)$. $T_j$ updates the key $k$ of the hash-table, $ht$. Thus, $(rvm_{ix}((ht, k, v) \in rvSet(T_i)) \cap (rvm_{ix}((ht, k, v) = H.firstKeyMth((ht, k), T_i)) \cap ((ht, k) \in updtSet(T_j)))) (3) T_i$’s $rvm$ completed before $T_j$’s $tryC$, i.e., $rvm_{ix} \prec_H^{MR} tryC_j$.

**Definition 2.** Co-opacity : A sequential history $H$ is conflict-opaque (or co-opaque) if there exists a serial history $S$ such that: (1) $S$ is equivalent to $H$, i.e., $evtS(H) \equiv evtS(S)$ (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. $\prec_H^{CO} \subseteq \prec_S^{CO}$) [17].

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(or rt edge):** $\{(T_i, T_j) : T_i \prec_H^{TR} T_j\}$

Consider the history $H_5 : l_1(h, k_1, NULL)l_2(h, k_2, NULL)l_3(h, k_1, v_1)i_4(h, k_4, v_1)c_1i_3(h, k_3, v_3)c_2d_2(h, k_4, v_1)c_3l_4(h, k_4, NULL)i_4(h, k_2, v_4)c_4$ shown in Figure 3.8.

![Figure 3.8: Graph Characterization of history $H_5$](image-url)

The legality and conflict notion established here are used to prove that histories generated by the HT-OSTM are correct or co-opaque[17] in Section 4.
3.2  **HT-OSTM Design**

We design the OSTM using hash-table where chaining is done using lazyskip-list, therefore we name it **HT-OSTM**. 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 its 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 3.9. 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 an inherent advantage of being search efficient. To illustrate this, consider the example in Figure 3.9 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 3.10) searching $k_8$ would have involved unnecessarily traversal over dead nodes represented by $k_1$, $k_3$ and $k_6$.

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 3.9, 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 3.11).

**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\[^\text{16}\] of the **HT-OSTM** execution( note storing the dead nodes is not specific to **HT-OSTM**, such a mechanism can always be used by **RWSTMs**). We further explain this using example in Figure 3.12 and Figure 3.13.
Figure 3.12: History H is not opaque

Figure 3.13: Opaque History H1

History H shown in Figure 3.12 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 $HT-OSTM$ scheduler knows that a conflicting operation $del_2(ht, k_1, v_0)$ has already been scheduled violating the time-order[10]. 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 3.13. 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 $HT-OSTM$.

3.2.1 $HT-OSTM$ data-structure design

In proposed $HT-OSTM$, 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.

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.

```java
class txlog {
    private :
        int t_id ;     // a log entry is uniquely identified using key and obj_id
        STATUS tx_status ;
        vector <key , le> ll_list ;

    public :
        txlog () ;    // txlog () ;     // createLLentry () ;
        setPredsnCurns() ; setOpnName () ; setOpStatus () ; setValue () ;
        setKey () ; setbucketId () ; getOpn () ; getOpStatus () ;
        getValue () ; getKey () ; getbucketId () ;
    } ;
}
```

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.

```java
class le {
    public :
```

19
enum OPERATION_NAME = {INSERT, DELETE, LOOKUP}

enum STATUS = {ABORT = 0, OK, FAIL, COMMIT}

enum VALIDATION_TYPE = {RV, TRYC}

enum LIST_TYPE = {RL, BL, RL_BL}

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 bl) respectively. Here, \langle k_m, k_q \rangle are predecessor (preds[0]) and current (currs[1]) node for bl and \langle k_n, k_p \rangle are predecessor (preds[1]) and current (currs[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. Table 3.1 describes the utility methods.

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setOpn()</td>
<td>store method name into ll_list of the txlog</td>
</tr>
<tr>
<td>setValue()</td>
<td>store value of the key into ll_list of the txlog</td>
</tr>
<tr>
<td>setOpStatus()</td>
<td>store status of method into ll_list of the txlog</td>
</tr>
<tr>
<td>setPreds&amp;Currs()</td>
<td>store location of preds and currs according to the key into ll_list of the txlog</td>
</tr>
<tr>
<td>getOpn()</td>
<td>give operation name from ll_list of the txlog</td>
</tr>
<tr>
<td>getValue()</td>
<td>give value of the key from ll_list of the txlog</td>
</tr>
<tr>
<td>getOpStatus()</td>
<td>give status of the method from ll_list of the txlog</td>
</tr>
<tr>
<td>getKey&amp;Objid()</td>
<td>give key and obj_id corresponding to the method from ll_list of the txlog</td>
</tr>
<tr>
<td>getAptCurr()</td>
<td>give the red or blue curr node from the log corresponding to the key of the txlog</td>
</tr>
<tr>
<td>getPreds&amp;Currs()</td>
<td>give location of preds and currs according to the node corresponding to the key from ll_list of the txlog</td>
</tr>
</tbody>
</table>

Table 3.1: Utility methods for each transaction to manipulate its log.

**Shared memory DS:**

*HT-OSTM* shared memory is the chained hash-table where each node (referred as LinkedHashNode in code) of the chain (lazyskip-list) is a key-value pairs of the form \langle k, v \rangle. Most of the notations used here are
A node $n$ when created is initialized as follows: (1) key and val is the key and value 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.

class LinkedHashNode
{
    public:
    int key, value;
    bool marked;

    /* stores the time stamp of last transaction that performed lookup, insert or delete respectively */
    struct max_ts { lookup; insert; delete; };

    pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER; /* lock */
    std::recursive_mutex lock;

    LinkedHashNode *red_next; /* next red node */
    LinkedHashNode *blue_next; /* next blue node */

    /* init the node with key, value */
    LinkedHashNode(int key, int value);
};

We adapt timestamp validation[10] to ensure schedules generated by proposed HT-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 tlookup$(ht, k)$, tinsert$(ht, k)$ and tdelete$(ht, k)$ respectively. $max_{ts}$, node and ll_entry form the part of the meta information for the HT-OSTM.

class HashMap
{
    private:
    /* hash table where each bucket is a lazyskip-list chain */
    LinkedHashNode **htable;

    public:
    HashMap();
    ~HashMap();
    /* Hash Function */
    int HashFunc(int key);

    /* Insert Element at a key */
    void lslIns(int key, int value, LinkedHashNode** preds,
                LinkedHashNode** currs, LIST_TYPE lst_type);
}
The hash-table object is of type HashMap which has buckets implemented as a lazyskip-list. Each bucket of the hash-table can be operated by *lsl_ins, lsldel* and *lsl_Seach* internal utility methods.

### 3.2.2 HT-OSTM execution cycle

![Figure 3.14: Transaction lifecycle of HT-OSTM](image)

Through out its life an HT-OSTM transaction may execute *STMBegin(), STMInsert(), STMlookup(), STMDelete() and STMtryC()* methods which are also exported to the user. A user can implement his/her applications using HT-OSTM which would provide efficient composability. 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. And, *STMInsert()* merely log its 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. This phase is executed by *STMtryC()* method. Figure 3.14 depicts the transaction life cycle.

**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. The variable prefixed with *sh_* are shared memory variables and can be accessed by multiple transactions concurrently, for instance *sh_preds[]*. *sh_preds[0]* & *sh_currs[1]* depict the blue nodes accessible by blue links and *sh_preds[1]* & *sh_currs[0]* depict the red nodes accessed by red links respectively. Also in pseudocode we call methods of Table 3.1 with *le*, this is simple to aid readers to understand that the method is called to manipulate the corresponding log entry in local log.

**rv_method execution phase:** Initially, in *rv_method execution phase* each transaction invokes *STMBegin()* of Algo 1 for getting unique transaction id and *local log*. Then transaction may encounter the upd_method or
rv_method. \textit{STM\_insert}() of Algo 5, first looks for the node corresponding to the \textit{key} into the ll\_list (Line 107). If \textit{key} is not found then it will create the ll\_entry and store the value, operation name and status (Line 109 to Line 114) into it which would be validated and realized in shared memory in \textit{STM\_tryC}().

\textit{STM\_tryC}() and rv\_method of HT-OSTM uses \textit{IslSearch()} to find the location at the lazyskip-list (thus the name) in lock free manner. Line 189 to Line 197 and Line 200 to Line 206 of Algo 7 find the location at lazyskip-list for \textit{bl} and \textit{rl} respectively. This is motivated by the search in lazylist \cite[section 9.7]{section}. The \textit{preds} and \textit{currs} thus are subjected to \textit{methodValidation()} of Algo 11 and \textit{transValidation()} of Algo 12 after acquiring locks on the \textit{preds} and \textit{currs} (Line 209 of Algo 7). If the validation succeeds \textit{IslSearch()} returns the correct location to the operation which invoked it, otherwise \textit{IslSearch()} retries (if concurrent interference detected) or aborts (if time order violated) post releasing locks (Line 213).

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 3.15. Consider the history in Figure 3.15(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 \textit{preds}($k_1$, $k_3$) and \textit{currs}($k_5$, $k_3$) with the help of \textit{IslSearch()} for key $k_5$ (refer Figure 3.15(ii)). Now, let \textit{Del}($k_5$) acquire the lock on the \textit{preds} and \textit{currs} before the \textit{Luo}($k_5$) and delete the node corresponding to the key $k_5$ from \textit{bl} leading to state $s_2$ (in Figure 3.15(iii)) and commit. Figure 3.15(ii) shows the state $s_2$ where key $k_5$ is the part of \textit{rl}. Now, \textit{methodValidation()} (in Algo 11) will identify that location of \textit{Luo}($k_5$) is no more valid due to (\textit{sh\_preds}[0].\textit{bl} $\neq$ \textit{sh\_currs}[1]) at Line 261 of Algo 11. Thus, \textit{IslSearch()} will retry to find the updated location for \textit{Luo}($k_5$) at state $s_3$ (in Figure 3.15(iii)) and eventually $T_2$ will commit.

![Figure 3.15: Interference Validation for conflicting concurrent methods on key $k_5$](image)

\textit{STM\_lookup}() & \textit{STM\_delete}() behaves similarly during \textit{rv\_method execution} phase except that \textit{STM\_delete}() is validated twice. First, in \textit{rv\_method execution} similar to \textit{STM\_lookup}() and secondly in \textit{upd\_method execution} (of \textit{STM\_tryC}()) to ensure opacity\cite{section}. We adopt lazy delete approach for \textit{STM\_delete}() method. Thus, nodes are marked for deletion and not physically deleted for \textit{STM\_delete}() method. In the current work we assume that a garbage collection mechanism is present and we donot worry about it.

\textbf{upd\_method execution phase}: Finally a transaction after executing the designated operations reaches the \textit{upd\_method execution} phase executed by the \textit{STM\_tryC}() method. It starts with modifying the log to \textit{ordered\_list} which contains the log entries in sorted order of the keys (so that locks can be acquired in an order, refer Line 122 of Algo 6) and contains only the \textit{upd\_method} (because we do not validate the lookup again for the reasons explained above for Figure 3.19). From Line 124 to Line 135 (in Algo 6) 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 \textit{rv\_method execution} phase. If the location for an operation has changed this block ensures that they are updated.

23
Now, $STM_{tryC}()$ enters the phase where it updates the shared memory using local data stored from Line 138 to Line 173 in Algo 6. Figure 3.16 & Figure 3.17 explain the execution of insert and delete in update phase of $STM_{tryC}()$ using $lslIns()$ and $lslDel()$ respectively. Figure 3.16(i) represents the case when $k_5$ is neither present in $bl$ and nor in $rl$ (Line 158 to Line 162 in Algo 6). It adds $k_5$ to lazyskip-list at location $preds(k_3, k_4)$ and $currs(k_8, k_8)$. Figure 3.16(i) is lazyskip-list before addition of $k_5$ and Figure 3.16(i)(a) is lazyskip-list state post addition. Similarly, Figure 3.16(ii) represents the case when $k_5$ is present in $rl$ (Line 153 to Line 157 in Algo 6). It adds $k_5$ to lazyskip-list at location $pred(k_3, k_4)$ and $curr(k_5, k_8)$. Figure 3.16(i)(c) is lazyskip-list before addition of $k_5$ into $bl$ and Figure 3.16(i)(d) is lazyskip-list state post addition. In case of $del(k_5)$ from lazyskip-list when $k_5$ is present in $bl$ (Line 167 to Line 173 in Algo 6) Figure 3.17(i) represent the lazyskip-list state before $k_5$ is deleted at location $preds(k_3, k_3)$ and $currs(k_5, k_5)$ and Figure 3.17(ii) represents the lazyskip-list state after deletion.

In $upd\ method\ execution$ phase two consecutive updates within same transaction having overlapping $preds$ and $currs$ may overwrite the previous method such that only effect of the later method is visible ($lost\ update$). This happens because the previous method while updating, changes the lazyskip-list causing the $preds$ &
currs of the next method working on the consecutive key to become obsolete. Figure 3.18 explains this lucidly. Suppose, \( T_1 \) is in update phase of \( STM\_tryC() \) at state \( s \) where \( ins_1(k_5) \) and \( ins_1(k_7) \) are waiting to take effect over the lazyskip-list. The lazyskip-list at \( s \) is as in Figure 3.18(i) also \( ins_1(k_5) \) and \( ins_1(k_7) \) have \( preds(k_3, k_5) \) and \( currs(k_8, k_8) \) as their location. Now, Lets say \( ins_1(k_5) \) adds \( k_5 \) between \( k_3 \) and \( k_8 \) and changes lazyskip-list (as in Figure 3.18(ii)) at state \( s_1 \) in Figure 3.18(iv). But, at \( s_1 \) bl preds and currs of \( ins_1(k_7) \) are still \( k_3 \) and \( k_8 \) thus it wrongly adds \( k_7 \) between \( k_3 \) and \( k_8 \) overwriting \( ins_1(k_5) \) as shown in Figure 3.18(iii) with dotted links. We correct this through \( intraTransValidation() \) which updates current upd_method’s preds and currs with the help of its \( ll\_entry \). We discuss lost update validation in detail at Algo 13. Next we elaborate the method of \( HT\_OSTM \).

### 3.3 \( HT\_OSTM \) Pseudocode

We now describe the implementation internals of the \( HT\_OSTM \). As discussed in life cycle of each transaction that every \( HT\_OSTM \) transacation executes in two phases \( rv\_method \& upd\_method \). methods executed in theses phases are \( STM\_begin, STM\_lookup(), STM\_insert(), STM\_delete(), STM\_tryC() \). We one by one explain each of the method in the ensuing text.

\( STM\_begin \) is the first function a transaction executes in its life cycle. It initiates the \( txlog \) (local log) for the transaction (Line 3) and provides an unique id to the transaction (Line 5).

