

**Design and Implementation of Efficient Techniques to
Achieve Compositionality using Object-based Software
Transactional Memory Systems**

Archit Somani

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



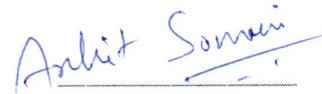
भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Department of Computer Science and Engineering

October 2019

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)

Archit Somani

(Name)

CS15RESCH01001

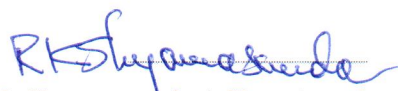
(Roll No.)

Approval Sheet

This Thesis entitled **Design and Implementation of Efficient Techniques to Achieve Compositionality using Object-based Software Transactional Memory Systems** by **Archit Somani** is approved for the degree of Doctor of Philosophy from IIT Hyderabad.



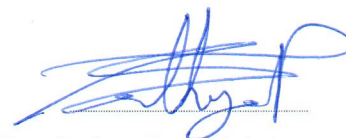
(Prof. Madhu Mutaym) Examiner 1
Dept. of CSE, IIT Madras



(Prof. R.K. Shyamasundar) Examiner 2
Dept. of CSE, IIT Bombay



(Dr. Manish Singh) Internal Examiner
Dept. of CSE, IITH



(Dr. Sathya Peri) Adviser
Dept. of CSE, IITH



(Prof. Raja Banerjee) Chairman
Dept. of Mechanical & Aerospace Engineering, IITH

Acknowledgements

Primarily, Lord Krishna, I bow at your feet and thank you for all the gifts and blessings you have given me in my life. I owe my deep gratitude to my advisor, Dr. Sathya Peri, for immense knowledge, patience, motivation, and research discussion throughout my Ph.D. His guidance helped me a lot in both research and career decision.

I would like to thank my doctoral committee members Dr. M.V. Panduranga Rao, Dr. Manish Singh, and Dr. Sumohana S. Channappayya for more than generous with their expertise and precious time. A special thanks to Prof. Sandeep Kulkarni from Michigan State University, USA for encouraging the research, carefully reading along with commenting, and re-writing on the revisions of our joint work. For the financial support and infrastructure of the lab establishment during my Ph.D., I heartily thank “Ministry of Electronics and Information Technology, Government of India” initiated “Visvesvaraya Ph.D. Scheme for Electronics and IT”.

I am thankful for and fortunate enough to get constant encouragement, support, and guidance from all teaching staff of Computer Science and Engineering of Indian Institute of Technology Hyderabad which helped me to complete my thesis successfully. Along with this, I would also like to extend my sincere esteem to all administrative staff and research groups for their timely support and suggestions. Especially, I would like to thank Ajay Singh, Sweta Kumari, Parwat Singh Anjana, Chirag Juyal, Sachin Rathor for the regular research discussions and memorable time. I am deeply grateful to surrounded by Mukesh Kumar Giluka, Sumanta Patro, Nagendra Kumar, and Sahil for their spiritual support. I am extremely privileged to take the opportunity to thank all my friends from IIT Hyderabad who made my stay pleasant and memorable.

Most importantly, I respect and thank my family members, especially my mother Sumitra Somani and father Mahesh Somani for their scarifies, patience, continuous love and support to me. They always encouraged me and being with me in all my pursuits. I am blessed with an elder brother Arpit Somani who always encouraged me mentally and financially. I also want to thank my younger sister Nikita Somani and Sister-in-law Shweta Maheshwari for their care and love. A special thanks to my wife Sweta Kumari for enthusiastic moral support and encouragement throughout the Ph.D. I am also grateful to everyone in my in-law’s family for their kindness and generosity. Hare Krishna!

Archit Somani

Dedication

I dedicate my dissertation to Lord Krishna, my parents, brother, sister, in-laws family, and my wife for their endless sacrifice, love, moral support, and encouragement.

Abstract

The rise of multi-core systems has necessitated the need for concurrent programming. However, developing correct, efficient concurrent programs is notoriously difficult. *Software Transactional Memory systems (STMs)* are a convenient programming interface for a programmer to access shared memory without worrying about concurrency issues such as priority-inversion, deadlock, livelock, etc. Another advantage of STMs is that they facilitate *compositionality* of concurrent programs with great ease. Different concurrent operations that need to be composed to form a single atomic unit is achieved by encapsulating them in a single transaction (a piece of code).

Most of the STMs proposed in the literature are based on read/write primitive operations on memory buffers. We represent them as *Read-Write STMs (RWSTMs)*. These read/write primitives result in unnecessary aborts. Instead, semantically rich higher-level methods such as hash table lookup, insert or delete, etc. aid in ignoring unimportant lower-level read/write conflicts and allow better concurrency. We call them *Object-based STMs (OSTMs)*.

In this thesis, we adapt the transaction tree model from databases to propose OSTM which enables efficient composition. We extend the traditional notion of conflicts and legality to higher-level methods using STMs and lay down detailed correctness proof to show that it is *conflict-opacity or co-opaque*. We implemented an efficient OSTM for two *Concurrent Data Structures (CDS)*, hash table and list as *HT-OSTM* and *list-OSTM* respectively. Both the OSTMs export the higher-level operations as transaction interface but it is generic to other data structures as well.

Experimental analysis of *HT-OSTM* outperforms state-of-the-art hash table based STMs (ESTM, RWSTM) by a factor of 3.8, 2.2 for lookup intensive workload (70% lookup, 10% insert, 20% delete) and by a factor of 6.7, 5.3 for update intensive workload (50% lookup, 25% insert, 25% delete) respectively. Similarly, *list-OSTM* outperforms state-of-the-art list based STMs, Trans-list, NOrec-list, and Boosting-list by a factor of 1.76, 1.89, 1.33 for lookup intensive workload and by a factor of 1.77, 1.77, 2.54 for update intensive workload respectively. Along with this, HT-OSTM and list-OSTM incurred negligible aborts as compared to state-of-the-art STMs considered in this thesis.

It has been shown in the literature of databases and RWSTMs that storing multiple versions corresponding to each key provides greater concurrency. So, to achieve greater concurrency than OSTM, we combine multiple versions with object semantics idea for harnessing greater concurrency in STMs. We propose the new and efficient notion of *Multi-Version Object-based STMs* or *MVOSTMs*. Specifically, we intro-

duce and implement *MVOSTM* for two efficient *CDS*, hash table and list object, represented as *HT-MVOSTM* and *list-MVOSTM* respectively but it is generic for other data structures as well. Initially, *HT-MVOSTM* and *list-MVOSTM* use an unbounded number of versions for each key that suffers from memory consumption. To address this issue, we developed two variants for both hash table and list data structures: (1) A *Garbage Collection (GC)* method in *MVOSTM* to delete the unwanted versions of a key, denoted as *MVOSTM-GC*. (2) *Finite version MVOSTM* (or *KOSTM*) which maintains at most K -versions corresponding to each key by replacing the oldest version when $(K + 1)^{th}$ version is created by the current transaction.

Experimental results show that hash table based *KOSTM* (*HT-KOSTM*) performs best among its variants (*HT-MVOSTM* and *HT-MVOSTM-GC*) and outperforms state-of-the-art hash table based STMs (*HT-OSTM* proposed by us, *ESTM*, *RWSTM*, *HT-MVTO*, *HT-KSTM*) by a factor of 3.5, 3.8, 3.1, 2.6, 1.8 for workload W1 (90% lookup, 5% insert, and 5% delete), by a factor of 1.4, 3, 4.85, 10.1, 7.8 for workload W2 (50% lookup, 25% insert, and 25% delete), and by a factor of 2, 4.25, 19, 69, 59 for workload W3 (10% lookup, 45% insert, and 45% delete) respectively.

Similarly, list based *KOSTM* (*list-KOSTM*) performs best among its variants (*list-MVOSTM* and *list-MVOSTM-GC*) and outperforms state-of-the-art list based STMs (*list-OSTM* proposed by us, *Trans-list*, *Boosting-list*, *NOrec-list*, *list-MVTO*, *list-KSTM*) by a factor of 2.2, 20, 22, 24, 12, 6 for workload W1, by a factor of 1.58, 20.9, 25.9, 29.4, 26.8, 19.68 for workload W2, and by a factor of 2, 35, 41, 47, 148, 112 for workload W3 respectively. We proved that *MVOSTMs* satisfy opacity and ensure that the transaction with lookup only methods does not abort if unbounded versions are used. To the best of our knowledge, this is the first work to explore the idea of using multiple versions in OSTMs to achieve greater concurrency.

To the *MVOSTMs* explained above, we performed a few more optimizations to harness greater concurrency further. We proposed the notion of *Optimized Multi-Version OSTMs (OPT-MVOSTMs)*. We propose the *OPT-MVOSTMs* for two efficient *CDS*, hash table and list objects as *OPT-HT-MVOSTM* and *OPT-list-MVOSTM* respectively but it is generic for other data structures as well. For memory utilization, we propose two variants of both the algorithms as *OPT-HT-MVOSTM-GC* (garbage collection on unwanted versions), *OPT-HT-KOSTM* (finite K -versions) and *OPT-list-MVOSTM-GC*, *OPT-list-KOSTM*.

Experimental analysis shows that *OPT-HT-KOSTM* performs best among its variants (*OPT-HT-MVOSTM* and *OPT-HT-MVOSTM-GC*) and outperforms all the state-of-the-art hash table based STMs (*HT-KOSTM* proposed by us, *HT-OSTM*

proposed by us, ESTM, RWSTM, HT-MVTO, HT-KSTM) by a factor of 1.05, 3.62, 3.95, 3.44, 2.75, 1.85 for workload W1, by a factor of 1.07, 1.44, 3.36, 5.45, 10.84, 8.42 for workload W2, and by a factor of 1.07, 2.11, 5.1, 19.8, 70.3, 60.23 for workload W3 respectively.

Similarly, *OPT-list-KOSTM* performs best among its variants (OPT-list-MVOSTM and OPT-list-MVOSTM-GC) and outperforms state-of-the-art list based STMs (list-KOSTM proposed by us, list-OSTM proposed by us, Trans-list, Boosting-list, NOrec-list, list-MVTO, list-KSTM) by a factor of 1.2, 2.56, 25.38, 23.57, 27.44, 13.1, 6.8 for W1, by a factor of 1.12, 2.11, 21.54, 26.27, 30.1, 27.89, 20.1 for W2, and by a factor of 1.11, 2.91, 36.1, 42.2, 48.89, 149.92, 114.89 for W3 respectively. We rigorously proved that *OPT-MVOSTMs* satisfy opacity.

In this thesis, we proposed multiple Object-based STM systems (OSTM, MVOSTM, OPT-MVOSTM). We proved that all the proposed OSTMs satisfy the popular correctness criteria as opacity. Experimental evaluation shows that all the proposed OSTMs achieved significant performance gain while reducing the number of aborts as compare to state-of-the-art STMs.

Contents

Declaration	ii
Approval Sheet	iii
Acknowledgements	iv
Abstract	vi
List of Abbreviations	xv
1 Introduction	1
1.1 Alternative to Locks: Software Transactional Memory systems (STMs)	4
1.2 Read-Write STMs	6
1.3 Motivation towards Object-based STMs	6
1.4 Advantages of Multi-Version STMs	9
1.4.1 Motivation towards Multi-Version Object-based STMs	10
1.4.2 Optimized Multi-Version Object-based STMs	12
1.5 Organization of the Thesis	15
2 System Model and Background	17
3 Object-based Software Transactional Memory systems	26
3.1 Introduction	26
3.2 A New Conflict Notion and Conflict-Opacity for OSTMs	29
3.3 OSTM Design and Data Structure	31
3.4 The Working of OSTM	34
3.5 Detailed Pseudocode of OSTM	37
3.5.1 Thread local Data Structure	37
3.5.2 Shared memory Data Structure	39
3.5.3 Pseudocode of OSTM	40
3.6 Correctness of OSTMs	60
3.6.1 Operational Level	60
3.6.2 Transactional Level	83

3.7	Experimental Evaluations	89
3.7.1	Pseudocode of Counter Application	90
3.7.2	Result Analysis	92
3.8	Summary	94
4	Multi-Version Object-based STMs	95
4.1	Introduction	95
4.2	Graph Characterization of Opacity	97
4.3	MVOSTM Design and Data Structure	101
4.4	The Working of MVOSTM	104
4.5	Correctness of MVOSTM	112
4.6	Experimental Evaluations	116
4.6.1	Result Analysis	117
4.7	Summary	123
5	Optimized-Multi-Version OSTMs	125
5.1	Introduction	125
5.2	OPT-MVOSTM Design and Data Structure	127
5.3	The Working of OPT-MVOSTM	129
5.4	Correctness of OPT-MVOSTM	136
5.5	Experimental Evaluations	136
5.5.1	Result Analysis	137
5.6	Summary	144
6	Conclusion and Future Work	146
6.1	Conclusion of the Thesis	147
6.2	Directions for Future Research	148
6.2.1	Nesting for Object-based STM systems	149
6.2.2	Object-based STM systems as an Application to Blockchain	150
6.2.3	Distributed Object-based STMs (DOSTMs)	151
6.3	Summary	151
	References	152

List of Figures

1.1	Motivational example for OSTMs	7
1.2	Not linearizable at lower-level as well as higher-level	8
1.3	Advantages of multi-version over single-version RWSTMs	10
1.4	Advantages of multi-version over single-version <i>OSTM</i>	11
2.1	T1 : A sample transaction on hash table (of Figure 1.1(i))	20
2.2	H2 : A non-sequential History.	21
2.3	A serial History	21
2.4	<i>STM_lookup()</i> returns the same value as previous method of the same transaction on same key	24
2.5	<i>STM_lookup()</i> returns the same value as previous closest conflicting method of committed transaction	24
2.6	<i>STM_lookup()</i> returns the same value as previous closest conflicting method of committed transaction	25
3.1	Graph Characterization of history <i>H5</i>	30
3.2	History H is not co-opaque	32
3.3	Co-opacity History H1	32
3.4	Searching k_8 over lazylist	33
3.5	Searching k_8 over lazyrb-list	33
3.6	Execution under lazyrb-list	33
3.7	Transaction lifecycle of <i>HT-OSTM</i>	34
3.8	Insert of k_5 in <i>STM_tryC()</i> . (i) BL & RL of k_5 is set to K_8 then BL of k_3 linked to K_5 & RL of k_4 is linked to k_5 . (ii) Only BL of k_5 is set to K_8 then BL of k_3 linked to K_5	36
3.9	Delete of k_5 in <i>STM_tryC()</i> . k_5 is unlinked from BL by linking BL of k_1 to ∞	37
3.10	Advantages of lookup validated once	43

3.11 Problem in execution without <code>intraTransValidation()</code> ($ins_1(k_5)$ and $ins_1(k_7)$). (i) lazyrb-list at state s . (ii) lazyrb-list at state s_1 . (iii) lazyrb-list at state s_2 (lost update problem).	47
3.12 k_{10} is not present in <i>BL</i> as well as <i>RL</i>	48
3.13 Adding k_{10} into <i>RL</i>	48
3.14 Execution of <code>list_ins()</code> : (i) key k_5 is present in <i>RL</i> and adding it into <i>BL</i> , (ii) key k_5 is not present in <i>RL</i> as well as <i>BL</i> and adding it into <i>RL</i> , (iii) key k_5 is not present in <i>RL</i> as well as <i>BL</i> and adding it into <i>RL</i> as well as <i>BL</i>	53
3.15 Non opaque history. Without timestamp validation	56
3.16 Opaque history H1. With timestamp validation	56
3.17 <code>intraTransValidation</code> for conflicting concurrent methods on key k_5	56
3.18 Advantage of validating <code>STM_delete()</code> once, if its returning FAIL in <code>rv_method execution</code> phase	60
3.19 Performance of <i>HT-OSTM</i> against State-of-the-art hash table based STMs	92
3.20 Abort Count of <i>HT-OSTM</i> against State-of-the-art hash table based STMs	93
3.21 Performance of list-OSTM against State-of-the-art list based STMs	93
3.22 Abort Count of list-OSTM against State-of-the-art list based STMs	94
4.1 History <i>H3</i> in time line view	97
4.2 $OPG(H3, \ll_{H3})$	99
4.3 <i>HT-MVOSTM</i> Design and Data Structure	102
4.4 Searching k_{12} over <i>lazy-list</i>	103
4.5 Searching k_{12} over <i>lazyrb-list</i>	103
4.6 <code>rv_Validation</code>	110
4.7 <code>tryC_Validation</code>	111
4.8 Intra transaction validation	112
4.9 Performance of <i>HT-KOSTM</i> and its variants (<i>HT-MVOSTM-GC</i> , <i>HT-MVOSTM</i>)	119
4.10 Performance of <i>list-KOSTM</i> and its variants (<i>list-MVOSTM-GC</i> , <i>list-MVOSTM</i>)	119
4.11 Aborts Count of <i>HT-KOSTM</i> and its variants	119
4.12 Aborts Count of <i>list-KOSTM</i> and its variants	120

4.13	Performance of <i>HT-KOSTM</i> against State-of-the-art hash table based STMs	120
4.14	Performance of <i>list-KOSTM</i> against State-of-the-art list based STMs	120
4.15	Aborts Count of <i>HT-KOSTM</i> against State-of-the-art hash table based STMs	121
4.16	Aborts Count of <i>list-KOSTM</i> against State-of-the-art list based STMs	121
4.17	Optimal Value of K	121
4.18	Memory Consumption	122
5.1	Optimized <i>HT-MVOSTM</i> design	128
5.2	Advantage of early validation in <code>STM_insert()</code>	132
5.3	Illustration of <code>tryC_Validation()</code>	135
5.4	Performance of <i>OPT-HT-KOSTM</i> and its variants (<i>OPT-HT-MVOSTM-GC</i> , <i>OPT-HT-MVOSTM</i>)	139
5.5	Abort Count of <i>OPT-HT-KOSTM</i> and its variants (<i>OPT-HT-MVOSTM-GC</i> , <i>OPT-HT-MVOSTM</i>)	139
5.6	Performance of <i>OPT-HT-KOSTM</i> against State-of-the-art hash table based STMs	140
5.7	Abort Count of <i>OPT-HT-KOSTM</i> against State-of-the-art hash table based STMs	140
5.8	Performance of <i>OPT-list-KOSTM</i> and its variants (<i>OPT-list-MVOSTM-GC</i> , <i>OPT-list-MVOSTM</i>)	140
5.9	Abort Count of <i>OPT-list-KOSTM</i> and its variants (<i>OPT-list-MVOSTM-GC</i> , <i>OPT-list-MVOSTM</i>)	141
5.10	Performance of <i>OPT-list-KOSTM</i> against State-of-the-art list based STMs	141
5.11	Abort Count of <i>OPT-list-KOSTM</i> against State-of-the-art list based STMs	141
5.12	Memory consumption among variants of <i>OPT-HT-KOSTMs</i> and <i>HT-KOSTMs</i> on hash table	142
5.13	Memory consumption among variants of <i>OPT-list-KOSTMs</i> and <i>list-KOSTMs</i> on list	142
5.14	Optimal Value of K for <i>OPT-HT-KOSTM</i> and <i>OPT-list-KOSTM</i> .	142
6.1	Challenges in Concurrent execution of SCTs	151

List of Tables

1.1	Difficulty in money transfer using multi-threading	2
1.2	Difficulty in money transfer using locks	3
3.1	Utility methods for each transaction to manipulate its log	39

List of Abbreviations

STMs	Software Transactional Memory systems
RWSTMs	Read-Write Software Transactional Memory systems
SV-RWSTMs	Single-Version Read-Write Software Transactional Memory systems
MV-RWSTMs	Multi-Version Read-Write Software Transactional Memory systems
OSTMs	Object-based Software Transactional Memory systems
SV-OSTMs	Single-Version Object-based Software Transactional Memory systems
HT-OSTMs	Hash Table based Object-based Software Transactional Memory systems
list-OSTMs	list based Object-based Software Transactional Memory systems
MVOSTMs	Multi-Version Object-based Software Transactional Memory systems
HT-MVOSTMs	Hash Table based Multi-Version Object-based Software Transactional Memory systems
list-MVOSTMs	list based Multi-Version Object-based Software Transactional Memory systems
KOSTMs	Finite K-Version Object-based Software Transactional Memory systems
HT-KOSTMs	Hash Table based Finite K-Version Object-based Software Transactional Memory systems
list-KOSTMs	list based Finite K-Version Object-based Software Transactional Memory systems
MVOSTM-GC	Garbage Collection in Multi-Version Object-based Software Transactional Memory systems
HT-MVOSTM-GC	Hash Table based Garbage Collection in Multi-Version Object-based Software Transactional Memory systems
list-MVOSTM-GC	list based Garbage Collection in Multi-Version Object-based Software Transactional Memory systems
OPT-MVOSTMs	Optimized Multi-Version Object-based Software Transactional Memory systems

HT-OPT-MVOSTMs	Hash Table based Optimized Multi-Version Object-based Software Transactional Memory systems
list-OPT-MVOSTMs	list based Optimized Multi-Version Object-based Software Transactional Memory systems
OPT-KOSTMs	Optimized Finite K-Version Object-based Software Transactional Memory systems
HT-OPT-KOSTMs	Hash Table based Optimized Finite K-Version Object-based Software Transactional Memory systems
list-OPT-KOSTMs	list based Optimized Finite K-Version Object-based Software Transactional Memory systems
OPT-MVOSTM-GC	Optimized Garbage Collection in Multi-Version Object-based Software Transactional Memory systems
HT-OPT-MVOSTM-GC	Hash Table based Optimized Garbage Collection in Multi-Version Object-based Software Transactional Memory systems
list-OPT-MVOSTM-GC	list based Optimized Garbage Collection in Multi-Version Object-based Software Transactional Memory systems
BTO	Basic Timestamp Ordering Protocol
MVTO	Multi-Version Timestamp Ordering Protocol
HT-MVTO	Hash Table based Multi-Version Timestamp Ordering Protocol
list-MVTO	list based Multi-Version Timestamp Ordering Protocol
KSTM	Finite K-version Timestamp Ordering Protocol
HT-KSTM	Hash Table based Finite K-version Timestamp Ordering Protocol
list-KSTM	list based Finite K-version Timestamp Ordering Protocol
ESTM	Elastic Software Transactional Memory system
HT	Hash Table
list	Linked-list
GC	Garbage Collection
CDS	Concurrent Data Structure
RL	Red Link
BL	Blue Link
lazyrb-list	Lazy Red-Blue List

Chapter 1

Introduction

A couple of decades ago, the performance of the single processor systems were relying on Moore's law. It says "The number of transistors on a single chip continues to double in approx two years". By following the Moore's law, the processing speeds of the systems has been improved consistently over the year. But around 2003-04, it was realized that single processor systems have reached their limits. Seymon Peyton Jones [1] said that, "The free lunch is over" i.e. we can not achieve better speed anymore by purchasing next-generation processor. To address this issue, the hardware manufacturers began to pack more processors, called as 'cores' per CPU.

Thus, nowadays multi-core systems have become ubiquitous. These systems as explained above, have more than one processors in a single chip. To harness the power of multi-core systems fully, the developed software should be parallel in nature. Therefore its a job of a software and operating system developer to develop efficient parallel software systems.

The solution to harness these multi-core system effectively is by multi-threaded parallel programming. But developing a correct multi-threaded program can be difficult due to synchronization issues. Here, more than one processors are on a single chip which connects using shared memory. Multiple threads may access the same shared memory location or shared variables simultaneously. This can possibly lead to synchronization issues which may cause incorrect output. Multi-threaded programming poses with some synchronization challenges as follows:

- Collaboration between threads normally involves sharing of data in memory or on secondary storage.
- Uncontrolled writes can lead to inconsistent data values called as *race condition*.
- Synchronized memory access is required since processors cannot modify multiple shared memory locations atomically.

- The granularity of access to shared memory, which is a deciding factor for the efficiency of the concurrent systems.

Consider Table 1.1 which demonstrates the issues with multi-threaded programming in money transfer. Here, we consider two threads Th_1 and Th_2 which execute concurrently. Thread Th_1 transfers money from account $A1$ to $A2$. While Th_2 computes the balance of all accounts in the bank. Initially, both the accounts $A1$ and $A2$ have a balance of \$500.

Now, suppose Th_1 wants to transfer \$500 from account $A1$ to $A2$. It achieves this by debiting an amount of \$500 from $A1$ and crediting it to $A2$. Now suppose Th_1 has debited the amount from $A1$ and is yet to credit to $A2$. At this point Th_2 comes in between and calculate the sum of both the accounts. So, Th_2 sees that \$500 missing in the bank. Hence, Th_2 sees a “Wrong sum”. This is due to concurrent multi-threaded execution. Thus to avoid these issues, threads need to collaborate and access common accounts in a synchronized manner.

Table 1.1: Difficulty in money transfer using multi-threading

Th_1	Time	Th_2
Read A1	1	
$A1 = A1 - 500$	2	
Write A1	3	
	4	Sum = 0
	5	Read A1
	6	Read A2
	7	Sum = Sum + A1
	8	Sum = Sum + A2
Read A2	9	
$A2 = A2 + 500$	10	
Write A2	11	
		Wrong Sum

Traditionally locks have been used to solve these synchronization issue between the threads. Locks ensure that a block of code which consists of multiple variables are executed as a critical section. It may be accessed by multiple threads but locks ensure that at a time only one thread will execute the critical section and access the variable. Locks solve the synchronization issues that arise with shared variable and multi-threading. If properly coded, locks ensure accessing the shared data items by

only one thread at a time. Any thread wants to access the shared variable will first have to acquire the corresponding lock and then access the variable. After the access is complete, the thread releases the lock. But, lock cause other problems. They are prone to livelocks, deadlocks, priority inversion, etc. If not coded properly, they can either bring down the performance of the multi-core systems or cause the program to be incorrect or both.

Table 1.2: Difficulty in money transfer using locks

Th_1	Time	Th_2
Lock A1	1	
Read A1	2	
$A1 = A1 - 500$	3	
Write A1	4	
Unlock A1	5	
	6	Sum = 0
	7	Lock A1, Lock A2
	8	Read A1
	9	Read A2
	10	Unlock A1, Unlock A2
	11	Sum = Sum + A1
	12	Sum = Sum + A2
Lock A2	13	
Read A2	14	
$A2 = A2 + 500$	15	
Write A2	16	
Unlock A2	17	
		Wrong Sum

Table 1.2 illustrates the difficulty with multi-threaded programming using locks in money transfer example. The scenario is the same as explained in Table 1.1. Here each thread acquires the lock before accessing any shared variable which in this case are accounts $A1$ and $A2$. After performing the respective operations such as read and write, thread releases the locks. Here, Th_1 wants to transfer the \$500 from $A1$ to $A2$ and Th_2 wants to view the sum of both the accounts. However, Th_1 acquired the lock on $A1$ before accessing it and debited the \$500 followed by releasing the lock. But, before crediting \$500 to $A2$ by Th_1 , suppose another thread Th_2 acquires the lock on $A1$ and $A2$. Following this, Th_2 reads the values of $A1$, $A2$ to compute the

sum of both the account. Unfortunately, this still results with “Wrong sum”. So, we can observe that incorrect locking leads to an inconsistent state. For other real-world software systems these kind of error may cause crashes or other undesirable outputs.

To handle these inconsistencies with programming, a popular locking mechanism identified in database literature is two-phase locking (2PL) [2,3]. As the name suggests it is having two phases. First is the locking phase where thread acquires the locks on all the shared variables and works on it. In the second phase, the thread releases the locks on all the shared variables after working on it. Once thread will release the lock on any shared variables, it will never acquire the lock again. This property of 2PL makes the transaction to be atomic. But improper use of two-phase locking also leads system into deadlock. Consider an example where Th_1 acquired the lock on account $A1$ and Th_2 acquired the lock on $A2$. After that, both the threads are waiting on each other to acquire the lock on next shared variable. Here, Th_1 and Th_2 is waiting for account $A2$ and $A1$ respectively. This situation leads system into deadlock.

To overcome these difficulties with parallel programming, researchers have developed the paradigm of *Software Transactional Memory systems (STMs)* which helps programmers to develop the correct concurrent programs without compromising on the efficiency of the multi-core systems which is explained in Section 1.1.

1.1 Alternative to Locks: Software Transactional Memory systems (STMs)

An alternative to locks is Software Transaction Memory Systems (STMs) [4] which exploit the cores of multi-core systems efficiently. STMs are a convenient programming interface for a programmer to access shared memory without worrying about concurrency issues such as priority-inversion, deadlock, livelock, etc. It is a promising research alternative which has gained a lot of interest by academia and industry. Another advantage of STMs is that they facilitate compositionality of concurrent programs with great ease. Different concurrent operations that need to be composed to form a single atomic unit is achieved by encapsulating them in a single transaction.

The transaction of STMs is a sequence of instructions/code which executed in the memory. This transaction is the same as the transaction defined in a database that executes the sequence of instructions on the shared variables and updates it. Along with this, a transaction of databases satisfies the ACID (Atomicity, Consistency, Isolation, and Durability) property and correctness criteria as serializability [5]. The transaction of STMs also satisfies all these properties except durability because STM

system are used to develop concurrent systems in the main memory.

The STM system is a new parallel programming paradigm in which transaction uses modern language constructs like *atomic* to get rid of synchronization issues. The idea of transactions can be implemented in hardware, software, and hybrid (combination of hardware and software) systems. The execution of money transfer using STMs is shown below:

```
Th1()
{
    initialization ();
    atomic
    {
        Read A1
        A1 = A1 - 500
        Write A1
        Read A2
        A2 = A2 + 500
        Write A2
    }
}
```

The consistent programming for money transfer is continued as follows:-

```
Th2()
{
    initialization ();
    atomic
    {
        Read A1
        Read A2
        Sum = Sum + A1
        Sum = Sum + A2
    }
}
```

In the past few years, several STMs have been proposed which address the synchro-

nization issues and provide greater concurrency. STMs hide the synchronization and communication difficulties among the multiple threads from the programmer while ensuring correctness and hence making programming easy. Another important feature that STMs facilitate is compositionality of concurrent programs with great ease. It composes different concurrent operations in a single atomic unit by encapsulating them in a transaction.

The atomic property of transactions helps to correctly compose together several different individual operations. For example, consider a real-world application on hash table that needs to perform the *move* operation. Move operation consists of delete and inserts method of the hash table. The implementation of move operation requires that a delete followed by insert method from the same or different hash table object appear to happen together. STM system ensures the compositionality of move operation by combining both delete and insert in a single transaction.

1.2 Read-Write STMs

Most of the STMs proposed in the literature (such as NOrec [6], ESTM [7]) are based on read/write operations on *transaction objects* (*t-objects*) or *keys* on shared memory. We denote them as *Read-Write STMs* or *RWSTMs*. These STMs typically export following methods: (1) *STM_begin()*: begins a transaction with a unique id, (2) *STM_read(k)* (or *r(k)*): reads the value of key *k* from shared memory, (3) *STM_write(k, v)* (or *w(k, v)*): writes the value of key *k* as *v* in its local log. This thesis considers the optimistic execution of STMs in which transactions are writing into its local log until the successful validation. (4) *STM_tryC()* (or *tryC()*): validates and tries to commit the transaction by writing values to the shared memory. If validation is successful, then it returns commit. Otherwise, it returns abort.

1.3 Motivation towards Object-based STMs

It has shown in databases that object-level systems provide greater concurrency than read/write systems [3, Chap 6]. They include more semantically rich operations such as enqueue/dequeue on queue objects, push/pop on stack objects and insert/lookup/delete on sets, trees or hash table objects depending upon the underlying data structure used to implement *Object-based STMs* (*OSTMs*). Along the same lines, we proposed a model to achieve composability with greater concurrency for *STMs* by considering higher-level objects which harness the richer semantics of object-level methods. We motivate this with an interesting example.

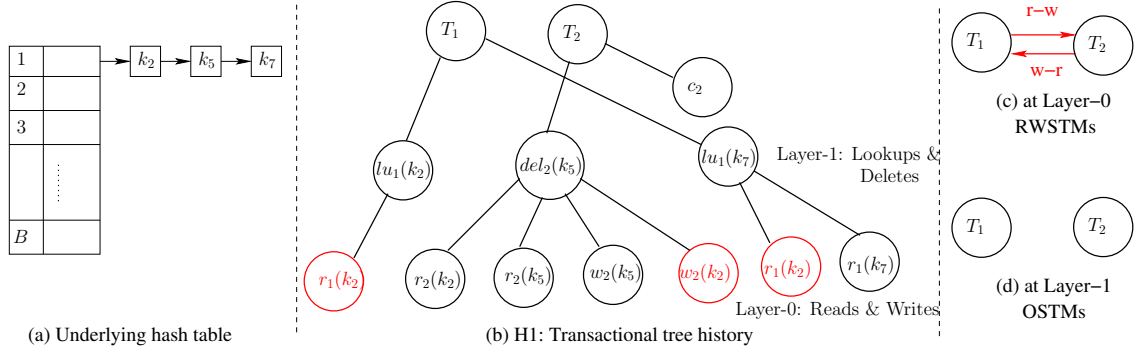


Figure 1.1: Motivational example for OSTMs

Object-based STMs: Consider an *OSTMs* operating on the hash table object called as *Hash table Object STM* or *HT-OSTM* which exports the following methods - (1) *STM_begin()*: begins a transaction with unique id (same as in *RWSTM*), (2) *STM_insert(k, v)* (or *ins(k, v)*): inserts a value v for key k in its local log, (3) *STM_delete(k)* (or *del(k)*): deletes the value associated with the key k , (4) *STM_lookup(k)* (or *lu(k)*): looks up the value associated with the key k from shared memory and, (5) *STM_tryC()* (or *tryC()*): validates and tries to commit the transaction by updating values to the shared memory. If validation is successful, then it returns commit. Otherwise, it returns abort.

A simple way to implement the concurrent *OSTM* is using a list (a single bucket) where each element of the list stores the $\langle \text{key}, \text{value} \rangle$ pair. The elements of the list are sorted by their keys similar to the set implementations discussed in [8, Chap 9]. It can be seen that the underlying list is a concurrent data structure manipulated by multiple transactions. So, we may use the lazy-list based concurrent set [9] to implement the operations of the list denoted as: *list_insert*, *list_del* and *list_lookup*. Thus, when a transaction invokes *STM_insert()*, *STM_delete()* and *STM_lookup()* methods, the STM internally invokes the *list_insert*, *list_del* and *list_lookup* methods respectively.

Benefit of *OSTMs* over *RWSTMs*: Consider an instance of list in which the nodes with keys $\langle k_2, k_5, k_7 \rangle$ are present in the hash table as shown in Figure 1.1(a) and transactions T_1 and T_2 are concurrently executing *STM_lookup*₁(k_2), *STM_delete*₂(k_5), and *STM_lookup*₁(k_7) as shown in Figure 1.1(b). In this setting, suppose a transaction T_1 of *OSTM* invokes methods *STM_lookup()* on the keys k_2, k_7 . This would internally cause the *OSTM* to invoke *list_lookup* method on keys $\langle k_2 \rangle$ and $\langle k_2, k_5, k_7 \rangle$ respectively.

Concurrently, suppose transaction T_2 invokes the method *STM_delete()* on key k_5 between the two *STM_lookup()* of T_1 . This would cause, *OSTM* to invoke *list_del* method of list on k_5 . Since, we are using lazy-list approach on the underlying

list, $list_del$ marks element k_5 for logical deletion (same as lazy deletion [9]) and points the next field of element k_2 to k_7 for physical deletion of the node k_5 . Thus $list_del$ of k_5 would execute the following sequence of read/write level operations- $r(k_2)r(k_5)w(k_5)w(k_2)$ where $r(k_5), w(k_5)$ denote read and write on the element k_5 with some value respectively. The execution of $OSTM$ denoted as a *history* can be represented as a transactional forest as shown in Figure 1.1(b). Here the execution of each transaction is a tree.

In this execution, we denote the read/write operations (leaves) as layer-0 and $STM_lookup()$, $STM_delete()$ methods as layer-1. Consider the history (execution) at layer-0 (while ignoring higher-level operations), denoted as $H0$. It can be verified that this history is not opaque [10]. This is because between the two reads of k_2 by T_1, T_2 writes to k_2 . It can be seen that if history $H0$ is input to a $RWSTM$ one of the transactions among T_1 and T_2 would be aborted to ensure correctness (in this case opacity [10]). Figure 1.1(c) shows the presence of a cycle in the conflict graph of $H0$. On the other hand consider the history $H1$ at layer-1 consisting of $STM_lookup()$, $STM_delete()$ methods while ignoring the underlying read/write operations. We ignore the underlying read and write operations since they do not overlap (referred to as pruning in [3, 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 [10] with equivalent serial history T_1T_2 (or T_2T_1) and the corresponding conflict graph shown in Figure 1.1(d).