#### Algorithm 1

\begin{align*}
\text{function } & \text{ STM\_begin(}t, \text{id} \uparrow) : \text{ initiates local transaction log and return the transaction id.} \\
1: & \text{ function STM\_begin(}t, \text{id} \uparrow) : \text{ initiates local transaction log and return the transaction id.} \\
2: & \text{ return } t, \text{id} \uparrow & \text{ OSTM as 0 */} \\
3: & \text{ txlog } \leftarrow \text{ new txlog()} & 5: t, \text{id} \leftarrow \text{ get\&inc(sh\_entry) } \uparrow) / \Phi_{ip} \\
4: & \text{ } & 6: \text{ return } (t, \text{id}); \\
\end{align*}

\( STM\_lookup() \) in Algo 2. 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 14 to Line 23 in Algo 2). If the last operation was an \( STM\_insert() \) (or \( STM\_lookup() \)) on same key then the subsequent \( STM\_lookup() \) of the same transaction returns the previous value (Line 18 in Algo 2) inserted (or observed) without accessing shared memory, and if the last operation was an \( STM\_delete() \) then \( STM\_lookup() \) returns the value \( NULL \) (Line 22 in Algo 2). Thus in this process subsequent methods also have same conflicts as the first method on same key within the same transaction (\textit{conflict inheritance}).

If \( STM\_lookup() \) is the first operation on a particular key then it has to do a wait free traversal (Line 70 in Algo 4) with the help of \( IsSearch() \) (Algo 7) 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). The commonLu&Del() algorithm is invoked at Line 27 of Algo 2. If the node is present as blue (or red) node then it updates the operation status as OK (or FAIL) and returns the value respectively (Line 77 to Line 86 in Algo 4). If node corresponding to the key is not found then it inserts that node (Line 87 to Line 92 in Algo 4) 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 3.12 & Figure 3.13. Finally, it updates the meta information in \( ll\_list \) and releases the locks acquired inside \( IsSearch() \) (Line 95 to Line 99).

We prefer \( STM\_lookup() \) to be validated instantly and is never validated again in \( STM\_tryC() \) as the design choice to aid performance. Let’s consider \( HT\_OSTM \) history in Figure 3.19(i), if we would have validated
$Lu(ht, k_1, v_0)$ again during tryC, $T_1$ would abort due to time order violation[10], but we can see that this history is acceptable where $T_1$ can be serialized before $T_2$ (Figure 3.19(ii)). Thus, HT-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 3.19(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.

![Figure 3.19: Advantages of lookup validated once](image)

**Algorithm 2**  
STM_lookup(t_id ↓, obj_id ↓, key ↓, value ↑, op_status ↑)
**DESCR**
If the transaction to which this operation belongs has locally done an operation on the same key then returns apt value and status(wrt the previous local operation). Else do the IsiSearch() to find the correct location of the key and validate it.

**IN**
: obj_id, key

**OUT**
: value, op_status

```
8: function STM_lookup
9:     STATUS, op_status ← RETRY ;
10:    /* get the txlog of the current transaction by t_id */;
11:    txlog ← getTxLog(t_id ↓);
12:    /* If already in log update the le with the current operation */
13:    if (txlog.findLL((t_id ↓, obj_id ↓, key ↓, le ↑)) then
14:        opn ← le.getOpn(obj_id ↓, key ↓);
15:        /* if previous operation is insert/lookup then current method would have value/op_status same as previous log entry */
16:        if ((INSERT = opn) || (LOOKUP = opn) then
17:            value ← le.getValue(obj_id ↓, key ↓);
18:            op_status ← le.getOpStatus(obj_id ↓, key ↓) ;
19:        /* if previous operation is delete then current method would have value as NULL and op_status as FAIL */
20:        if previous operation is delete then current method would
21:            else if (DELETE = opn) then
22:                value ← NULL ;
23:                op_status ← FAIL ;
24:            end if
25:        else
26:            /* common function for rv_method, if node corresponding to the key is the not part of underlying DS */
27:            commonLu&Del(t_id ↓, obj_id ↓, key ↓, value ↑ , op_status ↑);
28:        end if
29:    /* update the local log */
30:    le.setOpn(obj_id ↓, key ↓, LOOKUP ↓) ;
31:    le.setOpStatus(obj_id ↓, key ↓, op_status ↓) ;
32:    return (value, op_status) ;
33: end function
```

**STM_delete** (Algo 3) in *rv_method execution* phase executes as similar to rv_method and in *upd_method execution* phase executes as upd_method. In *rv_method execution* phase, the STM_delete() first checks if their is already a previous method on same key using the local log. In case their is already a method that executed on same key, STM_delete() does not need to touch shared memory and sees the effect of the previous method and returns accordingly (Line 39 to Line 57). For example if previous executed method is an insert then the current STM_delete() method will return OK (Line 42 to Line 46). If the previous executed method is an STM_delete() then the current STM_delete() should return FAIL (Line 48 to Line 51). In case previous method was STM_lookup() then current STM_delete() returns the status same as that of the previous STM_lookup() method also overwriting the log for the value and opn.
Algorithm 3  STM_delete(t_id ↓, obj_id ↓, key ↓, value ↑, op_status ↑)

34: function STM_delete  
35:     STATUS op_status ← RETRY;  
36:     /* get the txlog of the current transaction by t_id */  
37:     txlog ← getTxLog(t_id ↓);  
38:     /* if le(obj_id, key) already in log, update the le with the current operation */  
39:     if (txlog.findInList(t_id ↓, obj_id ↓, key ↓, le)) then  
40:         op ← le.getOpn(obj_id ↓, key ↓);  
41:         /* if previous local method is insert and current operation is delete then overall effect should be of delete, update log accordingly */  
42:         if (INSERT = opn) then  
43:             value ← le.getValue(obj_id ↓, key ↓);  
44:             le.setValue(obj_id ↓, key ↓, NULL ↓);  
45:             le.setOpn(obj_id ↓, key ↓, DELETE ↓);  
46:             op_status ← OK;  
47:         /* if previous local method is delete and current operation is delete then overall effect should be of delete, update log accordingly */  
48:         else if (DELETE = opn) then  
49:             le.setValue(obj_id ↓, key ↓, NULL ↓);  
50:             value ← NULL;  
51:             op_status ← FAIL;  
52:     else  
53:         /* if previous local method is lookup and current operation is delete then overall effect should be of delete, update log accordingly */  
54:         value ← le.getValue(obj_id ↓, key ↓);  
55:         le.setValue(obj_id ↓, key ↓, NULL ↓);  
56:         le.setOpn(obj_id ↓, key ↓, DELETE ↓);  
57:         op_status ← le.getOpStatus(obj_id ↓, key ↓);  
58:         end if  
59:     end if  
60:     /* common function for rv_method, if node corresponding to the key is not the part of underlying DS */  
61:     commonLu&Del(t_id ↓, obj_id ↓, key ↓, value ↑, op_status ↑);  
62:     end if  
63:     /* update the local log */  
64:     le.setOpStatus(obj_id ↓, key ↓, op status ↓);  
65:     return (value, op_status);  
66: end function

In case the current STM_delete() is not the first method on key then it touches the shared memory to identify the correct location over the hash-table from Line 59 to Line 65. IslSearch() gives the correct location for the current STM_delete() to take effect over the hash-table in form of preds and currs (Line 70 in Fig 4) along with the validation status which reveals weather the STM_delete() will succeed or abort. If the op_status is Abort, the method simply aborts the transaction. Otherwise, STM_delete() updates the local log and the time stamps of the corresponding nodes in the lazyskip-list of the hash-table from line Line 75 to Line 65.

From Line 77 to Line 81, STM_delete() observes that the node to be deleted is reachable from bl i.e. it is sh_currs[1] thus it updates it’s time-stamp field and returns op_status to OK with the value of sh_currs[1] (the update corresponding to this case takes place in STM_fryC() as represented in Figure 3.23). From Line 82 to Line 86, STM_delete() observes that the node to be deleted is reachable by rl i.e. it is sh_currs[0] thus it updates its time-stamp field and sets op_status to FAIL (as the node is dead node or marked for deletion) and value returned is NULL. Otherwise, in Line 87 to Line 92 the node is not at all present in lazyskip-list. Thus first STM_delete() adds a node in rl and updates its time-stamp and returns the value as NULL and sets the op_status as FAIL (Figure 3.20 and Figure 3.21 represents the case). Line 98, Line 99 and Line 64 sets the value, location and opn in local log respectively. At Line 95 the locks acquired(in invoked IslSearch()) to update shared memory time-stamps are released in order.

Figure 3.20: k10 is not present in bl as well as rl

Figure 3.21: Adding k10 into rl
Algorithm 4  commonLu&Del($t_{id} \downarrow, obj_{id} \downarrow, key \downarrow, value \uparrow, op_{status} \uparrow$)

68: function commonLu&Del
69:  /* le present for (obj_{id}, key), merely update the log */
70:  if (op_{status} = ABORT) then
71:      /* release all the locks */
72:      releasePred&CurrLocks(sh_{preds}[] ↓, sh_{curs}[] ↓);
73:  end if
74:  /\* if node (obj_{id}, key) is neither in blue or red list add the node in red list and update timestamp */
75:  if (read(sh_{curs}[0].max_ts.lookup, TS(t_{id})) = NULL) then
76:      value ← NULL;
77:  end if
78:  /\* if node (obj_{id}, key) is part of blue list */
79:  if (read(sh_{curs}[0].max_ts.lookup, TS(t_{id}));
80:      write(sh_{curs}[0].max_ts.lookup, TS(t_{id}));
81:      /* create new log entry in log */
82:  end if
83:  /* node (obj_{id}, key) is part of red list */
84:  if (read(sh_{curs}[0].max_ts.lookup, TS(t_{id})) = NULL) then
85:      write(sh_{curs}[0].max_ts.lookup, TS(t_{id}));
86:  end function

STM_insert() method in rv_method execution phase simply checks if there is a previous method that executed on the same key. If there 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 112, Line 113 and Line 114 respectively). In case the STM_insert() is the first method on key it creates a new log entry for the ll_list of txlog at Line 109. Finally the STM_insert() gets to modify the underlying hash-table using lslIns() at the upd_method execution phase in STM_tryC().

Algorithm 5  STM_insert ($t_{id} \downarrow, obj_{id} \downarrow, key \downarrow, value \downarrow, op_{status} \uparrow$) : updates log entry and return op_status locally.

103: function STM_insert
104:  STATUS op_{status} ← OK;
105:  /* get the txlog of the current transaction by t_{id} */
106:  txlog ← getTxLog(t_{id} ↓);
107:  if (txlog.findInLL(t_{id} ↓, obj_{id} ↓, key ↓, le ↑)) then
108:      /* no present for this (obj_{id}, key), create one */
109:      le ← new ll_entry(obj_{id} ↓, key ↓);
110:  end if
111:  /* le present for (obj_{id}, key), merely update the log */
112:  le.setValue(obj_{id} ↓, key ↓, value ↓, IN_INSERT ↓);
113:  /* update op_status to the transaction that invoked insert */
114:  return (op_{status});
115:  end function
Algorithm 6  STM_tryC(t_id ↓, tx_status ↑)

118: function STM_TRYC
119:    /* get the txlog of the current transaction by t_id */
120:    ll_list ← txlog.getLlList(t_id ↓);
121:    /* sort the local log in increasing order of keys and copy into ordered list */
122:    ordered_ll_list ← txlog.sort(ll_list ↓);
123:    /* identify the new preds and currs for all update methods of a tx and validate it */
124:    while (le ← next(ordered_ll_list)) do
125:        (key, obj_id) ← le.getKey&ObjId(le ↓);
126:        /* search correct location for the operation over lsl and lock the corresponding sh_preds[] and sh_currs[] */
127:        lslSearch(t_id ↓, obj_id ↓, key ↓, TRY C ↓, sh_preds[] ↑, sh_currs[] ↑, op_status ↑);
128:        /* if lslSearch return op_status as ABORT then method will return ABORT */
129:        if (op_status = ABORT) then
130:            /* release local memory in case lslSearch returns abort */
131:            handleAbort(t_id ↓);
132:            return (op_status);
133:        end if
134:        /* modify the log entry to help upcoming update method of same tx */
135:        le.setPreds&Currs(obj_id ↓, key ↓, sh_preds[] ↓, sh_currs[] ↓);
136:    end while
137:    /* get each update method one by one and take the effect in underlying DS */
138:    while (le ← next(ordered_ll_list)) do
139:        (key, obj_id) ← le.getKey&ObjId(le ↓);
140:        /* get the operation name to local log entry */
141:        opn ← le.opn;
142:        /* if operation is insert then after successful completion of it node corresponding to the key should be part of bl */
143:        if (INSERT = opn) then
144:            /* if node corresponding to the key is part of bl */
145:            if (read(sh_currs[1], key) = key) then
146:                /* get the value from local log */
147:                value ← le.getValue(obj_id ↓, key ↓);
148:                /* update the value into underlying DS */
149:                write(sh_currs[1], value);
150:                /* update the max_ts of insert for node corresponding to the key into underlying DS */
151:                write(sh_currs[1], max_ts.insert, TS(t_id ↓));
152:            else if (read(sh_currs[0], key) = key) then
153:                /* connect the node corresponding to the key to bl as well */
154:                insert(sh_preds[] ↓, sh_currs[] ↓, RL, BL ↓);
155:                /* update the max_ts of insert for node corresponding to the key into underlying DS */
156:                write(sh_currs[0], max_ts.insert, TS(t_id ↓));
157:            else
158:                /* if node corresponding to the key is not part of bl as well if then create the node with the help of lslIns() and add it into bl */
159:                lslIns(sh_preds[] ↓, sh_currs[] ↓, BL ↓);
160:                /* update the max_ts of insert for node corresponding to the key into underlying DS */
161:                write(node.max_ts.insert, TS(t_id ↓));
162:            end if
163:        end if
164:        /* if operation is delete then after successful completion of it node corresponding to the key should not be part of bl */
165:        if (DELETE = opn) then
166:            /* if node corresponding to the key is part of bl */
167:            if (read(sh_currs[1], key) = key) then
168:                /* delete the node corresponding to the key from the bl with the help of lslDel() */
169:                lslDel(sh_preds[] ↓, sh_currs[] ↓);
170:                /* update the max_ts of delete for node corresponding to the key into underlying DS */
171:                write(sh_currs[1], max_ts.delete, TS(t_id ↓));
172:            end if
173:        end if
174:    end if
175:    /* modify the preds and currs for the consecutive update methods which are working on overlapping zone in lazyskip-list */
176:    intranTransValidation(le ↓, sh_preds[] ↑, sh_currs[] ↑);
177:    end while
178:    /* release all the locks */
179:    releaseOrderedLocks(ordered_ll_list ↓);
180:    /* set the tx status as OK */
181:    tx_status ← OK;
182:    return (tx_status);
183: end function
Algorithm 7 lslSearch(t_id ↓, obj_id ↓, key ↓, val_type ↓, sh_preds[ ] ↑, sh_curs[ ] ↑, op_status ↑) : finds location (sh_preds[ ]& sh_curs[ ]) for given (obj_id, key) and returns them in locked state else returns ABORT.