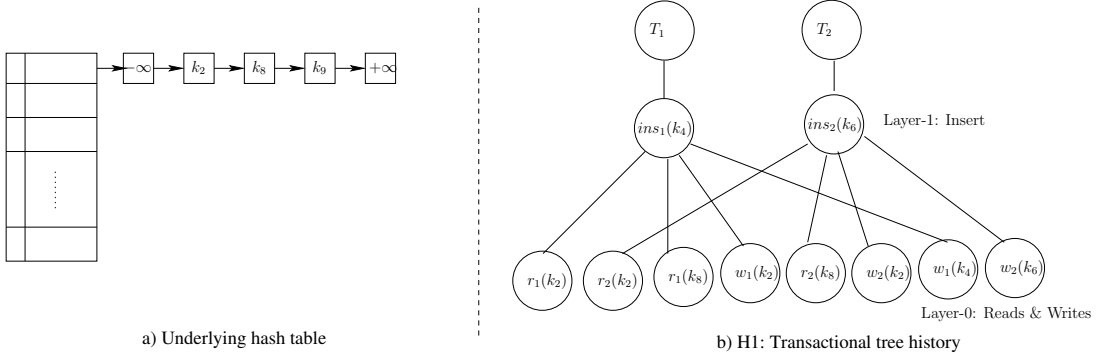


Figure 1.2: Not linearizable at lower-level as well as higher-level

The important idea in the above argument is that some conflicts at lower-level operations do not matter at higher level operations. Thus, such lower level conflicting operations may be ignored. The conflicts at lower-level do not matter at higher-level referred as *benign-conflicts* [11]. With object level modeling of histories, we get a higher number of acceptable schedules than read/write model. The history, $H1$ in Figure 1.1(b) clearly shows the advantage of considering STMs with higher level

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

While some lower-level conflict does matter at higher-level as shown in Figure 1.2. Here, we cannot ignore the underlying read and write conflicts at higher-level since they are overlapping to each other and unable to make it isolate/atomic. So, we cannot re-ordered in any way and prune this from lower-level to higher-level (referred to as pruning in [3, Chap 6]). Thus, to accept this history at higher-level one of the transaction have to return abort same as lower-level.

So, we can conclude that some conflicts at lower-level do not matter at higher-level which motivates us to work on Object-based STMs to achieves the greater concurrency than RWSTMs..

Experimental analysis of *HT-OSTM* outperforms state-of-the-art hash table based STMs (ESTM [7], RWSTM [3]) by a factor of 3.8, 2.2 for lookup intensive workload (70% lookup, 10% insert, 20% delete) and by a factor of 6.7, 5.3 for update intensive workload (50% lookup, 25% insert, 25% delete) respectively. Similarly, *list-OSTM* outperforms state-of-the-art list based STMs (Trans-list [12], NOrec-list [6] and Boosting-list [13]) by a factor of 1.76, 1.89, 1.33 for lookup intensive workload and by a factor of 1.77, 1.77, 2.54 for update intensive workload respectively. Along with this, HT-OSTM and list-OSTM incurred negligible aborts as compared to state-of-the-art STMs as shown in Section 3.7 of Chapter 3.

1.4 Advantages of Multi-Version STMs

It has been shown in the literature of databases and RWSTMs [14, 15] that greater concurrency can be obtained by storing multiple versions for each *transactional-object (t-object) or key*. Such RWSTMs are known as *Multi-Version RWSTMs (MV-RWSTMs)*. MV-RWSTM system reduces the number of aborts and increases the throughput as compared to *Single-Version RWSTMs (SV-RWSTMs or RWSTMs)* which maintains only one version for each key. We demonstrate this with another interesting example.

Figure 1.3 (a) illustrates the concurrent execution of two transactions T_1 and T_2 under SV-RWSTMs. Here, T_1 reads the t-object x from shared memory and returns the value as 0. We have assumed that all the t-objects are initialized with value 0. After reading x by T_1 , transaction T_2 wrote into two t-objects x and y with the value 10 and committed successfully while returning C_2 . Now, T_1 wants to read y and returns abort because T_1 is reading the older value of x whereas a newer value of y , so T_1 can not be atomic. This is reflected by a cycle in the corresponding conflict graph between T_1 and T_2 , as shown in Figure 1.3 (c). Hence, for the execution to be

correct, SV-RWSTMs will abort T_1 .

Figure 1.3 (b) demonstrates the execution under MV-RWSTMs. It has the same scenario as Figure 1.3 (a) but maintains multiple versions corresponding to each key. It shows the concurrent execution of two transactions T_1 and T_2 which access (read and write) two t-objects x and y while maintaining multiple versions corresponding to x and y . T_1 read x and returns the value as 0. After that T_2 writes to x and y with value 10 and returns commit as C_2 . Then T_1 read y and returns the value as 0 and finally, T_1 returns commit as C_1 . So, we can observe that if the MV-RWSTM system maintains multiple versions corresponding to y here as 0 as well as 10 then T_1 can return commits while reading the older value of y as 0 instead of newer value 10. In that sense, T_1 can be atomic and returns commit. Hence, while maintaining multiple versions both the transactions can return commit with equivalent serial schedule T_1T_2 . The corresponding conflict graph is shown in Figure 1.3 (d) does not have a cycle. So, we can conclude that maintaining multiple versions corresponding to each key reduces the number of aborts and improves concurrency.

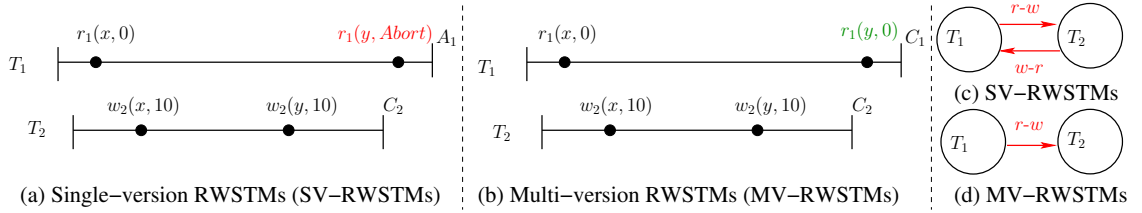


Figure 1.3: Advantages of multi-version over single-version RWSTMs

1.4.1 Motivation towards Multi-Version Object-based STMs

To further achieve greater concurrency with OSTMs, we have been motivated by the concept of multiple version explored in databases and RWSTMs. By storing multiple versions for each t-object (or key), greater concurrency can be obtained [15] as illustrated above. Specifically, maintaining multiple versions can ensure that more read operations succeed because the reading operation will have an appropriate version to read. So, in this subsection we explain how by using multiple version in OSTMs, denoted as *Multi-Version OSTMs (MVOSTMs)*, we have been able to identify a novel and efficient way to achieve greater concurrency on OSTMs. In Chapter 4, we show the benefit of MVOSTMs over single and multi-version RWSTMs as well as single-version OSTMs.

The potential benefit of *MVOSTMs* over *OSTMs* and multi-version *RWSTMs*: We now illustrate the advantage of *MVOSTMs* as compared to *Single-Version OSTMs (SV-OSTMs or OSTMs)* using hash table object having the same

operations as discussed above for OSTM: *ins*, *lu*, *del*. Figure 1.4 (a) represents a history H with two concurrent transactions T_1 and T_2 operating on a hash table ht . T_1 first tries to perform a *lu* on key k_2 . But due to the absence of key k_2 in ht , it obtains a value of *null*. Then T_2 invokes *ins* method on the same key k_2 and inserts the value v_2 in ht . Then T_2 deletes the key k_1 from ht and returns v_0 implying that some other transaction had previously inserted v_0 into k_1 . The second method of T_1 is *lu* on the key k_1 . With this execution, any *SV-OSTM* system has to return abort for T_1 's *lu* operation to ensure correctness, i.e., opacity. Otherwise, if T_1 would have obtained a return value *null* for k_1 , then the history would not be opaque anymore. This is reflected by a cycle in the corresponding conflict graph between T_1 and T_2 , as shown in Figure 1.4 (c). Thus to ensure opacity, *SV-OSTM* system has to return abort for T_1 's lookup on k_1 .

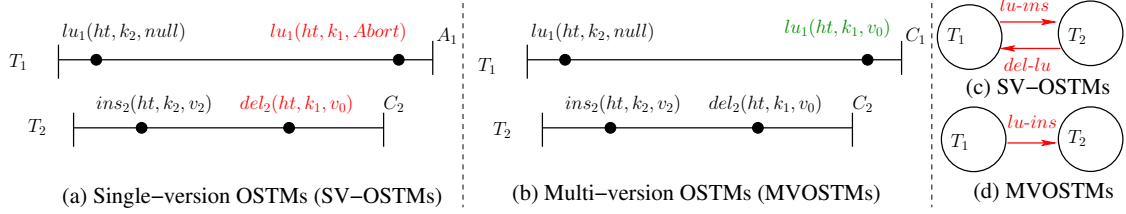


Figure 1.4: Advantages of multi-version over single-version *OSTM*

In an *MVOSTM* based on hash table, denoted as *HT-MVOSTM*, whenever a transaction inserts or deletes a key k , a new version is created. Consider the above example with a *HT-MVOSTM*, as shown in Figure 1.4 (b). Even after T_2 deletes k_1 , the previous value of v_0 is still retained. Thus, when T_1 invokes *lu* on k_1 after the delete on k_1 by T_2 , *HT-MVOSTM* returns v_0 (as previous value). With this, the resulting history is opaque with equivalent serial history being T_1T_2 . The corresponding conflict graph in Figure 1.4 (d) does not have a cycle.

Thus, *MVOSTM* reduces the number of aborts and achieves greater concurrency than *SV-OSTMs* while ensuring the compositionality. We believe that the benefit of *MVOSTM* over multi-version *RWSTM* is similar to *SV-OSTM* over single-version *RWSTM* as explained in Section 1.3.

MVOSTM is a generic concept which can be applied to any data structure. In this subsection of the thesis, we have considered two efficient *Concurrent Data Structures or CDS*, hash table and list based *MVOSTMs* as *HT-MVOSTM* and *list-MVOSTM* respectively. In the initial version of the algorithm developed, *HT-MVOSTM* and *list-MVOSTM* use an unbounded number of versions for each key. To address this issue, we developed two variants for both hash table and list data structures: (1) A *Garbage Collection (GC)* method in *MVOSTM* to delete the unwanted versions of a

key, denoted as *MVOSTM-GC*. Garbage collection gave a performance gain of 15% over *MVOSTM* without garbage collection in the best case. Thus, the overhead of garbage collection is less than the performance improvement due to improved memory usage. (2) Placing a limit of K on the number of versions in *MVOSTM*, resulting in *KOSTM*. This gave a performance gain of 21% over *MVOSTM* without garbage collection in the best case.

Experimental results show that hash table based *KOSTM* (HT-*KOSTM*) performs best among its variants (HT-*MVOSTM* and HT-*MVOSTM-GC*) and outperforms state-of-the-art hash table based STMs (HT-*OSTM* [16] proposed by us in Chapter 3, *ESTM* [7], *RWSTM* [3], HT-*MVTO* [15], HT-*KSTM* [15]) by a factor of 3.5, 3.8, 3.1, 2.6, 1.8 for workload W1 (90% lookup, 5% insert, and 5% delete), by a factor of 1.4, 3, 4.85, 10.1, 7.8 for workload W2 (50% lookup, 25% insert, and 25% delete), and by a factor of 2, 4.25, 19, 69, 59 for workload W3 (10% lookup, 45% insert, and 45% delete) respectively.

Similarly, list based *KOSTM* (list-*KOSTM*) performs best among its variants (list-*MVOSTM* and list-*MVOSTM-GC*) and outperforms state-of-the-art list based STMs (list-*OSTM* [16] proposed by us in Chapter 3, *Trans-list* [12], *Boosting-list* [13], *NOrec-list* [6], list-*MVTO* [15], list-*KSTM* [15]) by a factor of 2.2, 20, 22, 24, 12, 6 for workload W1, by a factor of 1.58, 20.9, 25.9, 29.4, 26.8, 19.68 for workload W2, and by a factor of 2, 35, 41, 47, 148, 112 for workload W3 respectively. We proved that *MVOSTMs* satisfy opacity [10] and ensure that the transaction with lookup only methods does not abort if unbounded versions are used. To the best of our knowledge, this is the first work to explore the idea of using multiple versions in *OSTMs* to achieve greater concurrency.

1.4.2 Optimized Multi-Version Object-based STMs

We made a few modifications to optimize the Multi-Version *OSTMs* (explained in SubSection 1.4.1) and propose the new notion of *Optimized Multi-Version OSTMs* (*OPT-MVOSTMs*) to harness the greater concurrency further while achieving the compositionality. In this subsection of the thesis, we have developed *OPT-MVOSTMs* for two efficient CDS, hash table and list objects as *OPT-HT-MVOSTM* and *OPT-list-MVOSTM* respectively. The detailed descriptions are explained in Chapter 5 of this thesis. *OPT-MVOSTM* is generic for other data structures as well. For efficient space utilization in *OPT-MVOSTMs* with unbounded versions, we developed *Garbage Collection (GC)* for *OPT-MVOSTM* (i.e. *OPT-MVOSTM-GC*) and bounded K -version *OPT-MVOSTM* (i.e. *OPT-KOSTM*) for both hash table and list data structures.

Experimental analysis shows that *OPT-HT-KOSTM* performs best among its variants (OPT-HT-MVOSTM and OPT-HT-MVOSTM-GC) and outperforms all the state-of-the-art hash table based STMs (HT-KOSTM [17] proposed by us in Chapter 4, HT-OSTM [16] proposed by us in Chapter 3, ESTM [7], RWSTM [3], HT-MVTO [15], HT-KSTM [15]) by a factor of 1.05, 3.62, 3.95, 3.44, 2.75, 1.85 for workload W1, by a factor of 1.07, 1.44, 3.36, 5.45, 10.84, 8.42 for workload W2, and by a factor of 1.07, 2.11, 5.1, 19.8, 70.3, 60.23 for workload W3 respectively.

Similarly, *OPT-list-KOSTM* performs best among its variants (OPT-list-MVOSTM and OPT-list-MVOSTM-GC) and outperforms state-of-the-art list based STMs (list-KOSTM [17] proposed by us in Chapter 4, list-OSTM [16] proposed by us in Chapter 3, Trans-list [12], Boosting-list [13], NOrec-list [6], list-MVTO [15], list-KSTM [15]) by a factor of 1.2, 2.56, 25.38, 23.57, 27.44, 13.1, 6.8 for W1, by a factor of 1.12, 2.11, 21.54, 26.27, 30.1, 27.89, 20.1 for W2, and by a factor of 1.11, 2.91, 36.1, 42.2, 48.89, 149.92, 114.89 for W3 respectively. We rigorously proved that *OPT-MVOSTMs* satisfy opacity [10].

Contributions of the Thesis:

- We proposed an efficient *Object-based STM (OSTM)* system for two *Concurrent Data Structures (CDS)*, hash table and list as *HT-OSTM* and *list-OSTM* respectively in Chapter 3 but it is generic to other data structures as well.
 - We proposed a generic framework for composing higher-level objects based on the notion of conflicts for objects in databases [3, Chap 6].
 - We developed a subclass of opacity [10] as conflict opacity or co-opacity for higher-level objects in Section 3.2.
 - We rigorously proved that proposed OSTMs satisfy the correctness criteria as co-opacity [18] in Section 3.6.
 - A simple modification of *HT-OSTM* gives us a concurrent list based OSTM or *list-OSTM*. Our experiments demonstrate that both HT-OSTM and list-OSTM provide greater concurrency and reduces the number of aborts as compared to state-of-the-art STMs in Section 3.7.
- To achieve the greater concurrency further, we proposed a novel and efficient technique as *Multi-Version Object-based STM system, MVOSTM* in Chapter 4.
 - Specifically, we developed it for two *Concurrent Data Structures (CDS)*,

- hash table and list objects as *HT-MVOSTM* and *list-MVOSTM* respectively but it is generic for other data structures as well in Section 4.3.
- We proved that *HT-MVOSTM* and *list-MVOSTM* satisfy *opacity* [10], standard correctness-criterion for STMs in Section 4.5.
 - For efficient space utilization in *MVOSTM* with unbounded versions, we developed *Garbage Collection (GC)* for *MVOSTM* (i.e. *MVOSTM-GC*) and bounded version *MVOSTM* (i.e. *KOSTM*) for both hash table and list data structures.
 - Our experiments demonstrate that both hash table based *KOSTM* (*HT-KOSTM*) and list based *KOSTM* (*list-KOSTM*) achieve greater performance and reduce the number of aborts as compared to single-version OSTMs, single and multi-version *RWSTMs* in Section 4.6. To the best of our knowledge, this is the first work to explore the idea of using multiple versions in *OSTMs* to achieve greater concurrency.
- We performed a few optimizations on the *MVOSTM* and proposed a new notion of *Optimized Multi-Version Object-based STM system* as *OPT-MVOSTM* to harness the greater concurrency further in Chapter 5.
 - We developed *OPT-MVOSTM* for two CDS, hash table and list objects called as *OPT-HT-MVOSTM* and *OPT-list-MVOSTM* respectively in Section 5.2. *OPT-MVOSTM* is generic for other data structures as well.
 - We proved that *OPT-HT-MVOSTM* and *OPT-list-MVOSTM* satisfy standard correctness-criterion of STMs, *opacity* [10] in Section 5.4.
 - For efficient space utilization in *OPT-MVOSTM* with unbounded versions, we developed *Garbage Collection (GC)* for *OPT-MVOSTM* (i.e. *OPT-MVOSTM-GC*) and bounded version *OPT-MVOSTM* (i.e. *OPT-KOSTM*) for both hash table and list data structures.
 - We did the experimental analysis of both hash table based *OPT-KOSTM* (*OPT-HT-KOSTM*) and list based *OPT-KOSTM* (*OPT-list-KOSTM*) with state-of-the-art STMs. Proposed *OPT-HT-KOSTM* and *OPT-list-KOSTM* provide greater concurrency and reduces the number of aborts as compared to single and multi-version OSTMs, single and multi-version *RWSTMs* while maintaining multiple versions corresponding to each key in Section 5.5.

1.5 Organization of the Thesis

In this thesis, we proposed multiple Object-based STM systems (OSTM, MVOSTM, OPT-MVOSTM). We proved that all the proposed OSTMs satisfy opacity as popular correctness criteria of STMs. Experimental evaluation shows that all the proposed OSTM systems achieved significant performance gain while reducing the number of aborts as compare to state-of-the-art STMs. The organization of the thesis is as follow:

Chapter 2 explains the system model and the background of the thesis. It includes assumptions about the processors/threads followed by definitions of Events, Global States, Methods, Histories, Sequential Histories, Transactions, Real-time Order & Serial Histories, Valid & Legal Histories, Linearizability, and Opacity as a popular correctness criterion of STMs.

Chapter 3 is organized as follows. Section 3.1 illustrates the motivation towards OSTM over RWSTMs. In Section 3.2, we build a new notion of legality, conflicts to describe opacity, co-opacity and the graph characterization. Section 3.3 represents the *OSTMs* design and data structure. Section 3.4 shows the working of *OSTMs* and its algorithms. We explain the detailed pseudocode of OSTM in Section 3.5. We formally prove the correctness of *OSTMs* in Section 3.6. In Section 3.7 we show the experimental evaluation of *OSTMs* with state-of-art-STMs. Finally, we summaries this chapter in Section 3.8.

Chapter 4 is organized as follows. Section 4.1 illustrates the advantage of MVOSTM over single-version OSTM, single and multi-version RWSTM. Section 4.2 shows the Graph Characterization of Opacity. Section 4.3 represents the *MVOSTMs* design and data structure. Section 4.4 shows the working of *MVOSTMs* and its algorithms. We formally prove the correctness of *MVOSTMs* in Section 4.5. In Section 4.6 we show the experimental evaluation of *MVOSTMs* with state-of-art-STMs. Finally, we summaries this chapter in Section 4.7.

Chapter 5 is organized as follows. Section 5.1 shows the optimization on MVOSTM to harness the greater concurrency than single and multi-version OSTM, single and multi-version RWSTM. Section 5.2 represents the *OPT-MVOSTMs* design and data structure. Section 5.3 shows the working of *OPT-HT-MVOSTMs* and its algorithms. We formally prove the correctness of *OPT-MVOSTMs* in Section 5.4. In Section 5.5 we show the experimental evaluation of *OPT-MVOSTMs* with state-of-art-STMs. Finally, we summaries this chapter in Section 5.6.

Chapter 6 describes the contributions of this thesis followed by direction for future research. This thesis mainly proposed three Object-based STM systems (OSTMs, MVOSTMs, OPT-MVOSTMs) and proved their correctness followed by the perfor-

mance evaluation with state-of-the-art STMs. The direction for future research of this thesis can be (1) Nesting for Object-based STM systems, (2) Object-based STM systems as an Application to Blockchain, and (3) Distributed Object-based STMs (DOSTMs).

Chapter 2

System Model and Background

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

Events: We assume that the threads execute atomic *events* and the events by different threads are (1) read/write on shared/local memory objects, (2) method invocations (or *inv*) event & responses (or *rsp*) event on higher-level shared-memory objects.

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

Methods: The n processes access a collection of *t-objects* via atomic *transactions*. Each transaction has a unique identifier typically denoted as T_i . Within a transaction, a process can invoke transactional methods on a *hash table* t-object. A hash table(ht) consists of multiple key-value pairs of the form $\langle k, v \rangle$. The keys and values are respectively from sets \mathcal{K} and \mathcal{V} . The methods that a transaction T_i can invoke are: (1) $STM_insert_i(ht, k, v)$ or $ins_i(ht, k, v)$: this method inserts the pair $\langle k, v \rangle$ into object ht and return *ok*. If ht already has a pair $\langle k, v' \rangle$ then v' gets replaced with v . (2) $STM_delete_i(ht, k, v)$ or $del_i(ht, k, v)$: if ht has a $\langle k, v \rangle$ pair then this operation deletes the pair and returns v . If no such $\langle k, v \rangle$ pair is present in ht , then the operation

returns *nil*. (3) $STM_lookup_i(ht, k, v)$ or $lu_i(ht, k, v)$: if ht has a $\langle k, v \rangle$ pair then this operation returns v . If no such $\langle k, v \rangle$ pair is present in ht , then the method returns *nil*. It can be seen that the return value of $STM_lookup()$ is similar to $STM_delete()$.

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

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

The *Object-based STM systems* (*OSTM*, *MVOSTM*, and *OPT-MVOSTM*) support two other methods: (4) $STM_tryC_i()$: this method tries to validate all the operations of the T_i . All proposed OSTM systems return *ok* if T_i is successfully committed. Otherwise, it returns \mathcal{A} implying abort. This method is invoked by a process after completing all its transactional operations. (5) $STM_tryA_i()$: this method returns \mathcal{A} and OSTM systems abort T_i .

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

Having described about methods of a transaction, we describe about the events invoked by these methods. We assume that each method consists of a *inv* and *rsp* event. Specifically, the *inv* & *rsp* events of the methods of a transaction T_i are: (1) $STM_insert_i(ht, k, v)$: $inv(STM_insert_i(ht, k, v))$ and $rsp(STM_insert_i(ht, k, v, ok/\mathcal{A}))$. (2) $STM_delete_i(ht, k, v)$: $inv(STM_delete_i(ht, k, v))$ and $rsp(STM_delete_i(h, k, v/nil/\mathcal{A}))$. (3) $STM_lookup_i(h, k, v)$: $inv(STM_lookup_i(h, k, v))$ and $rsp(STM_lookup_i(h, k, v/nil/\mathcal{A}))$. (4) $STM_tryC_i()$: $inv(STM_tryC_i())$ and $rsp(STM_tryC_i(ok/\mathcal{A}))$. (5) $STM_tryA_i()$: $inv(STM_tryA_i())$ and $rsp(STM_tryA_i(\mathcal{A}))$.

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

Formally, we denote a method m by the tuple $\langle evts(m), <_m \rangle$. Here, $evts(m)$ are all the events invoked by m and the $<_m$ a total order among these events. For instance,

the method $lu_{11}(k_5)$ of Figure 2.1 is represented as: $inv(lu_{11}(h, k_5)) r_{111}(k_2, o_2) r_{112}(k_5, o_5) rsp(lu_{11}(h, k_5, o_5))$. In our representation, we abbreviate $STM_insert()$ as ins , $STM_delete()$ as del and $STM_lookup()$ as lu . From our assumption, we get that for any read/write primitive rw of m , $inv(m) <_m rw <_m rsp(m)$.

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

Coming to transactions in H , the set of transactions in H is denoted as $txns(H)$. The set of committed (resp., aborted) transactions in H is denoted by $committed(H)$ (resp., $aborted(H)$). The *live* transactions in H are those which are neither committed nor aborted. On the other hand, the *terminated* transactions are those which have either committed or aborted.

We denote two histories H_1, H_2 as *equivalent* if their events are the same, i.e., $evts(H_1) = evts(H_2)$. A history H is qualified to be *well-formed* if: (1) all the methods of a transaction T_i in H are totally ordered, i.e. a transaction invokes a method only after it receives a response of the previous method invoked by it (2) T_i does not invoke any other method after it received an \mathcal{A} response or after $STM_tryC(ok)$ method. We only consider *well-formed* histories for all the proposed Object-based STM systems.

Sequential Histories: A method m_{ij} (j^{th} method of a transaction T_i) in a history H is said to be *isolated or atomic* if for any other event e_{pqr} (r^{th} event of method m_{pq}) belonging to some other method m_{pq} (of transaction T_p) either e_{pqr} occurs before $inv(m_{ij})$ or after $rsp(m_{ij})$. Formally, $\langle m_{ij} \in methods(H) : m_{ij} \text{ is isolated} \equiv (\forall m_{pq} \in methods(H), \forall e_{pqr} \in m_{pq} : e_{pqr} <_H inv(m_{ij}) \vee rsp(m_{ij}) <_H e_{pqr}) \rangle$. For instance in $H1$ shown in Figure 1.1(ii), $del_2(k_2)$ is isolated. In fact all the methods of $H1$ are isolated.

Consider history $H2$ shown in Figure 2.2. It can be seen that all the three methods in $H2$, $(lu_{11}, del_{21}, lu_{12})$ are not isolated.

A history H is said to be *sequential* (term used in [18, 19]) or *linearized* [20] if all the methods in it are complete and isolated. Thus, it can be seen that $H1$ is sequential whereas $H2$ is not.

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 $STM_tryA_k(\mathcal{A})$ immediately

after the last method of every transaction $T_k \in \text{incomp}(H)$. Since all the methods in a sequential history are complete, this definition only has to take care of completing transactions.

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

Transactions: Following the notations used in database multi-level transactions [3], we model a transaction as a two-level tree. Figure 2.1 shows a tree execution of a transaction T_1 . The leaves of the tree denoted as *layer-0* consist of read, write primitives on atomic objects. Hence, they are atomic. For simplicity, we have ignored the *inv* & *rsp* events in level-0 of the tree. *Level-1* of the tree consists of methods invoked by transaction. In the transaction shown in Figure 2.1, level-1 consists of *STM_lookup()* and *STM_delete()* methods operating on the hash table as also shown in Figure 1.1(i).

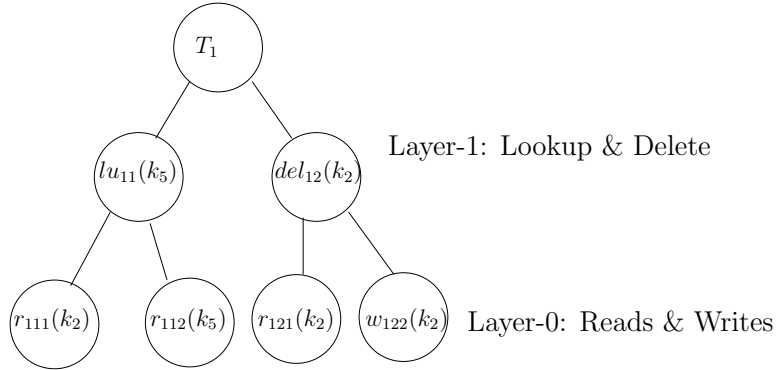


Figure 2.1: T_1 : A sample transaction on hash table (of Figure 1.1(i))

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 $\langle \text{evts}(T), <_T \rangle$. Here $\text{evts}(T)$ are all the read/write events (primitives) at level-0 of the transaction. $<_T$ is a total order among all the events of the transaction. For instance, the transaction T_1 of Figure 2.1 is: $\text{inv}(lu_{11}(ht, k_5)) r_{111}(k_2, o_2) r_{112}(k_5, o_5) \text{rsp}(lu_{11}(ht, k_5, o_5)) \text{inv}(del_{12}(ht, k_2)) r_{121}(k_2, o_2) w_{122}(k_2, o_2) \text{rsp}(del_{12}(ht, k_2, o_2))$. Given all level-0 events, it can be seen that the level-1 methods and the transaction tree can be constructed.

We denote the first and last events of a transaction T_i as $T_i.firstEvt$ and $T_i.lastEvt$. Given any other read/write event rw in T_i , we assume that $T_i.firstEvt <_{T_i} rw <_{T_i} T_i.lastEvt$.

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

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

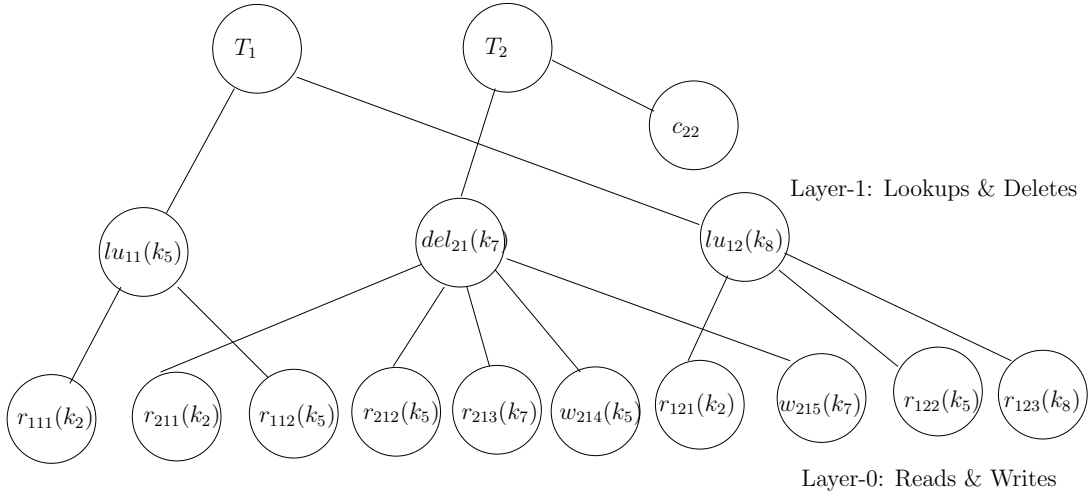


Figure 2.2: H2 : A non-sequential History.

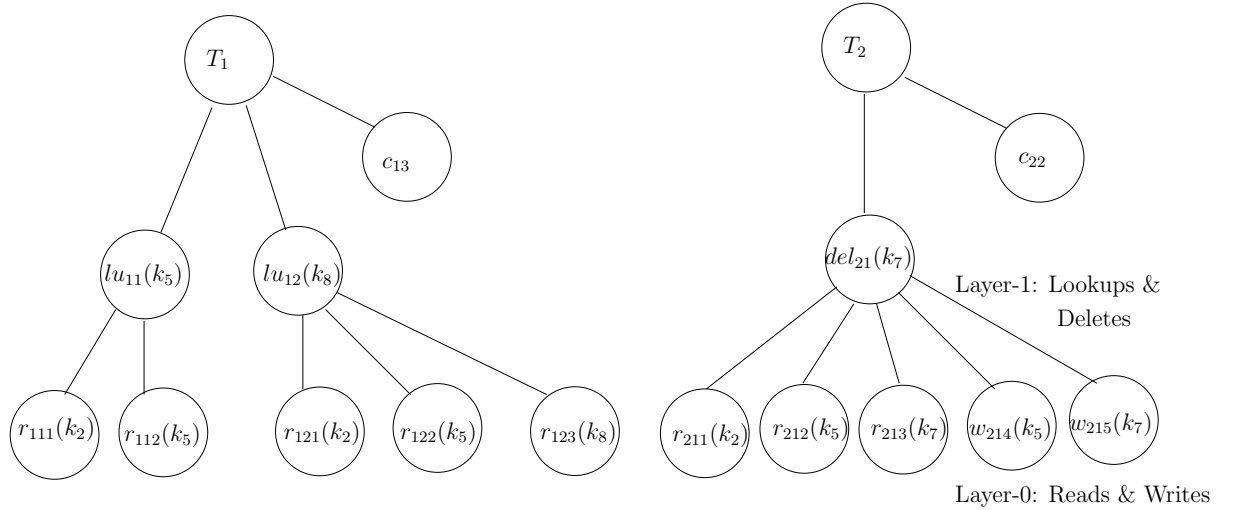


Figure 2.3: A serial History

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

We define a history H as *serial* [5] or *t-sequential* [19] if all the transactions in H have terminated and can be totally ordered w.r.t \prec_{TR} , i.e. all the transactions execute one after the other without any interleaving. Intuitively, a history H is serial if all its transactions can be isolated. Formally, $\langle (H \text{ is serial}) \implies (\forall T_i \in txns(H) : (T_i \in term(H)) \wedge (\forall T_i, T_p \in txns(H) : (T_i \prec_H^{TR} T_p) \vee (T_p \prec_H^{TR} T_i))) \rangle$. Since all the methods within a transaction are ordered, a serial history is also sequential. Refer Figure 2.3 in to shows a serial history.

Valid Histories: A `rv_method` ($STM_delete()$ and $STM_lookup()$) m_{ij} on key k is valid if it returns the value updated by any of the previous committed transaction that updated key k . A history H is said to valid if all the `rv_methods` of H are valid.

We formally prove validity for MVOSTM and OPT-MVOSTM in Section 4.5 and Section 5.4 and then show that proposed STMs histories are opaque.

Legal Histories To simplify our analysis, we assume that there exists an initial transaction T_0 that invokes $STM_delete()$ method on all the keys of all the hash tables used by any transaction.

We define *legality* of `rv_methods` ($STM_delete()$ & $STM_lookup()$) on sequential histories which we later use to define correctness criterion. Consider a sequential history H having a `rv_method` $rvm_{ij}(ht, k, v)$ (with $v \neq nil$) belonging to transaction T_i . We define this *rvm* method to be *legal* if:

- Rule1 If the rvm_{ij} is not first method of T_i to operate on $\langle ht, k \rangle$ and m_{ix} is the previous method of T_i to operate on $\langle ht, k \rangle$. Formally, $rvm_{ij} \neq H.firstKeyMth(\langle ht, k \rangle, T_i) \wedge (m_{ix}(ht, k, v') = H.prevKeyMth(\langle ht, k \rangle, T_i))$ (where v' could be nil). Then,
- (a) if $m_{ix}(ht, k, v')$ is a $STM_insert()$ method i.e. $STM_insert_{ix}(ht, k, v')$ then $v = v'$.
 - (b) if $m_{ix}(ht, k, v')$ is a $STM_lookup()$ method i.e. $STM_lookup_{ix}(ht, k, v')$ then $v = v'$.
 - (c) if $m_{ix}(ht, k, v')$ is a $STM_delete()$ method i.e. $STM_delete_{ix}(ht, k, v'/nil)$ then $v = nil$.

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

Rule2 If $rv_{m_{ij}}$ is the first method of T_i to operate on $\langle ht, k \rangle$ and v is not nil. Formally, $rv_{m_{ij}}(ht, k, v) = H.firstKeyMth(\langle ht, k \rangle, T_i) \wedge (v \neq nil)$. Then,

- (a) There is a $STM_insert()$ method $STM_insert_{pq}(ht, k, v)$ in $methods(H)$ such that T_p committed before $rv_{m_{ij}}$. Formally, $\langle \exists STM_insert_{pq}(ht, k, v) \in methods(H) : STM_tryC_p() \prec_H^{MR} rv_{m_{ij}} \rangle$.
- (b) There is no other update method up_{xy} of a transaction T_x operating on $\langle ht, k \rangle$ in $methods(H)$ such that T_x committed after T_p but before $rv_{m_{ij}}$. Formally, $\langle \nexists up_{xy}(ht, k, v'') \in methods(H) : STM_tryC_p() \prec_H^{MR} STM_tryC_x() \prec_H^{MR} rv_{m_{ij}} \rangle$.

In this case, we denote $STM_tryC_p()$ as the last update method of $rv_{m_{ij}}$, i.e., $STM_tryC_p(ht, k, v) = H.lastUpdt(rv_{m_{ij}}(ht, k, v))$.

Rule3 If $rv_{m_{ij}}$ is the first method of T_i to operate on $\langle ht, k \rangle$ and v is nil. Formally, $rv_{m_{ij}}(ht, k, v) = H.firstKeyMth(\langle ht, k \rangle, T_i) \wedge (v = nil)$. Then,

- (a) There is $STM_delete()$ method $STM_delete_{pq}(ht, k, v')$ in $methods(H)$ such that T_p (which could be T_0 as well) committed before $rv_{m_{ij}}$. Formally, $\langle \exists STM_delete_{pq}(ht, k, v') \in methods(H) : STM_tryC_p() \prec_H^{MR} rv_{m_{ij}} \rangle$. Here v' could be nil.
- (b) There is no other update method up_{xy} of a transaction T_x operating on $\langle ht, k \rangle$ in $methods(H)$ such that T_x committed after T_p but before $rv_{m_{ij}}$. Formally, $\langle \nexists up_{xy}(ht, k, v'') \in methods(H) : STM_tryC_p() \prec_H^{MR} STM_tryC_x() \prec_H^{MR} rv_{m_{ij}} \rangle$.

In this case, we denote $STM_tryC_p()$ as the last update method of $rv_{m_{ij}}$, i.e., $STM_tryC_p(ht, k, v) = H.lastUpdt(rv_{m_{ij}}(ht, k, v))$.

We assume that when a transaction T_i operates on key k of a hash table ht , the result of this method is stored in *local logs* of T_i for later methods to reuse. Thus, only the first rv_method operating on $\langle ht, k \rangle$ of T_i accesses the shared-memory. The other $rv_methods$ of T_i operating on $\langle ht, k \rangle$ do not access the shared-memory and they see the effect of the previous method from the *local logs*. This idea is utilized in Rule1. With reference to Rule2 and Rule3, it is possible that T_x could have aborted before $rv_{m_{ij}}$. For Rule3, since we are assuming that transaction T_0 has invoked a $STM_delete()$ method on all the keys used of all hash table objects, there exists at least one $STM_delete()$ method for every rv_method on k of ht . We formally prove legality for OSTM in Section 3.6 and then show that proposed STMs histories are opaque.

Coming to $STM_insert()$ methods, since a $STM_insert()$ method always returns ok as they overwrite the node if already present therefore they always take effect on the ht . Thus, we denote all $STM_insert()$ methods as legal. We denote a sequential history H as *legal* or *linearized* [20] if all its rvm methods are legal. While defining legality of a history, we are only concerned about rvm ($STM_lookup()$ and $STM_delete()$) methods since all $STM_insert()$ methods are by default legal.

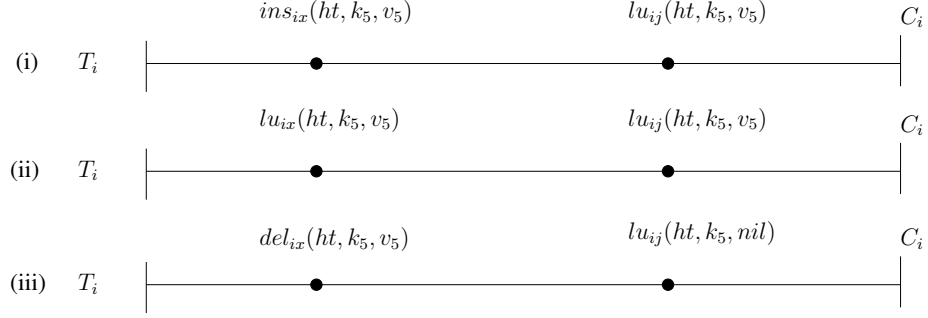


Figure 2.4: $STM_lookup()$ returns the same value as previous method of the same transaction on same key

Intuitive examples for Legality: If rv_method is not the first method of a transaction on any key then it will return the same value as the previous method of the same transaction on the same key. In Figure 2.4(i), previous method for $lu_{ij}(ht, k_5, v_5)$ of transaction T_i on same key k_5 is $ins_{ix}(ht, k_5, v_5)$. So, $lu_{ij}(ht, k_5, v_5)$ will return the same value which will be inserted by previous method $ins_{ix}(ht, k_5, v_5)$. Same technique will be followed in Figure 2.4(ii) and Figure 2.4(iii).

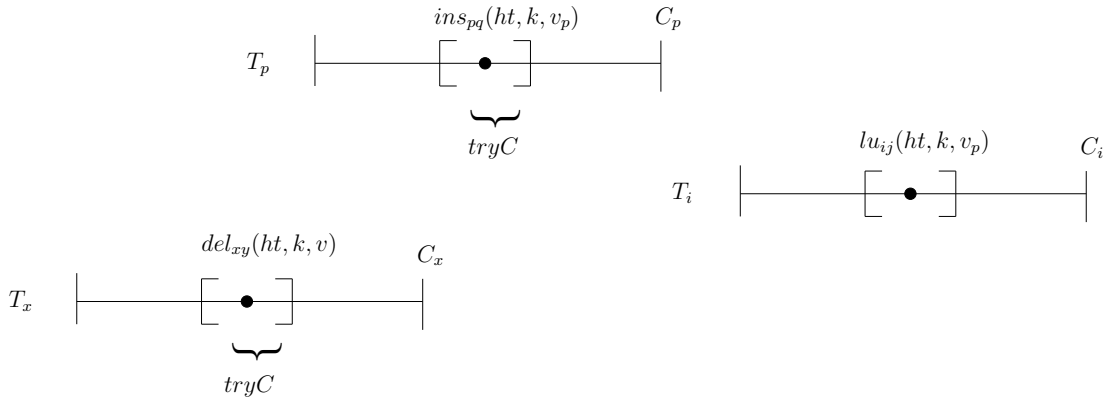


Figure 2.5: $STM_lookup()$ returns the same value as previous closest conflicting method of committed transaction

If rv_method is the first method of a transaction on any key and value is not null then the previous closest method of committed transaction should be inserted on the

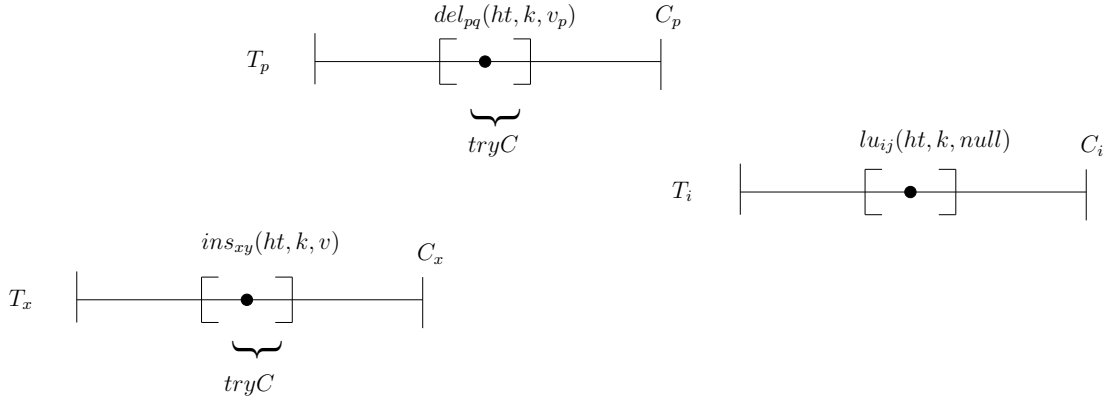


Figure 2.6: $STM_lookup()$ returns the same value as previous closest conflicting method of committed transaction

same key. In Figure 2.5, previous closest method for $lu_{ij}(ht, k, v_p)$ of transaction T_i on same key k is $ins_{pq}(ht, k, v_p)$ of transaction T_p . So, $lu_{ij}(ht, k, v_p)$ will return the same value which has been inserted by $ins_{pq}(ht, k, v_p)$ and there can't be any other transaction upd_method working on the same key between T_p and T_i . Figure 2.6 represents, previous closest method of committed transaction T_p is $del_{pq}(ht, k, v_p)$ on key k so $lu_{ij}(ht, k, null)$ 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* [10]. A sequential history H is opaque if there exists a serial history S such that: (1) S is equivalent to \overline{H} , i.e., $evts(\overline{H}) = evts(S)$ (2) S is legal and (3) S respects the transactional real-time order of H , i.e., $\prec_H^{TR} \subseteq \prec_S^{TR}$.

Linearizability: A linearizable [20] history H has following properties: (1) In order to get a valid sequential history the invocation and response events can be reordered. (2) The obtained sequential history should satisfy the sequential specification of the objects. (3) The real-time order should respect in sequential reordering as in H .

Chapter 3

Object-based Software Transactional Memory systems

3.1 Introduction

Software Transaction Memory systems (STMs) are a convenient programming interface for a programmer to access shared memory without worrying about concurrency issues [4, 21] and are natural choice for achieving composability [22].

Most of the *STMs* proposed in the literature are specifically based on read/write primitive operations (or methods) on memory buffers (or memory registers). These *STMs* typically export the following methods: (1) *STM_begin()*, (2) *STM_read(k)* (or *r(k)*), (3) *STM_write(k, v)* (or *w(k, v)*), (4) *STM_tryC()* (or *tryC()*) as explained in Section 1.2 of Chapter 1. We refer to these as *Read-Write STMs* or *RWSTM*. 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 cannot be commutated.

On the other hand, *Object-based STMs* or *OSTM* operate on higher level objects rather than read & write operations on memory locations. They include more semantically rich operations such as enqueue/dequeue on queue objects, push/pop on stack objects and insert/lookup/delete on sets, trees or hash table objects depending upon the underlying data structure used to implement *OSTM*.

It was shown in databases that object-level systems provide greater concurrency than read/write systems [3, Chap 6]. Along the same lines, we propose a model to achieve composability with greater concurrency for *STMs* by considering higher-level objects which leverage the richer semantics of object level methods. We motivated this with an interesting example in Section 1.2 of Chapter 1.

Consider an *OSTM* operating on the hash table object called as *Hash Table Object STM* or *HT-OSTM* which exports the following methods - (1) *STM_begin()*, (2) *STM_insert(k, v)* or *ins(k, v)*, (3) *STM_delete(k)* or *del(k)*, (4) *STM_lookup(k)* or *lu(k)* and, (5) *STM_tryC()* explained in Section 1.3 of Chapter 1.

The atomic property of transactions helps to correctly compose together several different individual operations. The above examples demonstrate that the concurrency in such STM can be enhanced by considering the object level semantics.

For correctness our framework considers, opacity [10] a popular correctness-criterion for STMs which is different from serializability commonly used in databases. It can be proved that verifying the membership of opacity similar to view-serializability is NP-Complete [5]. Hence, using conflicts we developed a subclass of opacity- *conflict opacity* or *co-opacity* for objects. We then developed polynomial time graph characterization for co-opacity based on conflict-graph acyclicity. The proposed correctness-criterion, co-opacity is similar to the notion of conflict-opacity developed for *RW-STM* by Kuznetsov & Peri [18]. To show the efficacy of this framework, we develop *OSTM* based on the idea of *Basic Timestamp Order (BTO)* scheduler developed in databases [3, Chap 4]. For showing correctness of *OSTM*, we prove that all the methods are linearizable [20] while the transactions are *co-opaque* by showing that the corresponding conflict graph is acyclic. We have considered two *Concurrent Data Structures (CDS)*, hash table and list based *OSTM* as *HT-OSTM* and *list-OSTM* (simple modification of *HT-OSTM* while considering the bucket size as 1), but we believe that this notion of conflicts can be extended to other high-level objects such as Stacks, Queues, Tries etc.

Contributions of the Chapter is as follows:

- We proposed a generic framework for composing higher-level objects based on the notion of conflicts for objects in databases [3, Chap 6].
- We developed a subclass of opacity [10] as conflict opacity or co-opacity for higher-level objects in Section 3.2.
- We rigorously proved that proposed *OSTMs* satisfy the correctness criteria as co-opacity [18] in Section 3.6.
- A simple modification of *HT-OSTM* gives us a concurrent list based STM or *list-OSTM*. Finally, we compared the performance of *HT-OSTM* against state-of-the-art hash table based STMs (*ESTM* [7] and *RWSTM* [3]). The *list-OSTM* is compared with state-of-the-art list based STMs (*Trans-list* [12], *NOREC-list*

[6] and Boosting-list [13]). The results show that *HT-OSTM* and *list-OSTM* reduces the number of aborts to minimal and show significant performance gain in comparison to state-of-the-art STMs in Section 3.7.

Related Work: Our work differs from databases model in with regard to correctness-criterion used for safety. While databases consider *Conflict Serializable Schedule* or *CSR* [3, Chap. 3], we consider linearizability [20] to prove the correctness of the methods of the transactions and opacity to show the correctness of the transactions. Earliest work of using the semantics of concurrent data structures for object level granularity include that of open nested transactions [23] and transaction boosting of Herlihy et al. [13] which is based on serializability (strict or commit order serializability) of generated schedules as correctness criteria. Herlihy’s model is pessimistic and uses undo logs for rollback. Our model is more optimistic in that sense and the underlying data structure is updated only after there is a guarantee that there is no inconsistency due to concurrency. Thus, we do not need to do rollbacks which keeps the log overhead minimal. This also solves the problem of irrevocable operations being executed during a transaction which might abort later otherwise.

Hassan et al. [24] have proposed Optimistic Transactional Boosting (OTB) that extends original transactional boosting methodology by optimizing and making it more adaptable to STMs. They further have implemented OTB on set data structure using lazy-linked list [24]. Although there seem similarities between their work and our implementation, we differ w.r.t the correctness-criterion which is co-opacity a subclass of opacity [18] in our case. Furthermore, we also differ in the development of the conflict-based theoretical framework which can be adapted to build other object-based STMs.

Zhang et al. [12] recently proposed a method to transform lockfree *CDS* to transactional lockfree linked *CDS* and base the correctness on *strict serializability*. The transactions are synchronized using CAS and they compare their work against STM based approaches. Our evaluation shows that *list-OSTM* implementation comprehensibly beats Zhang’s transactional lock free list (Trans-list) data structure in Section 3.7.

Fraser et al. [25] proposed OSTM based on shadow copy mechanism, which involves a level of indirection to access the shared objects through *OSTMOpenForReading* and *OSTMOpenForWriting* as exported methods. Contrary to it, our OSTM model exports the higher object level methods like *STM_lookup()*, *STM_insert()* and *STM_delete()* while hiding the internal read and write lower level primitives. The exported methods in OSTM by Fraser et al. [25] may allow *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 do not modify the underlying objects and only return some value by performing a search on them). Thus, we can say our motivation and implementation is different from Fraser OSTM and only the name happens to coincide.

Roadmap. This chapter is organized as follows. In Section 3.2, we build a new notion of legality, conflicts to describe opacity, co-opacity and the graph characterization. Section 3.3 represents the *OSTMs* design and data structure. Section 3.4 shows the working of *OSTMs* and its algorithms. We explain the detailed pseudocode of OSTM in Section 3.5. We formally prove the correctness of *OSTMs* in Section 3.6. In Section 3.7 we show the experimental evaluation of *OSTMs* with state-of-art-STMs. Finally, we summaries this chapter in Section 3.8.

3.2 A New Conflict Notion and Conflict-Opacity for OSTMs

In this section, we define the correctness of *OSTM* by extending opacity [10]. We then define a tractable subclass of opacity, co-opacity which is defined using conflict like CSR [3, Chap. 3] in databases.

Opacity is a popular correctness-criterion for STMs. But, as observed in Section 3.1, it can be proved that verifying the membership of opacity similar to view-serializability (VSR) in databases is NP-Complete [5]. To circumvent this issue, researchers in databases have identified an efficient sub-class of VSR, called conflict-serializability (CSR), based on the notion of conflicts. The membership of CSR can be verified in polynomial time using conflict graph characterization. Along the same lines, we develop the notion of conflicts for *OSTM* and identify a sub-class of opacity, co-opacity. The proposed correctness-criterion is extension of the notion of conflict-opacity developed for *RWSTM* by Kuznetsov & Peri [18].

We say two transactions T_i, T_j of a sequential history H for *OSTM* are in *conflict* if atleast one of the following conflicts holds:

- **tryC-tryC** conflict:(1) T_i & T_j are committed and (2) T_i & T_j update the same key k of the hash table, ht , i.e., $(\langle ht, k \rangle \in updtSet(T_i)) \wedge (\langle ht, k \rangle \in updtSet(T_j))$, where $updtSet(T_i)$ is update set of T_i . (3) T_i 's $STM_tryC()$ completed before T_j 's $STM_tryC()$, i.e., $STM_tryC_i() \prec_H^{MR} STM_tryC_j()$.
- **tryC-rv** conflict:(1) T_i is committed (2) T_i updates the key k of hash table, ht . T_j invokes a `rv_method` rvm_{jy} on the same key k of hash table ht which is the first method on $\langle ht, k \rangle$. Thus, $(\langle ht, k \rangle \in updtSet(T_i)) \wedge (rvm_{jy}(ht, k, v) \in$

$rvSet(T_j)) \wedge (rv_{m_{jy}}(ht, k, v) = H.firstKeyMth(\langle ht, k \rangle, T_j))$, where $rvSet(T_j)$ is return value set of T_j . (3) T_i 's $STM_tryC()$ completed before T_j 's rvm , i.e., $STM_tryC_i() \prec_H^{MR} rvm_{jy}$.

- **rv-tryC conflict:**(1) T_j is committed (2) T_i invokes a rv_method on the same key k of hash table ht which is the first method on $\langle ht, k \rangle$. T_j updates the key k of the hash table, ht . Thus, $(rv_{m_{ix}}(ht, k, v) \in rvSet(T_i)) \wedge (rv_{m_{ix}}(ht, k, v) = H.firstKeyMth(\langle ht, k \rangle, T_i)) \wedge (\langle ht, k \rangle \in updtSet(T_j))$ (3) T_i 's rvm completed before T_j 's $STM_tryC()$, i.e., $rv_{m_{ix}} \prec_H^{MR} STM_tryC_j()$.

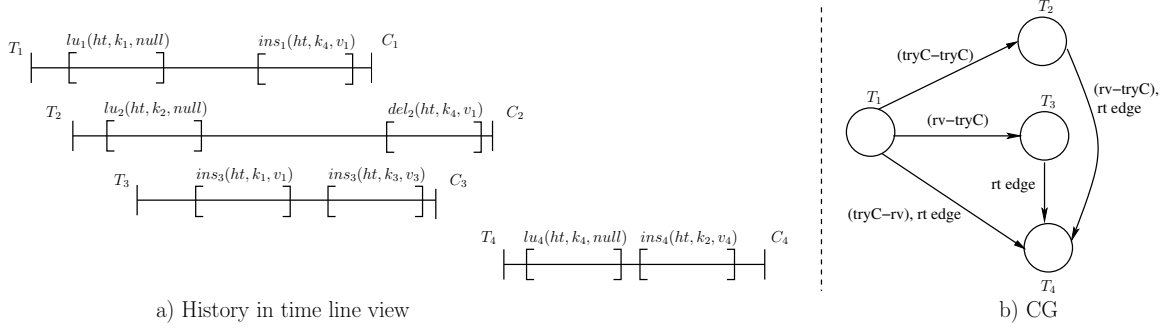


Figure 3.1: Graph Characterization of history $H5$

A rv_method $rv_{m_{ij}}$ conflicts with a $STM_tryC()$ method only if $rv_{m_{ij}}$ 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. $(STM_tryC_i(), STM_tryC_j())$, $(STM_tryC_i(), STM_lookup_j())$, $(STM_lookup_i(), STM_tryC_j())$, $(STM_tryC_i(), STM_delete_j())$ and $(STM_delete_i(), STM_tryC_j())$ can be the possible conflicting methods. For example, consider the history $H5 : lu_1(ht, k_1, null), lu_2(ht, k_2, null), ins_3(ht, k_1, v_1), ins_1(ht, k_4, v_1), c_1, ins_3(ht, k_3, v_3), c_3, del_2(ht, k_4, v_1), c_2, lu_4(ht, k_4, null), ins_4(ht, k_2, v_4), c_4$ in Figure 3.1. $(lu_1(ht, k_1, null), ins_3(ht, k_1, v_1))$ and $(lu_2(ht, k_2, null), ins_4(ht, k_2, v_4))$ are a conflict of type rv_tryC . Conflict type of $(ins_1(ht, k_4, v_1), del_2(ht, k_4, v_1))$ and $(ins_1(ht, k_4, v_1), lu_4(ht, k_4, null))$ are $tryC_tryC$ and $tryC_rv$ respectively.

Conflict Opacity: Using this conflict notion, we can now define *conflict-opaque* or *co-opacity*. A sequential history H is co-opaque if there exists a serial history S such that: (1) S is equivalent to \overline{H} (complete history), i.e., $evts(\overline{H}) = evts(S)$ (2) S is legal and (3) S respects the transactional real-time order of H , i.e., $\prec_H^{TR} \subseteq \prec_S^{TR}$ and (4) S preserves conflicts (i.e., $\prec_H^{CO} \subseteq \prec_S^{CO}$).

Thus from the above definition, it can be seen that any history that is co-opaque is also opaque.

Graph Characterization: We now develop a graph characterization of co-opacity.

For a sequential history H , we define *conflict-graph* of H , $CG(H)$ as the pair (V, E) where V is the set of $txns(H)$ and E can be of following types: (a) *conflict edges*: $\{(T_i, T_j) : (T_i, T_j) \in \text{conflict}(H)\}$ where, $\text{conflict}(H)$ is an ordered pair of transactions such that the transactions have one of the above pair of conflicts. (b) *real-time edge (or rt edge)*: $\{(T_i, T_j) : \text{Transaction } T_i \text{ precedes } T_j \text{ in real-time, i.e., } T_i \prec_H^{TR} T_j\}$.

3.3 OSTM Design and Data Structure

We design *OSTM* for a concurrent closed addressed hash table based on conflict notion defined in Section 3.2 and called as *HT-OSTM*. The *HT-OSTM* exports *STM_begin()*, *STM_insert()*, *STM_delete()*, *STM_lookup()*, *STM_tryC()* methods and has B number of buckets, which we refer to as size of the hash table. We propose list-OSTMs while considering the bucket size as 1 in HT-OSTM. The main part of interest from concurrency perspective is each bucket of the hash table implemented as *lazy red-blue list (or lazyrb-list)*, the shared memory data structure.

Lazyrb-list: $\langle \text{key, value, lock, marked, max.ts, RL, BL} \rangle$. It is a linked structure with immutable *head* and *tail* sentinel nodes. Each node of the list has *key, value, marked* (to have lazy deletion as popular in lazylists [8,9]) and *lock* (to implement exclusive access to the node) field. The *key* represents unique id of the node so that a transaction could differentiate between two nodes. The *key* values may range from $-\infty$ (key of head node) to $+\infty$ (key of tail node). The *value* field may accommodate any type ranging from a basic integer to a complex class type.

Lazyrb-list node has two links - *BL* (blue links) and *RL* (red links). First, the nodes which are not marked (not deleted) are reachable by *BL* from the head. Second, the nodes which are marked (i.e. logically deleted) and are only reached by *RL*. Thus, the name lazyrb-list (lazy red-blue list). All marked nodes are reachable via *RL* and all the unmarked nodes are reachable via *BL* & *RL* from the head. Thus nodes reachable by *BL* are the subset of the nodes reachable by *RL*. Every node of lazyrb-list is in increasing order of its key to avoid the deadlock.

Furthermore, every lazyrb-list node also has a *timestamp* field (*max.ts*) to record the ids of the transactions which most recently executed some method. Augmenting the underlying shared data structure with timestamps help in identifying conflicts by simulating the graph characterization of a generated history which is discussed with Figure 3.1 in Section 3.2. Thus, a transaction can decide if another conflicting transaction can cause a cycle in the execution and hence violate co-opacity [18].

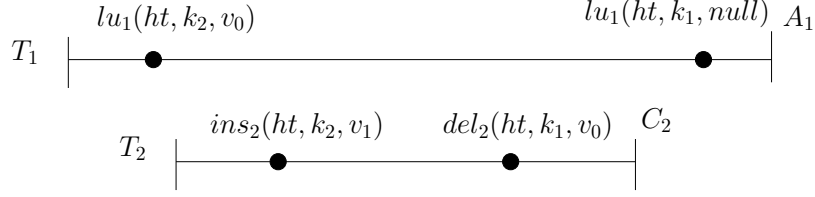


Figure 3.2: History H is not co-opaque

Now, we explain why we need to maintain deleted nodes through Figure 3.2 and 3.3. History H shown in Figure 3.2 is not co-opaque because there is no serial execution of T_1 and T_2 that can be shown co-opaque. In order to make it co-opaque $lu_1(ht, k_1, null)$ needs to be aborted. And $lu_1(ht, k_1, null)$ can only be aborted if *HT-OSTM* scheduler knows that a conflicting operation $del_2(ht, k_1, v_0)$ has already been scheduled and thus violating co-opacity. One way to have this information is that if the node represented by k_1 records the timestamp of the delete method so that the scheduler realizes the violation of the time-order [3] and aborts $lu_1(ht, k_1, null)$ to ensure co-opacity.

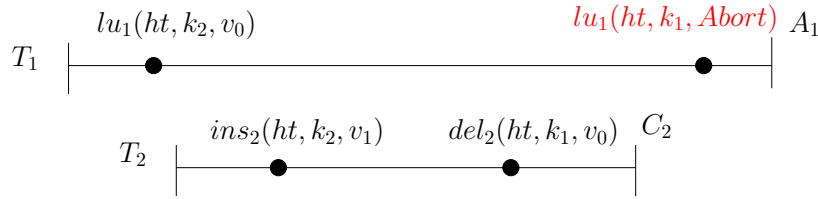


Figure 3.3: Co-opacity History H1

Thus, to ensure correctness, we need to maintain information about the nodes deleted from the hash table. This can be achieved by only marking node deleted from the list of hash table. But do not unlink it such that the marked node is still part of the list. This way, the information from deleted nodes can be used for ensuring co-opacity. In this case, after aborting $lu_1(ht, k_1)$, we get that the history is co-opaque with T_1 and T_2 being the equivalent serial history as shown in Figure 3.3. The deleted keys (nodes with marked field set) can be reused if another transaction comes and inserts the same key back.

But, the major hindrance in maintaining the deleted nodes as part of the ordinary lazy-list is that it would reduce search efficiency of the data structure. For example, in Figure 3.4 searching k_8 would unnecessary cause traversal over marked (marked for lazy deletion) nodes represented by k_1, k_3 and k_6 . We solve this problem in lazyrb-list by using two pointers. (1) **BL** (blue link): used to traverse over the actual inserted nodes and (2) **RL** (red link) used to traverse over the deleted nodes.

Hence, in Figure 3.5 to search for k_8 we can directly use BL saving significant search computations.

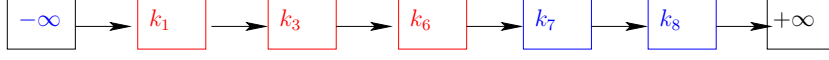


Figure 3.4: Searching k_8 over lazylist

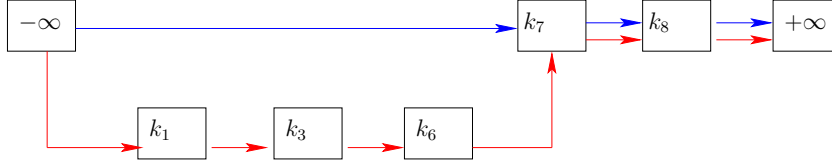


Figure 3.5: Searching k_8 over lazyrb-list

A question may arise that how would we maintain the timestamp of a node which has not yet been inserted? Such a case arises when $STM_lookup()$ or $STM_delete()$ is invoked from rv_method , and node corresponding to the key, say k is not present in BL and RL . Then the rv_method will create a node for key k and insert it into underlying data structure as deleted (marked field set) node.

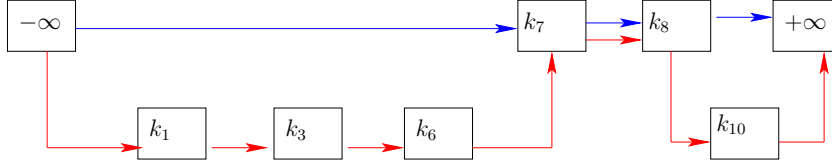


Figure 3.6: Execution under lazyrb-list

For example, lookup wants to search key k_{10} in Figure 3.5 which is not present in the BL as well as RL . Therefore, lookup method will create a new node corresponding to the key k_{10} and insert it into RL (refer the Figure 3.6). We discuss in detail the invariants and properties of the lazyrb-list and ensure that no duplicate nodes are inserted while proving the operational level correctness in Section 3.6.

Transaction local log. Each transaction maintains local log called $txlog$. It stores transaction id and status: *live*, *commit* or *abort* signifying that transaction is executing, has committed or has aborted due to some method failing the validation, respectively.

Each entry of the $txlog$ is called log_record (shortened as L_rec) stores the meta information of each method a transaction encounters as $updtSet()$ and $rvSet()$ formalized in Section 3.2. The L_rec is a tuple of type $\langle key, value, status, preds, currs \rangle$. A method may have *OK* and *FAIL* as it's status. The $preds$ and $currs$ are the array

of nodes in **RL** and **BL** identified during the traversal over the lazyrb-list by each method. It depicts the location over the lazyrb-list where the method would take effect.

3.4 The Working of OSTM

In this section, we explain the high-level idea of all the methods exported by HT-OSTM and the detailed description of it is provided in the Section 3.5.

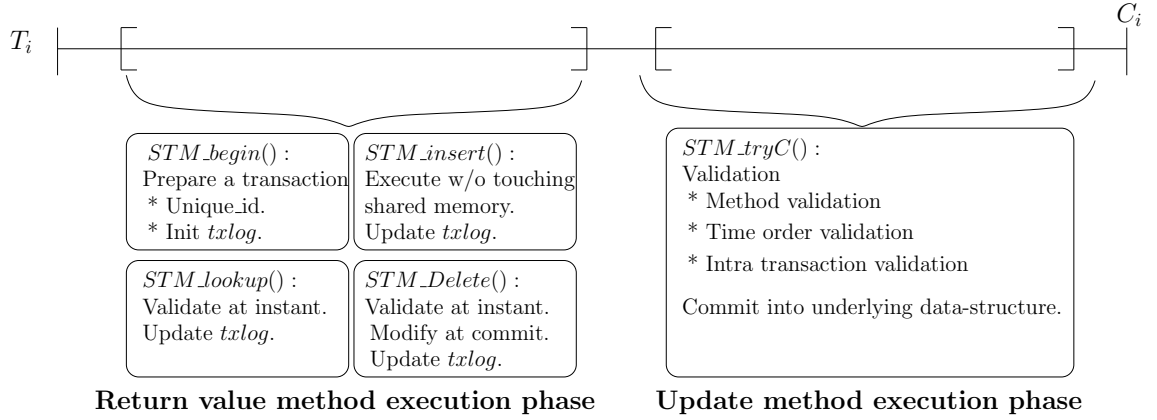


Figure 3.7: Transaction lifecycle of *HT-OSTM*

Through out its life a *HT-OSTM* transaction may execute *STM_begin()*, *STM_insert()*, *STM_lookup()*, *STM_delete()* and *STM_tryC()* methods which are also exported to the user. *STM_delete()*, *STM_lookup()* are rv methods while *STM_insert()*, *STM_delete()* are upd methods. Each transaction has a 1) *rv_method execution* phase: where upd_method and rv_method locally identify and logs the location to be worked upon and other meta information which would be needed for successful validation. Within *rv_method execution* phase rv_methods do lock free traversal and then validate while *STM_insert()* merely log there execution to be validated and updated during transaction commit. 2) *upd_method execution* phase: where it validates the upd_method executed during its lifetime and validates whether the transaction will commit and finally make changes in hash table atomically or it will abort and flush its log. Figure 3.7 depicts the transaction life cycle.

rv_method execution phase:

1. If $m_{ij}(k) \in \{STM_begin()\}$
 - (a) It invoked by a thread to being a new transaction T_i .
 - (b) It creates a local log and assign a unique id to each transaction.

2. $\forall m_{ij}(k) \in \{STM_lookup(), STM_delete()\}$
 - (a) If legality Rule1 is applicable.
 - i. Update the *txlog* and return.
 - (b) If legality Rule2 and Rule3 is applicable.
 - i. Traverse the underlying data structure to identify pred and curr nodes for both the **RL** and **BL** as done in lazy-lists or skip lists. Then, acquire ordered locks on the nodes.
 - ii. *Validate*. If the *Validate()* returns \mathcal{A} , the $m_{ij}(k)$ aborts followed by subsequently T_i is aborted and retried from step 1. Otherwise, if *Validate()* returns *retry* then $m_{ij}(k)$ is retried from step 2.(b).
 - iii. If validation succeeds, create a new *L_rec* in *txlog* and update the *L_rec*. And, insert a node in **RL** if the node is not present in lazyrb-list as explained in Figure 3.6.
 - iv. Release locks and return.
3. If $m_{ij}(k) \in \{STM_insert()\}$
 - (a) Update the *txlog* and return.

We validate *STM_lookup()* immediately and do not validate again in *STM_tryC()* unlike the implementation of Optimistic Transactional Boosting (OTB) by Hassan et al. [24]. This is required to ensure that the execution is opaque.

Validate():

1. First the current operation validates for any possible interference due to concurrent transactions through method validation.

Method Validation Rule: If the *preds* are marked and the next node of pred is not *curr*, implies a conflicting concurrent operation has also made changes. Thus, the current operation has to *retry*. Otherwise method validation is said to succeed.
2. Time order validation is performed when method validation succeeds.

Time Order Validation Rule [3, Chap 4]: When a transaction T_i with timestamp i want to access a node n . Also, Let T_j be a conflicting transaction with timestamp j which accessed n previously. Now, If $i < j$ then T_i is aborted. Else this method returns *ok*.
3. Return *abort* or *retry* or *ok*.

$STM_delete()$ in rv_method execution phase behaves as $STM_lookup()$ but it is validated twice. First, in rv_method execution similar to $STM_lookup()$ and secondly in upd_method execution to ensure co-opacity. We adopt lazy delete approach for $STM_delete()$. Thus, nodes are marked for deletion and not physically deleted for $STM_delete()$ method. In the current work we assume that a garbage collection mechanism is present and we do not worry about it.

upd_method execution phase. During this phase a transaction executes $STM_tryC()$. It begins by ordering the $txlog$ in increasing order of the keys. This way locks can be acquired in increasing order of keys to avoid deadlock. We re-validate upd_method in $txlog$ to ensure that the $pred$ and $curr$ for the methods has not changed since the point they were logged during rv_method execution phase. Please note that $txlog$ only contains the log record (L_rec) for upd_method . Because we do not validate the lookup and failed delete again in $STM_tryC()$.

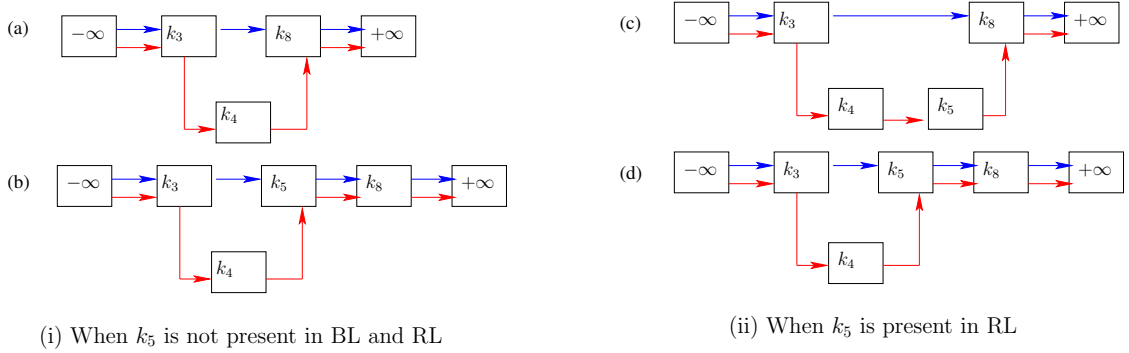


Figure 3.8: Insert of k_5 in $STM_tryC()$. (i) **BL** & **RL** of k_5 is set to K_8 then **BL** of k_3 linked to K_5 & **RL** of k_4 is linked to k_5 . (ii) Only **BL** of k_5 is set to K_8 then **BL** of k_3 linked to K_5 .

Now after successful validation, we update the shared lazyrb-list using the log records (L_rec) of the $txlog$ one by one. There may be two cases when a node is inserted into lazyrb-list by the $STM_insert()$. First, the node is not reachable by both **RL** and **BL** (not present in underlying data structure). Figure 3.8(i) represents this case when k_5 is neither reachable by **BL** and nor in **RL**. It adds k_5 to lazyrb-list at location $preds\langle k_3, k_4 \rangle$ and $currs\langle k_8, k_8 \rangle$ (in the notation, first and second index is the key reachable by **BL** and **RL**, respectively). Figure 3.8(i)(a) is lazyrb-list before addition of k_5 and Figure 3.8(i)(b) is lazyrb-list state post addition. Second, if the node is reached only by **RL**. Figure 3.8(ii) represents this case where k_5 is reached only by **RL**. It adds k_5 to lazyrb-list at location $pred\langle k_3, k_4 \rangle$ and $curr\langle k_5, k_8 \rangle$. Figure 3.8(i)(c) is lazyrb-list before addition of k_5 with **BL** and Figure 3.8(i)(d) is lazyrb-list state post addition.

During $STM_delete()$ if a node to be removed is reachable with BL then its marked field is set and the links are modified such that it is not reachable by BL . Figure 3.9 shows a case where a node k_5 needs to be deleted from the lazyrb-list in Figure 3.9 (a). So, here the node k_5 sets its marked field and then is detached from the BL (Figure 3.9 (b)).

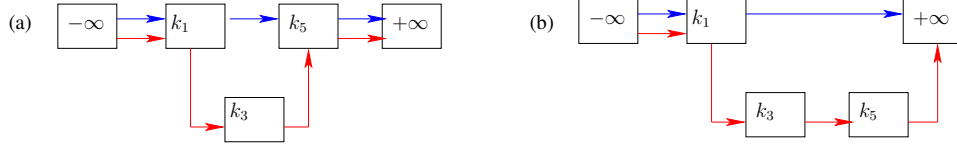


Figure 3.9: Delete of k_5 in $STM_tryC()$. k_5 is unlinked from BL by linking BL of k_1 to ∞ .

3.5 Detailed Pseudocode of OSTM

In this section we explain the working of proposed HT-OSTM methods in detail along with thread local and shared memory data structure. In proposed $HT-OSTM$, we use *thread local data structure (or 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.

3.5.1 Thread local Data Structure

Each transaction T_i maintains *local log* of type $Ltxlog$, which consists of transaction id and status as Lt_id and Ltx_status respectively. 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.

The *local log* also maintains a list ($Llist$) of meta information of each method a transaction executes in its life time. Each record of the $Llist$ is of type $Lrec$ which logs 1) L_key and L_val a method operates on, 2) L_opn : name of the method (or operation), 3) L_op_status : method's status (OK , $FAIL$) and 4) G_preds , G_currs : its *location* over the lazyrb-list.