185: function lslSearch
186: STATUS op_status ← RETRY;
187: while (op_status = RETRY) do
188:   /* get the head of the bucket in hash-table */
189:   head ← getHead(obj_id ↓, key ↓);
190:   /* init sh_preds[ ] to head */
191:   sh_preds[0] ← read(head);
192:   /* init sh_curs[1] to sh_preds[0].bl */
193:   sh_curs[1] ← read(sh_preds[0].bl);
194:   /* search node (obj_id, key) location in blue list */
195:   while (read(sh_curs[1].key) < key) do
196:     sh_curs[0] ← read(sh_curs[1].key);
197:     sh_curs[1] ← read(sh_curs[0].bl);
198:   end while
199:   /* init sh_preds[1] to sh_preds[0].bl */
200:   sh_preds[1] ← read(sh_preds[0].bl);
201:   /* init sh_curs[0] to sh_preds[0].rl */
202:   sh_curs[0] ← read(sh_preds[0].rl);
203:   /* search node (obj_id, key) location in red list between sh_preds[0]& sh_curs[1] */
204:   while (read(sh_curs[0].key) < key) do
205:     sh_curs[0] ← read(sh_curs[0].rl);
206:   end while
207:   if validation(t_id ↓, key ↓, sh_preds[ ] ↓, sh_curs[ ] ↓, val_type ↓, op_status ↑) then
208:     /* if validation returns op_status as RETRY or ABORT then release all the locks */
209:     releasePred&CurrLocks(sh_preds[ ], sh_curs[ ], op_status ↑);
210:     if (op_status = RETRY) ∨ (op_status = ABORT) then
211:       validation(t_id ↓, key ↓, sh_preds[ ] ↓, sh_curs[ ] ↓, val_type ↓, op_status ↑);
212:     end if
213:   end if
214:   flag := RETRY;
215:   /* insert the node into red list only */
216:   if sh_preds[0] ≠ sh_preds[1] then
217:     sh_preds[0] ← write(sh_preds[0].rl, node);
218:     sh_curs[0] ← write(sh_curs[0].bl, node);
219:     flag := RETRY;
220:   end if
221:   write(node.marked, True);
222:   write(node.rl, sh_preds[0].rl);
223:   write(node.bl, sh_curs[0].bl);
224: end function

Algorithm 8 lslIns(sh_preds[ ] ↓, sh_curs[ ] ↓, list_type ↓) : Inserts or overwrites a node in underlying hash table at location corresponding to preds & curs.

220: function lslIns
221:   /* inserting the node which is red list to bluelist */
222:   if ((list_type = RL) ∧ (BL)) then
223:     write(sh_preds[0].marked, false);
224:     write(sh_curs[0].bl, sh_curs[1].bl);
225:     write(sh_preds[0].bl, sh_curs[0].bl);
226:     /* inserting the node into red list only */
227:     else if ((list_type = RL)) then
228:       node = Create new node();
229:       write(node.marked, True);
230:       write(node.rl, sh_preds[0].rl);
231:       write(sh_preds[1].rl, node);
232:   else
233:     /* inserting the node into red as well as blue list */
234:     node = new node();
235:     /* after creating the node acquiring the lock on it */
236:     node.lock();
237:     write(node.rl, sh_preds[0].rl);
238:     write(node.bl, sh_curs[1].bl);
239:     write(sh_preds[1].rl, node);
240:     write(sh_preds[0].bl, node);
241:   end if
242:   return ( );
243: end function

Figure 3.22: Execution of lslIns(): (i) key k5 is present in rl and adding it into bl, (ii) key k5 is not present in rl as well as bl and adding it into rl, (iii) key k5 is not present in rl as well as bl and adding it into rl as well as bl.
**lslIns()** (Alg 8) 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()* is invoked in *upd_method execution* phase for the corresponding *STM_insert()* in local log represented by the block from Line 222 to Line 225. Here we first reset the sh_currs[0] mark field and update the bl to the sh_currs[1] and sh_preds[0] bl to sh_currs[0]. Thus the node is now reachable by bl also. Figure 3.22(i) represents the case.

- **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 227 to Line 231 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 230 and Line 231 the rl field of the node is updated to sh_currs[0] and rl field of the sh_preds[1] is modified to point to the node respectively. Figure 3.22(ii) represents the case.

- **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 222 to Line 225) or the node is not present at all. The later case is depicted in Line 232 to Line 240 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 3.22(iii) represents the case.

Algorithm 9  
*lslDel(sh_pspecs[] ↓, sh_currs[] ↓) : Deletes a node from blue link in underlying hash table at location corresponding to preds & currs.*

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>244:</td>
<td>function <em>lslDel</em></td>
</tr>
<tr>
<td>245:</td>
<td>/* mark the node (obj_id, key) for deletion */</td>
</tr>
<tr>
<td>246:</td>
<td>write(sh_currs[1].marked, True) ;</td>
</tr>
<tr>
<td>247:</td>
<td>/* set the update the blue links */</td>
</tr>
<tr>
<td>248:</td>
<td>write(sh_preds[0].bl, sh_currs[1].bl) ;</td>
</tr>
<tr>
<td>249:</td>
<td>return ⟨⟩ ;</td>
</tr>
<tr>
<td>250:</td>
<td>end function</td>
</tr>
</tbody>
</table>

 IslDel() 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 (sh_currs[1]) and changes the bl of sh_pspecs[1] to sh_currs[0] as shown in Line 246 and Line 248 of Alg 9 respectively. Figure 3.23 shows the deletion of node corresponding to k5.

![Figure 3.23: Execution of I宣讲Del(): (i) lazyskip-list before k5 is deleted, (ii) lazyskip-list after k5 is deleted from bl](image)

**validation:** *rv_method* and *upd_method* do the validation in *rv_method execution* phase and *upd_method execution* phase respectively. *validation* invokes *methodValidation()* and then does the *transValidation()* in the mentioned order. *methodValidation()* is the property of the method and *transValidation()* is the property of the transaction. Thus validating the method before the transaction intuitively make sense.

Algorithm 10  
*validation(t_id ↓, key ↓, sh_pspecs[] ↓, sh_currs[] ↓, val_type ↓, op_status ↑)*

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>251:</td>
<td>function <em>validation</em></td>
</tr>
<tr>
<td>252:</td>
<td>/* validate against concurrent updates */</td>
</tr>
<tr>
<td>253:</td>
<td>op_status ← methodValidation(sh_pspecs[] ↓) ;</td>
</tr>
<tr>
<td>254:</td>
<td>/* on successful full validation validate of transactional ordering to ensure opacity */</td>
</tr>
<tr>
<td>255:</td>
<td>if (RETRY ≠ op_status) then</td>
</tr>
<tr>
<td>256:</td>
<td>op_status ← transValidation(t_id ↓, key ↓, sh_currs[] ↓, val_type ↓, op_status ↑) ;</td>
</tr>
<tr>
<td>257:</td>
<td>end if</td>
</tr>
<tr>
<td>258:</td>
<td>return (op_status) ;</td>
</tr>
<tr>
<td>259:</td>
<td>end function</td>
</tr>
</tbody>
</table>
In `methodValidation()` each transaction ensures that no other transaction has concurrently updated the same location in lazyskip-list where it wants to perform the operation. This is done by checking that the `sh_preds[0]` and `sh_currs[1]` are not marked for deletion and next node of `sh_preds[0]` and `sh_preds[1]` is still the same as observed by lockfree traversal over the lazyskip-list.

```
Algorithm 11  methodValidation(sh_preds[], sh_currs[])
260: function METHODVALIDATION
261: if (read(sh_preds[0].marked) || read(sh_currs[1].marked) || read(sh_preds[0].bl) != sh_currs[1].rl) ≠ sh_currs[0]) then
262:    return ⟨RETRY⟩;
263:  else
264:    return ⟨OK⟩;
265:  end if
266: end function
```

In `transValidation()` `rv_method` always conflicts with the `upd_method` (as established in conflict notion Section 3.1.3). If the node corresponding to the `key` is present in the lazyskip-list (Line 274) we compare with time-stamp of the transaction that last executed the conflicting method on same `key`. If the current method that invoked the `transValidation()` is `rv_method` then Line 277 handles the case. Otherwise, if the invoking method is `upd_method` then Line 281 handles the case. Figure 3.24 and Figure 3.25 show the execution of `transValidation()`. Here `Lu_1((ht, k_1))` will return `Abort` in Figure 3.25 because `Del_2((ht, k_1))` of `T_2` has already updated the time-stamp at the node corresponding to `k_1`. So, when `Lu_1((ht, k_1))` does its `transValidation()` at Line 281, `TS(t_1) < curr.max_ts.delete(k)` holds true (since, `T_1 < T_2`) leading to `abort` of `T_1` at Line 282. This gives us a equivalent sequential schedule which can be shown co-opaque. Figure 3.24 shows the schedule where no sequential schedule is possible if `transValidation()` is not applied as there is no way to recognize the time-order violation.

```
Figure 3.24: non opaque history. Without time-stamp validation in `transValidation()`
```

```
Figure 3.25: opaque history H1. With time-stamp validation in `transValidation()`
```

`intraTransValidation()` handles the case where two consecutive updates within same transaction having overlapping `peds` and `currs` may overwrite the previous method such that only effect of the later method is visible. This happens because the previous method while updating, changes the lazyskip-list causing the `peds` & `currs` of the next method working on the consecutive key to become obsolete. Thus, `intraTransValidation()` corrects this by finding the new `peds` and `currs` of the current method on the consecutive key. There might be two cases (i) if previous method is `STM_insert()` or (ii) previous method is `STM_delete()`. For case(i) we find the `sh_preds[0]` (at Line 292 to Line 294 using previous log entry) and for case(ii) we find `sh_preds[0]` using previous log entry’s `sh_preds[0]` (Line 299) and finally find the new `sh_preds[1]` and `sh_currs[0]` between the new found `sh_preds[0]` and `sh_currs[1]` at Line 304 to Line 306.
Algorithm 12 \ transValidation(t\_id ↓, key ↓, sh\_curr s[] ↓, val\_type ↓, op\_status ↑) : Time-order validation for each transaction.

268: \textbf{function} \textbf{trans\_Validation}
269: \hspace{1em} /* by default setting the op\_status as RETRY */
270: \hspace{1em} STATUS op\_status ← OK ;
271: \hspace{1em} /* get the appropriate sh\_curr (red or blue) corresponding to key */
272: \hspace{1em} \textbf{let} get\_Apt\_Curr(sh\_curr s[]) ↓, key ↓, sh\_curr ↑;
273: \hspace{1em} /* if sh\_curr is not NULL and node corresponding to the key is equal to sh\_curr.key then check for TS */
274: \hspace{1em} if ((sh\_curr ≠ NULL) ∧ ((sh\_curr.key) = key)) then
275: \hspace{1em} /* if val\_type is RV then transaction validation for rv\_method */
276: \hspace{1em} if ((val\_type = RV) ∧ (TS(t\_id) < (\textbf{read}(sh\_curr.max\_js.insert(k))) ||
277: \hspace{1em} (TS(t\_id) < (\textbf{read}(sh\_curr.max\_js.delete(k)))))) then
278: \hspace{1em} op\_status ← ABORT ;
279: \hspace{1em} /* transaction validation for upd\_method */
280: \hspace{1em} else if ((TS(t\_id) < (\textbf{read}(sh\_curr.max\_js.insert(k))) ||
281: \hspace{1em} (TS(t\_id) < (\textbf{read}(sh\_curr.max\_js.delete(k)))) then
282: \hspace{1em} op\_status ← ABORT ;
283: \hspace{1em} end if
284: \hspace{1em} end if
285: \hspace{1em} return (op\_status) ;
286: \textbf{end function}

Algorithm 13 \ intraTransValidation(le ↓, sh\_preds[] ↑, sh\_curr s[] ↑)

287: \textbf{function} \textbf{intraTrans\_Validation}
288: \hspace{1em} \textbf{let} getAll\_Preds\_\&\_\_\textbf{C}urr\_\_\textbf{le} ↓, sh\_preds[] ↑, sh\_curr s[] ↑;
289: \hspace{1em} /* if sh\_preds[0] is marked or sh\_curr s[1] is not reachable from sh\_preds[0].bl then modify the next consecutive upd\_method sh\_preds[0] based on previous upd\_method */
290: \hspace{1em} if ((\textbf{read}(sh\_preds[0].marked)) || (\textbf{read}(sh\_preds[0].bl))
291: \hspace{1em} ⇒ sh\_curr s[1])) then
292: \hspace{1em} /* find \( k \), \( i \) such that le\_k contains previous update method on same bucket */
293: \hspace{1em} if ((le\_k.opn) = INSERT) then
294: \hspace{1em} le\_k, sh\_preds[0].unlock () ;
295: \hspace{1em} \textbf{sh} pred\_s[0] ← (le\_k, sh\_preds[0].bl) ;
296: \hspace{1em} \textbf{le\_k, sh\_preds[0].lock () ;}
297: \hspace{1em} /* upd\_method method sh\_preds[0] will be previous

\textbf{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 \textbf{key}. This is done by linearly traversing the log and finding an entry corresponding to the \textbf{key}. If the calling method is the first method of the transaction for the \textbf{key} then \textbf{findInLL} return false as it would not find any entry in the log of the transaction corresponding to the \textbf{key}. Since we consider that their can be multiple objects (hash\_table) so we need to find unique \( \textbf{obj.id, \textbf{key}} \) pair (refer Line 315).

While executing the \textbf{trans\_Validation()} the time\_stamp field of the corresponding \textbf{node} has to be updated. Such a node can be either the marked (dead or sh\_curr s[0]) or the unmarked (live sh\_curr s[1]). \textbf{get\_apt\_curr} in Algo 15 is the utility method which returns the appropriate \textbf{node} corresponding to the \textbf{key}.
**Algorithm 14** findInLL(\(t_{id} \downarrow, obj_{id} \downarrow, key \downarrow, le \uparrow\)) : Checks whether any operation corresponding to \((obj_{id}, key)\) is present in ll list.

311: function findInLL
312: ll list ← txlog.getLlList(\(t_{id} \downarrow\));
313: /* every method first identify the node corresponding to the key into local log */
314: while (\(le, \leftarrow \text{next}(ll list)\)) do
315: if ((\(le, \text{first} = obj_{id}\))\&\&(\(le, \text{first} = key\))) then
316: return \((TRUE, le)\);
317: end if
318: end while
319: return \((FALSE, le = NULL)\);
320: end function

**Algorithm 15** get_aptcurr(\(sh\_curreys[] \downarrow, key \downarrow, sh\_curr \uparrow\)) : Returns a curr node from underlying DS which corresponds to the key of \(le\).