We say a method identifies its *location* over the lazyrb-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

(curr[1]) node for *BL* and $\langle k_n, k_p \rangle$ are predecessor (preds[1]) and current (curr[0]) node for *RL*.

```

class L_txlog{
private:
int L_t_id;
STATUS L_tx_status;
/*A log record is uniquely identified using L_key and
L_obj_id in local record L_rec*/
vector <L_key, L_rec> L_list;
public :
L_txlog ();          ~L_txlog ();   find ();
setStatus ();       getList ();    sort ();          tryAbort ();
};

```

```

class L_rec{
public:
int L_obj_id, L_key, L_val;
node* G_preds, G_curr, G_node;
STATUS L_op_status;
OP_NAME L_opn;
getOpn ();          getPreds&Curr ();   getOpStatus ();
getKey&Objid ();  getVal ();          getAptCurr ();
setVal ();          setPreds&Curr ();
setOpStatus ();    setOpn ();
};
/*Types of method exported by the OSTIM*/
enum OP_NAME = {INSERT, DELETE, LOOKUP}
/*A transaction can ABORT/COMMIT and a method can ABORT,
OK, FAIL */
enum STATUS = {ABORT = 0, OK, FAIL, COMMIT}
/*To know whether validation is requested from STM_tryC ()
or rv_method (STM_lookup()/STM_delete())*/
enum VALIDATION_TYPE = {RV, TRYC}
/*To recognize on which list method has to be performed*/
enum LIST_TYPE = {RL, BL, RL_BL}

```

We use word location with G_preds and G_currs interchangeably in rest of the thesis. Each local and shared variables start with the prefix of “L” and “G” respectively to distinguish among them. Class L_rec also shows the getter and setter methods for each of the member variables which are self explanatory. Interested reader can find there description at Table 3.1.

Functions	Description
setOpn()	Set method name into transaction local log
setVal()	Set value of the key into transaction local log
setOpStatus()	Set status of method into transaction local log
setPred&Curr()	Set location of G_preds and G_currs according to the node corresponding to the key into transaction local log
getOpn()	Get method name from transaction local log
getVal()	Get value of the key from transaction local log
getOpStatus()	Get status of the method from transaction local log
getKey&Objid()	Get key and obj_id corresponding to the method from transaction local log
getPred&Curr()	Get location of G_preds and G_currs according to the node corresponding to the key from transaction local log

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

3.5.2 Shared memory Data Structure

```

struct G_node{
    int G_key, G_val;
    bool G_marked;
    struct G_max_ts;
    lock G_lock;
    G_node* G_rednext, G_bluenext;
};
/*Hash table where each bucket is a lazyrb-list chain*/
G_node* shared_ht [];

```

$HT-OSTM$ shared memory is the chained hash table where each node of the chain (lazyrb-list) is a key-value pairs of the form $\langle k, v \rangle$. Most of the notations used here are derived from [26]. A global node (G_node) n when created is initialized as follows:

(1) G_key and G_val is the key and value of the method that creates the node (2) G_marked is set to *false* (3) G_lock is null (4) $G_rednext$ (*RL*) and $G_bluenext$ (*BL*) are set to *nil* (5) Maximum timestamp (G_max_ts) is initialized to 0.

We adapt timestamp validation [3] to ensure schedules generated by proposed *HT-OSTM* are serial. Therefore we maintain $G_max_ts_lookup(ht, k)$, $G_max_ts_insert(ht, k)$ and $G_max_ts_delete(ht, k)$ that represent timestamp of last committed transaction which executed $STM_lookup(ht, k)$, $STM_insert(ht, k)$ and $STM_delete(ht, k)$ respectively. G_max_ts , G_node and L_rec form the part of the meta information for the *HT-OSTM*.

```

/*Stores the timestamp of last transaction that performed
lookup, insert or delete respectively */
struct G_max_ts {lookup; insert; delete;};

```

3.5.3 Pseudocode of OSTM

In this subsection we explain the detailed functionality of each method of proposed HT-OSTM along with there pseudocode.

Pseudocode convention: In each algorithm \downarrow represents the input parameter and \uparrow shows the output parameter (or return value) of the corresponding methods (such in and out variables are italicized). Each local and shared variables start with the prefix of “L” and “G” respectively to distingwish among them. Color of *preds* (*preds*[0], *preds*[1]) and *currs* (*currs*[1], *currs*[0]) in algorithm depicts the red or blue node which are accessed by red or blue links in the shared memory respectively.

rv_method execution phase: First, we explain the functionality of each method of proposed HT-OSTM in *rv_method* execution phase as follows:

STM_init(): This is the first function a transaction executes in its life cycle. It initialize the global counter (G_cnt) as 1 at Line 3 and return it.

STM_begin(): It initiates the L_txlog (local log) for the transaction (Line 8) and provides an unique id to the transaction (Line 10) using shared counter.

Then transaction may encounter the *upd_method* or *rv_method*.

STM_insert(): In *rv_method execution* phase, $STM_insert()$ 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*, *operation name* (L_opn) as insert and *operation status* L_op_status to *OK* (Line 22, Line 23 and Line 24 respectively). In case the $STM_insert()$ is the first

Algorithm 1 *STM_init()*: This method invokes at the start of the STM system. Initialize the global counter (*G_cnt*) as 1 at Line 3 and return it.

```

1: procedure STM_init(G_cnt ↑)
2:   /* Initializing the global counter */
3:   G_cnt ← 1;
4:   return ⟨G_cnt⟩;
5: end procedure

```

Algorithm 2 *STM_begin()*: It invoked by a thread to being a new transaction T_i . It creates transaction local log and allocate unique id at Line 8 and Line 10 respectively.

```

6: procedure STM_begin(G_cnt ↓, L_t_id ↑)
7:   /* Creating a local log for each transaction */
8:   L_txlog ← create new L_txlog();
9:   /* Getting transaction id (L_t_id) from G_cnt */
10:  L_txlog.L_t_id ← G_cnt;
11:  /* Incremented global counter atomically G_cnt */
12:  G_cnt ← get&inc(G_cnt ↓); //  $\Phi_{lp}$ (LinearizationPoint)
13:  return ⟨L_t_id⟩;
14: end procedure

```

method on *key* it creates a new log record for the *L_list* of *L_txlog* at Line 19. Finally the *STM_insert()* gets to modify the underlying hash table using *list_ins()* at the *upd_method execution* phase in *STM.tryC()*.

Algorithm 3 *STM_insert()*: Updates the log record and actual insertion happens in the *STM.tryC()*.

```

15: procedure STM_insert(L_t_id ↓, L_obj_id ↓, L_key ↓, L_val ↓, L_op_status ↑)
16:   /*First identify the node corresponding to the key into local log using find()
      function */
17:   if (!L_txlog.find(L_t_id ↓, L_obj_id ↓, L_key ↓, L_rec ↑)) then
18:     /* Create local log record and append it into increasing order of keys */
19:     L_rec ← create new L_rec(L_obj_id ↓, L_key ↓);
20:   end if
21:   /* Updating the local log */
22:   L_rec.setVal(L_obj_id ↓, L_key ↓, L_val ↓) ; //  $\Phi_{lp}$ 
23:   L_rec.setOpn(L_obj_id ↓, L_key ↓, INSERT ↓) ;
24:   L_rec.setOpStatus(L_obj_id ↓, L_key ↓, OK ↓) ;
25:   return ⟨void⟩;
26: end procedure

```

STM_lookup_i(ht, k): If this is the subsequent operation by a transaction T_i for a particular key k on hash table ht i.e. an operation on k has already been scheduled within the same transaction T_i , then this *STM_lookup()* returns the value from the L -list and does not access shared memory (Line 29 to Line 38 in Algorithm 4). 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 33) inserted (or observed) without accessing shared memory, and if the last operation was an *STM_delete()* then *STM_lookup()* returns the value NULL (Line 37). Thus in this process subsequent methods also have same conflicts as the first method on same key within the same transaction (*conflict inheritance*).

Algorithm 4 *STM_lookup()*: Returns the value corresponding to the key if exist

```

27: procedure STM_lookup( $L.t\_id \downarrow, L.obj\_id \downarrow, L.key \downarrow, L.val \uparrow, L.op\_status \uparrow$ )
28:   /* First identify the node corresponding to the key into local log */
29:   if ( $L.txlog.find(L.t\_id \downarrow, L.obj\_id \downarrow, L.key \downarrow, L.rec \uparrow)$ ) then
30:      $L.opn \leftarrow L.rec.getOpn(L.obj\_id \downarrow, L.key \downarrow)$  ;
31:     /* If previous operation is insert/lookup then current method would have
value/op_status same as previous log record */
32:     if ((INSERT ==  $L.opn$ ) || (LOOKUP ==  $L.opn$ )) then
33:        $L.val \leftarrow L.rec.getVal(L.obj\_id \downarrow, L.key \downarrow)$  ;
34:        $L.op\_status \leftarrow L.rec.getOpStatus(L.obj\_id \downarrow, L.key \downarrow)$  ;
35:       /* If previous operation is delete then current method would have value
as NULL and op_status as FAIL */
36:     else if (DELETE ==  $L.opn$ ) then
37:        $L.val \leftarrow \text{NULL}$  ;
38:        $L.op\_status \leftarrow \text{FAIL}$  ;
39:     end if
40:   else
41:     /* Common function for rv_method, if node corresponding to the key is
not the part of local log then search into underlying data structure */
42:      $commonLu\&Del(L.t\_id \downarrow, L.obj\_id \downarrow, L.key \downarrow, L.val \uparrow, L.op\_status \uparrow)$ ;
43:     /* update the local log */
44:      $L.rec.setOpn(L.obj\_id \downarrow, L.key \downarrow, LOOKUP \downarrow)$  ;
45:      $L.rec.setOpStatus(L.obj\_id \downarrow, L.key \downarrow, L.op\_status \downarrow)$  ;
46:   end if
47:   return  $\langle L.val, L.op\_status \rangle$ ;
48: end procedure

```

If $STM_lookup()$ is the first operation on a particular key then it has to do a wait free traversal (Line 82 in Algorithm 6) with the help of $list_lookup()$ (Algorithm 8) to identify the target node ($preds$ and $currs$) to be logged in L_list for subsequent methods in rv_method execution phase (discussed above for the case where $STM_lookup()$ is the subsequent method). If the node is present as blue (or red) node then it updates the operation status as OK (or FAIL) and returns the value respectively (Line 88 to Line 96 in Algorithm 6). If node corresponding to the key is not found then it inserts that node (Line 97 to Line 102 in Algorithm 6) corresponding to the key into RL of lazyrb-list. The inserted node can be accessed only via red links. Hence, it will not be visible to any subsequent $STM_lookup()$. The node is inserted to take care of situations as illustrated in Figure 3.2 and Figure 3.3 of Section 3.3. Finally, it updates the meta information in L_list and releases the locks acquired inside $list_lookup()$ (Line 105 to Line 109).

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 3.10(a), if we would have validated $lu(ht, k_1, v_0)$ again during $STM_tryC()$, T_1 would abort due to time order violation [3], but we can see that this history is acceptable where T_1 can be serialized before T_2 (Figure 3.10(b)). Thus, HT_OSTM prevents such unnecessary aborts. Another advantage for this design choice is that T_1 doesn't have to wait for $STM_tryC()$ to know that the transaction is bound to abort as can be seen in Figure 3.10(c). 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.

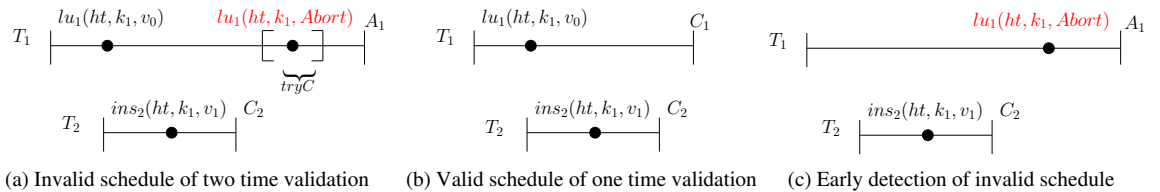


Figure 3.10: Advantages of lookup validated once

$STM_delete()$: In rv_method execution phase, $STM_delete()$ (Algorithm 5) 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 there is already a previous method on same key using the local log. In case there 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 51 to Line 69). For example if previous executed method is an *STM_insert()* then the current *STM_delete()* method will return *OK* (Line 54 to Line 58). If the previous executed method is an *STM_delete()* then the current *STM_delete()* should returns *FAIL* (Line 60 to Line 63). 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 *L_value* and *L_opn*. 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 71 to Line 76. *list_lookup()* gives the correct location for the current *STM_delete()* to take effect over the hash table in form of *preds* and *currs* (Line 82) along with the validation status which reveals weather the *STM_delete()* will succeed or abort. If the *L_op_status* is Abort, the method simply aborts the transaction. Otherwise, *STM_delete()* updates the local log and the timestamps of the corresponding nodes in the lazyrb-list of the hash table from Line 86 to Line 103. From Line 88 to Line 91, *STM_delete()* observes that the node to be deleted is reachable from *BL* i.e. it is *currs[1]* thus it updates its timestamp field and returns *L_op_status* to *OK* with the value of *currs[1]* (the update corresponding to this case takes place in *STM_tryC()* as represented in Figure 3.9 of Section 3.4 above).

Algorithm 5 *STM_delete()*: Actual deletion of node corresponding to the key happens in the *STM_tryC()* if it exist in *BL*.

```

49: procedure STM_delete(L.t_id ↓, L_obj_id ↓, L_key ↓, L_val ↑, L_op_status ↑)
50:   /* First identify the node corresponding to the key into local log */
51:   if (L.txlog.find(L.t_id ↓, L_obj_id ↓, L_key ↓, L_rec ↑)) then
52:     L_opn ← L_rec.getOpn(L_obj_id ↓, L_key ↓) ;
53:     /* If previous local method is insert and current operation is delete then
54:     overall effect should be of delete, update log accordingly */
55:     if (INSERT == L_opn) then
56:       L_val ← L_rec.getVal(L_obj_id ↓, L_key ↓) ;
57:       L_rec.setVal(L_obj_id ↓, L_key ↓, NULL ↓) ;
58:       L_rec.setOpn(L_obj_id ↓, L_key ↓, DELETE ↓) ;
59:       L_op_status ← OK ;
60:     /* If previous local method is delete and current operation is delete
61:     then overall effect should be of delete, update log accordingly */
62:     else if (DELETE == L_opn) then
63:       L_rec.setVal(L_obj_id ↓, L_key ↓, NULL ↓) ;
64:       L_val ← NULL ;

```

```

63:      L_op_status ← FAIL ;
64:      else
65:          /* If previous local method is lookup and current operation is delete
           then overall effect should be of delete, update log accordingly */
66:          L_val ← L_rec.getVal(L_obj_id ↓, L_key ↓) ;
67:          L_rec.setVal(L_obj_id ↓, L_key ↓, NULL ↓) ;
68:          L_rec.setOpn(L_obj_id ↓, L_key ↓, DELETE ↓) ;
69:          L_op_status ← L_rec.getOpStatus(L_obj_id ↓, L_key ↓) ;
70:      end if
71:      else
72:          /* Common function for rv_method, if node corresponding to the key is
           not the part of local log then search into underlying DS */
73:          commonLu&Del(L_t_id ↓, L_obj_id ↓, L_key ↓, L_val ↑, L_op_status ↑);
74:          /* Update the local log */
75:          L_rec.setOpn(L_obj_id ↓, L_key ↓, DELETE ↓) ;
76:          L_rec.setOpStatus(L_obj_id ↓, L_key ↓, L_op_status ↓) ;
77:      end if
78:      return ⟨L_val, L_op_status⟩;
79: end procedure

```

From Line 93 to Line 96, *STM_delete()* observes that the node to be deleted is reachable by *RL* i.e. it is *currs[0]* thus it updates its timestamp field and sets *L_op_status* to *FAIL* (as the node is dead node or marked for deletion) and value returned is *NULL*. Otherwise, in Line 97 to Line 102 the node is not at all present in lazyrb-list. Thus first *STM_delete()* adds a node in *RL* and updates its timestamp and returns the *value* as *NULL* and sets the *L_op_status* as *FAIL* (Figure 3.12 and Figure 3.13 represents the case). Line 108 and Line 109 sets the *value* and location in local log respectively. At Line 105 the locks acquired (in invoked *list_lookup()*) to update shared memory timestamps are released in order.

Algorithm 6 *commonLu&Del()*: This method is called by rv_methods (*STM_lookup()* and *STM_delete()*) to identify the node corresponding to the key from underlying data structure.

```

80: procedure commonLu&Del(L_t_id ↓, L_obj_id ↓, L_key ↓, L_val ↑
   , L_op_status ↑)
81:     /* If node corresponding to the key is not present in local log then search into
       underlying DS with the help of list lookup */

```

```

82: list_lookup(L_obj_id ↓, L_key ↓, G_pred ↑, G_curr ↑) ;
83: if (L_op_status == ABORT) then
84:     /* Release local memory in case list_lookup() returns abort */
85:     return ABORT ;
86: else
87:     /* If node corresponding to the key is part of BL */
88:     if (currs[1].key == L_key) then
89:         L_op_status ← OK ;
90:         currs[1].max_ts.lookup ← TS(L_t_id) ;
91:         L_val ← currs[1].val ;
92:         /*If node corresponding to the key part of RL*/
93:     else if (currs[0].key == L_key) then
94:         L_op_status ← FAIL ;
95:         currs[0].max_ts.lookup ← TS(L_t_id) ;
96:         L_val ← NULL ;
97:     else
98:         /* If node corresponding to the key is not part of RL as well as BL
99:         then create the node into RL with the help of list.Ins() */
100:        list_ins(G_pred ↓, G_curr ↓, G_node ↑, RL ↓) ;
101:        L_op_status ← FAIL ;
102:        G_node.max_ts.lookup ← TS(L_t_id) ;
103:        L_val ← NULL ;
104:    end if
105:    /* Release all the locks */
106:    releasePred&CurrLocks(G_preds[] ↓, G_currs[] ↓);
107:    /* Create local log record and append it into increasing order of keys */
108:    L_rec ← create new L_rec(L_obj_id ↓, L_key ↓);
109:    L_rec.setVal(L_obj_id ↓, L_key ↓, NULL ↓) ;
110:    L_rec.setPred&Curr(L_obj_id ↓, L_key ↓, G_pred ↓, G_curr ↓) ;
111: end if
112: return ⟨L_val, L_op_status⟩
113: end procedure

```

upd_method execution phase: Finally a transaction after executing the designated operations reaches the *upd_method execution* phase executed by the *STM-tryC()* method. It starts with modifying the log to *L_ordered_list* which contains the log records in sorted order of the keys (so that locks can be acquired in an order,

refer Line 117 of Algorithm 7) and contains only the `upd_method` (because we do not validate the lookup again for the reasons explained above in Figure 3.10). From Line 119 to Line 129 (in Algorithm 7) we re-validate the modified log operation to ensure that the location for the operations has not changed since the point they were logged during *rv_method execution* phase. If the location for an operation has changed this block ensures that they are updated.

Now, *STM_tryC()* enters the phase where it updates the shared memory using local data stored from Line 132 to Line 167 in Algorithm 7. Figure 3.8 and Figure 3.9 of Section 3.4 explain the execution of insert and delete in update phase of *STM_tryC()* using *list_ins()* and *list_del()* respectively. Figure 3.8(i) represents the case when k_5 is neither present in *BL* and nor in *RL* (Line 152 to Line 156 in Algorithm 7). It adds k_5 to lazyrb-list at location $preds\langle k_3, k_4 \rangle$ and $currs\langle k_8, k_8 \rangle$. Figure 3.8(i)(a) is lazyrb-list before addition of k_5 and Figure 3.8(i)(b) is lazyrb-list state post addition. Similarly, Figure 3.8(ii) represents the case when k_5 is present in *RL* (Line 147 to Line 151 in Algorithm 7). It adds k_5 to lazyrb-list at location $pred\langle k_3, k_4 \rangle$ and $curr\langle k_5, k_8 \rangle$. Figure 3.8(i)(c) is lazyrb-list before addition of k_5 into *BL* and Figure 3.8(i)(d) is lazyrb-list state post addition. In case of $del(k_5)$ from lazyrb-list when k_5 is present in *BL* (Line 161 to Line 167 in Algorithm 7) Figure 3.9(a) represent the lazyrb-list state before k_5 is deleted at location $preds\langle k_1, k_3 \rangle$ and $currs\langle k_5, k_5 \rangle$ and Figure 3.9(b) represents the lazyrb-list state after deletion.

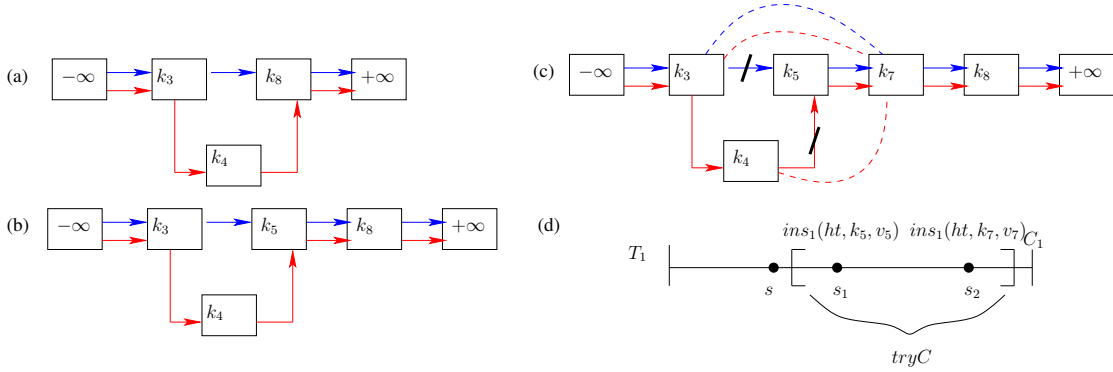


Figure 3.11: Problem in execution without `intraTransValidation()` ($ins_1(k_5)$ and $ins_1(k_7)$). (i) lazyrb-list at state s . (ii) lazyrb-list at state s_1 . (iii) lazyrb-list at state s_2 (lost update problem).

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 lazyrb-list causing the *preds* and *currs* of the next method working on the consecutive key to become obsolete. Figure 3.11 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 lazyrb-list. The lazyrb-list at s is as in Figure 3.11(a) also $ins_1(k_5)$ and $ins_1(k_7)$ have $preds\langle k_3, k_4 \rangle$ and $currs\langle k_8, k_8 \rangle$ as there location. Now, Lets say $ins_1(k_5)$ adds k_5 between k_3 and k_8 and changes lazyrb-list (as in Figure 3.11(b)) at state s_1 in Figure 3.11(d). 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 3.11(c) with dotted links. We correct this through $intraTransValidation()$ which updates current upd_method's $preds$ and $currs$ with the help of its L_rec . We discuss $intraTransValidation()$ in detail in at Algorithm 14.

Now, we illustrate the helping methods of rv_method and upd_method as follows:

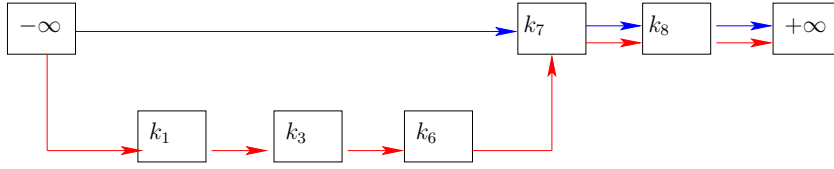


Figure 3.12: k_{10} is not present in BL as well as RL

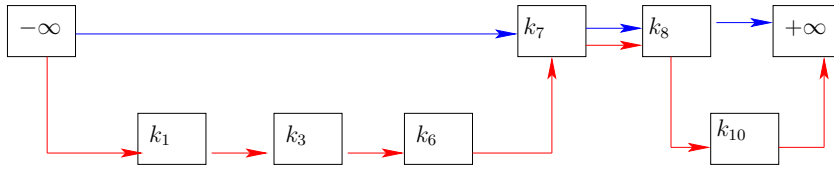


Figure 3.13: Adding k_{10} into RL

Algorithm 7 $STM_tryC()$: Actual effect of $STM_delete()$ and $STM_insert()$ will take place in this method after successful validation.

113: **procedure** $STM_tryC(L_t_id \downarrow, L_tx_status \uparrow)$

114: /* Get the txlog of the current transaction by t_id */

115: $L_list \leftarrow L_txlog.getList(L_t_id \downarrow);$

116: /* Sort the local log in increasing order of keys & copy into ordered list */

117: $L_ordered_list \leftarrow L_txlog.sort(L_list \downarrow);$

```

118:  /* Identify the new preds and currs for all update methods of a transaction
      (tx) and validate it */
119:  while ( $L\_rec_i \leftarrow \text{next}(L\_ordered\_list)$ ) do
120:    ( $L\_key, L\_obj\_id \leftarrow L\_rec.get\text{Key}\&\text{Objid}(L\_rec_i \downarrow)$ );
121:    /* Search correct location for the operation over list_lookup() and lock
      the corresponding G_preds[] and G_currs[] */
122:    list_lookup( $L\_obj\_id \downarrow, L\_key \downarrow, G\_pred \uparrow, G\_curr \uparrow$ );
123:    /* If list_lookup() return op_status as ABORT then method will return
      ABORT */
124:    if ( $L\_op\_status = \text{ABORT}$ ) then
125:      /* release local memory in case list_lookup() returns abort */
126:      return ABORT;
127:    end if
128:    /* Modify log record to help upcoming update method of same tx */
129:     $L\_rec.set\text{Pred}\&\text{Curr}(L\_obj\_id \downarrow, L\_key \downarrow, G\_pred \downarrow, G\_curr \downarrow)$ ;
130:  end while
131:  /* Get each update method one by one & take effect in underlying DS */
132:  while ( $L\_rec_i \leftarrow \text{next}(L\_ordered\_list)$ ) do
133:    ( $L\_key, L\_obj\_id \leftarrow L\_rec.get\text{Key}\&\text{Objid}(L\_rec_i \downarrow)$ );
134:    /* Get the operation name to local log record */
135:     $L\_opn \leftarrow L\_rec_i.L\_opn$ ;
136:    /* If operation is insert then after successful completion of it node
      corresponding to the key should be part of BL */
137:    if ( $\text{INSERT} == L\_opn$ ) then
138:      /* If node corresponding to the key is part of BL */
139:      if ( $currs[1].key == L\_key$ ) then
140:        /* Get the value from local log */
141:         $L\_val \leftarrow L\_rec.get\text{Val}(L\_obj\_id \downarrow, L\_key \downarrow)$ ;
142:        /* update the value into underlying DS */
143:         $currs[1].val \leftarrow L\_val$ ;
144:        /* Update the max_ts of insert for node corresponding to the key
          into underlying DS */
145:         $currs[1].max\_ts.insert \leftarrow \text{TS}(L\_t\_id)$ ;
146:        /* If node corresponding to the key is part of RL */
147:      else if ( $currs[0].key == L\_key$ ) then
148:        /* Connect the node corresponding to the key to BL as well */
149:        list_ins( $G\_pred \downarrow, G\_curr \downarrow, G\_node \uparrow, RL\_BL \downarrow$ );

```

```

150:         /* Update the max_ts of insert for node corresponding to the key
           into underlying DS */
151:         currs[0].max_ts.insert ← TS(L.t_id) ;
152:     else
153:         /* If node corresponding to the key is not part of BL as well as
           RL then create the node using list_ins() and add it into BL */
154:         list_ins(G_pred ↓, G_curr ↓, G_node ↑, BL ↓) ;
155:         /* Update the max_ts of insert for node corresponding to the key
           into underlying DS */
156:         G_node.max_ts.insert ← TS(L.t_id) ;
157:         /* Need to update the node field of log so that it can be released
           finally */
158:         L_reci.node ← preds[0].BL
159:     end if
160:     /* If operation is delete then after successful completion of it node
           corresponding to the key should not be part of BL */
161:     else if (DELETE == L_opn) then
162:         /* If node corresponding to the key is part of BL */
163:         if (currs[1].key == L_key) then
164:             /* Delete the node corresponding to the key from the BL with the
               help of list_del() */
165:             list_del(preds[] ↓, currs[] ↓) ;
166:             /* Update the max_ts of delete for node corresponding to the key
               into underlying DS */
167:             currs[1].max_ts.delete ← TS(L.t_id) ;
168:         end if
169:     end if
170:     /* Modify the preds and currs for the consecutive update methods which
           are working on overlapping zone in lazyskip-list */
171:     intraTransValidation(L_reci ↓, G_preds[] ↑, G_currs[] ↑) ;
172: end while
173:     /* release all the locks */
174:     releaseOrderedLocks(L_list ↓) ;
175:     /* set the tx status as OK */
176:     L_tx_status ← OK ;
177:     return ⟨L_tx_status⟩;
178: end procedure

```

Algorithm 8 *list_lookup()*: Identify the location of node corresponding to the key in the underlying data structure.

```

179: procedure list_lookup( $L\_t\_id \downarrow, L\_obj\_id \downarrow, L\_key \downarrow, L\_val\_type \downarrow, G\_preds[] \uparrow$ 
    ,  $G\_currs[] \uparrow, L\_op\_status \uparrow$ )
180:   /* By default setting the L_op_status as RETRY */
181:   STATUS  $L\_op\_status \leftarrow$  RETRY;
182:   while ( $L\_op\_status ==$  RETRY) do
183:     /* Get the head of the bucket in hash table */
184:      $G\_head \leftarrow$  getListHead( $L\_obj\_id \downarrow, L\_key \downarrow$ );
185:     /* Initialize (init)  $preds[0]$  to head */
186:      $preds[0] \leftarrow G\_head$  ;
187:     /* Init  $currs[1]$  to  $preds[0].BL$  */
188:      $currs[1] \leftarrow preds[0].BL$ ;
189:     /* Searching node corresponding to the key into BL */
190:     while ( $currs[1].key < L\_key$ ) do
191:        $preds[0] \leftarrow currs[1]$  ;
192:        $currs[1] \leftarrow currs[1].BL$ ;
193:     end while
194:     /*Init  $preds[1]$  to  $preds[0]$ */
195:      $preds[1] \leftarrow preds[0]$  ;
196:     /*Init  $currs[0]$  to  $preds[0].RL$ */
197:      $currs[0] \leftarrow preds[0].RL$ ;
198:     /*Searching node corresponding to the key into RL*/
199:     while ( $currs[0].key < L\_key$ ) do
200:        $preds[1] \leftarrow currs[0]$  ;
201:        $currs[0] \leftarrow currs[0].RL$ ;
202:     end while
203:     /* Acquire the locks on increasing order of keys */
204:     acquirePred&CurrLocks( $G\_preds[] \downarrow, G\_currs[] \downarrow$ );
205:     /* Validate the location recorded in  $G\_preds[]$  &  $G\_currs[]$ . Also verify if
    the transaction has to be aborted. */
206:     validation( $L\_t\_id \downarrow, L\_key \downarrow, G\_preds[] \downarrow, G\_currs[] \downarrow, L\_val\_type \downarrow,$ 
     $L\_op\_status \uparrow$ );
207:     /* If validation returns op_status as RETRY or ABORT then release all
    the locks */
208:     if ( $(L\_op\_status ==$  RETRY)  $\vee (L\_op\_status ==$  ABORT)) then
209:       /* Release all the locks */
210:       releasePred&CurrLocks( $G\_preds[] \downarrow, G\_currs[] \downarrow$ )

```

```

211:     end if
212: end while
213:   return  $\langle G\_preds[], G\_currs[], L\_op\_status \rangle$  ;
214: end procedure

```

Algorithm 9 *list_ins()*: Inserts or overwrites a node in underlying hash table at location corresponding to *preds* and *currs*.

```

215: procedure list_ins( $G\_preds[] \downarrow, G\_currs[] \downarrow, L\_list\_type \downarrow$ )
216:   /* Inserting the node from red list to blue list */
217:   if (( $L\_list\_type$ ) = ( $RL\_BL$ )) then
218:      $currs[0].marked \leftarrow FALSE$  ;
219:      $currs[0].BL \leftarrow currs[1]$  ;
220:      $preds[0].BL \leftarrow currs[0]$  ;
221:     /* Inserting the node into red list only */
222:   else if (( $L\_list\_type$ ) ==  $RL$ ) then
223:     node = Create new node() ;
224:     /* After creating the node acquiring the lock on it */
225:     node.lock();
226:     node.marked  $\leftarrow TRUE$  ;
227:     node. $RL \leftarrow currs[0]$  ;
228:      $preds[1].RL \leftarrow node$  ;
229:   else
230:     /* Inserting the node into red as well as blue list */
231:     node = Create new node() ;
232:     /* After creating the node acquiring the lock on it */
233:     node.lock();
234:     node. $RL \leftarrow currs[0]$  ;
235:     node. $BL \leftarrow currs[1]$  ;
236:      $preds[1].RL \leftarrow node$  ;
237:      $preds[0].BL \leftarrow node$  ;
238:   end if
239:   return  $\langle void \rangle$ ;
240: end procedure

```

list_ins(): It adds a new node to the lazyrb-list in the hash table (Algorithm 9). There can be following cases: **If node is present in RL and has to be inserted to BL**: such a case implies that the *list_ins()* is invoked in *upd_method execution*

phase for the corresponding $STM_insert()$ in local log represented by the block from Line 217 to Line 220. Here we first reset the $currs[0]$ mark field and update the BL to the $currs[1]$ and $preds[0]$ BL to $currs[0]$. Thus the node is now reachable by BL also. Figure 3.14(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 lazyrb-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 222 to Line 228 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 227 and Line 228 the RL field of the $node$ is updated to $currs[0]$ and RL field of the $preds[1]$ is modified to point to the $node$ respectively. Figure 3.14(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 217 to Line 220) or the node is not present at all. The later case is depicted in Line 229 to Line 237 which creates a new $node$ and add the node in both RL and BL note that order of insertion is important as the lazyrb-list can be concurrently accessed by other transactions since traversal is lock free. Figure 3.14(iii) represents the case.

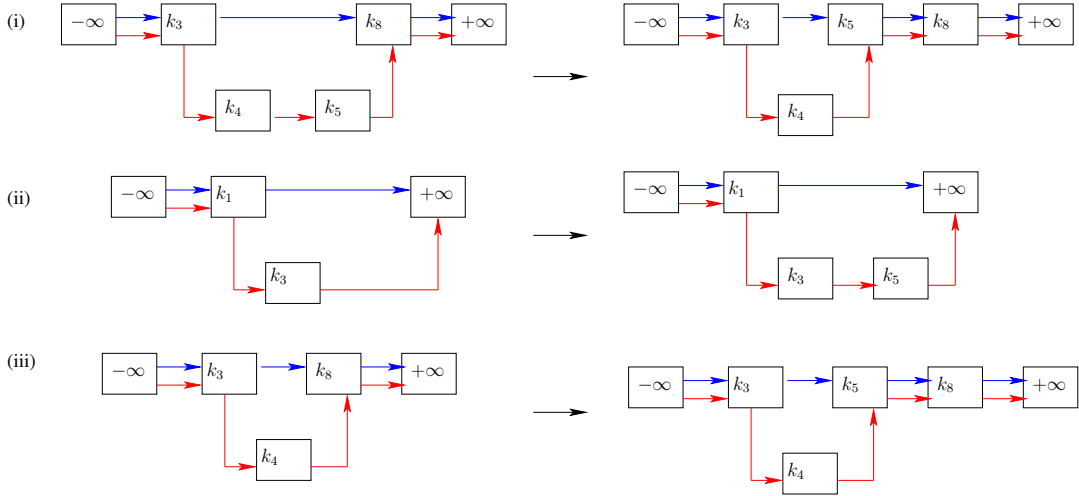


Figure 3.14: Execution of $list_ins()$: (i) key k_5 is present in RL and adding it into BL , (ii) key k_5 is not present in RL as well as BL and adding it into RL , (iii) key k_5 is not present in RL as well as BL and adding it into RL as well as BL

$list_del()$: It 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 ($currs[1]$) and changes the BL of $preds[0]$ to $currs[1].BL$ as shown in Line 243 and Line 245 of Algorithm 10 respectively. Figure 3.9 of Section 3.4 shows the deletion of node corresponding to k_5 .

Algorithm 10 *list_del()*: Deletes a node from blue link in underlying hash table at location corresponding to *preds* & *currs*.

```

241: procedure list_del(G_preds[] ↓, G_currs[] ↓)
242:   /* Mark the node⟨obj_id, key⟩ for deletion */
243:   currs[1].marked ← TRUE ;
244:   /* Update the blue link for physical deletion */
245:   preds[0].BL ← currs[1].BL;
246:   return ⟨void⟩;
247: end procedure

```

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

Algorithm 11 *validation()*: First, do the method validation. If its successful then it will do the time order validation for the transaction.

```

248: procedure validation(L_t_id ↓, L_key ↓, G_preds[] ↓, G_currs[] ↓, L_val_type ↓,
   L_op_status ↑)
249:   /* Validate against concurrent updates */
250:   L_op_status ← methodValidation(G_preds[] ↓, G_currs[] ↓);
251:   /* After successful method validation validate the transaction to ensure
   opacity */
252:   if (RETRY ≠ L_op_status) then
253:     L_op_status ← toValidation(L_key ↓, G_curr ↓, L_val_type ↓) ;
254:   end if
255:   return ⟨L_op_status⟩ ;
256: end procedure

```

Algorithm 12 *methodValidation()*: Identify the conflicts among the concurrent methods of the different transactions.

```

257: procedure methodValidation(G_preds[] ↓, G_currs[] ↓)
258:   if ((preds[0].marked) || (currs[1].marked) || (preds[0].BL) ≠
   currs[1] || (preds[1].RL) ≠ currs[0]) then
259:     return ⟨RETRY⟩ ;
260:   else

```

```

261:     return  $\langle OK \rangle$  ;
262:   end if
263: end procedure

```

Algorithm 13 *toValidation()*: Time order validation for each transaction.

```

264: procedure toValidation( $L.t\_id \downarrow, L.key \downarrow, G.currts[] \downarrow, L.val\_type \downarrow$ 
    ,  $L.op\_status \uparrow$ )
265:   /* Get the appropriate shared current node ( $sh\_curr$ ) corresponding to key
    from RL or BL */
266:    $L.rec.getAptCurr(G.currts[] \downarrow, L.key \downarrow, sh\_curr \uparrow)$  ;
267:   /* If  $sh\_curr$  is not NULL and node corresponding to the key is equal to
     $sh\_curr.key$  then check for TS */
268:   if ( $(sh\_curr \neq NULL) \wedge (sh\_curr.key == L.key)$ ) then
269:     /* If val_type is RV then transaction validation for rv_method */
270:     if ( $(L.val\_type = RV) \wedge (TS(L.t\_id) < sh\_curr.max\_ts.insert(k)) \parallel$ 
271:       ( $TS(L.t\_id) < sh\_curr.max\_ts.delete(k)$ )) then
272:        $L.op\_status \leftarrow ABORT$  ;
273:       /* Transaction validation for upd_method */
274:     else if ( $(TS(L.t\_id) < sh\_curr.max\_ts.insert(k)) \parallel TS(L.t\_id) < sh\_curr$ 
    . $max\_ts.delete(k) \parallel TS(L.t\_id) < sh\_curr.max\_ts.lookup(k)$ )) then
275:        $L.op\_status \leftarrow ABORT$  ;
276:     end if
277:   end if
278:   return  $\langle L.op\_status \rangle$  ;
279: end procedure

```

toValidation(): In *toValidation()*, *rv_method* always conflicts with the *upd_method* (as established in conflict notion Section 3.2). If the node corresponding to the *key* is present in the lazyrb-list (Line 268) we compare with timestamp of the transaction that last executed the conflicting method on same *key*. If the current method that invoked the *toValidation()* is *rv_method* then Line 271 handles the case. Otherwise, if the invoking method is *upd_method* then Line 274 handles the case. Figure 3.15 and Figure 3.16 show the execution of *toValidation()*. Here $lu_1(ht, k_1)$ will return *Abort* in Figure 3.16 because $del_2(ht, k_1)$ of T_2 has already updated the timestamp at the node corresponding to k_1 . So, when $lu_1(ht, k_1)$ does its *toValidation()* at Line 274, $TS(T_1) < curr.max_ts.delete(k)$ holds true (since, $T_1 < T_2$) leading to *abort* of T_1

at Line 275. This gives us a equivalent sequential schedule which can be shown co-opaque. Figure 3.15 shows the schedule where no sequential schedule is possible if *toValidation()* is not applied as there is no way to recognize the time-order violation.

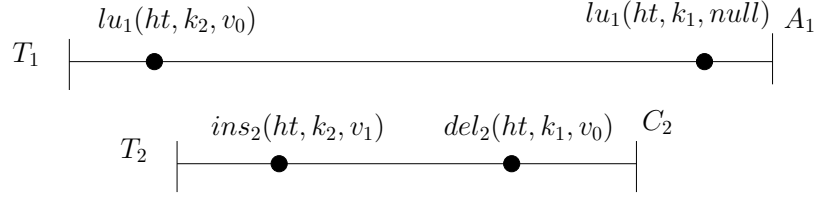


Figure 3.15: Non opaque history. Without timestamp validation

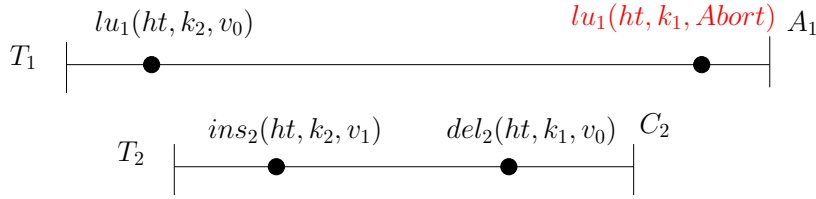


Figure 3.16: Opaque history H1. With timestamp validation

intraTrans Validation(): It handles the case where 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. This happens because the previous method while updating, changes the lazyrb-list causing the *preds* and *currs* of the next method working on the consecutive key to become obsolete. Thus, *intraTrans Validation()* corrects this by finding the new *preds* 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 `preds[0]` (at Line 285 to Line 287 using previous log record) and for case(ii) we find `preds[0]` using previous log record's `preds[0]` (Line 292) and finally find the new `preds[1]` and `currs[0]` between the new found `preds[0]` and `currs[1]` at Line 297 to Line 299.

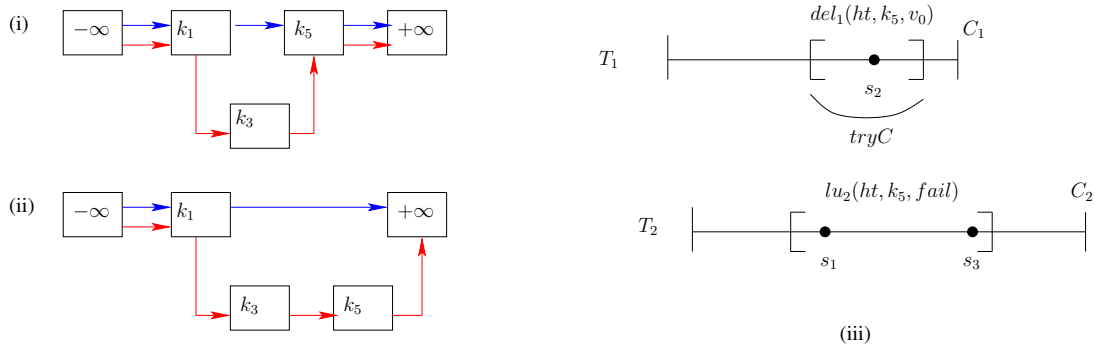


Figure 3.17: *intraTransValidation* for conflicting concurrent methods on key k_5

Algorithm 14 *intraTransValidation()*: Identify the consecutive upd_methods working on same location of the same transaction and update the preds and currs of second upd_method based on previous upd_method.

```

280: procedure intraTransValidation( $L\_rec \downarrow, G\_preds[] \uparrow, G\_currs[] \uparrow$ )
281:    $L\_rec.getAllPreds\&Currs(L\_rec \downarrow, G\_preds[] \uparrow, G\_currs[] \uparrow)$  ;
282:   /* If  $preds[0]$  is marked or  $currs[1]$  is not reachable from  $preds[0].BL$  then
      modify the next consecutive upd_method  $preds[0]$  based on previous upd-
      method */
283:   if ( $(preds[0].marked) || (preds[0].BL) \neq currs[1]$ ) then
284:     /* Find  $k < i$ ; such that  $L\_rec_k$  contains previous update method on
      same bucket */
285:     if ( $(L\_rec_k.L\_opn) == INSERT$ ) then
286:        $L\_rec_i.preds[0].unlock()$  ;
287:        $preds[0] \leftarrow (L\_rec_k.preds[0].BL)$  ;
288:        $L\_rec_i.preds[0].lock()$  ;
289:     else
290:       /* upd_method  $preds[0]$  will be previous method  $preds[0]$  */
291:        $L\_rec_i.preds[0].unlock()$  ;
292:        $preds[0] \leftarrow (L\_rec_k.preds[0])$  ;
293:        $L\_rec_i.preds[0].lock()$  ;
294:     end if
295:   end if
296:   /* If  $currs[0]$  &  $preds[1]$  is modified by previous operation then update them
      also */
297:   if ( $preds[1].RL \neq currs[0]$ ) then
298:      $L\_rec_i.preds[1].unlock()$ 
299:      $preds[1] \leftarrow (L\_rec_k.preds[1].RL)$  ;
300:      $L\_rec_i.preds[1].lock()$ 
301:   end if
302:   return  $\langle G\_preds[], G\_currs[] \rangle$ ;
303: end procedure

```

This can be illustrated with Figure 3.17. Consider the history in Figure 3.17(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 lazyrb-list at that instant. Let at s_1 both the methods record the same $preds\langle k_1, k_3 \rangle$ and $currs\langle k_5, k_5 \rangle$ with the help of *list_lookup()* for key k_5 (refer Figure 3.17(i)). Now, let $del_1(k_5)$ acquire the lock on the *preds* and *currs* before the $lu_2(k_5)$ and delete the node corresponding to the key k_5 from *BL*

leading to state s_2 (in Figure 3.17(iii)) and commit. Figure 3.17(ii) shows the state s_2 where key k_5 is the part of RL . Now, $intraTransValidation()$ (in Algorithm 14) will identify that location of $lu_2(k_5)$ is no more valid due to $(preds[0].BL \neq currs[1])$ at Line 283 of Algorithm 14. Thus, $list_lookup()$ will retry to find the updated location for $lu_2(k_5)$ at state s_3 (in Figure 3.17(iii)) and eventually T_2 will commit.

find(): It is an utility method that returns true to the method that has invoked it, if the calling method is not the first method of the transaction on the *key*. This is done by linearly traversing the log and finding an entry corresponding to the *key*. If the calling method is the first method of the transaction for the *key* then *find* return false as it would not find any entry in the log of the transaction corresponding to the *key*. Since we consider that there can be multiple objects (hash table) so we need to find unique $\langle obj_id, key \rangle$ pair (refer Line 308).

Algorithm 15 *find()*: Checks whether any operation corresponding to $\langle obj_id, key \rangle$ is present in L_list.

```

304: procedure find( $L\_t\_id \downarrow, L\_obj\_id \downarrow, L\_key \downarrow, L\_rec \uparrow$ )
305:   L_list  $\leftarrow L\_txlog.getList(L\_t\_id \downarrow)$  ;
306:   /* Every method first identify the node corresponding to key in local log */
307:   while ( $L\_rec_i \leftarrow next(L\_list)$ ) do
308:     if ( $(L\_rec_i.first == L\_obj\_id) \& \& (L\_rec_i.first == L\_key)$ ) then
309:       return  $\langle TRUE, L\_rec \rangle$  ;
310:     end if
311:   end while
312:   return  $\langle FALSE, L\_rec = NULL \rangle$  ;
313: end procedure

```

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

Algorithm 16 *get_aptcurr()*: Returns a curr node from underlying DS which corresponds to the key of L_rec_i .

```

314: procedure get_aptcurr( $G\_currs[] \downarrow, L\_key \downarrow, sh\_curr \uparrow$ )
315:   /* By default set curr to NULL */
316:   sh_curr  $\leftarrow NULL$ ;

```

```

317:  /* If node corresponding to the key is part of BL then curr is currs[1] */
318:  if (currs[1].key == L_key) then
319:    sh_curr ← currs[1] ;
320:    /* If node corresponding to key is part of RL then curr is currs[0] */
321:  else if (currs[0].key == L_key) then
322:    sh_curr ← currs[0] ;
323:  end if
324:  return ⟨sh_curr⟩ ;
325: end procedure

```

release_ordered_locks(): It is an utility method to release the locks in order.

Algorithm 17 *release_ordered_locks()*: Release all locks taken during *list_lookup()*.

```

326: procedure release_ordered_locks(L_ordered_list ↓)
327:  /* Releasing all the locks on preds, currs and node */
328:  while (L_reci ← next(L_ordered_list)) do
329:    L_reci.preds[0].unlock() ;// $\Phi_{lp}$ 
330:    L_reci.preds[1].unlock() ;
331:    if (L_reci.node) then
332:      L_reci.node.unlock()
333:    end if
334:    L_reci.currs[0].unlock() ;
335:    L_reci.currs[1].unlock() ;
336:  end while
337:  return ⟨void⟩;
338: end procedure

```

Algorithm 18 *acquirePred&CurrLocks()*: Acquire all locks taken during *list_lookup()*.

```

339: procedure acquirePred&CurrLocks(G_preds[] ↓, G_currs[] ↓)
340:  preds[0].lock();
341:  preds[1].lock();
342:  currs[0].lock();
343:  currs[1].lock();
344:  return ⟨void⟩;
345: end procedure

```

Algorithm 19 *releasePred&CurrLocks()*: Release all locks taken during *list-lookup()*.

```

346: procedure releasePred&CurrLocks( $G\_preds[] \downarrow, G\_currs[] \downarrow$ )
347:   preds[0].unlock(); //  $\Phi_{lp}$ 
348:   preds[1].unlock();
349:   currs[0].unlock();
350:   currs[1].unlock();
351:   return  $\langle void \rangle$ ;
352: end procedure

```

Optimization: If *STM_delete()* returns FAIL in *rv_method execution* phase then no need to validate it in *STM_tryC()* (*upd_method execution* phase). This is similar to the case of *STM_lookup()* is being validated once. Since a failed *STM_delete()* method implies that it would not change the underlying hash table and in behaviour it is similar to a lookup as discussed in Figure 3.18.

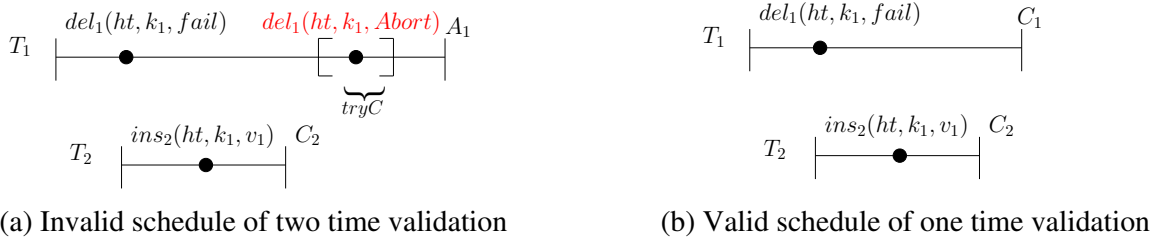


Figure 3.18: Advantage of validating *STM_delete()* once, if its returning FAIL in *rv_method execution* phase

3.6 Correctness of OSTMs

In this section we rigorously prove the correctness of proposed HT-OSTM at operational level and transactional level. We prove that all the methods are linearizable at operational level while the transactions are co-opaque by showing that the corresponding conflict graph is acyclic at transactional level.

3.6.1 Operational Level

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

Definition 1 *PublicNodes*: Which is having a incoming *RL*, except head node.

Definition 2 *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,

$$S.Abs.RL := \{\forall n | (n \in S.PublicNodes) \wedge (S.Head \rightarrow_{RL}^* S.n)\}.$$

$$S.Abs.BL = \{\forall n | (n \in S.PublicNodes) \wedge (\neg S.n.marked) \wedge (S.Head \rightarrow_{BL}^* S.n)\}$$

Observation 1 Consider a global state S which has a node n . Then in any future state S' of S , n is a node in S' as well. Formally, $\langle \forall S, S' : (n \in S.nodes) \wedge (S \sqsubset S') \Rightarrow (n \in S'.nodes) \rangle$.

With Observation 1, we assume that nodes once created do not get deleted (ignoring garbage collection for now).

Observation 2 Consider a global state S which has a node n , initialized with key k . Then in any future state S' the key of n does not change. Formally, $\langle \forall S, S' : (n \in S.nodes) \wedge (S \sqsubset S') \Rightarrow (n \in S'.nodes) \wedge (S.n.key = S'.n.key) \rangle$.

Observation 3 Consider a global state S which is the post-state of return event of the function `list_lookup()` invoked in the `STM_delete()` or `STM_tryC()` or `STM_lookup()` methods. Suppose the `list_lookup()` method returns $(preds[0], preds[1], currs[0], currs[1])$. Then in the state S , we have,

$$3.1 \ (preds[0] \wedge preds[1] \wedge currs[0] \wedge currs[1]) \in S.PublicNodes$$

$$3.2 \ (S.preds[0].locked) \wedge (S.preds[1].locked) \wedge (S.currs[0].locked) \wedge (S.currs[1].locked)$$

$$3.3 \ (\neg S.preds[0].marked) \wedge (\neg S.currs[1].marked) \wedge (S.preds[0].BL = S.currs[1]) \wedge (S.preds[1].RL = S.currs[0])$$

In Observation 3, `list_lookup()` method returns only if validation succeed at Line 206.

Lemma 4 Consider a global state S which is the post-state of return event of the function `list_lookup()` invoked in the `STM_delete()` or `STM_tryC()` or `STM_lookup()` methods. Suppose the `list_lookup()` method returns $(preds[0], preds[1], currs[0], currs[1])$. Then in the state S , we have,

$$4.1 \ ((S.preds[0].key) < key \leq (S.currs[1].key)).$$

$$4.2 \ ((S.preds[1].key) < key \leq (S.currs[0].key)).$$

Proof:

4.1 ($S.preds[0].key < key \leq S.curr[s][1].key$) :

Line 186 of *list_lookup()* method of Algorithm 8 initializes $S.preds[0]$ to point head node. Also, ($S.curr[s][1] = S.preds[0].BL$) by line 188. As in penultimate execution of line 190 ($S.curr[s][1].key < key$) and at line 191 ($S.preds[0] = S.curr[s][1]$) this implies,

$$(S.preds[0].key < key) \tag{3.1}$$

The node key doesn't change as known by Observation 2. So, before executing of line 195, we know that,

$$(key \leq S.curr[s][1].key) \tag{3.2}$$

From eq(3.1) and eq(3.2), we get,

$$(S.preds[0].key < key \leq S.curr[s][1].key) \tag{3.3}$$

From Observation 3.2 and Observation 3.3 we know that these nodes are locked and from Observation 2, we have that key is not changed for a node, so the lemma holds even when *list_lookup()* method of Algorithm 8 returns.

4.2 ($S.preds[1].key < key \leq S.curr[s][0].key$) :

Line 195 of *list_lookup()* method of Algorithm 8 initializes $S.preds[1]$ to point $S.preds[0]$. Also, ($S.curr[s][0] = S.preds[0].RL$) by line 197. As in penultimate execution of line 199 ($S.curr[s][0].key < key$) and at line 200 ($S.preds[1] = S.curr[s][0]$) this implies,

$$(S.preds[1].key < key) \tag{3.4}$$

The node key doesn't change as known by Observation 2. So, before executing of line 204, we know that

$$(key \leq S.curr[s][0].key) \tag{3.5}$$

From eq(3.4) and eq(3.5), we get,

$$(S.preds[1].key < key \leq S.curr[s][0].key) \tag{3.6}$$

From Observation 3.2 and Observation 3.3 we know that these nodes are locked and from Observation 2, we have that key is not changed for a node, so the lemma holds even when *list_lookup()* method of Algorithm 8 returns.

Lemma 5 *For a node n in any global state S , we have that, $\langle \forall n \in S.nodes : (S.n.key < S.n.RL.key) \rangle$.*

Proof: We prove by Induction on events that change the *RL* field of the node (as these affect reachability), which are Lines 227, 228, 234 & 236 of *list_ins()* method of Algorithm 9. It can be seen by observing the code that *list_del()* method of Algorithm 10 do not have any update events of *RL*.

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

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

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 227 of *list_ins()* method:** By observing the code, we notice that Line 227 (*RL* field changing event) can be executed only after the *list_lookup()* method of Algorithm 8 returns. Line 223 of the *list_ins()* method creates a new node, *node* with *key* and at line 226 set the (*S.node.marked* = *true*) (because inserting the node only into the red link). Line 227 then sets (*S.node.RL* = *S.currns[0]*). Since this event doest not change the *RL* field of any node reachable from the head of the list (because *node* \notin *S.PublicNodes*), the lemma is not violated.
2. **Line 228 of *list_ins()* method:** By observing the code, we notice that Line 228 (*RL* field changing event) can be executed only after the *list_lookup()* method of Algorithm 8 returns. From Lemma 4.2, we know that when *list_lookup()* method of Algorithm 8 returns then,

$$(S.preds[1].key) < key \leq (S.currns[0].key) \quad (3.7)$$

To reach line 228 of *list_ins()* method, line 97 of *commonLu&Del()* method of Algorithm 6 should ensure that,

$$(S.currns[0].key \neq key) \xrightarrow{eq(3.7)} (S.preds[1].key) < key < (S.currns[0].key) \quad (3.8)$$

From Observation 3.3, we know that,

$$(S.preds[1].RL = S.curr[s][0]) \quad (3.9)$$

Also, the atomic event at line 228 of *list_ins()* sets,

$$\begin{aligned} (S.preds[1].RL = node) &\xrightarrow{eq(3.8)} (S.preds[1].key < node.key) \\ &\implies (S.preds[1].key < S.preds[1].RL.key) \end{aligned} \quad (3.10)$$

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

3. Line 234 of *list_ins()* method: By observing the code, we notice that Line 234 (*RL* field changing event) can be executed only after the *list_lookup()* method of Algorithm 8 returns. Line 231 of the *list_ins()* method creates a new node, *node* with *key*. Line 234 then sets $(S.node.RL = S.curr[s][0])$. 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 236 of *list_ins()* method: By observing the code, we notice that Line 236 (*RL* field changing event) can be executed only after the *list_lookup()* Algorithm 8 method returns. From Lemma 4.2, we know that when *list_lookup()* method of Algorithm 8 returns then,

$$(S.preds[1].key) < key \leq (S.curr[s][0].key) \quad (3.11)$$

To reach line 236 of *list_ins()* method, line 152 of *STM_tryC()* method of Algorithm 7 should ensure that,

$$(S.curr[s][0].key \neq key) \xrightarrow{eq(3.11)} (S.preds[1].key) < key < (S.curr[s][0].key) \quad (3.12)$$

From Observation 3.3, we know that,

$$(S.preds[1].RL = S.curr[s][0]) \quad (3.13)$$

Also, the atomic event at line 236 of *list_ins()* sets,

$$\begin{aligned} (S.preds[1].RL = node) &\xrightarrow{eq(3.12)} (S.preds[1].key < node.key) \\ &\implies (S.preds[1].key < S.preds[1].RL.key) \end{aligned} \quad (3.14)$$

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

Lemma 6 *In a global state S , any public node n is reachable from Head via red links. Formally, $\langle \forall S, n : n \in S.PublicNodes \implies S.Head \rightarrow_{RL}^* S.n \rangle$.*

Proof: We prove by Induction on events that change the RL field of the node (as these affect reachability), which are Lines 227, 228, 234 & 236 of *list_ins()* method of Algorithm 9. It can be seen by observing the code that *list_del()* method of Algorithm 10 do not have any update events of RL .

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

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

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

1. **Line 227 of *list_ins()* method:** Line 223 of the *list_ins()* method creates a new node, *node* with *key* and at line 226 set the $(S.node.marked = true)$ (because inserting the node only into the red link). Line 227 then sets $(S.node.RL = S.curr[0])$. 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 228 of *list_ins()* method:** By observing the code, we notice that Line 228 (RL field changing event) can be executed only after the *list_lookup()* method of Algorithm 8 returns. From line 227 & 228 of *list_ins()* method, $(S.node.RL = S.curr[0]) \wedge (S.preds[1].RL = S.node) \wedge (node \in S.PublicNodes) \wedge (S.node.marked = true)$ (because inserting the node only into the red link). It is to be noted that (from Observation 3.2), $(preds[0], preds[1], curr[0], curr[1])$ are locked, hence no other thread can change marked field of $S.preds[1]$ and $S.curr[0]$ simultaneously. Also, from Observation 2, a node's key field does not change after initialization. Before executing line 228, $preds[1]$ is reachable from head by RL (from induction hypothesis). After line 228, we know that from $preds[1]$, public marked node, *node* is also reachable. Thus, we know that *node* is also reachable from head. Formally, $(S.Head \rightarrow_{RL}^* S.preds[1]) \wedge (S.preds[1] \rightarrow_{RL}^* S.node) \implies (S.Head \rightarrow_{RL}^* S.node)$.

3. Line 234 of `list_ins()` method: Line 231 of the `list_ins()` method creates a new node, `node` with `key`. Line 234 then sets $(S.node.RL = S.curr[0])$. 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 236 of `list_ins()` method: By observing the code, we notice that Line 236 (`RL` field changing event) can be executed only after the `list_lookup()` method of Algorithm 8 returns. From line 234 & 236 of `list_ins()` method, $(S.node.RL = S.curr[0]) \wedge (S.preds[1].RL = S.node) \wedge (node \in S.PublicNodes) \wedge (node.marked = false)$ (because new node is created by default with unmarked field). It is to be noted that (from Observation 3.2), $(preds[0], preds[1], curr[0], curr[1])$ are locked, hence no other thread can change marked field of `S.preds[1]` and `S.curr[0]` simultaneously. Also, from Observation 2, a node's key field does not change after initialization. Before executing line 236, `preds[1]` is reachable from head by `RL` (from induction hypothesis). After line 236, we know that from `preds[1]`, public unmarked node, `node` is also reachable. Thus, we know that `node` is also reachable from head. Formally, $(S.Head \rightarrow_{RL}^* S.preds[1]) \wedge (S.preds[1] \rightarrow_{RL}^* S.node) \Rightarrow (S.Head \rightarrow_{RL}^* S.node)$.

Corollary 7 *Each node is associated with an unique key, i.e. at any given state S , there cannot be two nodes with same key.*

As every node is reachable by red links and has a strict ordering and from Observation 1 and Observation 2 we get this.

Corollary 8 *Consider the global state S such that for any public node n , if there exists a key strictly greater than $n.key$ and strictly smaller than $n.RL.key$, then the node corresponding to the key does not belong to $S.Abs$. Formally, $\langle \forall S, n, key : S.PublicNodes \wedge (S.n.key < key < S.n.RL.key) \implies node(key) \notin S.Abs \rangle$.*

Observation 9 *Consider a global state S which has a node n is reachable from head via `RL`. Then in any future state S' of S , node n is also reachable from head via `RL` in S' as well. Formally, $\langle \forall S, S' : (n \in S.nodes) \wedge (S \sqsubseteq S') \wedge (S.head \rightarrow_{RL}^* S.n) \Rightarrow (n \in S'.nodes) \wedge (S'.head \rightarrow_{RL}^* S'.n) \rangle$.*

Proof: From Observation 1, 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 red links are preserved during delete update events (refer `list_del()` method of Algorithm 10).

Lemma 10 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 219, 220, 235 & 237 of $list_ins()$ method of Algorithm 9 and Line 245 of $list_del()$ method of Algorithm 10 .

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

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

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

1. **Line 219 & 220 of $list_ins()$ method:** By observing the code, we notice that Line 219 & 220 (BL field changing event) can be executed only after the $list_lookup()$ method of Algorithm 8 returns. From Lemma 4.1 and Lemma 4.2, we know that when $list_lookup()$ method of Algorithm 8 returns then,

$$\begin{aligned} ((S.preds[0].key) < key \leq (S.curr[s][1].key)) \wedge ((S.preds[1].key) < key \\ \leq (S.curr[s][0].key)) \end{aligned} \quad (3.15)$$

To reach line 219 of $list_ins()$ method, line 147 of $STM_tryC()$ method of Algorithm 7 should ensure that,

$$\begin{aligned} (S.curr[s][1].key \neq key) \wedge (S.curr[s][0].key = key) &\xrightarrow{eq(3.15)} \\ ((S.preds[0].key) < key < (S.curr[s][1].key)) & \\ \wedge ((S.preds[1].key) < (key = S.curr[s][0].key)) & \end{aligned} \quad (3.16)$$

From Observation 3.3, we know that,

$$(S.preds[0].BL = S.curr[s][1]) \wedge (S.preds[1].RL = S.curr[s][0]) \quad (3.17)$$

The atomic event at line 219 of $list_ins()$ sets,

$$\begin{aligned} (S.curr[s][0].BL = S.curr[s][1]) &\xrightarrow[Lemma 5]{eq(3.16), Lemma 6} (S.curr[s][0].key) \\ < (S.curr[s][1].key) \implies (S.curr[s][0].key) < (S.curr[s][0].BL.key) & \end{aligned} \quad (3.18)$$

Also, the atomic event at line 220 of *list_ins()* sets,

$$\begin{aligned} & (S.preds[0].BL = S.curr[s][0]) \xrightarrow{eq(3.16)} (S.preds[0].key) \\ & < (S.curr[s][0].key) \implies (S.preds[0].key) < (S.preds[0].BL.key). \end{aligned} \quad (3.19)$$

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

2. Line 235 of *list_ins()* method: By observing the code, we notice that Line 235 (*BL* field changing event) can be executed only after the *list_lookup()* method of Algorithm 8 returns. Line 231 of the *list_ins()* method creates a new node, *node* with *key*. Line 235 then sets $(S.node.BL = S.curr[s][1])$. Since this event does not change the *BL* field of any node reachable from the head of the list (because $node \notin S.PublicNodes$), the lemma is not violated.

3. Line 237 of *list_ins()* method: By observing the code, we notice that Line 237 (*BL* field changing event) can be executed only after the *list_lookup()* method of Algorithm 8 returns. From Lemma 4.1 and Lemma 4.2, we know that when *list_lookup()* method of Algorithm 8 returns then,

$$\begin{aligned} (S.preds[0].key) < key \leq (S.curr[s][1].key) \wedge (S.preds[1].key) < key \\ \leq (S.curr[s][0].key) \end{aligned} \quad (3.20)$$

To reach line 237 of *list_ins()* method, line 152 of *STM_tryC()* method of Algorithm 7 should ensure that,

$$\begin{aligned} & (S.curr[s][0].key \neq key) \wedge (S.curr[s][1].key \neq key) \xrightarrow{eq(3.20)} \\ & (S.preds[0].key) < key < (S.curr[s][1].key) \\ & \wedge (S.preds[1].key) < key < (S.curr[s][0].key) \end{aligned} \quad (3.21)$$

From Observation 3.3, we know that,

$$(S.preds[0].BL = S.curr[s][1]) \quad (3.22)$$

Also, the atomic event at line 237 of *list_ins()* sets,

$$\begin{aligned} & (S.preds[0].BL = S.node) \xrightarrow{eq(3.21)} (S.preds[0].key < S.node.key) \\ & \implies (S.preds[0].key < S.preds[0].BL.key) \end{aligned} \quad (3.23)$$

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 245 of $list_del()$ method:** By observing the code, we notice that Line 245 (BL field changing event) can be executed only after the $list_lookup()$ method of Algorithm 8 returns. From Lemma 4.1, we know that when $list_lookup()$ method of Algorithm 8 returns then,

$$(S.preds[0].key) < key \leq (S.curr[s][1].key) \quad (3.24)$$

To reach line 245 of $list_del()$ method, line 163 of $STM_tryC()$ method of Algorithm 7 should ensure that,

$$(S.curr[s][1].key = key) \xrightarrow{eq(3.24)} (S.preds[0].key) < (key = S.curr[s][1].key) \quad (3.25)$$

From Observation 3.3, we know that,

$$(S.preds[0].BL = S.curr[s][1]) \quad (3.26)$$

We know from Induction hypothesis,

$$(curr[s][1].key < curr[s][1].BL.key) \quad (3.27)$$

Also, the atomic event at line 245 of $list_del()$ sets,

$$(S.preds[0].BL = S.curr[s][1].BL) \xrightarrow{eq(3.25), eq(3.27)} (S.preds[0].key < S.curr[s][1].BL.key) \implies (S.preds[0].key < S.preds[0].BL.key) \quad (3.28)$$

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

Lemma 11 *In a global state S , any unmarked public node n is reachable from Head via blue links. Formally, $\langle \forall S, n : (S.PublicNodes) \wedge (\neg S.n.marked) \implies (S.Head \rightarrow_{BL}^* S.n) \rangle$.*

Proof: We prove by Induction on events that change the BL field of the node (as these affect reachability), which are Line 219, 220, 235 & 237 of $list_ins()$ method of Algorithm 9 and line 245 of $list_del()$ method of Algorithm 10.

Base condition: Initially, before the first event that changes the BL field of any

node, we know that $(head, tail) \in S.PublicNodes \wedge \neg(S.head.marked) \wedge \neg(S.tail.marked) \wedge (S.head \rightarrow_{BL}^* S.tail)$.

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

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

1. **Line 219 & 220 of `list_ins()` method:** By observing the code, we notice that Line 219 & 220 (BL field changing event) can be executed only after the `list_lookup()` method of Algorithm 8 returns. It is to be noted that (from Observation 3.2), $(preds[0], preds[1], currs[0], currs[1])$ are locked, hence no other thread can change $S.preds[0].marked$ and $S.currs[1].marked$ simultaneously. Also, from Observation 2, a node's key field does not change after initialization. Before executing line 219, from Observation 3.3 ,

$$(S.preds[0].marked = false) \wedge (S.currs[1].marked = false) \quad (3.29)$$

And from Lemma 6 and induction hypothesis,

$$(S.Head \rightarrow_{RL}^* S.currs[0]) \wedge (S.Head \rightarrow_{BL}^* S.currs[1]) \quad (3.30)$$

After line 219, we know that from $currs[0]$, public unmarked node, $currs[1]$ is also reachable, implies that,

$$(S.currs[0] \rightarrow_{BL}^* S.currs[1]) \quad (3.31)$$

Also, before executing line 220, from induction hypothesis and Lemma 6 ,

$$(S.Head \rightarrow_{BL}^* S.preds[0]) \wedge (S.Head \rightarrow_{RL}^* S.currs[0]) \quad (3.32)$$

After line 220, we know that from $preds[0]$, public unmarked node (from line 218 of `list_ins()` method), $currs[0]$ is also reachable via BL , implies that,

$$(S.preds[0] \rightarrow_{BL}^* S.currs[0]) \wedge (S.currs[0].marked = false) \quad (3.33)$$

From eq(3.31) and eq(3.33),

$$(S.preds[0] \rightarrow_{BL}^* S.currs[0]) \wedge (S.currs[0] \rightarrow_{BL}^* S.currs[1]) \wedge (S.currs[0].marked = false) \quad (3.34)$$

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

2. Line 235 of *list_ins()* method: Line 231 of the *list_ins()* method creates a new node, *node* with *key*. Line 235 then sets $(S.node.BL = S.currs[1])$. Since this event does not change the *BL* field of any node reachable from the head of the list (because $node \notin S.PublicNodes$), the lemma is not violated.

3. Line 237 of *list_ins()* method: By observing the code, we notice that Line 237 (*BL* field changing event) can be executed only after the *list_lookup()* method of Algorithm 8 returns. It is to be noted that (from Observation 3.2), $(preds[0], preds[1], currs[0], currs[1])$ are locked, hence no other thread can change $S.preds[0].marked$ and $S.currs[1].marked$ simultaneously. Also, from Observation 2, a node's key field does not change after initialization. Before executing line 235, from Observation 3.3,

$$(S.preds[0].marked = false) \wedge (S.currs[1].marked = false) \quad (3.35)$$

And from induction hypothesis,

$$(S.Head \rightarrow_{BL}^* S.currs[1]) \quad (3.36)$$

After line 235, we know that from *node*, public unmarked node, $currs[1]$ is also reachable via *BL*, implies that,

$$(S.node \rightarrow_{BL}^* S.currs[1]) \quad (3.37)$$

Also, before executing line 237, from induction hypothesis,

$$(S.Head \rightarrow_{BL}^* S.preds[0]) \quad (3.38)$$

After line 237, we know that from $preds[0]$, public unmarked node (because new node is created by default with unmarked field), *node* is also reachable via *BL*, implies that,

$$(S.preds[0] \rightarrow_{BL}^* S.node) \wedge (S.node.marked = false) \quad (3.39)$$

From eq(3.37) and eq(3.39),

$$(S.preds[0] \rightarrow_{BL}^* S.node) \wedge (S.node \rightarrow_{BL}^* S.curr[1]) \wedge (S.node.marked = false) \quad (3.40)$$

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

Corollary 12 *All public node n , is reachable from head via blue list is subset of all public node n , is reachable from head via red list. Formally, $\langle \forall S, n : (n \in S.nodes) \wedge (S.head \rightarrow_{BL}^* S.n) \subseteq (S.head \rightarrow_{RL}^* S.n) \rangle$.*

Proof: From Lemma 6 , we know that all public nodes either marked or unmarked are reachable from head by *RL*, also from Lemma 11 we have that all unmarked public nodes are reachable by *BL*. Unmarked public nodes are subset of all public nodes thus the corollary.

Definition 3 *First unlocking point of each successful method is the Linearization Point, 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_begin()*: $(G.cnt++)$ at Line 8 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 105 of *STM_delete()*.
4. *STM_tryC(ht, k, OK/FAIL/ABORT)*: *L_rec_i.preds[0].unlock()* at Line 329 of *releaseOrderedLocks()*. Which is called at Line 174 of *STM_tryC()*.

Lemma 13 *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 timestamp. Formally, $\langle (node(key) \in ([E^H.Post(m_i.LP)].Abs.RL)) \wedge (node.max_ts = TS(T_i)) \rangle$.*

Proof:

1. **For `rv_method`:** By observing the code, each `rv_method` first invokes `list_lookup()` method of Algorithm 8 (line 82 of `commonLu&Del()` method of Algorithm 6). From Lemma 5 & Lemma 10 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 7. So, when the `list_lookup()` is invoked from a method, it returns correct location (`preds[0]`, `preds[1]`, `currs[0]`, `currs[1]`) of corresponding `key` as observed from Observation 3 & Lemma 4 and all are locked, hence no other thread can change simultaneously (from Observation 3.2).