321: function get_aptcurr
322: /* by default set curr to NULL */
323: sh curr ← NULL;
324: /* if node corresponding to the key is part of bl then curr is \(sh\_curreys[1]\) */
325: if (\(sh\_curreys[1].key = key\)) then
326: sh curr ← \(sh\_curreys[1]\);
327: /* if node corresponding to the key is part of rl then curr is \(sh\_curreys[0]\) */
328: else if (\(sh\_curreys[0].key = key\)) then
329: sh curr ← \(sh\_curreys[0]\);
330: end if
331: return \((sh\_curr)\);
332: end function

**release_ordered_locks** in Algo 16 is an utility method to release the locks in order of the keys to avoid deadlock.

**Algorithm 16** release_ordered_locks(\(ordered\_ll\_list \downarrow\)) : Release all locks taken during lslSearch().

333: function release_ordered_locks
334: /* releasing all the locks on preds, curr and node */
335: while (\(le, \leftarrow \text{next}(ordered\_ll\_list)\)) do
336: \(le, \text{sh\_preds[0].unlock()}\); //\(\phi\)
337: \(le, \text{sh\_preds[1].unlock()}\);
338: if \(le, \text{node}\) then
339: \(le, \text{node.unlock()}\)
340: end if
341: \(le, \text{sh\_curreys[0].unlock()}\); //\(\phi\)
342: \(le, \text{sh\_curreys[1].unlock()}\);
343: end while
344: return ();
345: end function

**acquirePred&CurrLocks** in Algo 17 & **releasePred&CurrLocks** in Algo 18 do what their names denote. They are used as helping methods in Algo 7.

**Algorithm 17** acquirePred&CurrLocks(\(sh\_preds[] \downarrow, sh\_curreys[] \downarrow\)) : acquire all locks taken during lslSearch().

346: function acquirePred&CurrLocks
347: \(sh\_preds[0].lock()\);
348: \(sh\_preds[1].lock()\);
349: \(sh\_curreys[0].lock()\);
350: \(sh\_curreys[1].lock()\);
351: return ();
352: end function

**Algorithm 18** releasePred&CurrLocks(\(sh\_preds[] \downarrow, sh\_curreys[] \downarrow\)) : Release all locks taken during lslSearch().

353: function releasePred&CurrLocks
354: \(sh\_preds[0].unlock()\); //\(\phi\)
355: \(sh\_preds[1].unlock()\);
356: \(sh\_curreys[0].unlock()\);
357: \(sh\_curreys[1].unlock()\);
358: return ();
359: end function
3.4 Optimizations

In case a \textit{STM\_delete()} method returns FAIL then it would just behave as a \textit{STM\_lookup()} because it does not modify the underlying data structure. Thus, we do not need to revalidate such failed \textit{STM\_delete()} method in upd\_method phase inside \textit{STM\_tryC()}. This helps in saving extra computation and time spent during upd\_method phase leading to speedup of the transaction.

Furthermore, twice validating the failed \textit{STM\_delete()} also may lead to unnecessary aborts as shown with an example in Figure 3.26. The Figure 3.26(i) shows the schedule where \( T_1 \) validates \( del_1(k_1) \) two times. During \textit{STM\_tryC()} it aborts realizing during its validation that \( T_2 \) has scheduled a conflicting insert operation on same node. On the other hand, if would not have validated this failed delete in \textit{STM\_tryC()} the schedule can be accepted hence saving an unnecessary abort as shown in Figure 3.26(ii).

![Figure 3.26: Advantage of validating \textit{STM\_delete()} once, if its returning \textit{FAIL} in \textit{rv\_method execution} phase](image)

Second optimization could be that during \textit{IslSearch()} if node corresponding to the node is part of the underlying data structure and the corresponding \textit{methodValidation()} returns a retry (unsuccessful) then instead of retrying again we can do a \textit{transValidation()} so that in case the transaction is doomed to abort we would avoid unnecessary computation in retrying a transaction that is bound to abort.
Chapter 4

Proof Of Correctness

Brief Summary:

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 HT-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 HT-OSTM history at transaction level. We present the proof sketch in this section.

HT-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 (Observation 24), 5) all unmarked nodes are reachable by bl (Lemma 15) and every node (marked or unmarked) is reachable by rl (Lemma 10). From code it can be observed lslSearch() is guaranteed to return correct location for a method (Observation 7 and Lemma 8).

Linearization Points: Here, we list the linearization points (LPs) of each method. Note that each method of the list can return either OK, FAIL or ABORT. So, we define the LP for all the methods:

1. STM_begin(): (global_cntn++) at Line 5 of STM_begin().

2. STM_insert(ht, k, OK/FAIL/ABORT): Linearization point for the STM_insert() follows the LPs of the STM_tryC().

3. STM_delete(ht, k, OK/FAIL/ABORT): preds[0].unlock() at Line 95 of STM_delete().

4. STM_tryC(ht, k, OK/FAIL/ABORT): ll_entry, preds[0].unlock() at Line 336 of releaseOrdered-Locks(). Which is called at Line 180 of STM_tryC().

Operational level correctness: Here we establish the above HT-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 HT-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 HT-OSTM method (Definition 5)) 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 [10] as observed in Observation 25 and Observation 26. We prove that lost update validation is not violated by subsequent updates in STM_tryC() in Lemma 18.
Lemma 1. Consider a concurrent history, \( E^H \), let there be a successful STM\_tryC() method of a transaction \( T_i \) which last updated the node corresponding to \( k \). Now, Consider a successful \( rv \_\text{method} \) of a transaction \( T_j \) on key \( k \) then,

1.1 If in the the pre-state of LP event of the \( rv \_\text{method} \), node corresponding to the key \( k \) is part of bl and value is \( v \). Then the last upd\_method of STM\_tryC() would be insert on same key \( k \) and value \( v \) and it should be the previous closest to the \( rv \_\text{method} \).

1.2 If in the the pre-state of LP event of the \( rv \_\text{method} \), node corresponding to the key \( k \) is not part of the bl. Then the last upd\_method in STM\_tryC() would be delete on same key \( k \) and it should be the previous closest to the \( rv \_\text{method} \).

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 \( HT\_OSTM \). We then show that it is co-opaque[17] by showing its conflict graph is acyclic. Since our algorithm uses time-order validation[10], we show that conflict graph is acyclic by showing that all the edge follow timestamp order as proved in Lemma 2, Lemma 3.

Lemma 2. If \((T_i, T_j) \in \text{conflict}(H) \Rightarrow TS(T_i) < TS(T_j)\).

Lemma 3. If \((T_1, T_2 \cdots T_n)\) is a path in \( CG(H) \), this implies that \((TS(T_1) < TS(T_2) < \cdots < TS(T_n))\).

Finally, using the fact that \( HT\_OSTM \) generates legal histories whose conflict graph is acyclic. We show that \( HT\_OSTM \) histories are co-opaque [17] as stated below (proved in Theorem 48).

Theorem 4. A legal history \( H \) is co-opaque iff \( CG(H) \) is acyclic.

Safety of \( HT\_OSTM \): We formally say that \( HT\_OSTM \) generates linearizable history at operational level (Observation 34) and the conflict graph generated by \( HT\_OSTM \) history is acyclic (Theorem 47). For a complete proof of all the above lemmas and theorem please refer the Appendix ???. Above discussion gives enough intuition to believe that \( HT\_OSTM \) will indeed be co-opaque[17] hence opaque[16]. Moreover, depending upon the lock implementation \( HT\_OSTM \) can be starvation free(if locks provide starvation free mutual exclusion).

Deadlock freedom of \( HT\_OSTM \): 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.
4.1 Proof Sketch of OSTMs

4.1.1 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 $evts(S) \subseteq evts(S')$. In this case, we denote $S \sqsubseteq S'$. 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, $\langle S.Abs = S.Abs.rl \cup S.Abs.bl \rangle$, where,

$S.Abs.rl := \{ \forall n | (n \in S.PublicNodes) \land (S.Head \rightarrow_{rl} S.n) \}$

$S.Abs.bl = \{ \forall n | (n \in S.PublicNodes) \land (\neg S.n.marked) \land (S.Head \rightarrow_{bl} S.n) \}$

**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, $\forall S, S': (n \in S.nodess) \land (S \sqsubseteq S') \Rightarrow (n \in S'.nodes)$.

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, $\forall S, S': (n \in S.nodess) \land (S \sqsubseteq S') \Rightarrow (n \in S'.nodes) \land (S.n.key = S'.n.key)$.

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

1. $(preds[0] \land preds[1] \land currs[0] \land currs[1]) \in S.PublicNodes$
2. $(S.preds[0].locked) \land (S.preds[1].locked) \land (S.currs[0].locked) \land (S.currs[1].locked)$
3. $(\neg S.preds[0].marked) \land (\neg S.currs[0].marked) \land (S.preds[0].bl = S.currs[1]) \land (S.preds[1].rl = S.currs[0])$

In Observation 7, lslSearch() method returns only if validation succeed at Line 211.

**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,

1. $((S.preds[0].key) < key \leq (S.currs[1].key))$.
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 191 of lslSearch() method of Algo 7 initializes $S.preds[0]$ to point head node. Also, $(S.currs[1] = \ldots)$.
As in penultimate execution of line 195 \((S.currs[1].key < key)\) and at line 196 \((S.preds[0] = S.currs[1])\) this implies,

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

The node key doesn’t change as known by Observation 6. So, before executing of line 200, we know that,

\[(key \leq S.currs[1].key)\]

From eq(4.1) and eq(4.2), we get,

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

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 7 returns.

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

Line 200 of \(lslSearch()\) method of Algo 7 initializes \(S.preds[1]\) to point \(S.preds[0]\). Also, \((S.currs[0] = S.preds[0].rl)\) by line 202. As in penultimate execution of line 204 \((S.currs[0].key < key)\) and at line 205 \((S.preds[1] = S.currs[0])\) this implies,

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

The node key doesn’t change as known by Observation 6. So, before executing of line 209, we know that

\[(key \leq S.currs[0].key)\]

From eq(4.4) and eq(4.5), we get,

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

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 7 returns.

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

**Proof.** We prove by Induction on events that change the \(rl\) field of the node (as these affect reachability), which are Line 230, 231, 237 & 239 of \(lslIns()\) method of Algo 8. It can be seen by observing the code that \(lslDel()\) method of Algo 9 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 \((S.head.key < S.tail.key) \land (head, tail) \in S.nodes\).
**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 230 of \( lslIns() \) method**: By observing the code, we notice that Line 230 (\( rl \) field changing event) can be executed only after the \( lslSearch() \) method of Algo 7 returns. Line 228 of the \( lslIns() \) method creates a new node, \( node \) with \( key \) and at line 229 set the \((S.node.marked = true)\) (because inserting the node only into the redlink). Line 230 then sets \((S.node.rl = S.currs[0])\). Since this event does not change the \( rl \) field of any node reachable from the head of the list (because \( node \notin S.PublicNodes \)), the lemma is not violated.

2. **Line 231 of \( lslIns() \) method**: By observing the code, we notice that Line 231 (\( rl \) field changing event) can be executed only after the \( lslSearch() \) method of Algo 7 returns. From Lemma 8.2, we know that when \( lslSearch() \) method of Algo 7 returns then,

\[
(S.preds[1].key) < key \leq (S.currs[0].key)
\]  
(4.7)

To reach line 231 of \( lslIns() \) method, line 87 of commonLu&Del() method of Algo 4 should ensure that,

\[
(S.currs[0].key \neq key) \implies (S.preds[1].key) < key < (S.currs[0].key)
\]  
(4.8)

From Observation 7.3, we know that,

\[
(S.preds[1].rl = S.currs[0])
\]  
(4.9)

Also, the atomic event at line 231 of \( lslIns() \) sets,

\[
(S.preds[1].rl = node) \implies (S.sh.preds[1].key < node.key)
\]  
(4.10)

\[
\implies (S.preds[1].key < S.preds[1].rl.key)
\]

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 237 of \( lslIns() \) method**: By observing the code, we notice that Line 237 (\( rl \) field changing event) can be executed only after the \( lslSearch() \) method of Algo 7 returns. Line 234 of the \( lslIns() \) method creates a new node, \( node \) with \( key \). Line 237 then sets \((S.node.rl = S.currs[0])\). Since this event does not change the \( rl \) field of any node reachable from the head of the list (because \( node \notin S.PublicNodes \)), the lemma is not violated.

4. **Line 239 of \( lslIns() \) method**: By observing the code, we notice that Line 239 (\( rl \) field changing event) can be executed only after the \( lslSearch() \) Algo 7 method returns. From Lemma 8.2, we know that when \( lslSearch() \) method of Algo 7 returns then,

\[
(S.preds[1].key) < key \leq (S.currs[0].key)
\]  
(4.11)
To reach line 239 of \textit{IssIns()} method, line 158 of \textit{STM\_tryC()} method of Algo 6 should ensure that,

\[
(S.\text{currs}[0].\text{key} \neq \text{key}) \xRightarrow{(4.11)} (S.\text{preds}[1].\text{key}) < \text{key} < (S.\text{currs}[0].\text{key})
\]  

From Observation 7.3, we know that,

\[
(S.\text{preds}[1].rl = S.\text{currs}[0])
\]  

Also, the atomic event at line 239 of \textit{IssIns()} sets,

\[
(S.\text{preds}[1].rl = \text{node}) \xRightarrow{(4.12)} (S.sh.\text{preds}[1].\text{key} < \text{node}.\text{key})
\]  

\[
\implies (S.\text{preds}[1].\text{key} < S.\text{preds}[1].rl.\text{key})
\]  

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

\[
\square
\]

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

\textbf{Proof.} We prove by Induction on events that change the \(rl\) field of the node (as these affect reachability), which are Line 230, 231, 237 & 239 of \textit{IssIns()} method of Algo 8. It can be seen by observing the code that \textit{IssDel()} method of Algo 9 do not have any update events of \(rl\).

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

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

\textbf{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. \textbf{Line 230 of \textit{IssIns()} method:} Line 228 of the \textit{IssIns()} method creates a new node, \(\text{node}\) with \(\text{key}\) and at line 229 set the \((S.\text{node}.\text{marked} = \text{true})\) (because inserting the node only into the redlink). Line 230 then sets \((S.\text{node}.rl = S.\text{currs}[0])\). Since this event does not change the \(rl\) field of any node reachable from the head of the list (because \(\text{node} \notin S.\text{PublicNodes}\)), the lemma is not violated.