In the pre-state of `LP` event of `rv_method`, if $(node.key \in S.Abs.RL)$, means `key` is already there in `RL` and timestamp of that node is less than the `rv_method` transaction timestamp, from `toValidation()` method of Algorithm 13, then in the post-state of `LP` event of `rv_method`, `node.key` should be the part of `RL` from Observation 9 and `key` can't be changed from Observation 2 and it just updates the `max_ts` field for corresponding node `key` by method transaction timestamp else abort.

In the pre-state of `LP` event of `rv_method`, if $(node.key \notin S.Abs.RL)$, means `key` is not there in `RL` then, in the post-state of `LP` event of `rv_method`, insert the `node` corresponding to the `key` into `RL` by using `list_ins()` method of Algorithm 9 and update the `max_ts` field for corresponding node `key` by method transaction timestamp. Since, `node.key` should be the part of `RL` from Observation 9 and `key` can't be change from Observation 2, in post-state of `LP` event of `rv_method`.

2. **For `upd_method`:** By observing the code, each `upd_method` also first invokes `list_lookup()` method of Algorithm 8 (line 122 of `STM_tryC()` method of Algorithm 7). From Lemma 5 and Lemma 10 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 7. So, when the `list_lookup()` is invoked from a method, it returns correct location (`preds[0]`, `preds[1]`, `currs[0]`, `currs[1]`) of corresponding `key` as observed from Observation 3 & Lemma 4 and all are locked, hence no other thread can change simultaneously (from Observation 3.2).

- (a) **If `upd_method` is `insert`:** In the pre-state of `LP` event of `upd_method`, if $(node.key \in S.Abs.RL)$, means `key` is already there in `RL` and timestamp of that node is less than the `upd_method` transaction timestamp, from `toValidation()` method of Algorithm 13, then in the post-state of `LP` event of `upd_method`, `node.key` should be the part of `BL` and it just update the

max_ts field for corresponding node key by method transaction timestamp else abort.

In the pre-state of LP event of upd_method , if $(node.key \notin S.Abs.RL)$, means key is not there in RL then in the post-state of LP event of upd_method , it will insert the $node$ corresponding to the key into the RL as well as BL , from $list_ins()$ method of Algorithm 9 at line 149 of $STM_tryC()$ method of Algorithm 7 and update the max_ts field for corresponding node key by method transaction timestamp. Once a node is created it will never get deleted from Observation 9 and node corresponding to a key can't be modified from Observation 2.

- (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 timestamp of that node is less then the upd_method transactions timestamp, from $toValidation()$ method of Algorithm 13 , then in the post-state of LP event of upd_method , $node.key$ should be the part of RL , from $list_del()$ method of Algorithm 10 at line 165 of $STM_tryC()$ method of Algorithm 7 and it just update the max_ts field for corresponding node key by method transaction timestamp 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 Algorithm 5 must have already inserted a node in the underlying data structure prior to $STM_tryC()$ method of Algorithm 7 . Thus, $(node.key \in S.Abs.RL)$ and update the max_ts field for corresponding node key by method transaction timestamp 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 $(preds[0], preds[1], currs[0], currs[1])$, 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 14 $intraTransValidation()$ 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 ($preds[0], preds[1], currs[0], currs[1]$) then effect of first upd_method will be overwritten by the next upd_method .

By observing the code at line 171 of $STM_tryC()$ method of Algorithm 7, current upd_method will go for $intraTransValidation()$ and at line 283 of $intraTransValidation()$ method of Algorithm 14, current upd_method will validate its ($preds[0].marked$) and ($preds[0].BL! = currs[1]$). If any condition is true then, at line 285 of $intraTransValidation()$ method of Algorithm 14, will check for previous upd_method . If the previous upd_method is insert then the current upd_method update its $preds[0]$ to previous upd_method , $node.key$ else set current upd_method $preds[0]$ to previous upd_method $preds[0]$.

After that at line 297 of $intraTransValidation()$ method of Algorithm 14, current upd_method validate its ($preds[1].RL! = currs[0]$). If condition is true then current upd_method set its $preds[1]$ to previous upd_method , $node.key$.

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

Observation 15 For any global state S , the $intraTransValidation()$ in $STM_tryC()$ preserves the properties of $list_lookup()$ as proved in Observation 3 and Lemma 4.

Lemma 16 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,

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

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

Proof: By observing the code, each upd_method also first invokes $list_lookup()$ method of Algorithm 8 (line 122 of $STM_tryC()$ method of Algorithm 7). From Lemma 5 and Lemma 10 we have that the nodes in the underlying data structure are in increasing order of there keys, thus the key on which the method is working has a unique location in underlying data structure from Corollary 7. So, when the $list_lookup()$ is invoked from a method, it returns correct location ($preds[0], preds[1], currs[0], currs[1]$) of corresponding key as observed from Observation 3 & Lemma 4 and all are locked, hence no other thread can change simultaneously (from Observation 3.2).

16.1 If `upd_method` is `insert`: In the pre-state of LP event of `upd_method` at Line 139, 147 of $STM_tryC()$ method of Algorithm 7, 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 timestamp, from $toValidation()$ method of Algorithm 13, 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 152 of $STM_tryC()$ method of Algorithm 7, 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 $list_ins()$ method of Algorithm 9 at line 154 of $STM_tryC()$ method of Algorithm 7 and update the $value$ as v . Once a node is created it will never get deleted from Observation 9 and node corresponding to a key can't be modified from Observation 2.

16.2 If `upd_method` is `delete`: In the pre-state of LP event of `upd_method` at Line 163 of $STM_tryC()$ method of Algorithm 7, if $(node.key \in S.Abs.BL)$, means key is already there in BL and timestamp of that node is less then the `upd_method` transactions timestamp, from $toValidation()$ method of Algorithm 13, then in the post-state of LP event of `upd_method`, $node.key$ should not be the part of BL , from $list_del()$ method of Algorithm 10 at line 165 of $STM_tryC()$ method of Algorithm 7.

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 Algorithm 5 must have already inserted a node in the underlying data structure prior to $STM_tryC()$ method of Algorithm 7 .

Lemma 17 Consider a concurrent history, E^H , where S be the pre-state of LP event of successful rv_method (rvm), in that, if node corresponding to the key is the part of BL and $node.val$ is equal to v then, rv_method return OK and value v . Formally, $\langle (node(key) \in ([E^H.Pre(m_i.LP)].Abs.BL)) \wedge (S.node.val = v) \implies rvm(key, OK, v) \rangle$.

Proof: Let the rv_method is $STM_lookup()$ method of Algorithm 4 and it is the first key method of the transaction, we ignore the abort case for simplicity.

From line 82 of $commonLu\&Del()$ method of Algorithm 6, when $list_lookup()$ method of Algorithm 8 returns we have $(preds[0], preds[1], currs[0], currs[1] \in S.PublicNodes)$ and are locked (from Observation 3.1 and Observation 3.2) until $STM_lookup()$ method

of Algorithm 4 return. Also, from Lemma 4.1,

$$(S.preds[0].key < key \leq S.currns[1].key) \quad (3.41)$$

To return OK, $S.currns[1]$ should be reachable from the head via blue list from Definition 2, in the pre-state of LP of rv_method . And after observing code, at line 88 of `commonLu&Del()` method of Algorithm 6,

$$(S.currns[1].key = key) \xrightarrow{eq(3.41)} (S.preds[0].key < (key = S.currns[1].key)) \quad (3.42)$$

Also, from Observation 3.3,

$$(S.preds[0].BL = S.currns[1]) \quad (3.43)$$

And ($currns[1] \in S.nodes$), we know ($currns[1] \in S.Abs.BL$) where S is the pre-state of the LP event of the method. From Lemma 16.1, there should be a prior upd_method which have to be *insert* and $currns[1].val$ is equal to v . Since Observation 2 tells, no node changes its *key* value after initialization. Hence ($node(key) \in ([E^H.Pre(m_i.LP)].Abs.BL) \wedge (S.node.val = v)$).

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

Lemma 18 Consider a concurrent history, E^H , where S be the pre-state of LP event of successful rv_method , in that, if node corresponding to the key is not the part of BL then, rv_method return *FAIL*. Formally, $\langle (node(key) \notin ([E^H.Pre(m_i.LP)].Abs.BL)) \implies rvm(key, FAIL) \rangle$.

Proof: Let the rv_method is $STM_lookup()$ method of Algorithm 4 and it is the first key method of the transaction, we ignore the abort case for simplicity.

1. From line 82 of `commonLu&Del()` method of Algorithm 6, when $list_lookup()$ method of Algorithm 8 returns we have ($preds[0], preds[1], currns[0], currns[1] \in S.PublicNodes$) and are locked (from Observation 3.1 and Observation 3.2) until $STM_lookup()$ method of Algorithm 4 return. Also, from Lemma 4.2,

$$(S.preds[1].key < key \leq S.currns[0].key) \quad (3.44)$$

To return FAIL, $S.currns[0]$ should not be reachable from the head via bluelist from Definition 2, in the pre-state of LP of rv_method . And after observing

code, at line 93 of `commonLu&Del()` method of Algorithm 6,

$$(S.curr[s][0].key = key) \xrightarrow{eq(3.44)} (S.preds[1].key < (key = S.curr[s][0].key)) \quad (3.45)$$

Also, from Observation 3.3,

$$(S.preds[1].RL = S.curr[s][0]) \quad (3.46)$$

And ($curr[s][0] \in S.nodes$), we know ($curr[s][0] \in S.Abs.RL$) where S is the pre-state of the LP event of the method and ($S.curr[s][0].marked = true$). Thus, ($curr[s][0] \notin S.Abs.BL$) from Definition 2. Hence ($node(key) \notin ([E^H.Pre(m_i.LP)].Abs.BL)$)

2. From line 82 of `commonLu&Del()` method of Algorithm 6, when `list_lookup()` method of Algorithm 8 returns we have ($preds[0], preds[1], curr[s][0], curr[s][1] \in S.PublicNodes$) and are locked (from Observation 3.1 and Observation 3.2) until `STM_lookup()` method of Algorithm 4 return. Also, from Lemma 4.2,

$$(S.preds[1].key < key \leq S.curr[s][0].key) \quad (3.47)$$

And after observing code, at line 97 of `commonLu&Del()` method of Algorithm 6,

$$(S.curr[s][1].key \neq key) \wedge (S.curr[s][0].key \neq key) \xrightarrow{eq(3.47)} (S.preds[1].key < key < S.curr[s][0].key) \quad (3.48)$$

Also, from Observation 3.3,

$$(S.preds[1].RL = S.curr[s][0]) \quad (3.49)$$

From eq(3.48), we can say that, ($node(key) \notin S.Abs$) and from Corollary 8, we conclude that $node(key)$ not in the state after `list_lookup()` returns. Since Observation 2 tells, no node changes its key value after initialization. Hence ($node(key) \notin ([E^H.Pre(m_i.LP)].Abs.BL)$).

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

Observation 19 *Only the successful `STM_tryC()` method working on the key k can update the `Abs.BL`.*

By observing the code, only the successful $STM_tryC()$ method of Algorithm 7 is changing the BL . There is no line which is changing the BL in $STM_delete()$ method of Algorithm 5 and $STM_lookup()$ method of Algorithm 4. Such that rv_method is not changing the BL .

Observation 20 *If $STM_tryC()$ and rv_method want to update Abs on the key k , then first it has to acquire the lock on the node corresponding to the key k .*

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

Observation 21 *Two concurrent conflicting methods of different transaction can't acquire the lock on the same node corresponding to the key k simultaneously.*

Observation 22 *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 $ul(m_i(k))$ (unlocking of $m_i(k)$) happen before the $l(m_j(k))$ (locking of $m_j(k)$) then $LP(m_i)$ happen before $LP(m_j)$. Formally, $\langle (ul(m_i(k)) \prec l(m_j(k))) \Rightarrow (LP(m_i) \prec LP(m_j)) \rangle$*

If two concurrent conflicting methods are working on the same key k and want to update Abs then they have to acquire the lock on the node corresponding to the key k from Observation 20 and one of them succeed from Observation 21. If $ul(m_i(k))$ happen before the $l(m_j(k))$ then from Definition 3, $LP(m_i)$ happen before the $LP(m_j)$.

Lemma 23 *Consider two state, S_1, S_2 s.t. $S_1 \sqsubset S_2$ and $S_1.BL.value(k) \neq S_2.BL.value(k)$ then there exist S' s.t. $S' \sqsubset S_2$ and S' contain the $STM_tryC()$ method on the same key k . Formally, $\langle (S_1.BL.value(k) \neq (S_2.BL.value(k)) \Rightarrow \exists(S' \text{ s.t.}, S_1.BL \prec S'.LP(STM_tryC())) \prec S_2.BL) \rangle$. 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 19, 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 20. Such that, $l(STM_tryC(k))$ happen before the $l(S_2(k))$ from Observation 21, then, $ul(STM_tryC(k))$ happen before the $l(S_2(k))$ then $LP(STM_tryC())$ happen before the $LP(S_2)$ from Observation 22.

Lemma 24 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,

24.1 If in 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 .

24.2 If in 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 in $STM_tryC()$ would be delete on same key k and it should be the previous closest to the rv_method .

Proof:

24.1 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 $STM_tryC()$ method should having the last upd_method as insert on the same key k from Corollary 7, 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 23, there should be some other transaction $STM_tryC()$ method working on the same key k and its LP should lies in between these two methods LP . Therefore that intermediate $STM_tryC()$ should be the previous closest method for the rv_method and it will return the same value as previous closest method inserted.

24.2 For proving this we are taking contradiction that previous closest successful $STM_tryC()$ 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 $STM_tryC()$ method, node corresponding to the key k should be the part of BL from Lemma 16.1. 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 $STM_tryC()$ method should not having last upd_method as insert on the same key k . Hence contradiction.

Corollary 25 The sequential history generated by HT-OSTM at operation level is legal.

Corollary 26 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 there LP order for returning the same *return value*.

Lemma 27 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 timestamp. Formally, $\langle (node(key) \in (P.Abs.RL)) \wedge (P.node.max_ts = TS(T_i)) \rangle$, where P is the post-state of the method.

Proof:

1. **For rv_method :** By observing the code, each rv_method first invokes $list_lookup()$ method of Algorithm 8 (line 82 of $commonLu\&Del()$ method of Algorithm 6). From Lemma 5 and Lemma 10 we have that the nodes in the underlying data structure are in increasing order of there keys, thus the key on which the method is working has a unique location in underlying data structure from Corollary 7. So, when the $list_lookup()$ is invoked from a method, it returns correct location ($preds[0], preds[1], currs[0], currs[1]$) of corresponding key as observed from Observation 3 & Lemma 4 and all are locked, hence no other thread can change simultaneously (from Observation 3.2).

In the pre-state of rv_method , if $(node.key \in S.Abs.RL)$, means key is already there in RL and timestamp of that node is less then the rv_method transactions timestamp, from $toValidation()$ method of Algorithm 13, then in the post-state of rv_method , $node.key$ should be the part of RL from Observation 9 and key can not be change from Observation 2 and it just update the max_ts field for corresponding node key by method transaction timestamp else abort.

In the pre-state of rv_method , if $(node.key \notin S.Abs.RL)$, means key is not there in RL then, in the post-state of rv_method , insert the $node$ corresponding to the key into RL by using $list_ins()$ method of Algorithm 9 and update the max_ts field for corresponding node key by method transaction timestamp. Since, $node.key$ should be the part of RL from Observation 9 and key can not be change from Observation 2, in post-state of rv_method .

2. **For upd_method :** By observing the code, each upd_method also first invokes $list_lookup()$ method of Algorithm 8 (line 122 of $STM_tryC()$ method of Algorithm 7). From Lemma 5 and Lemma 10 we have that the nodes in the underlying data structure are in increasing order of there keys, thus the key on

which the method is working has a unique location in underlying data structure from Corollary 7. So, when the *list_lookup()* is invoked from a method, it returns correct location (*preds*[0], *preds*[1], *currs*[0], *currs*[1]) of corresponding *key* as observed from Observation 3 & Lemma 4 and all are locked, hence no other thread can change simultaneously (from Observation 3.2).

- (a) **If *upd_method* is *insert*:** In the pre-state of *upd_method*, if ($node.key \in S.Abs.RL$), means *key* is already there in *RL* and timestamp of that node is less then the *upd_method* transactions timestamp, from *toValidation()* method of Algorithm 13, then in the post-state of *upd_method*, *node.key* should be the part of *BL* and it just update the *max_ts* field for corresponding node *key* by method transaction timestamp else abort.

In the pre-state of *upd_method*, if ($node.key \notin S.Abs.RL$), means *key* is not there in *RL* then in the post-state of *upd_method*, it will insert the *node* corresponding to the *key* into the *RL* as well as *BL*, from *list_ins()* method of Algorithm 9 at line 154 of *STM_tryC()* method of Algorithm 7 and update the *max_ts* field for corresponding node *key* by method transaction timestamp. Once a node is created it will never get deleted from Observation 9 and node corresponding to a key can not be modified from Observation 2.

- (b) **If *upd_method* is *delete*:** In the pre-state of *upd_method*, if ($node.key \in S.Abs.RL$), means *key* is already there in *RL* and timestamp of that node is less then the *upd_method* transactions timestamp, from *toValidation()* method of Algorithm 13, then in the post-state of *upd_method*, *node.key* should be the part of *RL*, from *list_del()* method of Algorithm 10 at line 165 of *STM_tryC()* method of Algorithm 7 and it just update the *max_ts* field for corresponding node *key* by method transaction timestamp else abort.

In the pre-state of *upd_method*, ($node.key \notin S.Abs.RL$) this should not be happen because execution of *STM_delete()* method of Algorithm 5 must have already inserted a node in the underlying data structure prior to *STM_tryC()* method of Algorithm 7. Thus, ($node.key \in S.Abs.RL$) and update the *max_ts* field for corresponding node *key* by method transaction timestamp else abort.

Corollary 28 *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 timestamp.*

3.6.2 Transactional Level

From Section 3.6.1 we are guaranteed to have a linearizable history. Now we prove that such linearizable history obtained from *HT-OSTM* is opaque.

Observation 29 *H* is a sequential history obtained from *HT-OSTM*, as shown at operational level using *LP*.

Definition 4 $CG(H)$ is a conflict graph of *H*.

Lemma 30 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 there exist a cycle path in the form $(T_1, T_2 \cdots T_k, T_1)$, ($k \geq 1$). So, transitively,

$$\begin{aligned} ((T_1.lastEvt \prec_H T_k.firstEvt) \wedge (T_k.lastEvt \prec_H T_1.firstEvt)) \Rightarrow \\ (T_1.lastEvt \prec_H T_1.firstEvt) \end{aligned} \quad (3.50)$$

This contradict our assumption as eq(3.50) is impossible, from definition of program order of a transaction. Thus, cycle is not possible in serial history.

Observation 31 H_2 is an history generated by applying topological sort on $CG(H_1)$.

Observation 32 Topological sort maintains conflict-order and real-time order of the original history H_1 .

Definition 5 $conflict(H)$ is a set of ordered pair (T_i, T_j) , such that there exists conflicting methods m_i, m_j in T_i and T_j respectively, such that $m_i \prec_H^{MR} m_j$. And it is represented as \prec_H^{CO} .

Lemma 33 H_1 is legal and $CG(H_1)$ is acyclic. then,

33.1 H_1 is equivalent to $H_2 \Rightarrow (methods(H_1) = methods(H_2))$.

33.2 $\prec_{H_1}^{CO} \subseteq \prec_{H_2}^{CO}$. i.e. H_1 preserves the conflicts of H_2

Proof: Lemma 33.2

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

Lets assume that there exists a conflict (T_i, T_j) in $\prec_{H_1}^{CO}$ but not in $\prec_{H_2}^{CO}$. But, from

Observation 31 and Observation 32 we know that $(T_i, T_j) \in \prec_{H_2}^{CO}$. Thus, $\prec_{H_1}^{CO} \subseteq \prec_{H_2}^{CO}$.

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

Lemma 34 *Let H_1 and H_2 be equivalent histories such that $\prec_{H_1}^{CO} \subseteq \prec_{H_2}^{CO}$. Then, H_1 is legal $\implies H_2$ is legal.*

Proof: We know H_1 is legal, without loss of generality, let us say $(rvm_j(ht, k, v) \in \text{methods}(H_1))$, such that $(upd_p(ht, k, v_p) = H_1.lastUpdt(rvm_j(ht, k, v)))$ where, $(v = v_p \neq nil)$, if $(upd_p(ht, k, v_p) = STM.insert_p(ht, k, v_p))$ or $(v = nil)$, if $(upd_p(ht, k, v_p) = STM.delete_p(ht, k, v_p))$. From the *conflict-notion* $\text{conflict}(H_1)$ has,

$$upd_p(ht, k, v_p) \prec_{H_1}^{MR} rvm_j(ht, k, v) \quad (3.51)$$

Let us assume H_2 is not legal. Since, H_1 is equivalent to H_2 from Lemma 33.1 such that $(rvm_j(ht, k, v) \in \text{methods}(H_2))$. Since H_2 is not legal, there exist a $(upd_r(ht, k, v_r) \in \text{methods}(H_2))$ such that $(upd_r(ht, k, v_r) = H_2.lastUpdt(rvm_j(ht, k, v)))$. So $\text{conflict}(H_2)$ has,

$$upd_r(ht, k, v_r) \prec_{H_2}^{MR} rvm_j(ht, k, v) \quad (3.52)$$

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

$$upd_p(ht, k, v_p) \prec_{H_2}^{MR} rvm_j(ht, k, v) \quad (3.53)$$

From Lemma 33.1 $(upd_r(ht, k, v_r) \in \text{methods}(H_1))$. Since H_1 is legal $upd_r(ht, k, v_r)$ can occur only in one of following *conflicts*,

$$upd_r(ht, k, v_r) \prec_{H_1}^{MR} upd_p(ht, k, v_p) \quad (3.54)$$

or

$$rvm_j(ht, k, v) \prec_{H_1}^{MR} upd_r(ht, k, v_r) \quad (3.55)$$

In H_1 eq(3.55) is not possible, because if $(eq(3.55) \in \text{conflict}(H_1))$ implies $(eq(3.55) \in \text{conflict}(H_2))$ from $(\prec_{H_1}^{CO} \subseteq \prec_{H_2}^{CO})$ and in H_2 eq(3.52) and eq(3.55) cannot occur together. Thus only possible way $upd_r(ht, k, v_r)$ can occur in H_1 is via eq(3.54). From eq(3.54) we have,

$$upd_r(ht, k, v_r) \prec_{H_2}^{MR} upd_p(ht, k, v_p) \quad (3.56)$$

From eq(3.52), eq(3.53) and eq(3.56) we have,

$$upd_r(ht, k, v_r) \prec_{H_2}^{MR} upd_p(ht, k, v_p) \prec_{H_2}^{MR} rvm_j(ht, k, v)$$

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

Observation 35 *Each transaction is assigned a unique timestamp in `STM_begin()` method using a shared counter which always increases atomically.*

Observation 36 *Each successful method of a transaction is assigned the timestamp of its own transaction.*

Lemma 37 *Consider a global state S which has a node n , initialized with max_ts . Then in any future state S' the max_ts of n should be greater then or equal to S . Formally, $\langle \forall S, S' : (n \in S.Abs) \wedge (S \sqsubset S') \Rightarrow (n \in S'.Abs) \wedge (S.n.max_ts \leq S'.n.max_ts) \rangle$.*

Proof: We prove by Induction on events that change the max_ts field of a node associated with a key, which are Line 90, 95 & 101 of `commonLu&Del()` method of Algorithm 6 and Line 145, 151, 156 & 167 of `STM_tryC()` method of Algorithm 7.

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

Lets assume, a node corresponding to the key is already the part of underlying RL which is having a timestamp of m_1 as T_1 from Observation 36. Let say m_2 of T_2 wants to perform on that node, by observing the code at line 6 of `toValidation()` method of Algorithm 13, if timestamp $TS(T_2) < curr.max_ts.m_1()$, T_2 will return abort, else to succeed, $TS(T_2) > curr.max_ts.m_1()$ should evaluate to true. Thus, for successful completion of m_2 of T_2 , $TS(T_2)$ should be greater then the $TS(T_1)$. Hence, node corresponding to the key, max_ts field should be updated in increasing order of TS values.

Induction Hypothesis: Say, upto k events that change the max_ts field of a node associated with a key always in increasing TS value.

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

1. Line 90, 95 & 101 of `commonLu&Del()` method of Algorithm 6: By observing the code, line 69 of `commonLu&Del()` method of Algorithm 6 first invokes `list_lookup()` method of Algorithm 8 for finding the node corresponding to the key. Inside the `list_lookup()` method of Algorithm 8, it will do the `toValidation()` method of Algorithm 13, if $(curr.key = key)$.

From induction hypothesis, node corresponding to the key is already the part of underlying *RL* which is having a timestamp of m_k of T_k from Observation 36. Let say m_{k+1} of T_{k+1} wants to perform on that node, by observing the code at Line 271 of *toValidation()* method of Algorithm 13, if $TS(T_{k+1}) < curr.max_ts.m_k()$, T_{k+1} will return abort, else to succeed, $TS(T_{k+1}) > curr.max_ts.m_k()$ should evaluate to true. Thus, for successful completion of m_{k+1} of T_{k+1} , $TS(T_{k+1})$ should be greater then the $TS(T_k)$. Hence, node corresponding to the key, *max_ts* field should be updated in increasing order of TS values.

2. Line 145, 151, 156 & 167 of *STM_tryC()* method of Algorithm 7: By observing the code, line 122 of *STM_tryC()* method of Algorithm 7 first invokes *list_lookup()* method of Algorithm 8 for finding the node corresponding to the key. Inside the *list_lookup()* method of Algorithm 8, it will do the *toValidation()* method of Algorithm 13, if (*curr.key = key*).

From induction hypothesis, node corresponding to the key is already the part of underlying *RL* which is having a timestamp of m_k as T_k from Observation 36. Let say m_{k+1} of T_{k+1} wants to perform on that node, by observing the code at Line 274 of *toValidation()* method of Algorithm 13, if $TS(T_{k+1}) < curr.max_ts.m_k()$, T_{k+1} will return abort, else to succeed, $TS(T_{k+1}) > curr.max_ts.m_k()$ should evaluate to true. Thus, for successful completion of m_{k+1} of T_{k+1} , $TS(T_{k+1})$ should be greater then the $TS(T_k)$. Hence, node corresponding to the key, *max_ts* field should be updated in increasing order of TS values.

Corollary 38 *Every successful methods update the max_ts field of a node associated with a key always in increasing TS values.*

Lemma 39 *If $STM_begin(T_i)$ occurs before $STM_begin(T_j)$ then $TS(T_i)$ precedes $TS(T_j)$. Formally, $\langle \forall T \in H : (STM_begin(T_i) \prec STM_begin(T_j)) \Leftrightarrow (TS(T_i) < TS(T_j)) \rangle$.*

Proof: (Only if) If $(STM_begin(T_i) \prec STM_begin(T_j))$ then $(TS(T_i) < TS(T_j))$. Lets assume $(TS(T_j) < TS(T_i))$. From Observation 35,

$$STM_begin(T_j) \prec_H STM_begin(T_i) \tag{3.57}$$

but we know that,

$$STM_begin(T_j) \succ_H STM_begin(T_i) \tag{3.58}$$

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 35,

$$TS(T_j) < TS(T_i) \quad (3.59)$$

but we know that,

$$TS(T_j) > TS(T_i) \quad (3.60)$$

Again, a contradiction.

Lemma 40 *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,

$$T_i.lastEvt \prec_H T_j.firstEvt \quad (3.61)$$

And from program order of T_i ,

$$T_i.firstEvt \prec_H T_i.lastEvt \Rightarrow STM_begin(T_i) \prec_H T_i.lastEvt \quad (3.62)$$

From eq(3.61) and eq(3.62) implies that,

$$\begin{aligned} T_i.firstEvt \prec_H T_j.firstEvt \Rightarrow STM_begin(T_i) \prec_H STM_begin(T_j) \\ \xrightarrow{\text{Lemma 39}} TS(T_i) < TS(T_j) \end{aligned} \quad (3.63)$$

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 5 we get, $m_i \prec_H^{MR} m_j$.

m_i can be *rv_methods* or *upd_methods* (which are taking the effects in *STM-tryC()* method of Algorithm 7) and we know that after the *LP* of m_i of T_i , *node* corresponding to the *key* should be there in *RL* (from Corollary 28 & Definition 2) and the timestamp of that *node* corresponding to *key* should be equal to timestamp of this method transaction timestamp from Corollary 28 and Observation 36 .

From Lemma 5 and Lemma 10 we have that the nodes in the underlying data structure are in increasing order of there keys, thus the key on which the operation is working has a unique location in underlying data structure from Corollary 7. So, when the *list_lookup()* is invoked from a method m_j of T_j , it

returns correct location ($preds[0], preds[1], currs[0], currs[1]$) of corresponding *key* as observed from Observation 3 and Lemma 4.

Now, m_j similar to m_i take effect on the same node represented by key k (from Observation 2 & Corollary 7) and from Observation 9 we know that the *node* corresponding to the key k is still reachable via *RL*. Thus, we know that T_i and T_j will work on same node with key k .

By observing the code at Line 271 and Line 274 of *toValidation()* method of Algorithm 13, we know since, $TS(T_j) < curr.max.ts.m_i()$, T_j will return abort from Corollary 38. In Algorithm 13 for *toValidation()* to succeed, $TS(T_j) > curr.max.ts.m_i()$ should evaluate to true from Corollary 38. 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 41 *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 40 ($TS(T_1) < TS(T_2)$).

Induction Hypothesis: The claim holds for a path of length $(n - 1)$. That is,

$$TS(T_1) < TS(T_2) < \cdots < TS(T_{n-1}) \quad (3.64)$$

Induction Step: Let T_n is a transaction in a path of length n . Then, (T_{n-1}, T_n) is path in $CG(H)$. Thus, it follows from Lemma 40 that,

$$TS(T_{n-1}) < TS(T_n) \xrightarrow{eq(3.64)} (TS(T_1) < TS(T_2) < \cdots < TS(T_n)) \quad (3.65)$$

Hence, the lemma.

Theorem 42 *$CG(H)$ is acyclic.*

Proof: Assume that $CG(H)$ is cyclic, then there exist a cycle say of form $(T_1, T_2 \cdots T_n, T_1)$, for all $(n \geq 1)$. From Lemma 41,

$$TS(T_1) < TS(T_2) \cdots < TS(T_n) < TS(T_1) \Rightarrow TS(T_1) < TS(T_1) \quad (3.66)$$

But, this is impossible as each transaction has unique timestamp, refer Observation 35. Hence the theorem.

Theorem 43 *A legal history H is co-opaque iff $CG(H)$ is acyclic.*

Proof: (Only if) If H is co-opaque and legal, then $CG(H)$ is acyclic: Since H is co-opaque, there exists a legal t-sequential history S equivalent to \bar{H} and S respects \prec_H^{RT} and \prec_H^{CO} (from co-opacity [18]). 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 30. Any sub graph of an acyclic graph is also acyclic. Hence $CG(H)$ is also acyclic.

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

Since every two operations related by the conflict relation in S are also related by $\prec_{\bar{H}}^{CO}$, we obtain $\prec_{\bar{H}}^{CO} \subseteq \prec_S^{CO}$. Since H is legal, \bar{H} is also legal. Combining this with Lemma 34, We get that S is also legal. This satisfies all the conditions necessary for H to be co-opaque.

3.7 Experimental Evaluations

This section describes the experimental analysis of proposed OSTMs with state-of-the-art STMs. We have two main goals in this section: (1) Evaluate the benefit of proposed OSTMs over the state-of-the-art Object-based STMs, and (2) Analyze the benefit of proposed OSTMs over Multi-Version and Single-Version Read-Write STMs.

Experimental system: The Experimental system is a large-scale 2-socket Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz with 14 cores per socket and two hyper-threads (HTs) per core, for a total of 56 threads. Each core has a private 32KB L1 cache and 256 KB L2 cache (which is shared among HTs on that core). All cores on a socket share a 35MB L3 cache. The machine has 32GB of RAM and runs Ubuntu 16.04.2 LTS. All code was compiled with the GNU C++ compiler (G++) 5.4.0 with the build target x86_64-Linux-gnu and compilation option `-std=c++1x -O3`. We consider the same experimental setup in Chapter 4 and Chapter 5 as well.

STM implementations: To show the performance of proposed OSTM against state-of-the-art STMs, we have taken the implementation of Norec-list [6], Boosting-list [13], Trans-list [12], ESTM [7], and RWSTM [3] directly from the TLDS framework¹. We implemented our algorithms in C++. We use *Counter Application* defined in SubSection 3.7.1 where each STM algorithm first creates N -threads, each thread, in turn, spawns a transaction. Each transaction exports *STM_begin()*, *STM_insert()*,

¹TLDS Framework: <https://ucf-cs.github.io/tlds/>

STM_lookup(), *STM_delete()* and *STM_tryC()* methods as described in Section 3.4.

Methodology:² We have considered two types of workloads: (1) Lookup Intensive (70% lookup, 10% insert, 20% delete) and (2) Update Intensive (50% lookup, 25% insert, 25% delete) workloads. The experiments are conducted by varying number of threads from 2 to 64 in power of 2, with 1000 keys randomly chosen. We assume that the hash table of *HT-OSTM* has five buckets and each of the bucket (or list in case of list-OSTM) can have a maximum size of 1000 keys. Each transaction, in turn, executes 10 operations which include *STM_lookup()*, *STM_delete()*, and *STM_insert()* operations. For accuracy, we take an average over 10 results for the final result in which the first run is discarded and considered as a warm-up result for each experiment.

3.7.1 Pseudocode of Counter Application

To analyze the absolute benefit of OSTM, we use a *Counter Application* which provides us the flexibility to create a high contention environment. In this subsection we describe the high-level overview of *Counter Application* though pseudocode as follows:

Algorithm 20 *main()*: The main function invoked by *Counter Application*.

```

353: procedure main()
354:   /*Each thread  $th_i$  log abort counts and average time taken by each transaction
      to commit in  $abortCount_{th_i}$  and  $timeTaken_{th_i}$  respectively;*/
355:   for all (numOfThreads) do /*Multiple threads call the helper function*/
356:     helperFun();
357:   end for
358:   for all (numOfThreads) do
359:     /*Join all the threads*/
360:   end for
361:   for all (numOfThreads) do
362:     /*Calculate the Total Abort Count*/
363:      $totalAbortCount += abortCount_{th_i}$ ;
364:     /*Calculate the Average Time Taken*/
365:      $AvgTimeTaken /= TimeTaken_{th_i}$ ;
366:   end for
367: end procedure

```

²Proposed OSTM code is available here: <https://github.com/PDCRL/HT-OSTM>

Algorithm 21 *helperFun()*: Multiple threads invoke this function.

```
368: procedure helperFun()
369:   Initialize the Transaction Count  $txCount_i$  of  $T_i$  as 0;
370:   /*Execute until number of transactions are non zero*/
371:   while (numOfTransactions) do
372:      $startTime_{th_i} \leftarrow \text{timeRequest}()$ ; /*get the start time of thread  $th_i$ */
373:     /*Execute the transactions  $T_i$  by invoking testSTM functions;*/
374:      $abortCount_{th_i} \leftarrow \text{testSTM}_i()$ ;
375:     Increment the  $txCount_i$  of  $T_i$  by one.
376:      $endTime_{th_i} \leftarrow \text{timeRequest}()$ ; /*get the end time of thread  $th_i$ */
377:     /*Calculate the Total Time Taken by each thread  $th_i$ */
378:      $timeTaken_{th_i} += (endTime_{th_i} - startTime_{th_i})$ ;
379:     Atomically, decrement the numOfTransactions;
380:   end while
381:   /*Calculate the Average Time taken by each thread  $th_i$ */
382:    $TimeTaken_{th_i} /= txCount_i$ ;
383: end procedure
```

Algorithm 22 *testSTM_i()*: Main function which executes the methods of the transaction T_i (or i) by thread th_i .

```
384: procedure testSTMi()
385:   while (true) do
386:      $STM\_begin()$ ; /*Get the unique timestamp  $i$  of each transaction  $T_i$ */
387:     for all (numOfMethods) do
388:        $k_i \leftarrow \text{rand}()\%totalKeys$ ; /*Select the key randomly*/
389:        $m_i \leftarrow \text{rand}()\%100$ ; /*Select the method randomly*/
390:       switch ( $m_i$ ) do
391:         case ( $m_i \leq STM\_lookup()$ ):
392:            $v \leftarrow STM\_lookup(k_i)$ ; /*Lookup key  $k$  from a shared memory*/
393:           if ( $v == abort$ ) then
394:              $txAbortCount_i ++$ ; /*Increment the transaction abort
count*/
395:             goto Line 386;
396:           end if
397:         case ( $STM\_lookup() < m_i \leq STM\_insert()$ ):
398:           /*Insert key  $k_i$  into  $T_i$  local memory with value  $v$ */
```

```

399:         STM_insert( $k_i, v$ );
400:     case (STM_insert() <  $m_i \leq$  STM_delete());
401:         /*Actual deletion happens after successful STM_tryC()*/
402:         STM_delete( $k_i$ );
403:     case default:
404:         /*Neither lookup nor insert/delete on shared memory*/
405:          $v \leftarrow$  STM_tryC(); /*Validate all the methods of  $T_i$  in tryC*/
406:         if ( $v == abort$ ) then
407:             txAbortCount $_i$  ++;
408:             goto Line 386;
409:         end if
410:     end for
411:     return  $\langle txAbortCount_i \rangle$ ;
412: end while
413: end procedure

```

3.7.2 Result Analysis

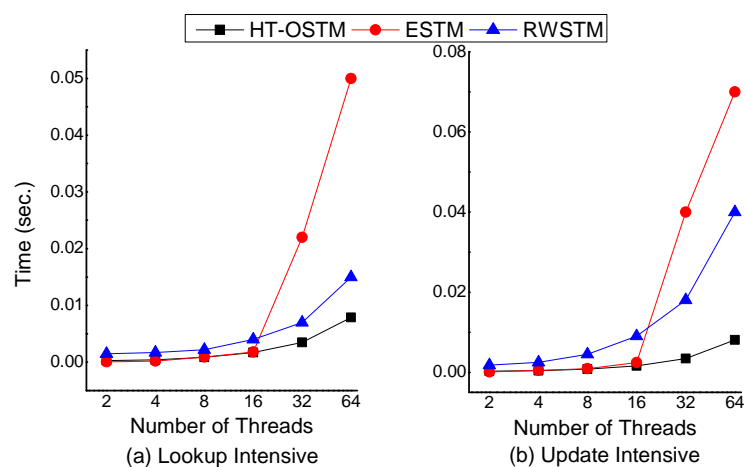


Figure 3.19: Performance of *HT-OSTM* against State-of-the-art hash table based STMs

Experimental analysis of *HT-OSTM* outperforms state-of-the-art hash table based STMs (ESTM [7], RWSTM [3]) by a factor of 3.8, 2.2 for lookup intensive workload and by a factor of 6.7, 5.3 for update intensive workload respectively as demonstrated in Figure 3.19. ESTM and RWSTM work on lower level which has false conflicts as explained in Figure 1.1 of Chapter 1 whereas *HT-OSTM* works on higher level which

ignores the false conflicts and provides greater concurrency. Initially, the performance of HT-OSTM and ESTM as same (upto 16 threads) but as the number of threads increases, the performance HT-OSTM is far better than ESTM.