2. \textbf{Line 231 of \textit{IssIns()} method:} By observing the code, we notice that Line 231 \((rl\) field changing event) can be executed only after the \textit{IssSearch()} method of Algo 7 returns. From line 230 \\& 231 of \textit{IssIns()} method, \((S.\text{node}.rl = S.sh.\text{currs}[0]) \land (S.sh.\text{preds}[1].rl = S.\text{node}) \land (\text{node} \in S.\text{PublicNodes}) \land (S.\text{node}.\text{marked} = \text{true})\) (because inserting the node only into the redlink). It is to be noted that (from Observation 7.2), \((sh.\text{preds}[0], sh.\text{preds}[1], sh.\text{currs}[0], sh.\text{currs}[1])\) are locked, hence no other thread can change marked field of \(S.sh.\text{preds}[1]\) and \(S.sh.\text{currs}[0]\) simultaneously. Also, from Observation 6, a node’s key field does not change after initialization. Before executing line 231, \(sh.\text{preds}[1]\) is reachable from head by \(rl\) (from induction
hypothesis). After line 231, we know that from $sh.preds[1]$, public marked node, $node$ is also reachable. Thus, we know that $node$ is also reachable from head. Formally, $(S.Head \rightarrow^*_ril S.sh.preds[1]) \land (S.sh.preds[1] \rightarrow^*_ril S.node) \Rightarrow (S.Head \rightarrow^*_ril S.node)$.

3. **Line 237 of lslIns() method**: Line 234 of the lslIns() method creates a new node, $node$ with key. Line 237 then sets $(S.node.rl = S.currs[0])$. Since this event does not change the $rl$ field of any node reachable from the head of the list (because $node \notin S.PublicNodes$), the lemma is not violated.

4. **Line 239 of lslIns() method**: By observing the code, we notice that Line 239 ($rl$ field changing event) can be executed only after the lslSearch() method of Algo 7 returns. From line 237 & 239 of lslIns() method, $(S.node.rl = S.sh.currs[0]) \land (S.sh.preds[1].rl = S.node) \land (node \in S.PublicNodes) \land (node.marked = false)$ (because new node is created by default with unmarked field). It is to be noted that (from Observation 7.2), $(S.preds[0], S.preds[1], sh.currs[0], sh.currs[1])$ are locked, hence no other thread can change marked field of $S.sh.preds[1]$ and $S.sh.currs[0]$ simultaneously. Also, from Observation 6, a node’s key field does not change after initialization. Before executing line 239, $sh.preds[1]$ is reachable from head by $rl$ (from induction hypothesis). After line 239, we know that from $sh.preds[1]$, public unmarked node, $node$ is also reachable. Thus, we know that $node$ is also reachable from head. Formally, $(S.Head \rightarrow^*_ril S.sh.preds[1]) \land (S.sh.preds[1] \rightarrow^*_ril S.node) \Rightarrow (S.Head \rightarrow^*_ril S.node)$.

\[ \square \]

**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, $(\forall S, n, key : S.PublicNodes \land (S.n.key < key < S.n.rl.key) \implies node(key) \notin S.Abs)$.

**Observation 13.** Consider a global state $S$ which has a node $n$ is reachable from head via $rl$. Then in any future state $S'$ of $S$, node $n$ is also reachable from head via $rl$ in $S'$ as well. Formally, $(\forall S, S' : (n \in S.nodes) \land (S \subseteq S') \land (S.head \rightarrow^*_ril S.n) \Rightarrow (n \in S'.nodes) \land (S'.head \rightarrow^*_ril S'.n))$.

**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 9).

\[ \square \]

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

**Proof.** We prove by Induction on events that change the $bl$ field of the node (as these affect reachability), which are Line 224, 225, 238 & 240 of lslIns() method of Algo 8 and Line 248 of lslDel() method of Algo 9.

**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 $(S.head.key < S.tail.key) \land (head, tail) \in S.nodes$. 

42
**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 224 & 225 of *lslIns()* method:** By observing the code, we notice that Line 224 & 225 (\( bl \) field changing event) can be executed only after the *lslSearch()* method of Algo 7 returns. From Lemma 8.1 and Lemma 8.2, we know that when *lslSearch()* method of Algo 7 returns then,

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

(4.15)

To reach line 224 of *lslIns()* method, line 153 of *STMtryC()* method of Algo 6 should ensure that,

\[
(S.curs[1].key \neq key) \land (S.curs[0].key = key) \stackrel{eq(4.15)}{=} ((S.preds[0].key) < key < (S.curs[1].key)) \land ((S.preds[1].key) < (key = S.curs[0].key))
\]  

(4.16)

From Observation 7.3, we know that,

\[
(S.preds[0].bl = S.curs[1]) \land (S.preds[1].rl = S.curs[0])
\]  

(4.17)

The atomic event at line 224 of *lslIns()* sets,

\[
(S.curs[0].bl = S.curs[1]) \stackrel{eq(4.16)}{\subseteq} (S.curs[0].key) < (S.curs[1].key) \Rightarrow (S.curs[0].key) < (S.curs[0].bl.key)
\]  

(4.18)

Also, the atomic event at line 225 of *lslIns()* sets,

\[
(S.preds[0].bl = S.curs[0]) \stackrel{eq(4.16)}{=} (S.preds[0].key) < (S.curs[0].key) \Rightarrow (S.preds[0].key) < (S.preds[0].bl.key).
\]  

(4.19)

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

2. **Line 238 of *lslIns()* method:** By observing the code, we notice that Line 238 (\( bl \) field changing event) can be executed only after the *lslSearch()* method of Algo 7 returns. Line 234 of the *lslIns()* method creates a new node, \( node \) with \( key \). Line 238 then sets \((S.node.bl = S.curs[1])\). 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 240 of *lslIns()* method:** By observing the code, we notice that Line 240 (\( bl \) field changing event) can be executed only after the *lslSearch()* method of Algo 7 returns. From Lemma 8.1
and Lemma 8.2, we know that when \textit{IslSearch}() method of Algo 7 returns then,

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

(4.20)

To reach line 240 of \textit{IslIns}() method, line 158 of \textit{STM} \texttt{tryC}() method of Algo 6 should ensure that,

\[(S.currs[0].key \neq key) \land (S.currs[1].key \neq key) \equiv (4.20)\]

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

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

(4.21)

From Observation 7.3, we know that,

\[(S.preds[0].bl = S.currs[1])\]  

(4.22)

Also, the atomic event at line 240 of \textit{IslIns}() sets,

\[(S.preds[0].bl = S.node) \equiv (4.21)\]

\[(S.preds[0].key < S.node.key) \implies (S.preds[0].key < S.preds[0].bl.key)\]  

(4.23)

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 248 of \textit{IslDel}() method**: By observing the code, we notice that Line 248 (bl field changing event) can be executed only after the \textit{IslSearch}() method of Algo 7 returns. From Lemma 8.1, we know that when \textit{IslSearch}() method of Algo 7 returns then,

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

(4.24)

To reach line 248 of \textit{IslDel}() method, line 169 of \textit{STM} \texttt{tryC}() method of Algo 6 should ensure that,

\[(S.currs[1].key = key) \equiv (4.24)\]

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

(4.25)

From Observation 7.3, we know that,

\[(S.preds[0].bl = S.currs[1])\]  

(4.26)

We know from Induction hypothesis,

\[(currs[1].key < currs[1].bl.key)\]  

(4.27)

Also, the atomic event at line 248 of \textit{IslDel}() sets,

\[(S.preds[0].bl = S.currs[1].bl) \equiv (4.25, 4.27)\]

\[(S.preds[0].key < S.currs[1].bl.key) \implies (S.preds[0].key < S.preds[0].bl.key)\]  

(4.28)
Lemma 15. In a global state $S$, any unmarked public node $n$ is reachable from Head via blue links. Formally, $(\forall S, n : (S.PublicNodes) \land (\neg S.n.marked) \implies (S.Head \rightarrow bl S.n))$.

Proof. We prove by Induction on events that change the $bl$ field of the node (as these affect reachability), which are Line 224, 225, 238 & 240 of $lslIns()$ method of Algo 8 and line 248 of $lslDel()$ method of Algo 9.

**Base condition**: Initially, before the first event that changes the $bl$ field of any node, we know that $(head, tail) \in S.PublicNodes$ and $\neg (S.head.marked) \land \neg (S.tail.marked) \land (S.head \rightarrow bl S.tail)$.

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

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

1. **Line 224 & 225 of $lslIns()$ method**: By observing the code, we notice that Line 224 & 225 ($bl$ field changing event) can be executed only after the $lslSearch()$ method of Algo 7 returns. It is to be noted that (from Observation 7.2), $(sh.preds[0], sh.preds[1], sh.currs[0], sh.currs[1])$ are locked, hence no other thread can change $S.sh.preds[0].marked$ and $S.sh.currs[1].marked$ simultaneously. Also, from Observation 6, a node’s key field does not change after initialization. Before executing line 224, from Observation 7.3,

   $$(S.sh.preds[0].marked = false) \land (S.sh.currs[1].marked = false) \quad (4.29)$$

And from Lemma 10 and induction hypothesis,

$$(S.Head \rightarrow rl S.sh.currs[0]) \land (S.Head \rightarrow bl S.sh.currs[1]) \quad (4.30)$$

After line 224, we know that from $sh.currs[0].marked = false$, public unmarked node, $sh.currs[1]$ is also reachable, implies that,

$$(S.sh.currs[0] \rightarrow bl S.sh.currs[1]) \quad (4.31)$$

Also, before executing line 225, from induction hypothesis and Lemma 10,

$$(S.Head \rightarrow bl S.sh.preds[0]) \land (S.Head \rightarrow rl S.sh.currs[0]) \quad (4.32)$$

After line 225, we know that from $sh.preds[0].marked = false$, public unmarked node (from line 223 of $lslIns()$ method), $sh.currs[0]$ is also reachable via $bl$, implies that,

$$(S.sh.preds[0] \rightarrow bl S.sh.currs[0]) \land (S.sh.currs[0].marked = false) \quad (4.33)$$

From eq(4.31) and eq(4.33),

$$(S.sh.preds[0] \rightarrow bl S.sh.currs[0]) \land (S.sh.currs[0] \rightarrow bl S.sh.currs[1]) \land (S.sh.currs[0].marked = false) \quad (4.34)$$
Since \((sh\_preds[0], sh\_currs[0]) \in S.\text{PublicNode}\) and hence, \((S.\text{Head} \to_{bl}^* S.\text{sh\_preds}[0]) \land (S.\text{sh\_preds}[0] \to_{bl}^* S.\text{sh\_currs}[0]) \land (S.\text{sh\_currs}[0].\text{marked} = false) \Rightarrow (S.\text{Head} \to_{bl}^* S.\text{sh\_currs}[0]).\)

2. **Line 238 of `lslIns()` method**: Line 234 of the `lslIns()` method creates a new node, \(n\) with key. Line 238 then sets \((S.\text{node}.bl = S.\text{currs}[1])\). Since this event does not change the \(bl\) field of any node reachable from the head of the list (because \(n \notin S.\text{PublicNodes}\)), the lemma is not violated.

3. **Line 240 of `lslIns()` method**: By observing the code, we notice that Line 240 (bl field changing event) can be executed only after the `lslSearch()` method of Algo 7 returns. It is to be noted that (from Observation 7.2), \((sh\_preds[0], sh\_preds[1], sh\_currs[0], sh\_currs[1])\) are locked, hence no other thread can change \(S.\text{sh\_preds}[0].\text{marked}\) and \(S.\text{sh\_currs}[1].\text{marked}\) simultaneously. Also, from Observation 6, a node’s key field does not change after initialization. Before executing line 238, from Observation 7.3,

\[(S.\text{sh\_preds}[0].\text{marked} = false) \land (S.\text{sh\_currs}[1].\text{marked} = false)\] (4.35)

And from induction hypothesis,

\[(S.\text{Head} \to_{bl}^* S.\text{sh\_currs}[1])\] (4.36)

After line 238, we know that from \(n\), public unmarked node, \(sh\_currs[1]\) is also reachable via \(bl\), implies that,

\[(S.\text{node} \to_{bl}^* S.\text{sh\_currs}[1])\] (4.37)

Also, before executing line 240, from induction hypothesis,

\[(S.\text{Head} \to_{bl}^* S.\text{sh\_preds}[0])\] (4.38)

After line 240, we know that from \(sh\_preds[0]\), public unmarked node (because new node is created by default with unmarked field), \(n\) is also reachable via \(bl\), implies that,

\[(S.\text{sh\_preds}[0] \to_{bl}^* S.\text{node}) \land (S.\text{node}.\text{marked} = false)\] (4.39)

From eq(4.37) and eq(4.39),

\[(S.\text{sh\_preds}[0] \to_{bl}^* S.\text{node}) \land (S.\text{node} \to_{bl}^* S.\text{sh\_currs}[1]) \land (S.\text{node}.\text{marked} = false)\] (4.40)

Since \((sh\_preds[0], n) \in S.\text{PublicNode}\) and hence, \((S.\text{Head} \to_{bl}^* S.\text{sh\_preds}[0]) \land (S.\text{sh\_preds}[0] \to_{bl}^* S.\text{node}) \land (S.\text{node}.\text{marked} = false) \Rightarrow (S.\text{Head} \to_{bl}^* S.\text{node}).\)

\[\square\]

**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, \(\forall S, n : (n \in S.\text{nodes}) \land (S.\text{head} \to_{bl}^* S.\text{n}) \subseteq (S.\text{head} \to_{rl}^* S.\text{n}).\)
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.

\[ \square \]

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, \((\text{node}(\text{key}) \in ([E^H . \text{Post}(m_i . \text{LP}]) . \text{Abs} . rl)) \land (\text{node}.max Ts = TS(T_i))\).

Proof.

1. For \text{rv} method: By observing the code, each \text{rv} method first invokes \text{IslSearch()} method of Algo 7 (line 70 of \text{commonLu&Del()} 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 \text{IslSearch()} is invoked from a method, it returns correct location \((sh.preds[0], sh.preds[1], sh.curs[0], sh.curs[1])\) 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 \( LP \) event of \text{rv} method, if \((\text{node}.key \in S.Abs.rl)\), means key is already there in \( rl \) and time-stamp of that node is less then the \text{rv} method transactions time-stamp, from \text{transValidation()} method of Algo 12, then in the post-state of \( LP \) event of \text{rv} method, \text{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 \text{key} by method transaction time-stamp else abort.

   In the pre-state of \( LP \) event of \text{rv} method, if \((\text{node}.key \notin S.Abs.rl)\), means key is not there in \( rl \) then, in the post-state of \( LP \) event of \text{rv} method, insert the node corresponding to the key into \( rl \) by using \text{IslIns()} method of Algo 8 and update the \( max Ts \) field for corresponding node \text{key} by method transaction time-stamp. Since, \text{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 \text{rv} method.