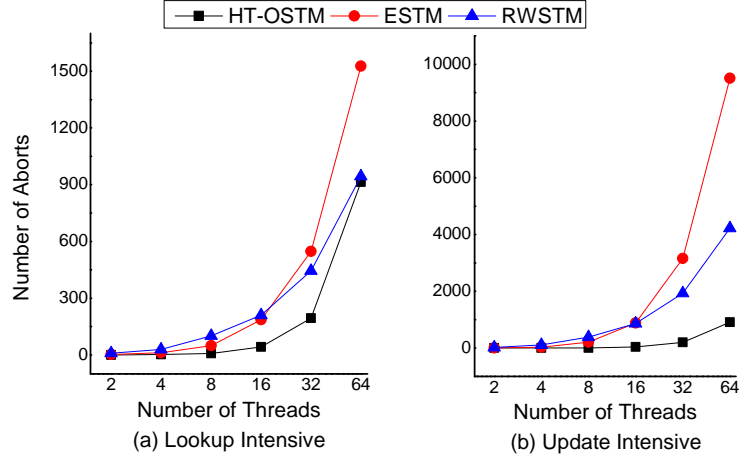


Figure 3.20: Abort Count of *HT-OSTM* against State-of-the-art hash table based STMs

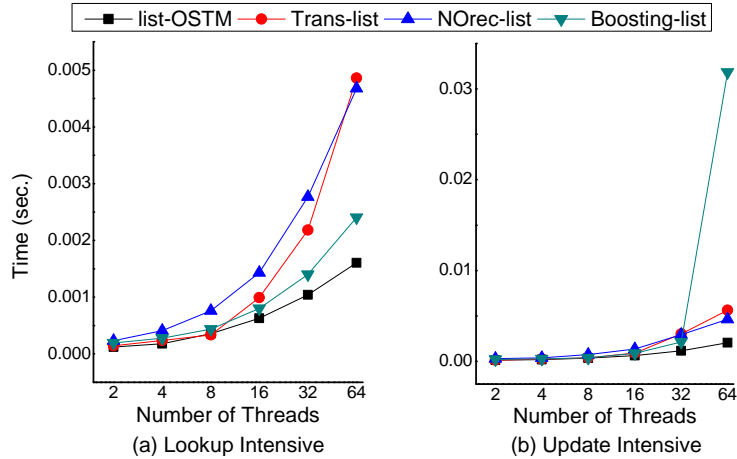


Figure 3.21: Performance of *list-OSTM* against State-of-the-art list based STMs

The number of aborts are directly proportional to the time taken by transactions to commit. Figure 3.20 represents that HT-OSTM has lesser number of aborts by a factor of 3, 2 for lookup intensive workload and by a factor of 7, 8 for update intensive workload with state-of-the-art hash table based STMs (ESTM [7] and RWSTM [3]) respectively.

Experimental analysis of *list-OSTM* outperforms state-of-the-art list based STMs (Trans-list [12], NOrec-list [6] and Boosting-list [13]) by a factor of 1.76, 1.89, 1.33 for lookup intensive workload and by a factor of 1.77, 1.77, 2.54 for update intensive

workload respectively as demonstrated in Figure 3.21. NOrec-list works on lower level which has some false conflicts whereas Trans-list, Boosting-list, and HT-OSTM works on higher level which do not have false conflicts. Boosting-list is a pessimistic approach which rollback the transaction on inconsistency whereas Trans-list is an optimistic approach in which rollbacks are not required. So, boosting-list is performing better for high contention workloads.

The results demonstrate that *list-OSTM* reduce the number of aborts to minimal as comparison to Trans-list [12], NOrec-list [6], and Boosting-list [13] shown in Figure 3.22. *list-OSTM* works on higher level with optimistic approach so, aborts are nominal.

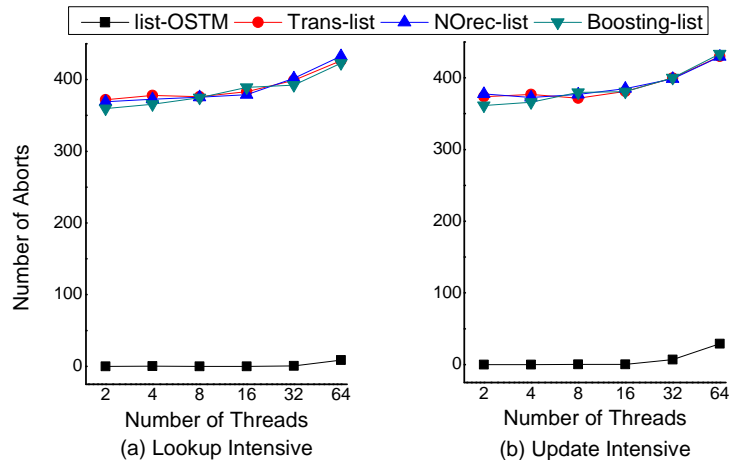


Figure 3.22: Abort Count of list-OSTM against State-of-the-art list based STMs

3.8 Summary

In this chapter of the thesis, we build a model for building highly concurrent and composable data structures with object level transactions called OSTMs. We showed that higher concurrency can be obtained by considering OSTMs as compared to traditional *RWSTM* by leveraging richer object-level semantics. We proposed a comprehensive theoretical model based on legality semantics and conflict notions for hash table based OSTMs, *HT-OSTM*. Using these notions we extend the definition of opacity and co-opacity for *HT-OSTMs* in Section 3.2. Then, based on this model, we developed a practical implementation of *HT-OSTM* and *list-OSTM* (simple modification in *HT-OSTM* while setting the bucket size as one) to verify the gains achieved as demonstrated in Section 3.7. Further, we proved that proposed model is co-opaque [18] thus composable.

Chapter 4

Multi-Version Object-based STMs

4.1 Introduction

The rise of multi-core systems has necessitated the need for concurrent programming. However, developing correct concurrent programs without compromising on efficiency is a big challenge. Software Transactional Memory Systems (STMs) are a convenient programming interface for a programmer to access shared memory without worrying about concurrency issues. Another advantage of STMs is that they facilitate compositionality of concurrent programs with great ease. Different concurrent operations that need to be composed to form a single atomic unit is achieved by encapsulating them in a single transaction. Next, we discuss different types of STMs considered in the literature and identify the need to develop *Multi-Version Object-based STMs* proposed in this chapter of the thesis.

Read-Write STMs: Most of the STMs proposed in the literature (such as NOrec [6], ESTM [7]) are based on read/write operations on *transaction objects* (*t-objects*) or *keys*. We denote them as *Read Write STMs* or *RWSTMs*. These STMs typically export following methods: (1) *STM.begin()*: begins a transaction with unique id, (2) *STM.read(k)* (or *r(k)*): reads the value of key *k* from shared memory, (3) *STM.write(k, v)* (or *w(k, v)*): writes the value of key *k* as *v* in its local log. This thesis considers the optimistic execution of STMs in which transactions are writing into its local log until the successful validation. (4) *STM.tryC()* (or *tryC()*): validates and tries to commit the transaction by writing values to the shared memory. If validation is successful, then it returns commit. Otherwise, it returns abort.

Object-based STMs: Some STMs have been proposed that work on higher level operations such as hash table. We call them *Object-based STMs* or *OSTMs*. OSTM exports the following methods: (1) *STM.begin()*: begins a transaction with unique id

(same as in *RWSTM*), (2) *STM_insert(k, v)* (or *ins(k, v)*): inserts a value v for key k in its local log, (3) *STM_delete(k)* (or *del(k)*): deletes the value associated with the key k , (4) *STM_lookup(k)* (or *lu(k)*): looks up the value associated with the key k from shared memory and, (5) *STM_tryC()* (or *tryC()*): validates and tries to commit the transaction by updating values to the shared memory. If validation is successful, then it returns commit. Otherwise, it returns abort. The concept of Boosting by Herlihy et al. [13], the optimistic variant by Hassan et al. [24] and *HT-OSTM* system by Peri et al. [16] (proposed work shown in Chapter 3) are some examples that demonstrate the performance benefits achieved by *OSTMs*. We have shown that *HT-OSTMs* provide greater concurrency than *RWSTM* in Chapter 3.

Multi-Version Object-based STMs: It has been shown in the literature of databases and STMs [14,15] that greater concurrency can be obtained by storing multiple versions for each t-object (or *key*). Having seen the advantage achieved by *OSTMs* in Chapter 3, we combine multiple versions with object semantics idea for harnessing greater concurrency in STMs. We proposed the new and efficient notion of *Multi-Version Object-based STMs* or *MVOSTMs*.

Specifically, maintaining multiple versions can ensure that more read operations succeed because the reading operation will have an appropriate version to read. Our goal is to evaluate the benefit of *MVOSTMs* over both multi-version and single-version *RWSTMs* as well as single-version *OSTMs*.

Potential benefit of *MVOSTMs* over *OSTMs* and multi-version *RWSTMs* explained in SubSection 1.4.1 of Chapter 1.

Contributions of the Chapter is as follows:

- We proposed a new notion of *Multi-Version Object-based STM system, MVOSTM*. Specifically developed it for two CDS, hash table and list objects as *HT-MVOSTM* and *list-MVOSTM* respectively.
- We show *HT-MVOSTM* and *list-MVOSTM* satisfy *opacity* [10], a standard correctness-criterion for STMs.
- For efficient space utilization in *MVOSTM* with unbounded versions we develop *Garbage Collection* for *MVOSTM* (i.e. *MVOSTM-GC*) and bounded version *MVOSTM* (i.e. *KOSTM*).
- Our experiments show that both hash table based *KOSTM (HT-KOSTM)* and list based *KOSTM (list-KOSTM)* provides greater concurrency and reduces the number of aborts as compared to single-version *OSTMs*, single and multi-version *RWSTMs*. We achieve this by maintaining multiple versions correspond-

ing to each key. To the best of our knowledge, this is the first work to explore the idea of using multiple versions in OSTMs to achieve greater concurrency.

Roadmap. This chapter is organized as follows. Section 4.2 shows the Graph Characterization of Opacity. Section 4.3 represents the *MVOSTMs* design and data structure. Section 4.4 shows the working of *MVOSTMs* and its algorithms. We formally prove the correctness of *MVOSTMs* in Section 4.5. In Section 4.6 we show the experimental evaluation of *MVOSTMs* with state-of-art-STMs. Finally, we summaries this chapter in Section 4.7.

4.2 Graph Characterization of Opacity

To prove that an STM system satisfies opacity, it is useful to consider graph characterization of histories. In this section, we describe the graph characterization of Guerraoui and Kapalka [27] modified for sequential histories.

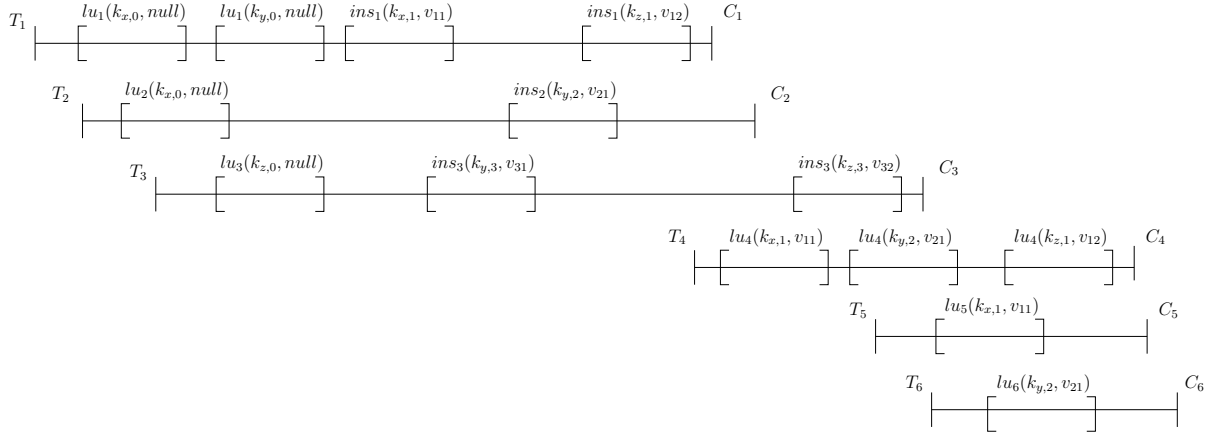


Figure 4.1: History $H3$ in time line view

Consider a history H which consists of multiple version for each t-object. The graph characterization uses the notion of *version order*. Given H and a t-object k , we define a version order for k as any (non-reflexive) total order on all the versions of k ever created by committed transactions in H . It must be noted that the version order may or may not be the same as the actual order in which the versions of k are generated in H . A version order of H , denoted as \ll_H is the union of the version orders of all the t-objects in H .

Consider the history $H3$ as shown in Figure 4.1 : $lu_1(k_{x,0}, null), lu_2(k_{x,0}, null), lu_1(k_{y,0}, null), lu_3(k_{z,0}, null), ins_1(k_{x,1}, v_{11}), ins_3(k_{y,3}, v_{31}), ins_2(k_{y,2}, v_{21}), ins_1(k_{z,1}, v_{12}), c_1, c_2, lu_4(k_{x,1}, v_{11}), lu_4(k_{y,2}, v_{21}), ins_3(k_{z,3}, v_{32}), c_3, lu_4(k_{z,1}, v_{12}), lu_5(k_{x,1}, v_{11}), lu_6(k_{y,2}, v_{21}),$

c_4, c_5, c_6 . Using the notation that a committed transaction T_i writing to k_x creates a version $k_{x,i}$, a possible version order for $H3 \ll_{H3}$ is: $\langle k_{x,0} \ll k_{x,1} \rangle, \langle k_{y,0} \ll k_{y,2} \ll k_{y,3} \rangle, \langle k_{z,0} \ll k_{z,1} \ll k_{z,3} \rangle$.

We define the graph characterization based on a given version order. Consider a history H and a version order \ll . We then define a graph (called opacity graph) on H using \ll , denoted as $OPG(H, \ll) = (V, E)$. The vertex set V consists of a vertex for each transaction T_i in \overline{H} . The edges of the graph are of three kinds and are defined as follows:

1. *rt*(real-time) edges: If the commit of T_i happens before beginning of T_j in H , then there exist a real-time edge from v_i to v_j . We denote set of such edges as $rt(H)$.
2. *rvf*(return value-from) edges: If T_j invokes `rv_method` on key k_1 from T_i which has already been committed in H , then there exists a return value-from edge from v_i to v_j . If T_i is having `upd_method` as insert on the same key k_1 then $ins_i(k_{1,i}, v_{i1}) <_H c_i <_H rvm_j(k_{1,i}, v_{i1})$. If T_i is having `upd_method` as delete on the same key k_1 then $del_i(k_{1,i}, null) <_H c_i <_H rvm_j(k_{1,i}, null)$. We denote set of such edges as $rvf(H)$.
3. *mv*(multi-version) edges: This is based on version order. Consider a triplet with successful methods as $upd_i(k_{1,i}, u), rvm_j(k_{1,i}, u), upd_k(k_{1,k}, v)$, where $u \neq v$. As we can observe it from $rvm_j(k_{1,i}, u), c_i <_H rvm_j(k_{1,i}, u)$. if $k_{1,i} \ll k_{1,k}$ then there exist a multi-version edge from v_j to v_k . Otherwise $(k_{1,k} \ll k_{1,i})$, there exist a multi-version edge from v_k to v_i . We denote set of such edges as $mv(H, \ll)$.

We now show that if a version order \ll exists for a history H such that it is acyclic, then H is opaque.

Using this construction, the $OPG(H3, \ll_{H3})$ for history $H3$ and \ll_{H3} is given above is shown in Figure 4.2. The edges are annotated. The only mv edge from T_4 to T_3 is because of t-objects k_y, k_z . T_4 lookups value v_{12} for k_z from T_1 whereas T_3 also inserts v_{32} to k_z and commits before $lu_4(k_{z,1}, v_{12})$.

Given a history H and a version order \ll , consider the graph $OPG(\overline{H}, \ll)$. While considering the *rt* edges in this graph, we only consider the real-time relation of H and not \overline{H} . It can be seen that $\prec_H^{RT} \subseteq \prec_{\overline{H}}^{RT}$ but with this assumption, $rt(H) = rt(\overline{H})$. Hence, we get the following property,

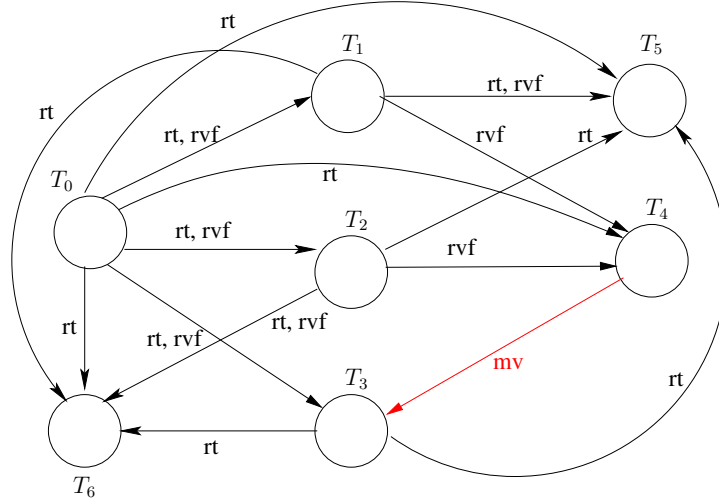


Figure 4.2: $OPG(H3, \ll_{H3})$

Property 44 *The graphs $OPG(H, \ll)$ and $OPG(\overline{H}, \ll)$ are the same for any history H and \ll .*

Definition 6 *For a t -sequential history S , we define a version order \ll_S as follows: For two version $k_{x,i}, k_{x,j}$ created by committed transactions T_i, T_j in S , $\langle k_{x,i} \ll_S k_{x,j} \Leftrightarrow T_i <_S T_j \rangle$.*

Now we show the correctness of our graph characterization using the following lemmas and theorem.

Lemma 45 *Consider a legal t -sequential history S . Then the graph $OPG(S, \ll_S)$ is acyclic.*

Proof: We numerically order all the transactions in S by their real-time order by using a function ord . For two transactions T_i, T_j , we define $ord(T_i) < ord(T_j) \Leftrightarrow T_i <_S T_j$. Let us analyze the edges of $OPG(S, \ll_S)$ one by one:

- rt edges: It can be seen that all the rt edges go from a lower ord transaction to a higher ord transaction.
- rvf edges: If T_j lookups k_x from T_i in S then T_i is a committed transaction with $ord(T_i) < ord(T_j)$. Thus, all the rvf edges from a lower ord transaction to a higher ord transaction.
- mv edges: Consider a successful rv_method $rvm_j(k_x, u)$ and a committed transaction T_k writing v to k_x where $u \neq v$. Let c_i be $rvm_j(k_x, u)$'s lastWrite. Thus, $upd_i(k_{x,i}, u) \in evts(T_i)$. Thus, we have that $ord(T_i) < ord(T_j)$. Now there are

two cases w.r.t T_i : (1) Suppose $ord(T_k) < ord(T_i)$. We now have that $T_k \ll T_i$. In this case, the mv edge is from T_k to T_i . (2) Suppose $ord(T_i) < ord(T_k)$ which implies that $T_i \ll T_k$. Since S is legal, we get that $ord(T_j) < ord(T_k)$. This case also implies that there is an edge from $ord(T_j)$ to $ord(T_k)$. Hence, in this case as well the mv edges go from a transaction with lower ord to a transaction with higher ord.

Thus, in all the three cases the edges go from a lower ord transaction to higher ord transaction. This implies that the graph is acyclic.

Lemma 46 *Consider two histories H, H' that are equivalent to each other. Consider a version order \ll_H on the t-objects created by H . The mv edges $mv(H, \ll_H)$ induced by \ll_H are the same in H and H' .*

Proof: Since the histories are equivalent to each other, the version order \ll_H is applicable to both of them. It can be seen that the mv edges depend only on events of the history and version order \ll . It does not depend on the ordering of the events in H . Hence, the mv edges of H and H' are equivalent to each other.

Using these lemmas, we prove the following theorem.

Theorem 47 *A valid history H is opaque iff there exists a version order \ll_H such that $OPG(H, \ll_H)$ is acyclic.*

Proof: (if part): Here we have a version order \ll_H such that $G_H = OPG(H, \ll)$ is acyclic. Now we have to show that H is opaque. Since the G_H is acyclic, a topological sort can be obtained on all the vertices of G_H . Using the topological sort, we can generate a t-sequential history S . It can be seen that S is equivalent to \overline{H} . Since S is obtained by a topological sort on G_H which maintains the real-time edges of H , it can be seen that S respects the rt order of H , i.e $\prec_H^{RT} \subseteq \prec_S^{RT}$.

Similarly, since G_H maintains return value-from (rvf) order of H , it can be seen that if T_j lookups k_x from T_i in H then T_i terminates before $lu_j(k_x)$ and T_j in S . Thus, S is valid. Now it remains to be shown that S is legal. We prove this using contradiction. Assume that S is not legal. Thus, there is a successful rv_method $rvm_j(k_x, u)$ such that its lastWrite in S is c_k and T_k updates value $v(\neq u)$ to k_x , i.e $upd_k(k_{x,k}, v) \in evts(T_k)$. Further, we also have that there is a transaction T_i that inserts u to k_x , i.e $upd_i(k_{x,i}, u) \in evts(T_i)$. Since S is valid, as shown above, we have that $T_i \prec_S^{RT} T_k \prec_S^{RT} T_j$.

Now in \ll_H , if $k_{x,k} \ll_H k_{x,i}$ then there is an edge from T_k to T_i in G_H . Otherwise ($k_{x,i} \ll_H k_{x,k}$), there is an edge from T_j to T_k . Thus, in either case, T_k can not be in between T_i and T_j in S contradicting our assumption. This shows that S is legal.

(Only if part): Here we are given that H is opaque and we have to show that there exists a version order \ll such that $G_H = OPG(H, \ll)(= OPG(\bar{H}, \ll)$, Property 44) is acyclic. Since H is opaque there exists a legal t-sequential history S equivalent to \bar{H} such that it respects real-time order of H . Now, we define a version order for S , \ll_S as in Definition 6. Since the S is equivalent to \bar{H} , \ll_S is applicable to \bar{H} as well. From Lemma 45, we get that $G_S = OPG(S, \ll_S)$ is acyclic. Now consider $G_H = OPG(\bar{H}, \ll_S)$. The vertices of G_H are the same as G_S . Coming to the edges,

- rt edges: We have that S respects real-time order of H , i.e. $\prec_H^{RT} \subseteq \prec_S^{RT}$. Hence, all the rt edges of H are a subset of S .
- rvf edges: Since \bar{H} and S are equivalent, the return value-from relation of \bar{H} and S are the same. Hence, the rvf edges are the same in G_H and G_S .
- mv edges: Since the version-order and the operations of the H and S are the same, from Lemma 46 it can be seen that \bar{H} and S have the same mv edges as well.

Thus, the graph G_H is a subgraph of G_S . Since we already know that G_S is acyclic from Lemma 45, we get that G_H is also acyclic.

4.3 MVOSTM Design and Data Structure

This section shows the design and data structure of proposed MVOSTM. *HT-MVOSTM* is a hash table based *MVOSTM* that explores the idea of using multiple versions in *OSTMs* for hash table object to achieve greater concurrency. The design of *HT-MVOSTM* is similar to our proposed *HT-OSTM* [16] (explained in Section 3.3) consisting of B buckets. If the bucket size B of hash table becomes one then hash table based MVOSTMs boils down to the list based MVOSTMs (list-MVOSTM). All the keys of the hash table in the range \mathcal{K} are statically allocated to one of these buckets.

Each bucket consists of linked-list of nodes along with two sentinel nodes *head* and *tail* with values $-\infty$ and $+\infty$ respectively. The structure of each node is as $\langle key, lock, marked, vl, nnext \rangle$. The *key* is a unique value from the set of all keys \mathcal{K} . All the nodes are stored in increasing order in each bucket as shown in Figure 4.3 (a), similar to any linked-list based concurrent set implementation [9, 28]. In the rest of the document, we use the terms key and node interchangeably. To perform any operation on a key, the corresponding *lock* is acquired. *marked* is a boolean field which represents whether the key is deleted or not. The deletion is performed in a lazy manner similar

to the concurrent linked-lists structure [9]. If the *marked* field is true then key corresponding to the node has been logically deleted; otherwise, it is present. The *vl* field of the node points to the version list (shown in Figure 4.3 (b)) which stores multiple versions corresponding to the key. The last field of the node is *nnext* which stores the address of the next node. It can be seen that the list of keys in a bucket is as an extension of *lazy-list* [9]. Given a node n in the linked-list of bucket B , we denote its fields as $n.key(k.key)$, $n.lock(k.lock)$, $n.marked(k.marked)$, $n.vl(k.vl)$, $n.nnext(k.nnext)$.

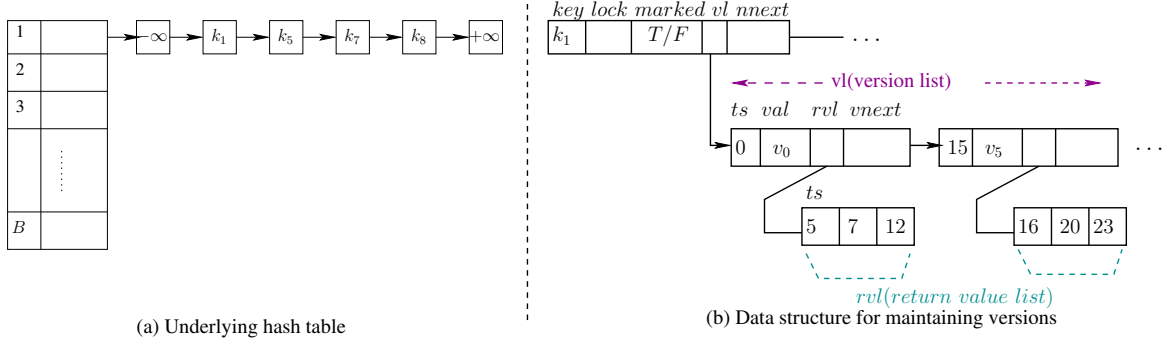


Figure 4.3: *HT-MVOSTM* Design and Data Structure

The structure of each version in the *vl* of a key k is $\langle ts, val, rvl, vnext \rangle$ as shown in Figure 4.3 (b). The field *ts* denotes the unique timestamp of the version. In our algorithm, every transaction is assigned a unique timestamp when it begins which is also its *id*. Thus *ts* of this version is the timestamp of the transaction that created it. All the versions in the *vl* of k are sorted by *ts*. Since the timestamps are unique, we denote a version, *ver* of a node n with key k having *ts* j as $n.vl[j].ver$ or $k.vl[j].ver$. The corresponding fields in the version as $k.vl[j].ts$, $k.vl[j].val$, $k.vl[j].rvl$, $k.vl[j].vnext$.

The field *val* contains the value updated by an update transaction. If this version is created by an insert method $STM_insert_i(ht, k, v)$ by transaction T_i , then *val* will be v . On the other hand, if the method is $STM_delete_i(ht, k)$ with the return value v , then *val* will be *null*. In this case, as per the algorithm, the node of key k will also be marked. *HT-MVOSTM* algorithm does not immediately physically remove deleted keys from the hash table. The need for this is explained below. Thus a *rv_method* ($STM_delete()$ or $STM_lookup()$) on key k can return *null* when it does not find the key or encounters a *null* value for k .

The *rvl* field stands for *return value list* which is a list of all the transactions that executed *rv_method* on this version, i.e., those transactions which returned *val*. The field *vnext* points to the next available version of that key.

Number of versions in *vl* (the length of the list) as per *HT-MVOSTM* can be bounded or unbounded. It can be bounded by having a limit on the number of

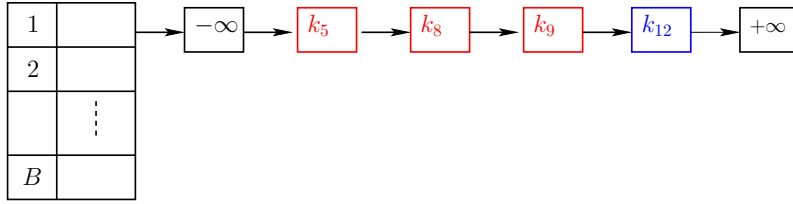


Figure 4.4: Searching k_{12} over *lazy-list*

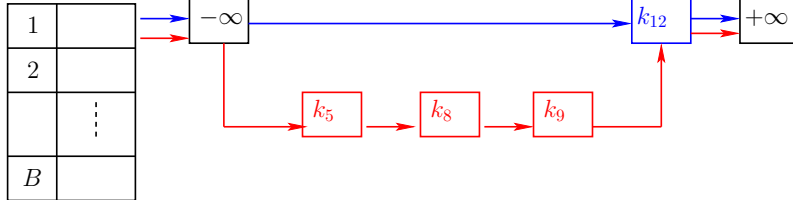


Figure 4.5: Searching k_{12} over *lazyrb-list*

versions such as K . Whenever a new version ver is created and is about to be added to vl , the length of vl is checked. If the length becomes greater than K , the version with lowest ts (i.e., the oldest) is replaced with the new version ver and thus maintaining the length back to K . If the length is unbounded, then we need a garbage collection scheme to delete unwanted versions for efficiency.

Marked Nodes: *HT-MVOSTM* stores keys even after they have been deleted (nodes which have *marked* field as true). This is because some other concurrent transactions could read from a different version of this key and not the *null* value inserted by the deleting transaction. Consider for instance the transaction T_1 performing $STM_lookup(ht, k)$ as shown in Figure 1.4 (b) of SubSection 1.4.1. Due to the presence of previous version v_0 , *HT-MVOSTM* could return this earlier version v_0 for $STM_lookup(ht, k)$ method. Whereas, it is not possible for *HT-OSTM* to return the version v_0 because k has been removed from the system after the delete by T_2 . In that case, T_1 would have to be aborted. Thus as explained in Section 4.1, storing multiple versions increases the concurrency.

To store deleted keys along with live keys (or unmarked node) in a lazy-list will increase the traversal time to access unmarked nodes. Consider the Figure 4.4, in which there are four keys $\langle k_5, k_8, k_9, k_{12} \rangle$ present in the list. Here $\langle k_5, k_8, k_9 \rangle$ are marked (or deleted) nodes while k_{12} is unmarked. Now, consider an access the key k_{12} as by *HT-MVOSTM* as a part of one of its methods. Then *HT-MVOSTM* would have to unnecessarily traverse the marked nodes to reach key k_{12} .

This motivated us to modify the lazy-list structure of nodes in each bucket to form a skip list based on red and blue links. We denote it as *lazy red-blue list* or *lazyrb-list*. This idea has been inherited from proposed *HT-OSTM* [16] described in Section 3.3. *lazyrb-list* consists of nodes with two links, red link (or **RL**) and blue link (or **BL**).

The nodes which are not marked (or not deleted) are accessible from the head via **BL**. While all the nodes including the marked ones can be accessed from the head via **RL**. With this modification, let us consider the above example of accessing unmarked key k_{12} . It can be seen that k_{12} can be accessed much more quickly through **BL** as shown in Figure 4.5. Using the idea of *lazyrb-list*, we have modified the structure of each node as $\langle key, lock, marked, vl, RL, BL \rangle$. Further, for a bucket B , we denote its linked-list as $B.lazyrb-list$.

4.4 The Working of MVOSTM

In this section, we show the working of each MVOSTM method along with their pseudocode. As explained in Section 4.1, *HT-MVOSTM* also exports the same methods exported by proposed HT-OSTM as $STM_begin()$, $STM_insert()$, $STM_delete()$, $STM_lookup()$, $STM_tryC()$ methods. $STM_delete()$, $STM_lookup()$ are *rv_methods* while $STM_insert()$, $STM_delete()$ are *upd_methods*. We treat $STM_delete()$ as both *rv_method* as well as *upd_method*. The *rv_methods* return the current value of the key. The *upd_methods*, update to the keys are first noted down in local log, $txLog$. Then in the $STM_tryC()$ method after validations of these updates are transferred to the shared memory. In this section, we now explain the high-level idea of all the methods as follows:

$STM_begin()$: A thread invokes a new transaction T_i using this method. The transaction T_i local log $txLog_i$ is initialized at Line 3. This method returns a unique id to the invoking thread by incrementing an atomic counter at Line 5. This unique id is also the timestamp of the transaction T_i . For convenience, we use the notation that i is the timestamp (or id) of the transaction T_i .

Algorithm 23 $STM_begin()$: It provides the local log and unique id to each transaction.

```

1: procedure  $STM\_begin()$ 
2:   /*Initialize the local log of transaction*/
3:    $txLog \leftarrow new\ txLog()$ .
4:   /*Get the unique transaction id ( $t\_id$ ) while incrementing the counter
   atomically*/
5:    $t\_id \leftarrow get\&inc(counter)$ .
6:   return  $\langle t\_id \rangle$ .
7: end procedure

```

rv_methods - $STM_delete_i(ht, k, v)$ and $STM_lookup_i(ht, k, v)$: Both these methods return the current value of key k . Algorithm 24 gives the high-level overview of these

methods. First, the algorithm checks to see if the given key is already in the local log, $txLog$ of T_i (Line 9). If the key is already there then the current rv_method is not the first method on k and is a subsequent method of T_i on k . So, we can return the value of k from the $txLog_i$.

If the key is not present in the $txLog_i$, then $HT-MVOSTM$ searches into shared memory. Specifically, it searches the bucket to which k belongs to. Every key in the range \mathcal{K} is statically allocated to one of the B buckets. So the algorithms search for k in the corresponding bucket, say B_k to identify the appropriate location, i.e., identify the correct *predecessor* or *pred* and *current* or *curr* keys in the lazyrb-list of B_k without acquiring any locks similar to the search in lazy-list [9]. Since each key has two links, **RL** and **BL**, the algorithm identifies four node references: two *pred* and two *curr* according to red and blue links. They are stored in the form of an array with $preds[0]$ and $currs[1]$ corresponding to blue links; $preds[1]$ and $currs[0]$ corresponding to red links. If both $preds[1]$ and $currs[0]$ nodes are unmarked then the *pred*, *curr* nodes of both red and blue links will be the same, i.e., $preds[0] = preds[1]$ and $currs[0] = currs[1]$. Thus depending on the marking of *pred*, *curr* nodes, a total of two, three or four different nodes will be identified. Here, the search ensures that $preds[0].key \leq preds[1].key < k \leq currs[0].key \leq currs[1].key$.

Next, the re-entrant locks on all the *pred*, *curr* keys are acquired in increasing order to avoid the deadlock. Then all the *pred* and *curr* keys are validated by $rv_Validation()$ in Line 14 as follows: (1) If *pred* and *curr* nodes of blue links are not marked, i.e, $(\neg preds[0].marked) \ \&\& \ (\neg currs[1].marked)$. (2) If the next links of both blue and red *pred* nodes point to the correct *curr* nodes: $(preds[0].BL = currs[1]) \ \&\& \ (preds[1].RL = currs[0])$.

If any of these checks fail, then the algorithm retries to find the correct *pred* and *curr* keys. It can be seen that the validation check is similar to the validation in concurrent lazy-list [9].