2. For \text{upd} method: By observing the code, each \text{upd} method also first invokes \text{IslSearch()} method of Algo 7 (line 127 of \text{STM.TryC()} method of Algo 6). From Lemma 9 & Lemma 14 we have that the nodes in the underlying data-structure are in increasing order of their keys, thus the key on which the method is working has a unique location in underlying data-structure from Corollary 11. So, when the \text{IslSearch()} is invoked from a method, it returns correct location \((sh.preds[0], sh.preds[1], sh.curs[0], sh.curs[1])\) 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 \text{upd} method is insert: In the pre-state of \( LP \) event of \text{upd} method, if \((\text{node}.key \in S.Abs.rl)\), means key is already there in \( rl \) and time-stamp of that node is less then the \text{upd} method transactions time-stamp, from \text{transValidation()} method of Algo 12, then in the post-state of \( LP \) event of \text{upd} method, \text{node}.key should be the part of \( rl \) and it just update the \( max Ts \) field for corresponding node \text{key} by method transaction time-stamp else abort.

   In the pre-state of \( LP \) event of \text{upd} method, if \((\text{node}.key \notin S.Abs.rl)\), means key is not there in \( rl \) then in the post-state of \( LP \) event of \text{upd} method, it will insert the node corresponding to the key into the \( rl \) as well as \( bl \), from \text{IslIns()} method of Algo 8 at line 167 of \text{STM.TryC()} method of Algo 6 and update the \( max Ts \) field for corresponding node \text{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 transValidation() 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 9 at line 173 of STM\_tryC() method of Algo 6 and it just update the maxTs 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 3 must have already inserted a node in the underlying data-structure prior to STM\_tryC() method of Algo 6 . Thus, (node.key \( \in \) S.Abs.rl) and update the maxTs field for corresponding node key by method transaction time-stamp else abort.

In HT-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 (sh\_preds[0], sh\_preds[1], sh\_currs[0], sh\_currs[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 intra trans validation after each upd\_method in STM\_tryC(), during upd\_method execution phase.

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

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

By observing the code at line 177 of STM\_tryC() method of Algo 6, current upd\_method will go for intraTransValidation() and at line 290 of intraTransValidation() method of Algo 13 , current upd\_method will validate its (sh\_preds[0].marked) and (sh\_preds[0].bl! = sh\_currs[1]). If any condition is true then, at line 292 of intraTransValidation() method of Algo 13, will check for previous upd\_method. If the previous upd\_method is insert then the current upd\_method update its sh\_preds[0] to previous upd\_method, node.key else set current upd\_method sh\_preds[0] to previous upd\_method sh\_preds[0].

After that at line 304 of intraTransValidation() method of Algo 13, current upd\_method validate its (sh\_preds[1].rl! = sh\_currs[0]). If condition is true then current upd\_method set its sh\_preds[1] to previous upd\_method, node.key.

If we will not update the current method preds and currs using intraTransValidation() then effect of first upd\_method will be overwritten by the next upd\_method.

**Observation 19.** For any global state S, the intraTransValidation() 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, $\langle (node\_key \in \{E^H.\text{Post}(m_1, LP)\}.Abs.bl) \land (node\_val = v) \rangle$.

20.2 If $upd\_method$ is delete, then node corresponding to the key should not be part of $bl$. Formally, $\langle (node\_key \notin \{E^H.\text{Post}(m_1, LP)\}.Abs.bl) \rangle$.

**Proof.** By observing the code, each $upd\_method$ also first invokes $lslSearch()$ method of Algo 7 (line 127 of STM\_tryC() method of Algo 6). 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 $\langle sh\_preds[0], sh\_preds[1], sh\_currs[0], sh\_currs[1]\rangle$ 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 145, 153 of STM\_tryC() method of Algo 6, if $(node\_key \in S.Abs.rl)$, means $key$ is already there in $rl$ and timestamp of that node is less then the $upd\_method$ transactions time-stamp, from $transValidation()$ 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 158 of STM\_tryC() method of Algo 6, 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 8 at line 160 of STM\_tryC() method of Algo 6 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 169 of STM\_tryC() method of Algo 6, if $(node\_key \in S.Abs.bl)$, means $key$ is already there in $bl$ and time-stamp of that node is less then the $upd\_method$ transactions time-stamp, from $transValidation()$ 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 9 at line 169 of STM\_tryC() method of Algo 6.

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 3 must have already inserted a node in the underlying data-structure prior to STM\_tryC() method of Algo 6.

□

**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, $\langle (node\_key \in \{E^H.\text{Pre}(m_1, LP)\}.Abs.bl) \land (S.node\_val = v) \Rightarrow rvm(key, OK, v) \rangle$.

**Proof.** Let the $rv\_method$ is STM\_lookup() method of Algo 2 and it is the first key method of the transaction, we ignore the abort case for simplicity.

From line 70 of commonLu&Del() method of Algo 4, when $lslSearch()$ method of Algo 7 returns we
have \((\text{preds}[0], \text{preds}[1], \text{currs}[0], \text{currs}[1]) \in S.\text{PublicNodes}\) and are locked (from Observation 7.1 & Observation 7.2) until \(\text{STM} \text{lookup}()\) method of Algo 2 return. Also, from Lemma 8.1,

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

To return OK, \(S.\text{currs}[1]\) should be reachable from the head via bluelist from Definition 4, in the pre-state of \(LP\) of \(\text{rv}_{\text{method}}\). And after observing code, at line 77 of commonLu&Del() method of Algo 4,

\[
(S.\text{currs}[1].\text{key} = \text{key}) \equiv (S.\text{preds}[0].\text{key} < (\text{key} = S.\text{currs}[1].\text{key}))
\]

Also, from Observation 7.3,

\[
(S.\text{preds}[0].\text{bl} = S.\text{currs}[1])
\]

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

*Same argument can be extended to \(\text{STM.delete}()\) method.

\[\Box\]

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

**Proof.** Let the \(\text{rv}_{\text{method}}\) is \(\text{STM} \text{lookup}()\) method of Algo 2 and it is the first key method of the transaction, we ignore the abort case for simplicity.

1. From line 70 of commonLu&Del() method of Algo 4, when \(\text{IslSearch}()\) method of Algo 7 returns we have \((\text{preds}[0], \text{preds}[1], \text{currs}[0], \text{currs}[1]) \in S.\text{PublicNodes}\) and are locked (from Observation 7.1 & Observation 7.2) until \(\text{STM} \text{lookup}()\) method of Algo 2 return. Also, from Lemma 8.2,

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

To return FAIL, \(S.\text{currs}[0]\) should not be reachable from the head via bluelist from Definition 4, in the pre-state of \(LP\) of \(\text{rv}_{\text{method}}\). And after observing code, at line 82 of commonLu&Del() method of Algo 4,

\[
(S.\text{currs}[0].\text{key} = \text{key}) \equiv (S.\text{preds}[1].\text{key} < (\text{key} = S.\text{currs}[0].\text{key}))
\]

Also, from Observation 7.3,

\[
(S.\text{preds}[1].\text{rl} = S.\text{currs}[0])
\]

And \((\text{currs}[0] \in S.\text{nodes})\), we know \((\text{currs}[0] \in S.\text{Abs.rl})\) where \(S\) is the pre-state of the LP event of the method and \((S.\text{sh}_{\text{currs}[0].\text{marked}} = \text{true})\). Thus, \((\text{sh}_{\text{currs}[0]} \notin S.\text{Abs.bl})\) from Definition 4. Hence \((\text{node}(\text{key}) \notin ([E^H.\text{Pre}(m_i, LP)].\text{Abs.bl}))\)

2. From line 70 of commonLu&Del() method of Algo 4, when \(\text{IslSearch}()\) method of Algo 7 returns we
have \((\text{preds}[0], \text{preds}[1], \text{curre}[0], \text{curre}[1]) \in S.\text{PublicNodes}\) and are locked (from Observation 7.1 & Observation 7.2) until \(STM.\text{lookup}()\) method of Algo 2 return. Also, from Lemma 8.2,

\[
(S.\text{preds}[1].\text{key} < \text{key} \leq S.\text{curre}[0].\text{key})
\]  

And after observing code, at line 87 of commonLu&Del() method of Algo 4,

\[
(S.\text{curre}[1].\text{key} \neq \text{key}) \land (S.\text{curre}[0].\text{key} \neq \text{key}) \stackrel{\text{eq(4.47)}}{\Rightarrow} (S.\text{preds}[1].\text{key} < \text{key} < S.\text{curre}[0].\text{key})
\]  

Also, from Observation 7.3,

\[
(S.\text{preds}[1].\text{rl} = S.\text{curre}[0])
\]  

From eq(4.48), we can say that, \(\text{node}(\text{key}) \notin S.\text{Abs}\) and from Corollary 12, we conclude that \(\text{node}(\text{key})\) not in the state after \(\text{Is} \text{Search()}\) returns. Since Observation 6 tells, no node changes its key value after initialization. Hence \(\text{node}(\text{key}) \notin ([E.\text{Pre}(m_i.\text{LP})], \text{Abs.bl})\).

*Same argument can be extended to \(STM.\text{delete}()\) method.

\[\square\]

**Observation 23.** Only the successful \(STM.\text{tryC()}\) method working on the key \(k\) can update the Abs.bl.

By observing the code, only the successful \(STM.\text{tryC()}\) method of Algo 6 is changing the \(bl\). There is no line which is changing the \(bl\) in \(STM.\text{delete}()\) method of Algo 3 and \(STM.\text{lookup}()\) method of Algo 2. Such that \(rv.\text{method}\) is not changing the \(bl\).

**Observation 24.** If \(STM.\text{tryC()}\) and \(rv.\text{method}\) 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 \(STM.\text{tryC()}\) and \(rv.\text{method}\) 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 5.** First unlocking point of each successful method is the LP.

**Linearization Points:** Here, we list the linearization points (LPs) of each method. Note that each method of the list can return either \(OK, FAIL\) or \(ABORT\). So, we define the LP for all the methods:

1. \(STM.\text{begin()}:\) (global_centr++) at Line 5 of \(STM.\text{begin}()\).
2. \(STM.\text{insert(ht, k, OK/FAIL/ABORT)}:\) Linearization point for the \(STM.\text{insert()}\) follows the LPs of the \(STM.\text{tryC()}\).
3. \(STM.\text{delete(ht, k, OK/FAIL/ABORT)}:\) \text{preds}[0].\text{unlock()} at Line 95 of \(STM.\text{delete}()\).
4. \(STM.\text{tryC(ht, k, OK/FAIL/ABORT)}:\) \text{ll.entry, preds}[0].\text{unlock()} at Line 336 of \(releaseOrderedLocks()\). Which is called at Line 180 of \(STM.\text{tryC()}\).

**Observation 25.** Two concurrent conflicting methods of different transaction can’t acquire the lock on the same node corresponding to the key \(k\) simultaneously.
Observation 26. 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, $\langle ul(m_i(k)) \prec l(m_j(k)) \rangle \Rightarrow (LP(m_i) \prec LP(m_j))$

If two concurrent conflicting methods are working on the same key $k$ and want to update Abs then they have to acquire the lock on the node corresponding to the key $k$ from Observation 24 and one of them succeed from Observation 25. If $ul(m_i(k))$ happen before the $l(m_j(k))$ then from Definition 5, $LP(m_i)$ happen before the $LP(m_j)$.

Lemma 27. Consider two state, $S_1$, $S_2$ s.t. $S_1 \sqsubseteq S_2$ and $S_1.bl.value(k) \neq S_2.bl.value(k)$ then there exist $S'$ s.t. $S' \sqsubseteq S_2$ and $S'$ contain the STM\_tryC() method on the same key $k$. Formally, $\langle (S_1.bl.value(k) \neq (S_2.bl.value(k)) \Rightarrow \exists (S'.s.t., S_1.bl \prec S'.LP(tryC) \prec S_2.bl) \rangle$. 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 25, then, $ul(tryC(k))$ happen before the $l(S_2(k))$ then $LP(tryC)$ happen before the $LP(S_2)$ from Observation 26.

Lemma 28. Consider a concurrent history, $E^H$, let there be a successful STM\_tryC() method of a transaction $T_i$ which last updated the node corresponding to $k$. Now, Consider a successful rv\_method of a transaction $T_j$ on key $k$ then,

28.1 If in the the pre-state of LP event of the rv\_method , node corresponding to the key $k$ is part of bl and value is $v$. Then the last upd\_method of STM\_tryC() would be insert on same key $k$ and value $v$ and it should be the previous closest to the rv\_method.

28.2 If in the the pre-state of LP event of the rv\_method , node corresponding to the key $k$ is not part of the bl. Then the last upd\_method of STM\_tryC() would be delete on same key $k$ and it should be the previous closest to the rv\_method.

Proof. 28.1 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 27, 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.

28.2 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.
Theorem 29. The sequential history generated by HT-OSTM at operation level is legal.

Theorem 30. The legal sequential history generated by HT-OSTM at operation level is Linearizable.

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 31. Let there be a successful STM\_tryC() method of a transaction \( T_i \) which last updated the node corresponding to \( k \). Now, consider a successful rv\_method of a transaction \( T_j \) on key \( k \) then,

31.1 If in the the pre-state of rv\_method , node corresponding to the key \( k \) is part of bl and value is \( v \). Then the last upd\_method of STM\_tryC() would be insert on same key \( k \) and value \( v \) and it should be the previous closest to the rv\_method.

31.2 If in the the pre-state of rv\_method , node corresponding to the key \( k \) is not part of the bl. Then the last upd\_method in STM\_tryC() would be delete on same key \( k \) and it should be the previous closest to the rv\_method.

Proof. 31.1 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 27 , 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.

31.2 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.

Lemma 32. Consider a sequential history, \( E^S \), for any successful method which is call by transaction \( T_i \), after the post-state of the method, node corresponding to the key should be part of rl and max\_ts of that node should be equal to method transaction time-stamp. Formally, \((\text{node}(\text{key}) \in (P.Abs.rl)) \land (P.node.max\_ts = TS(T_i)))\). Where P is the post-state of the method.

Proof. 1. For rv\_method method: By observing the code, each rv\_method first invokes lslSearch() method of Algo 7 (line 70 of commonLu&Del() 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 \((sh\_preds[0], sh\_preds[1], sh\_curs[0], sh\_curs[1])\) 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 \textit{rv} method, if \((\text{node.key} \in S.Abs.rl)\), means \textit{key} is already there in \(rl\) and time-stamp of that node is less then the \textit{rv} method transactions time-stamp, from \texttt{transValidation()} method of Algo 12, then in the post-state of \textit{rv} method, \textit{node.key} should be the part of \(rl\) from Observation 13 and \textit{key} can’t be change from Observation 6 and it just update the \textit{max.ts} field for corresponding node \textit{key} by method transaction time-stamp else abort.

In the pre-state of \textit{rv} method, if \((\text{node.key} \notin S.Abs.rl)\), means \textit{key} is not there in \(rl\) then, in the post-state of \textit{rv} method, insert the \textit{node} corresponding to the \textit{key} into \(rl\) by using \texttt{lslIns()} method of Algo 8 and update the \textit{max.ts} field for corresponding node \textit{key} by method transaction time-stamp. Since, \textit{node.key} should be the part of \(rl\) from Observation 13 and \textit{key} can’t be change from Observation 6, in post-state of \textit{rv} method.

2. \textbf{For \texttt{upd} method}: By observing the code, each \texttt{upd} method also first invokes \texttt{lslSearch()} method of Algo 7 (line 127 of \texttt{STM_TRYC()} method of Algo 6). From Lemma 9 & Lemma 14 we have that the nodes in the underlying data-structure are in increasing order of their keys, thus the key on which the method is working has a unique location in underlying data-structure from Corollary 11. So, when the \texttt{lslSearch()} is invoked from a method, it returns correct location \((sh.preds[0], sh.preds[1], sh.curns[0], sh.curns[1])\) of corresponding \textit{key} as observed from Observation 7 & Lemma 8 and all are locked, hence no other thread can change simultaneously (from Observation 7.2).

\begin{enumerate}
\item \textbf{If \texttt{upd} method is insert}: In the pre-state of \texttt{upd} method, if \((\text{node.key} \in S.Abs.rl)\), means \textit{key} is already there in \(rl\) and time-stamp of that node is less then the \texttt{upd} method transactions time-stamp, from \texttt{transValidation()} method of Algo 12, then in the post-state of \texttt{upd} method, \textit{node.key} should be the part of \(rl\) and it just update the \textit{max.ts} field for corresponding node \textit{key} by method transaction time-stamp else abort.

In the pre-state of \texttt{upd} method, if \((\text{node.key} \notin S.Abs.rl)\), means \textit{key} is not there in \(rl\) then in the post-state of \texttt{upd} method, it will insert the \textit{node} corresponding to the \textit{key} into \(rl\) as well as \(bl\), from \texttt{lslIns()} method of Algo 8 at line 162 of \texttt{STM_TRYC()} method of Algo 6 and update the \textit{max.ts} field for corresponding node \textit{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.

\item \textbf{If \texttt{upd} method is delete}: In the pre-state of \texttt{upd} method, if \((\text{node.key} \in S.Abs.rl)\), means \textit{key} is already there in \(rl\) and time-stamp of that node is less then the \texttt{upd} method transactions time-stamp, from \texttt{transValidation()} method of Algo 12, then in the post-state of \texttt{upd} method, \textit{node.key} should be the part of \(rl\), from \texttt{lslDel()} method of Algo 9 at line 173 of \texttt{STM_TRYC()} method of Algo 6 and it just update the \textit{max.ts} field for corresponding node \textit{key} by method transaction time-stamp else abort.

In the pre-state of \texttt{upd} method, \((\text{node.key} \notin S.Abs.rl)\) this should not happen because execution of \texttt{STM_DELETE()} method of Algo 3 must have already inserted a node in the underlying data-structure prior to \texttt{STM_TRYC()} method of Algo 6. Thus, \((\text{node.key} \in S.Abs.rl)\) and update the \textit{max.ts} field for corresponding node \textit{key} by method transaction time-stamp else abort.
\end{enumerate}

\textbf{Corollary 33.} \textit{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.}
4.1.2 Transactional Level

From Section 4.1.1 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 HT-OSTM is opaque.

**Observation 34.** $H$ is a sequential history obtained from HT-OSTM, as shown at operational level using LP.

**Definition 6.** $CG(H)$ is a conflict graph of $H$.

**Lemma 35.** 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,

$$(T_1.lastEvt \prec_H T_k.firstEvt) \land (T_k.lastEvt \prec_H T_1.firstEvt) \Rightarrow (T_1.lastEvt \prec_H T_1.firstEvt) \quad (4.50)$$

This contradict our assumption as eq(4.50) is impossible, from definition of program order of a transaction. Thus, cycle is not possible in serial history.

**Observation 36.** $H_2$ is an history generated by applying topological sort on $CG(H_1)$.

**Observation 37.** Topological sort maintains conflict-order and real-time order of the original history $H_1$.

**Definition 7.** 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 $m_i \prec_H m_j$. And it is represented as $\prec_{CO}^H$.

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

38.1 $H_1$ is equivalent to $H_2 \Rightarrow (\text{methods}(H_1) = \text{methods}(H_2))$.

38.2 $\prec_{CO}^{H_1} \subseteq \prec_{CO}^{H_2}$, i.e. $H_1$ preserves the conflicts of $H_2$

**Proof.** Lemma 38.2

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

Lets assume that their exists a conflict $(T_i, T_j)$ in $\prec_{CO}^{H_1}$ but not in $\prec_{CO}^{H_2}$. But, from Observation 36 & Observation 37 we know that $(T_i, T_j) \in \prec_{CO}^{H_2}$. Thus, $\prec_{CO}^{H_1} \subseteq \prec_{CO}^{H_2}$.

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

**Lemma 39.** Let $H_1$ and $H_2$ be equivalent histories such that $\prec_{CO}^{H_1} \subseteq \prec_{CO}^{H_2}$. Then, $H_1$ is legal $\implies$ $H_2$ is legal.

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

$$up_p(ht, k, v_p) \prec_{H_1}^{MR} rv_j(ht, k, v) \quad (4.51)$$
Let us assume $H_2$ is not legal. Since, $H_3$ is equivalent to $H_2$ from Lemma 38.1 such that $(rv_j(h_k, v) \in methods(H_2))$. Since $H_2$ is not legal, there exist a $(up_r(h_k, v_r) \in methods(H_2))$ such that $(up_r(h_k, v_r) = H_2.lastUpdt(rv_j(h_k, v)))$. So conflict($H_2$) has,

$$up_r(h_k, v_r) \prec H_2^{MR} rv_j(h_k, v) \quad (4.52)$$

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

$$up_p(h_k, v_p) \prec H_2^{MR} rv_j(h_k, v) \quad (4.53)$$

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

$$up_r(h_k, v_r) \prec H_1^{MR} up_p(h_k, v_p) \quad (4.54)$$

or

$$rv_j(h_k, v) \prec H_1^{MR} up_r(h_k, v_r) \quad (4.55)$$

In $H_1$ eq(4.55) is not possible, because if $(eq(4.55) \in conflict(H_1))$ implies $(eq(4.55) \in conflict(H_2))$ from $(\prec CO \subseteq \prec CO)_{H_1}$ and in $H_2$ eq(4.52) and eq(4.55) cannot occur together. Thus only possible way $up_r(h_k, v_r)$ can occur in $H_1$ is via eq(4.54). From eq(4.54) we have,

$$up_r(h_k, v_r) \prec H_2^{MR} up_p(h_k, v_p) \quad (4.56)$$

From eq(4.52), eq(4.53) and eq(4.56) we have,

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

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

**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, $∀S, S'$ : $(n \in S.Abs) \land (S \sqsubseteq S') ⇒ (n \in S'.Abs) \land (S.n.max.ts ≤ S'.n.max.ts))$.

**Proof.** We prove by induction on events that change the max.ts field of a node associated with a key, which are Line 80, 85 & 91 of commonLu&Del() method of Algo 4 and Line 151, 157, 162 & 173 of STM.stryC() method of Algo 6.

**Base condition:** Initially, before the first event that changes the max.ts field of a node associated with a key, we know the underlying lazy-skipped-list has immutable $S.head$ and $S.tail$ nodes with $(S.head.bl = S.tail)$ and $(S.head.rl = S.tail)$.

Let assume, a node corresponding to the key is already the part of underlying $rl$ which is having a time-stamp of $m_1$ as $T_1$ from Observation 41 . Let say $m_2$ of $T_2$ wants to perform on that node, by observing the code at line 6 of transValidation() 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

56
updated in increasing order of TS values.

**Induction Hypothesis:** Say, upto \( k \) events that change the \( \text{max} \_\text{ts} \) 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 \( \text{max} \_\text{ts} \) field be only one of the following:

1. **Line 80, 85 & 91 of commonLu&Del() method of Algo 4:** By observing the code, line 57 of commonLu&Del() method of Algo 4 first invokes lslSearch() method of Algo 7 for finding the node corresponding to the key. Inside the lslSearch() method of Algo 7, it will use the transValidation() method of Algo 12, if \((\text{curr.key} = \text{key})\).

   From induction hypothesis, node corresponding to the key is already the part of underlying \( rl \) which is having a time-stamp of \( m_k \) 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 transValidation() method of Algo 12, if \( \text{TS}(T_{k+1}) < \text{curr.max}\_\text{ts}.m_k() \), \( T_{k+1} \) will return abort, else to succeed, \( \text{TS}(T_{k+1}) > \text{curr.max}\_\text{ts}.m_k() \) should evaluate to true. Thus, for successful completion of \( m_k+1 \) of \( T_{k+1} \), \( \text{TS}(T_{k+1}) \) should be greater then the \( \text{TS}(T_k) \).

   Hence, node corresponding to the key, \( \text{max} \_\text{ts} \) field should be updated in increasing order of TS values.

2. **Line 151, 157, 162 & 173 of STM.tryC() method of Algo 6:** By observing the code, line 127 of STM.tryC() method of Algo 6 first invokes lslSearch() method of Algo 7 for finding the node corresponding to the key. Inside the lslSearch() method of Algo 7, it will use the transValidation() method of Algo 12, if \((\text{curr.key} = \text{key})\).

   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 transValidation() method of Algo 12, if \( \text{TS}(T_{k+1}) < \text{curr.max}\_\text{ts}.m_k() \), \( T_{k+1} \) will return abort, else to succeed, \( \text{TS}(T_{k+1}) > \text{curr.max}\_\text{ts}.m_k() \) should evaluate to true. Thus, for successful completion of \( m_k+1 \) of \( T_{k+1} \), \( \text{TS}(T_{k+1}) \) should be greater then the \( \text{TS}(T_k) \).

   Hence, node corresponding to the key, \( \text{max} \_\text{ts} \) field should be updated in increasing order of TS values.

**Corollary 43.** Every successful methods update the \( \text{max} \_\text{ts} \) 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, \( (\forall T \in H : (\text{STM.begin}(T_i) \prec \text{STM.begin}(T_j)) \Leftrightarrow (TS(T_i) < TS(T_j))) \).

**Proof.** (Only if) If \((\text{STM.begin}(T_i) \prec \text{STM.begin}(T_j))\) then \((TS(T_i) < TS(T_j))\). Lets assume \((TS(T_i) < TS(T_i))\). From Observation 40,

\[
\text{STM.begin}(T_j) \prec_H \text{STM.begin}(T_i) \tag{4.57}
\]

but we know that,

\[
\text{STM.begin}(T_j) \succ_H \text{STM.begin}(T_i) \tag{4.58}
\]

Which is a contradiction thus, \((TS(T_i) < TS(T_j))\).
(if) If \((TS(T_i) < TS(T_j))\) then \((STM\_begin(T_i) \prec STM\_begin(T_j))\). Let us assume \((STM\_begin(T_j) \prec STM\_begin(T_i))\). From Observation 40, \[TS(T_j) < TS(T_i)\] (4.59)

but we know that, \[TS(T_j) > TS(T_i)\] (4.60)

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, \[T_i\_lastEvt \prec_H T_j\_firstEvt\] (4.61)

And from program order of \(T_i\), \[T_i\_firstEvt \prec_H T_i\_lastEvt \Rightarrow STM\_begin(T_i) \prec_H T_i\_lastEvt\] (4.62)

From eq(4.61) and eq(4.62) implies that, \[T_i\_firstEvt \prec_H T_j\_firstEvt \Rightarrow STM\_begin(T_i) \prec_H STM\_begin(T_j) \Rightarrow Lemma 44 \Rightarrow TS(T_i) < TS(T_j)\] (4.63)

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 7 we get, \(m_i \prec^{MR}_H m_j\).

\(m_i\) can be \textit{rv\_methods} or \textit{upd\_methods} (which are taking the effects in \textit{STM\_tryC}() method of Algo 6) and we know that after the \textit{LP} of \(m_i\) of \(T_i\), node corresponding to the \textit{key} should be there in \textit{rl} (from Corollary 33 & Definition 4) and the time-stamp of that \textit{node} corresponding to \textit{key} should be equal to time-stamp of this method transaction time-stamp from Corollary 33 & 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 \textit{IslSearch()} is invoked from a method \(m_j\) of \(T_j\), it returns correct location \((sh\_preds[0], sh\_preds[1], sh\_currs[0], sh\_currs[1])\) of corresponding \textit{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 \textit{node} corresponding to the key \(k\) is still reachable via \textit{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 \textit{transValidation()} method of Algo 12, we know since, \(TS(T_j) < curr\_max\_ts\_m_i()\), \(T_j\) will return \text{abort} from Corollary 43. In Algo 12 for \textit{transValidation()} 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,

\[
TS(T_1) < TS(T_2) < \cdots < TS(T_{n-1})
\]

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

\[
TS(T_{n-1}) < TS(T_n) \overset{\text{eq}(4.64)}{=} (TS(T_1) < TS(T_2) < \cdots < TS(T_n))
\]

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 ,

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

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 \(\prec_{RT} H\) and \(\prec_{CO} H\) (from Definition 2). 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 35. 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 \(\prec_{RT} H = \prec_{RT} H\) and \(\prec_{CO} H = \prec_{CO} H\).

Since every two operations related by the conflict relation in \(S\) are also related by \(\prec_{CO} H\), we obtain \(\prec_{CO} H \subseteq \prec_{S} \). 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.
Chapter 5

Results

Setup: We evaluate OSTM against the lockfree hash-table of Synchrobench[31] benchmark’s ESTM and a concurrent hash-table implementation with the read/write STM[18]. We perform two kind of experiments for lookup-intensive and update intensive workloads. In first experiment, we measure throughput (transactions/second) of hash-table with OSTM, ESTM and read/write STM (with basic time stamp ordering protocol) against the varying number of threads in power of 2. In the second experiment we measure the throughput against varying range of transaction object Id’s (100 to 1000) i.e. bucket size which represents varying contention at each lazyskip-list (bucket). We perform all the experiments on Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz machine with 56 NUMA CPUs. For better readability of the plots we re-scale the throughputs to \( \log_2 \) scale.

Parameters: For all the experiments we have considered a hash-table of size 5. Each bucket of the hash-table may have node id (transaction object Id) ranging from 1 to 1000. Please note lesser the range of the transaction objects (lazyskip-list (or bucket) size) higher would be the contention amongst the transactions. The transactions in the applications may be allowed to run for a 100, 1000 or 100000 milli-second time window. Within this time window each transaction randomly decides to execute a method (insert, lookup or delete) based on the type of work load (lookup intensive or update intensive.) For the experiments where throughput is compared against the transcation object range we use 64 number of threads. Each transaction can generate 10 methods. The throughput is averaged over 10 runs of the application.
5.1 Test application

The test application is designed to evaluate the throughput (transactions/second) of the HT-OSTM for the composability of the underlying hash-table methods. Each transaction that executes the Algo 19 randomly generates the STM\_insert(), STM\_delete(), STM\_lookup() methods based on the workload distribution. Each thread executes the transaction unless the timeout seconds. Finally, the number of transactions committed by the HT-OSTM are reported for the given number of threads or the transaction object range (the range of keys allowed in the hash-table), depending upon the type of the experiment.

Algorithm 19 \texttt{test\_app()}: test application to evaluate the composability of the HT-OSTM.

```
1: function \texttt{TEST\_APP}
2: STATUS ops, txs ← ABORT;
3: int* val ← new int;
4: bool retry ← true;
5: /*keep on executing transactions unless timeout*/
6: while !timeout do
7: /*keep retrying until the transaction commits*/
8: while retry do
9: txlog ← lib.begin();
10: for int op; op < num\_op\_per\_tx; op++ do
11: int opn ← rand()%100;
12: /* generate operations with given probability */
13: if opn < prinsert then
14: opn ← INSERT;
15: else if opn < (prinsert+prdelete) then
16: opn ← DELETE;
17: else
18: opn ← LOOKUP;
19: end if
20: /*Execute the randomly generated method*/
21: if INSERT == opn then /*INSERT*/
22: ops ← lib\_t\_insert(txlog);
23: else if DELETE == opn then
24: ops ← lib\_t\_delete(txlog);
25: else
26: ops ← lib\_t\_lookup(txlog);
27: end if
28: /*commit the transaction*/
29: if ABORT == ops then
30: break;
31: end if
32: end for
33: /*commit the transaction*/
34: if ABORT ! = ops then
35: txs ← lib\_try\_Commit(txlog);
36: end if
37: if ABORT == ops || ABORT == txs then
38: retry ← true;
39: else
40: retry ← false;
41: end if
42: end while
43: end while
44: return txs;
45: end function
```
5.2 Lookup intensive workload

This section presents the evaluation results for the two experiments of lookup intensive workload i.e. \textit{STM lookup()} operation is produced with 80\% & 50\% probability in Section 5.2.1 & Section 5.2.2 respectively.

5.2.1 Experiment 1: 80\% lookup

Table 5.1 states various parameters during the evaluation.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>transaction object range</td>
<td>1000</td>
</tr>
<tr>
<td>hash-table size</td>
<td>5</td>
</tr>
<tr>
<td>lookup%</td>
<td>80</td>
</tr>
<tr>
<td>delete%</td>
<td>5</td>
</tr>
<tr>
<td>insert%</td>
<td>15</td>
</tr>
<tr>
<td>num operation per transaction</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 5.1: Evaluation parameters for 80\% lookup

plots:

Figure 5.1 shows the throughput against the varying number of threads for the comparison of ESTM and \textit{HT-OSTM}. \textit{HT-OSTM} comprehensively beats ESTM and the difference is of 1000 transactions per second in magnitude. Similarly, Figure 5.1b shows the throughput evaluation against the varying range of allowed key (transaction object) range in the underlying hash-table of \textit{HT-OSTM}. The number of threads for Figure 5.1b are 64.

(a) varying number of threads
(b) varying range of transaction objects

Figure 5.1: ESTM vs \textit{HT-OSTM}
Figure 5.2a shows the throughput against the varying number of threads for the comparison of HT-OSTM and BTO (basic time stamp ordering) protocol of RWSTMs. HT-OSTM comprehensively beats BTO. Similarly, Figure 5.2b shows the throughput evaluation against the varying range of allowed key (transaction object) range in the underlying hash-table of HT-OSTM. It can be seen that initially for lower number of threads (2 to 16) and lower transaction object range BTO is comparable to HT-OSTM. This can be attributed to the overheads in HT-OSTM. The logging or validation overhead exceeds the performance benefits of HT-OSTM for lower number of threads. The number of threads for Figure 5.2b are 64.

5.2.2 Experiment 2: 50% lookup

Table 5.2 states various parameters during the evaluation.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time window</td>
<td>100 ms</td>
</tr>
<tr>
<td>transaction object range</td>
<td>1000</td>
</tr>
<tr>
<td>hash-table size</td>
<td>5</td>
</tr>
<tr>
<td>lookup%</td>
<td>50</td>
</tr>
<tr>
<td>delete%</td>
<td>10</td>
</tr>
<tr>
<td>insert%</td>
<td>40</td>
</tr>
<tr>
<td>num operation per transaction</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 5.2: Evaluation parameters for 50% lookup.
Figure 5.3: ESTM vs HT-OSTM

Figure 5.3a shows the throughput against the varying number of threads for the comparison of ESTM and HT-OSTM. HT-OSTM comprehensively beats ESTM and the difference is of 1000 transactions per second in magnitude. Similarly, Figure 5.3b shows the throughput evaluation against the varying range of allowed key (transaction object) range in the underlying hash-table of HT-OSTM. To enhance the readability of the plots, the Y axis is plotted on log₂ scale.

Figure 5.4: ESTM vs rwSTM(BTO protocol)

Figure 5.4a shows the throughput against the varying number of threads for the comparison of HT-OSTM and BTO (basic time stamp ordering) protocol of RWSTMs. HT-OSTM comprehensively beats BTO and the difference is of 1000 transactions per second in magnitude. Similarly, Figure 5.4b shows the throughput evaluation against the varying range of allowed key (transaction object) range in the underlying hash-table of HT-OSTM. It can be seen that initially for lower number of threads (2 to 16) and lower transaction object range BTO is comparable to HT-OSTM. This can be attributed to the overheads in HT-OSTM. The logging or validation overhead exceeds the performance benifits of HT-OSTM for lower number of threads.
5.3 Update intensive workload

This section presents the evaluation results for the update intensive workload i.e. upd_method (insert and delete operation) is produced with 70% probability. Each of ensuing subsections present the results for experiments with different time windows.

5.3.1 Experiment 1: 100 ms window

parameters:

<table>
<thead>
<tr>
<th>Time window 100 ms</th>
<th>Parameters</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>transaction object range</td>
<td>1000</td>
</tr>
<tr>
<td>hash-table size</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>lookup%</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>delete%</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>insert%</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>num operation per transaction</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: Evaluation parameters for 100ms window and update intensive workload.

plots:

Figure 5.5: ESTM vs HT-OSTM

Figure 5.5a shows the throughput against the varying number of threads for the comparison of ESTM and HT-OSTM. HT-OSTM comprehensively beats ESTM and the difference is of 1000 transactions per second in magnitude. Similarly, Figure 5.5b shows the throughput evaluation against the varying range of allowed key (transaction object) range in the underlying hash-table of HT-OSTM. To enhance the readability of the plots, the Y axis is plotted on $log_2$ scale. The number of threads for Figure 5.5b are 64.
Figure 5.6a shows the throughput against the varying number of threads for the comparison of HT-OSTM and BTO (basic time stamp ordering) protocol of RWSTMs. HT-OSTM comprehensively beats BTO and the difference is of 1000 transactions per second in magnitude. Similarly, Figure 5.6b shows the throughput evaluation against the varying range of allowed key (transaction object) range in the underlying hash-table of HT-OSTM. It can be seen that initially for lower number of threads (2 to 16) BTO is comparable to HT-OSTM. This can be attributed to the overheads in HT-OSTM. The logging or validation overhead exceeds the performance benefits of HT-OSTM for lower number of threads. Please note that the number of threads for Figure 5.6b are 64.

![Figure 5.6: ESTM vs rwSTM(BTO protocol)](image)

5.3.2 Experiment 2: 1000 ms window

parameters:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time window 1000 ms</td>
<td></td>
</tr>
<tr>
<td>transaction object range</td>
<td>1000</td>
</tr>
<tr>
<td>hash-table size</td>
<td>5</td>
</tr>
<tr>
<td>lookup%</td>
<td>30</td>
</tr>
<tr>
<td>delete%</td>
<td>20</td>
</tr>
<tr>
<td>insert%</td>
<td>50</td>
</tr>
<tr>
<td>num operation per transaction</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 5.4: Evaluation parameters for 1000ms window and update intensive workload.

plots:
Figure 5.7a shows the throughput against the varying number of threads for the comparison of ESTM and HT-OSTM. HT-OSTM comprehensively beats ESTM and the difference is of 1000 transactions per second in magnitude. Similarly, Figure 5.7b shows the throughput evaluation against the varying range of allowed key (transaction object) range in the underlying hash-table of HT-OSTM. To enhance the readability of the plots, the Y axis is plotted on $\log_2$ scale. The number of threads for Figure 5.7b are 64.

Figure 5.8a shows the throughput against the varying number of threads for the comparison of HT-OSTM and BTO (basic time stamp ordering) protocol of RWSTMs. HT-OSTM comprehensively beats BTO and the difference is of 1000 transactions per second in magnitude. Similarly, Figure 5.8b shows the throughput evaluation against the varying range of allowed key (transaction object) range in the underlying hash-table of HT-OSTM. It can be seen that initially for lower number of threads (2 to 16) BTO is comparable to HT-OSTM. This can be attributed to the overheads in HT-OSTM. The logging or validation overhead exceeds the performance benefits of HT-OSTM for lower number of threads. Please note that the number of threads for Figure 5.8b are 64.
Figure 5.8: ESTM vs rwSTM(IITHSTM)

5.3.3 Experiment 3: 10000 ms window

parameters:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time window 10000 ms</td>
<td></td>
</tr>
<tr>
<td>transaction object range</td>
<td>1000</td>
</tr>
<tr>
<td>hash-table size</td>
<td>5</td>
</tr>
<tr>
<td>lookup%</td>
<td>30</td>
</tr>
<tr>
<td>delete%</td>
<td>20</td>
</tr>
<tr>
<td>insert%</td>
<td>50</td>
</tr>
<tr>
<td>num operation per transaction</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 5.5: Evaluation parameters for 10000ms window and update intense workload.

plots:
Figure 5.9a shows the throughput against the varying number of threads for the comparison of ESTM and HT-OSTM. HT-OSTM comprehensively beats ESTM and the difference is of 1000 transactions per second in magnitude. Similarly, Figure 5.9b shows the throughput evaluation against the varying range of allowed key (transaction object) range in the underlying hash-table of HT-OSTM. To enhance the readability of the plots, the Y axis is plotted on $\log_2$ scale. The number of threads for Figure 5.9b are 64.

Figure 5.10a shows the throughput against the varying number of threads for the comparison of HT-OSTM and BTO (basic time stamp ordering) protocol of RWSTMs. HT-OSTM comprehensively beats BTO. Similarly, Figure 5.10b shows the throughput evaluation against the varying range of allowed key (transaction object) range in the underlying hash-table of HT-OSTM. It can be seen that initially for lower number of threads (2 to 16) BTO is comparable to HT-OSTM. This can be attributed to the overheads in HT-OSTM. The logging or validation overhead exceeds the performance benefits of HT-OSTM for lower number of threads. Please note that the number of threads for Figure 5.10b are 64.
Chapter 6

Conclusion

In this dissertation we develop OSTM: an alternative theoretical model for building highly concurrent and composable data structures which are heart of any software application trying to leverage underlying multi-core architecture in presence of multiple threads. OSTM utilizes the software transactional memory approach to synchronize the access to underlying shared memory which is basically the concurrent data structure. We differ from the classic STM approach where the interface is mere read/write operations. The read/write operation are naive as they do not offer any other useful information apart from the fact that a write operation on a shared memory always conflicts with any concurrent read/write operation. On the other hand, we consider semantically rich higher level operations of the underlying shared data structure. These higher level primitives are exported to the programmers instead of mere read/writes. The enhanced semantics available through the OSTM interface provide better concurrency and performance as corroborated by the evaluation results.

We implement the proposed model using an address addressed hash table named HT-OSTM. Each bucket of the underlying concurrent hash-table is a lazyskip-list. The lazyskip-list is shared data structure which is augmented by the meta-information needed for ensuring consistency in concurrent executions. Thus, we do not use any separate data structure to store the information. This aids efficient memory usage and avoids access or maintenance over heads of maintaining meta-information. HT-OSTM exports STMbegin, STMinsert(), STMdellte(), STMlookup() and STMTryC(). The STMinsert(), STMdellte() and STMlookup() are the semantically rich higher level methods. Each transaction in HT-OSTM has methods executing in rv_method execution phase and upd_method execution phase. In rv_method phase the STMdellte(), STMlookup() (which return a value) and STMinsert() execute without modifying the underlying data structure. In upd_method phase the STMdellte() and STMinsert() methods execute for modifying the underlying hash-table inside STMTryC().

The STMlookup() is validated during the rv_method phase returning its fate at the point of its execution thus avoiding unnecessary work if the transaction is eventually supposed to abort because of this STMlookup() operation. The STMdellte() is validated twice once during rv_method phase (to avoid doing unnecessary execution of the transaction that is destined to abort) and next during upd_method to ensure consistency during concurrent executions.

In the STMTryC() we perform intraTransValidation() which aids in updating the underlying data structure without losing any update of the same transaction in case they happen to occur at same location. This help us in solving the problem where irrevocable updates may occur within a transaction which might abort. Also, the validation strategy employed help us to perform rollback-free commit.
We provide detailed proof of correctness for HT-OSTM where we establish the properties of the underlying data structure and prove that each of the method is linearizable, resulting in a legal sequential history. Further we show that such a legal sequential history is co-opaque by showing that the time order validation strategy ensures that HT-OSTM generates an history which would be equivalent to some serial history. please note that Peri et. al.[17] has shown that co-opacity is subset of opacity. Hence, HT-OSTM is co-opaque.

It can be seen easily that the OSTM model can be easily extended with underlying list, set or queue data structure. Implementing OSTM with tree as underlying data structure may need some extra effort.

HT-OSTM combines the scalable abstraction and ease of programming from STMs with our efficient mechanism of achieving composability using object level semantics. Our prototype implementation of HT-OSTM shows significant performance gain (as detailed in Chapter 5) over composable hash-table implementation of Synchrobench against ESTM and RWSTM with basic time-stamp ordering protocol.
Awards and Publications

Awards

1. Awarded fellowship under Charpak Research Internship Program 2017 by French government.

2. Awarded travel scholarship for HiPC 2017 to attend and present the poster at SRS of HiPC 2017.

Publications


References