Next, we check if k is in $B_k.lazyrb-list$. If k is not in B_k , then we create a new node for k as: $\langle key = k, lock = false, marked = false, vl = v, RL = \phi, BL = \phi \rangle$ and insert it into $B_k.lazyrb-list$ such that it is accessible only via **RL** since this node is marked (Line 21). This node will have a single version v as: $\langle ts = 0, val = null, rvl = i, vnext = \phi \rangle$. Here invoking transaction T_i is creating a version with timestamp 0 to ensure that $rv_methods$ of other transactions will never abort. As we have explained in Figure 1.4 (b) of SubSection 1.4.1, even after T_2 deletes k_1 , the previous value of v_0 is still retained. Thus, when T_1 invokes lu on k_1 after the delete on k_1 by T_2 , $HT-MVOSTM$ will return v_0 (as previous value). Hence, each $rv_methods$ will find a version to read while maintaining the infinite version corresponding to each key

k . In rvl , T_i adds the timestamp as i in it and $vnext$ is initialized to empty value. Since val is null and the n , this version and the node is not technically inserted into $B_k.lazyrb-list$.

If k is in $B_k.lazyrb-list$ then, k is the same as $curr_s[0]$ or $curr_s[1]$ or both. Let n be the node of k in $B_k.lazyrb-list$. We then find the version of n , ver_j which has the timestamp j such that j has the largest timestamp smaller than i (timestamp of T_i). Add i to ver_j 's rvl (Line 29). Then release the locks, update the local log $txLog_i$ in Line 31 and return the value stored in $ver_j.val$ in Line 33.

Algorithm 24 $rv_method(ht, k, v)$: Could be either $STM_delete_i(ht, k, v)$ or $STM_lookup_i(ht, k, v)$ on key k that maps to bucket B_k .

```

8: procedure  $rv\_method_i(ht, k, v)$ 
9:   if ( $k \in txLog_i$ ) then
10:     Update the local log and return  $val$ .
11:   else
12:     Search in lazyrb-list to identify the  $preds[]$  and  $curr_s[]$  for  $k$  using BL and RL in bucket  $B_k$ .
13:     Acquire the locks on  $preds[]$  and  $curr_s[]$  in increasing order.
14:     if ( $\neg rv\_Validation(preds[], curr_s[])$ ) then
15:       Release the locks and goto Line 12.
16:     end if
17:     if ( $k \notin B_k.lazyrb-list$ ) then
18:       Create a new node  $n$  with key  $k$  as:  $\langle key = k, lock = false, marked = false, vl = v, RL = \phi, BL = \phi \rangle$ .
19:       /*The  $vl$  consists of a single element  $v$  with  $ts$  as 0*/
20:       Create the version  $v$  as:  $\langle ts = 0, val = null, rvl = i, vnext = \phi \rangle$ .
21:       Insert  $n$  into  $B_k.lazyrb-list$  such that it is accessible only via RLs. /* $n$  is marked*/
22:       Release the locks; update the  $txLog_i$  with  $k$ .
23:       return  $\langle null \rangle$ .
24:     end if
25:     Identify the version  $ver_j$  with  $ts = j$  such that  $j$  is the largest timestamp smaller than  $i$ .
26:     if ( $ver_j == null$ ) then
27:       goto Line 18.
28:     end if
29:     Add  $i$  into the  $rvl$  of  $ver_j$ .
30:      $retVal = ver_j.val$ .
31:     Release the locks; update the  $txLog_i$  with  $k$  and  $retVal$ .
32:   end if
33:   return  $\langle retVal \rangle$ .
34: end procedure

```

STM_insert(): The actual insertion will happen in the *STM_tryC()* method. First, it identifies the node corresponding to the key in local log at Line 37. If the node exists then it updates the local log with useful information like value, operation name and status for the node corresponding to the key at Line 40 for later use in *STM_tryC()*. Otherwise, it will create a local log at Line 38 and update it at Line 40.

Algorithm 25 *STM_insert()*: Actual insertion happens in the *STM_tryC()*.

```

35: procedure STM_insert()
36:   /*First identify the node corresponding to the key into local*/
37:   if ( $k \notin txLog_i$ ) then
38:     Create local log and append it into increasing order of keys.
39:   else
40:     Update the local log with value, operation name and status.
41:   end if
42: end procedure

```

upd_methods - *STM_insert()* and *STM_delete()*: Both the methods create a version corresponding to the key k . The actual effect of *STM_insert()* and *STM_delete()* in shared memory will take place in *STM_tryC()*. Algorithm 26 represents the high-level overview of *STM_tryC()*.

Initially, to avoid deadlocks, algorithm sorts all the *keys* in increasing order which are present in the local log, $txLog_i$. In *STM_tryC()*, $txLog_i$ consists of *upd_methods* (*STM_insert()* or *STM_delete()*) only. For all the *upd_methods* (opn_i) it searches the key k in the shared memory corresponding to the bucket B_k . It identifies the appropriate location (*pred* and *curr*) of key k using **BL** and **RL** (Line 48) in the lazyrb-list of B_k without acquiring any locks similar to *rv_method* explained above.

Next, it acquires the re-entrant locks on all the *pred* and *curr* keys in increasing order. After that, all the *pred* and *curr* keys are validated by *tryC_Validation()* in Line 50 as follows: (1) It does the *rv_Validation()* as explained above in the *rv_method*. (2) If key k exists in the $B_k.lazyrb-list$ and let n as a node of k . Then algorithm identifies the version of n , ver_j which has the timestamp j such that j has the largest timestamp smaller than i (timestamp of T_i). If any higher timestamp k of T_k than timestamp i of T_i exist in $ver_j.rvl$ then algorithm returns *Abort* in Line 51.

If all the above steps are true then each *upd_methods* exist in $txLog_i$ will take the effect in the shared memory after doing the *intraTransValidation()* in Line 56. If two *upd_methods* of the same transaction have at least one common shared node among its recorded *pred* and *curr* keys, then the previous *upd_method* effect may overwrite if the current *upd_method* of *pred* and *curr* keys are not updated according to the updates done by the previous *upd_method*. Thus to solve this we have *intraTransValidation()*

that modifies the *pred* and *curr* keys of current operation based on the previous operation in Line 56.

Next, we check if *upd_method* is *STM_insert()* and *k* is in *B_k.lazyrb-list*. If *k* is not in *B_k*, then create a new node *n* for *k* as: $\langle key = k, lock = false, marked = false, vl = v, RL = \phi, BL = \phi \rangle$. This node will have two versions *ver* as $\langle ts = 0, val = null, rvl = \phi, max_{rvl} = \phi, vnext = i \rangle$ for *T₀* and $\langle ts = i, val = v, rvl = \phi, max_{rvl} = \phi, vnext = \phi \rangle$ for *T_i*. *T_i* is creating a version with timestamp 0 to ensure that *rv_methods* of other transactions will never abort. For second version, *i* is the timestamp of the transaction *T_i* invoking this method; *marked* field sets to false because the node is inserted in the **BL**. *rvl* and *vnext* are initialized to empty values. We set the *val* as *v* and insert *n* into *B_k.lazyrb-list* such that it is accessible via **RL** as well as **BL** and set the lock field to be *true* (Line 60). If *k* is in *B_k.lazyrb-list* then, *k* is the same as *currs[0]* or *currs[1]* or both. Let *n* be the node of *k* in *B_k.lazyrb-list*. Then, we create the version *v* as: $\langle ts = i, val = v, rvl = \phi, vnext = \phi \rangle$ and insert the version into *B_k.lazyrb-list* such that it is accessible via **RL** as well as **BL** (Line 62).

Subsequently, we check if *upd_method* is *STM_delete()* and *k* is in *B_k.lazyrb-list*. Let *n* be the node of *k* in *B_k.lazyrb-list*. Then create the version *v* as: $\langle ts = i, val = null, rvl = \phi, vnext = \phi \rangle$ and insert the version into *B_k.lazyrb-list* such that it is accessible only via **RL** (Line 65).

Finally, at Line 67 it updates the *pred* and *curr* of *opn_i* in local log, *txLog_i*. At Line 69 releases the locks on all the *pred* and *curr* in increasing order of keys to avoid deadlocks and return *Commit*.

We illustrate the helping methods of *rv_method* and *upd_method* as follows:

rv_Validation(): It is called by both the *rv_method* and *upd_method*. It identifies the conflicts among the concurrent methods of different transactions. Consider an example shown in Figure 4.6, where two concurrent conflicting methods of different transactions are working on the same key *k₃*. Initially, at stage *s₁* in Figure 4.6 (c) both the conflicting method optimistically (without acquiring locks) identify the same *pred* and *curr* keys for key *k₃* from *B_k.lazyrb-list* in Figure 4.6 (a). At stage *s₂* in Figure 4.6 (c), method *ins₁(k₃)* of transaction *T₁* acquired the lock on *pred* and *curr* keys and inserted the node into *B_k.lazyrb-list* as shown in Figure 4.6 (b). After successful insertion by *T₁*, *pred* and *curr* has been changed for *lu₂(k₃)* at stage *s₃* in Figure 4.6 (c). So, the above modified information is delivered by *rv_Validation()* method at Line 72 when $(preds[0].BL \neq currs[1])$ for *lu₂(k₃)*. After that again it will find the new *pred* and *curr* for *lu₂(k₃)* and eventually it will commit.

Algorithm 26 *STM_tryC(T_i)*: Validate the upd_methods of the transaction and then commit.

```

43: procedure STM_tryC(Ti)
44:   /*Operation name (opn) could be either STM_insert() or STM_delete ()*/
45:   /*Sort the keys of txLogi in increasing order.*/
46:   for all (opni ∈ txLogi) do
47:     if ((opni == STM_insert()) || (opni == STM_delete())) then
48:       Search in lazyrb-list to identify the preds[] and currs[] for k of opni
       using BL and RL in bucket Bk.
49:       Acquire the locks on preds[] and currs[] in increasing order.
50:       if (!tryC_Validation()) then
51:         return ⟨Abort⟩.
52:       end if
53:     end if
54:   end for
55:   for all (opni ∈ txLogi) do
56:     intraTransValidation() modifies the preds[] and currs[] of current
     operation which would have been updated by the previous operation of
     the same transaction.
57:     if ((opni == STM_insert()) && (k ∉ Bk.lazyrb-list)) then
58:       Create new node n with k as: ⟨key = k, lock = false, marked = false,
       vl = v, RL = φ, BL = φ⟩.
59:       Create two versions ver as: ⟨ts=0, val=null, rvl=φ, maxrvl = φ,
       vnext=i⟩ for T0 and ⟨ts=i, val=v, rvl=φ, maxrvl = φ, vnext=φ⟩
       for Ti.
60:       Insert node n into Bk.lazyrb-list such that it is accessible via RL as
       well as BL /*lock sets true*/.
61:     else if (opni == STM_insert()) then
62:       Add the version v as: ⟨ts = i, val = v, rvl = φ, vnext = φ⟩ into
       Bk.lazyrb-list such that it is accessible via RL as well as BL.
63:     end if
64:     if (opni == STM_delete()) then
65:       Add the version i as: ⟨ts=i, val=null, rvl=φ, vnext=φ⟩ into
       Bk.lazyrb-list such that it is accessible only via RL.
66:     end if
67:     Update the preds[] and currs[] of opni in txLogi.
68:   end for
69:   Release the locks; return ⟨Commit⟩.
70: end procedure

```

Algorithm 27 *rv_Validation(preds[], currs[])*: Validate against the conflicting method of different transactions.

```

71: procedure rv_validation(preds[], currs[])
72:   if      ((preds[0].marked) || (currs[1].marked) || (preds[0].BL)       $\neq$ 
           currs[1] || (preds[1].RL)  $\neq$  currs[0]) then
73:     return  $\langle$ false $\rangle$ .
74:   else
75:     return  $\langle$ true $\rangle$ .
76:   end if
77: end procedure

```

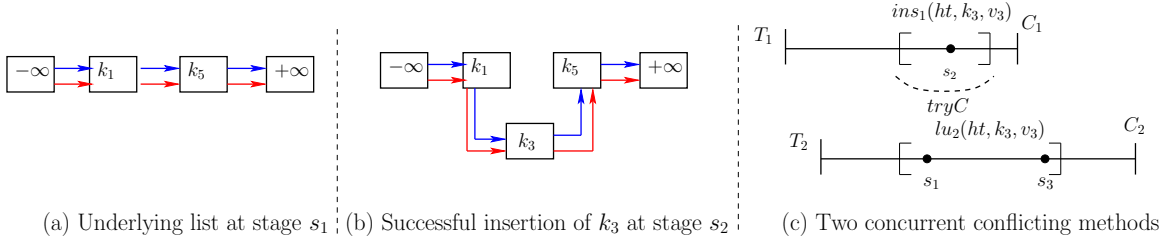


Figure 4.6: *rv_Validation*

Algorithm 28 *tryC_Validation()*: It maintains the order among the transactions.

```

78: procedure tryC_validation()
79:   if (!rv_Validation(preds[], currs[])) then
80:     Release the locks and retry.
81:   end if
82:   if ( $k \in B_k.lazyrb-list$ ) then
83:     Identify the version  $ver_j$  with  $ts = j$  such that  $j$  is the largest timestamp
           smaller than  $i$ .
84:     for all  $T_k$  in  $ver_j.rvl$  do
85:       if ( $TS(T_k) > TS(T_i)$ ) then
86:         return  $\langle$ false $\rangle$ .
87:       end if
88:     end for
89:   end if
90:   return  $\langle$ true $\rangle$ .
91: end procedure

```

tryC_Validation(): It is called by `upd_method` in `STM_tryC()`. First it does the `rv_Validation()` in Line 79. If its successful and key k exists in the $B_k.lazyrb-list$ and let n as a node of k . Then algorithm identifies the version of n , ver_j which has the timestamp j such that j has the largest timestamp smaller than i (timestamp of

T_i). If any higher timestamp T_k than timestamp T_i exist in $ver_j.rvl$ then algorithm returns false (in Line 86) and eventually return *Abort* in Line 51. Consider an example as shown in Figure 4.7 (a), where second method $ins_1(k_3)$ of transaction T_1 returns *Abort* because higher timestamp of transaction T_2 is already present in the *rvl* of version T_0 identified by T_1 in Figure 4.7 (b).

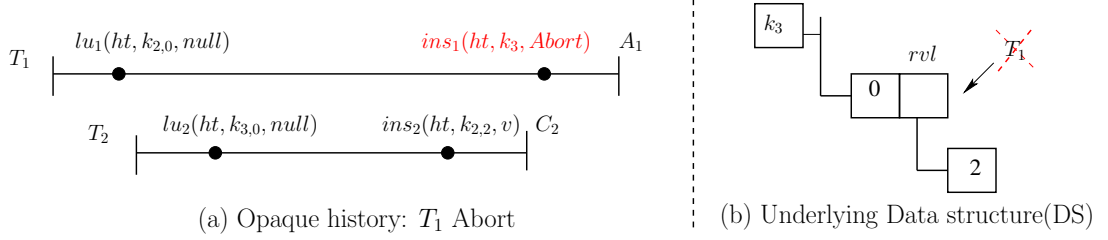


Figure 4.7: tryC_Validation

Algorithm 29 *intraTransValidation()*: It helps the upcoming method of the same transaction.

```

92: procedure intraTransValidation()
93:   if ((preds[0].marked) || (preds[0].BL  $\neq$  currs[1])) then
94:     if (opnk == Insert) then
95:       /*Modify the pred of current transaction  $T_i$  with the help of previous
96:        transaction  $T_k$ */
97:       /*Set the  $T_i$  preds[0] as  $T_k$  currs[1]*/
98:       preds[0]i  $\leftarrow$  preds[0]k.BL.
99:     else
100:      /*Set the  $T_i$  preds[0] as  $T_k$  preds[0]*/
101:      preds[0]i  $\leftarrow$  preds[0]k.
102:    end if
103:  if (preds[1].RL  $\neq$  currs[0]) then
104:    /*Set the  $T_i$  preds[1] as  $T_k$  currs[0]*/
105:    preds[1]i  $\leftarrow$  preds[1]k.RL.
106:  end if
107: end procedure

```

intraTransValidation(): It is called by *upd_method* in *STM_tryC()*. If two *upd_methods* of the same transaction have at least one common shared node among its recorded *pred* and *curr* keys, then the previous *upd_method* effect may overwrite if the current *upd_method* of *pred* and *curr* keys are not updated according to the updates

done by the previous *upd_method*. Thus to solve this we have *intraTransValidation()* that modifies the *pred* and *curr* keys of current operation based on the previous operation from Line 93 to Line 106. Consider an example as shown in Figure 4.8, where two *upd_methods* of transaction T_1 are $ins_{s_{11}}(k_3)$ and $ins_{s_{12}}(k_5)$ in Figure 4.8 (c). At stage s_1 in Figure 4.8 (c) both the *upd_methods* identify the same *pred* and *curr* from *underlying data structure* as $B_k.lazyrb-list$ shown in Figure 4.8 (a). After the successful insertion done by first *upd_method* at stage s_2 in Figure 4.8 (c), key k_3 is part of $B_k.lazyrb-list$ (Figure 4.8 (b)). At stage s_3 in Figure 4.8 (c), $ins_{s_{12}}(k_5)$ identified ($preds[0].BL \neq currs[1]$) in *intraTransValidation()* at Line 93. So, it updates the $preds[0]$ in Line 97 for correct updation in $B_k.lazyrb-list$.

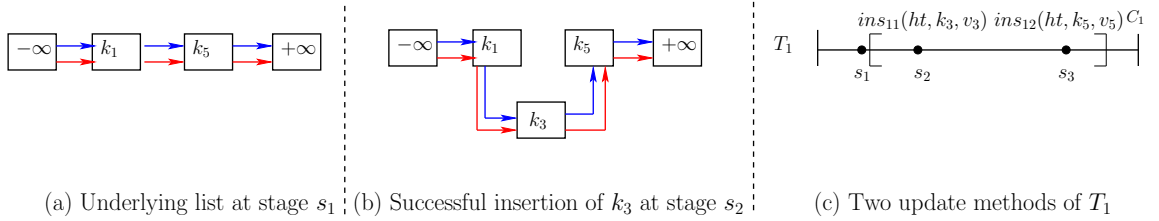


Figure 4.8: Intra transaction validation

4.5 Correctness of MVOSTM

In this section, we will prove that our implementation satisfies opacity. Consider the history H generated by *MVOSTM* algorithm. Recall that only the *STM_begin()*, *rv_method*, *STM_insert()*, *upd_method* (or *STM_tryC()*) access shared memory.

Note that H is not necessarily sequential: the transactional methods can execute in an overlapping manner. To reason about correctness, we have to prove H is opaque. Since we defined opacity for histories which are sequential, we order all the overlapping methods in H to get an equivalent sequential history. We then show that this resulting sequential history satisfies opacity.

We order overlapping methods of H as follows: (1) two overlapping *STM_begin()* methods based on the order in which they obtain lock over the *counter*; (2) two *rv_*-methods accessing the same key k by their order of unlocking over $\langle preds[0], preds[1], currs[0], currs[1] \rangle$ of k ; (3) an *rv_method* $rvm_i(k)$ and a *STM_insert_j()*, of a transaction T_j accessing the same key k , are ordered by their order of unlocking over $\langle preds[0], preds[1], currs[0], currs[1] \rangle$ of k ; (4) an *rv_method* $rvm_i(k)$ and a *STM_tryC_j()*, of a transaction T_j which has written to k , are similarly ordered by their order of unlocking over $\langle preds[0], preds[1], currs[0], currs[1] \rangle$ of k ; (5) two *STM_insert()* methods

accessing the same key k by their order of unlocking over $\langle \text{preds}[0], \text{preds}[1], \text{currs}[0], \text{currs}[1] \rangle$ of k ; (6) a $STM_insert_i()$ and a $STM_tryC_j()$, of a transaction T_j which has written to k , are similarly ordered by their order of unlocking over $\langle \text{preds}[0], \text{preds}[1], \text{currs}[0], \text{currs}[1] \rangle$ of k ; (7) similarly, two $STM_tryC()$ methods based on the order in which they unlock over $\langle \text{preds}[0], \text{preds}[1], \text{currs}[0], \text{currs}[1] \rangle$ of same key k .

Combining the real-time order of events with above-mentioned order, we obtain a partial order which we denote as $lockOrder_H$. (It is a partial order since it does not order overlapping $rv_methods$ on different $keys$ or an overlapping rv_method and a STM_tryC which do not access any common key).

In order for H to be sequential, all its methods must be ordered. Let α be a total order or *linearization* of methods of H such that when this order is applied to H , it is sequential. We denote the resulting history as $H^\alpha = linearize(H, \alpha)$. We now argue about the validity of histories generated by the algorithm.

Lemma 48 *Consider a history H generated by the MVOSTM algorithm. Let α be a linearization of H which respects $lockOrder_H$, i.e. $lockOrder_H \subseteq \alpha$. Then $H^\alpha = linearize(H, \alpha)$ is valid.*

Proof: Consider a successful rv_method $rvm_i(k)$ that returns value v . The rv_method first obtains the lock on $\langle \text{preds}[0], \text{preds}[1], \text{currs}[0], \text{currs}[1] \rangle$ of key k . Thus the value v returned by the rv_method must have already been stored in k 's version list by a transaction, say T_j when it successfully returned OK from its STM_tryC method. For this to have occurred, T_j must have successfully locked and released $\langle \text{preds}[0], \text{preds}[1], \text{currs}[0], \text{currs}[1] \rangle$ of k prior to T_i 's locking method. Thus from the definition of $lockOrder_H$, we get that $STM_tryC_j(ok)$ occurs before $rvm_i(k, v)$ which also holds in α .

It can be seen that for proving correctness, any linearization of a history H is sufficient as long as the linearization respects $lockOrder_H$. The following lemma formalizes this intuition,

Lemma 49 *Consider a history H . Let α and β be two linearizations of H such that both of them respect $lockOrder_H$, i.e. $lockOrder_H \subseteq \alpha$ and $lockOrder_H \subseteq \beta$. Then, $H^\alpha = linearize(H, \alpha)$ is opaque if $H^\beta = linearize(H, \beta)$ is opaque.*

Proof: From Lemma 48, we get that both H^α and H^β are valid histories. Now let us consider each case

If: Assume that H^α is opaque. Then, we get that there exists a legal t-sequential history S that is equivalent to $\overline{H^\alpha}$. From the definition of H^β , we get that $\overline{H^\alpha}$ is equivalent to $\overline{H^\beta}$. Hence, S is equivalent to $\overline{H^\beta}$ as well. We also have that,

$\prec_{H^\alpha}^{RT} \subseteq \prec_S^{RT}$. From the definition of $lockOrder_H$, we get that $\prec_{H^\alpha}^{RT} = \prec_{lockOrder_H}^{RT} = \prec_{H^\beta}^{RT}$. This automatically implies that $\prec_{H^\beta}^{RT} \subseteq \prec_S^{RT}$. Thus H^β is opaque as well.

Only if: This proof comes from symmetry since H^α and H^β are not distinguishable.

This lemma shows that, given a history H , it is enough to consider one sequential history H^α that respects $lockOrder_H$ for proving correctness. If this history is opaque, then any other sequential history that respects $lockOrder_H$ is also opaque.

Consider a history H generated by $MVOSTM$ algorithm. We then generate a sequential history that respects $lockOrder_H$. For simplicity, we denote the resulting sequential history of $MVOSTM$ as H_{to} . Let T_i be a committed transaction in H_{to} that writes to k (i.e. it creates a new version of k).

To prove the correctness, we now introduce some more notations. We define $H_{to}.stl(T_i, k)$ as a committed transaction T_j such that T_j has the *smallest timestamp larger (or stl)* than T_i in H_{to} that writes to k in H_{to} . Similarly, we define $H_{to}.lts(T_i, k)$ as a committed transaction T_k such that T_k has the *largest timestamp smaller (or lts)* than T_i that writes to k in H_{to} . Using these notations, we describe the following properties and lemmas on H_{to} ,

Property 50 *Every transaction T_i is assigned a unique numeric timestamp i .*

Property 51 *If a transaction T_i begins after another transaction T_j then $j < i$.*

Lemma 52 *If a transaction T_k looks up key k_x from (a committed transaction) T_j then T_j is a committed transaction updating to k_x with j being the largest timestamp smaller than k . Formally, $T_j = H_{to}.lts(T_k, k_x)$.*

Proof: We prove it by contradiction. So, assume that transaction T_k looks up key k_x from T_i that has committed before T_j so, from Property 51, $i < k$ and $k < j$ i.e. i is not largest timestamp smaller than k . But given statement in this lemma is $i < j < k$ which contradicts our assumption. Hence, T_k looks up key k_x from T_j which is the largest timestamp smaller than k .

Lemma 53 *Suppose a transaction T_k looks up k_x from (a committed transaction) T_j in H_{to} , i.e. $\{\text{upd}_j(k_{x,j}, v), \text{rvm}_k(k_{x,i}, v)\} \in \text{evts}(H_{to})$. Let T_i be a committed transaction that updates to k_x , i.e. $\text{upd}_i(k_{x,i}, u) \in \text{evts}(T_i)$. Then, the timestamp of T_i is either less than T_j 's timestamp or greater than T_k 's timestamp, i.e. $i < j \oplus k < i$ (where \oplus is XOR operator).*

Proof: We will prove this by contradiction. Assume that $i < j \oplus k < i$ is not true. This implies that, $j < i < k$. But from the implementation of `rv_method` and `STM-tryC` methods, we get that either transaction T_i is aborted or T_k looks up k from T_i in H . Since neither of them are true, we get that $j < i < k$ is not possible. Hence, $i < j \oplus k < i$.

To show that H_{to} satisfies opacity, we use the graph characterization developed above in Section 4.2. For the graph characterization, we use the version order defined using timestamps. Consider two committed transactions T_i, T_j such that $i < j$. Suppose both the transactions write to key k . Then the versions created are ordered as $k_i \ll k_j$. We denote this version order on all the *keys* created as \ll_{to} . Now consider the opacity graph of H_{to} with version order as defined by \ll_{to} , $G_{to} = OPG(H_{to}, \ll_{to})$. In the following lemmas, we will prove that G_{to} is acyclic.

Lemma 54 *All the edges in $G_{to} = OPG(H_{to}, \ll_{to})$ are in timestamp order, i.e. if there is an edge from T_j to T_i then the $j < i$.*

Proof: To prove this, let us analyze the edges one by one,

- rt edges: If there is an rt edge from T_j to T_i , then T_j terminated before T_i started. Hence, from Property 51 we get that $j < i$.
- rvf edges: This follows directly from Lemma 52.
- mv edges: The mv edges relate a committed transaction T_k updates to a key k , $up_k(k, v)$; a successful `rv_method` $rv_m_j(k, u)$ belonging to a transaction T_j looks up k updated by a committed transaction T_i , $up_i(k, u)$. Transactions T_i, T_k create new versions k_i, k_k respectively. According to \ll_{to} , if $k_k \ll_{to} k_i$, then there is an edge from T_k to T_i . From the definition of \ll_{to} this automatically implies that $k < i$.

On the other hand, if $k_i \ll_{to} k_k$ then there is an edge from T_j to T_k . Thus, in this case, we get that $i < k$. Combining this with Lemma 53, we get that $j < k$.

Thus in all the cases, we have shown that if there is an edge from T_j to T_i then the $j < i$.

Theorem 55 *Any history H_{to} generated by `MVOSTM` is opaque.*

Proof: From the definition of H_{to} and Lemma 48, we get that H_{to} is valid. We show that $G_{to} = OPG(H_{to}, \ll_{to})$ is acyclic. We prove this by contradiction. Assume that G_{to} contains a cycle of the form, $T_{c1} \rightarrow T_{c2} \rightarrow \dots T_{cm} \rightarrow T_{c1}$. From Lemma 54 we get

that, $c1 < c2 < \dots < cm < c1$ which implies that $c1 < c1$. Hence, a contradiction. This implies that G_{to} is acyclic. Thus from Theorem 47, we get that H_{to} is opaque.

Now, it is left to show that our algorithm is *live*, i.e., under certain conditions, every operation eventually completes. We have to show that the transactions do not deadlock. This is because all the transactions lock all $\langle preds[0], preds[1], currs[0], currs[1] \rangle$ of *keys* in a predefined order. As discussed earlier, the STM system orders all $\langle preds[0], preds[1], currs[0], currs[1] \rangle$ of *keys*. We denote this order as *accessOrder* and denote it as \prec_{ao} . Thus $k_1 \prec_{ao} k_2 \prec_{ao} \dots \prec_{ao} k_n$.

From *accessOrder*, we get the following property

Property 56 *Suppose transaction T_i accesses shared objects p and q in H . If p is ordered before q in *accessOrder*, then $lock(p)$ by transaction T_i occurs before $lock(q)$. Formally, $(p \prec_{ao} q) \Leftrightarrow (lock(p) <_H lock(q))$.*

Theorem 57 *MVOSTM with unbounded versions ensures that *rv_methods* do not abort.*

Proof: This is self-explanatory with the help of *MVOSTM* algorithm because each *key* is maintaining multiple versions in the case of unbounded versions. So *rv_method* always finds a correct version to read it from. Thus, *rv_methods* do not *abort*.

4.6 Experimental Evaluations

This section presents the experimental analysis of *MVOSTM*. We have two main goals in this section: (1) evaluating the benefits of *Multi-Version Object-based STMs (MVSOTM)* over the *Single-Version Object-based STMs (OSTM)*, and (2) evaluating the benefit of *Multi-Version Object-based STMs* over *Single and Multi-Version Read-Write STMs*. We use the hash table based *MVOSTM (HT-MVOSTM)* described in Section 4.4 as well as the corresponding list based *MVOSTM (list-MVOSTM)* which implements the list object. We also consider extensions of these multi-version object-based STMs to reduce the memory usage. Specifically, we consider a variant that implements garbage collection with unbounded versions and another variant where the number of versions never exceeds a given threshold K .

STM implementations: To show the performance of proposed *MVOSTM* against state-of-the-art STMs, we have taken the implementation of *NRec-list* [6], *Boosting-list* [13], *Trans-list* [12], *ESTM* [7], and *RWSTM* [3] directly from the *TLDS* framework¹. Along with this, we have considered the implementation of *OSTM* proposed

¹TLDS Framework: <https://ucf-cs.github.io/tlds/>

by us in Section 3.7 and MVTO [15] defined in PDCRL library². We implemented our algorithms in C++. We use counter application (explained in SubSection 3.7.1) where each STM algorithm first creates N-threads, each thread, in turn, spawns a transaction. Each transaction exports the following methods as follows: *STM_begin()*, *STM_insert()*, *STM_lookup()*, *STM_delete()* and *STM_tryC()* as described in Section 4.4.

Methodology:³ We have considered three types of workloads: (W1) Lookup Intensive (90% lookup, 5% insert and 5% delete), (W2) Mid Intensive (50% lookup, 25% insert and 25% delete), and (W3) Update Intensive (10% lookup, 45% insert and 45% delete). The experiments are conducted by varying number of threads from 2 to 64 in power of 2, with 1000 keys randomly chosen. We assume that the hash table of *HT-MVOSTM* has five buckets and each of the bucket (or list in case of *list-MVOSTM*) can have a maximum size of 1000 keys. Each transaction, in turn, executes 10 operations which include *STM_lookup()*, *STM_delete()* and *STM_insert()* operations. For accuracy, we take an average over 10 results for the final result in which the first run is discarded and considered as a warm-up result for each experiment.

4.6.1 Result Analysis

For efficient memory utilization, we developed two variants of *MVOSTM*. The first, *MVOSTM-GC*, uses unbounded versions but performs *Garbage Collection*. **This is achieved by deleting non-latest versions whose timestamp is less than the timestamp of the least live transaction.** For the sake of better understanding, the detailed description of garbage collection method is explained below. *MVOSTM-GC* gave a performance gain of 15% over *MVOSTM* without garbage collection in the best case. The second, *KOSTM*, keeps at most K -versions by deleting the oldest version when $(K + 1)^{th}$ version is created by a current transaction. As *KOSTM* has limited number of versions while *MVOSTM-GC* can have infinite versions, the memory consumed by *KOSTM* is 21% less than *MVOSTM*.

We have integrated these variants in both hash table based *MVOSTM* (i.e. *HT-MVOSTM-GC*, *HT-KOSTM*) and linked-list based *MVOSTMs* (i.e. *list-MVOSTM-GC*, *list-KOSTM*), we observed that these two variants increase the performance, concurrency and reduces the number of aborts as compared to *MVOSTM*.

Experimental results show that *KOSTM* performs better than *MVOSTM-GC* and *MVOSTM*. Here, *MVOSTM* and *MVOSTM-GC* maintain unbounded versions which

²PDCRL Library: <https://github.com/PDCRL>

³Proposed *MVOSTM* code is available here: <https://github.com/PDCRL/MVOSTM>

increase the search time to find the correct version and *GC* behaves as an overhead whereas *KOSTM* maintains only *K* versions corresponding to each key so, *KOSTM* reduces the search time to find the correct version to return/replace, does not use *GC*, and performs best. Proposed *HT-KOSTM* achieves a performance speedup of 1.22, 1.15 for workload *W1*, 1.1, 1.06 for workload *W2*, and 1.15, 1.08 for workload *W3* as compared to proposed *HT-MVOSTM* and *HT-MVOSTM-GC* respectively as shown in Figure 4.9. Whereas, proposed *list-KOSTM* gives a speedup of 1.1, 1.07 for workload *W1*, 1.25, 1.20 for workload *W2*, and 1.25, 1.13 for workload *W3* over the proposed *list-MVOSTM* and *list-MVOSTM-GC* respectively as demonstrated in Figure 4.10. *HT-KOSTM* and *list-KOSTM* have the least number of aborts than its variants for all the type of workloads illustrated in Figure 4.11 and Figure 4.12.

Figure 4.13 demonstrates that *HT-KOSTM* outperforms state-of-the-art hash table based STMs (*HT-OSTM* [16] proposed in Chapter 3, *ESTM* [7], *RWSTM* [3], *HT-MVTO* [15], *HT-KSTM* [15]) by a factor of 3.5, 3.8, 3.1, 2.6, 1.8 for workload *W1*, by a factor of 1.4, 3, 4.85, 10.1, 7.8 for workload *W2*, and by a factor of 2, 4.25, 19, 69, 59 for workload *W3* respectively. Here, *ESTM*, *RWSTM*, and *HT-MVTO* work on lower level. Among them *ESTM* and *RWSTM* maintain single-version corresponding to each key but *HT-MVTO* maintains multiple versions. Apart from these, *HT-OSTM* and *HT-KOSTM* works on higher level and *HT-OSTM* maintains single-version whereas *HT-KOSTM* maintains multiple versions. Hence, *HT-KOSTM* performs best.

Similarly, *list-KOSTM* outperforms state-of-the-art list based STMs (*list-OSTM* [16] proposed in Chapter 3, *Trans-list* [12], *Boosting-list* [13], *NOrec-list* [6], *list-MVTO* [15], *list-KSTM* [15]) by a factor of 2.2, 20, 22, 24, 12, 6 for workload *W1*, by a factor of 1.58, 20.9, 25.9, 29.4, 26.8, 19.68 for workload *W2*, and by a factor of 2, 35, 41, 47, 148, 112 for workload *W3* respectively shown in Figure 4.14. Though, *list-OSTM*, *Trans-list*, and *boosting-list* work on higher level but all of them maintain single-version corresponding to each key whereas *list-KOSTM* work on higher level and maintains multiple versions. So, *list-KOSTM* performs best.

list-KOSTM performs much better than *list-OSTM* for lookup intensive workload because of maintaining multiple versions, most of the lookup found a correct version to return. Whereas for update intensive workload, replacing the oldest version is taking time. So, *list-KOSTM* performs slightly better than *list-OSTM* for update intensive workload.

Due to minimum time taken by transactions to commit, *HT-KOSTM* and *list-KOSTM* have the least number of aborts against state-of-the-art STMs for all the type of workloads illustrated in Figure 4.15 and Figure 4.16.

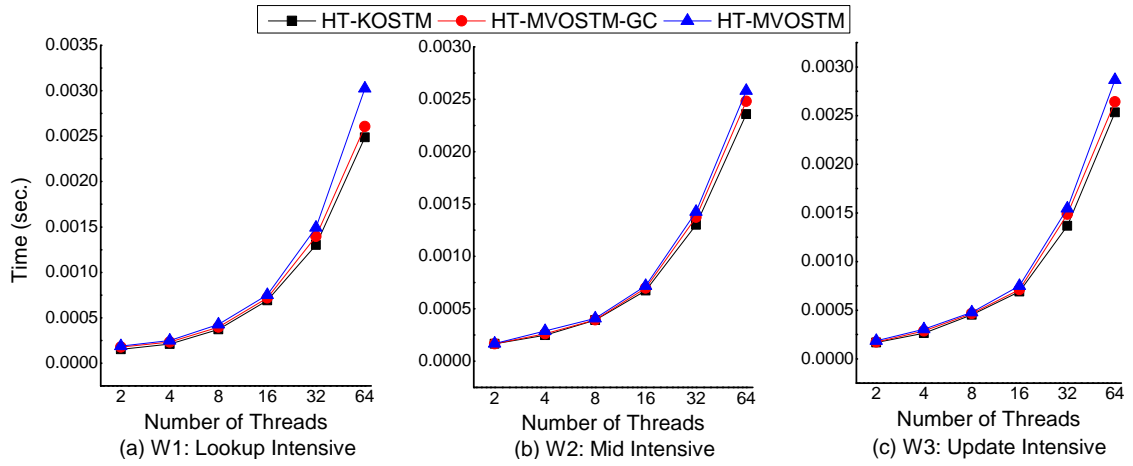


Figure 4.9: Performance of *HT-KOSTM* and its variants (*HT-MVOSTM-GC*, *HT-MVOSTM*)

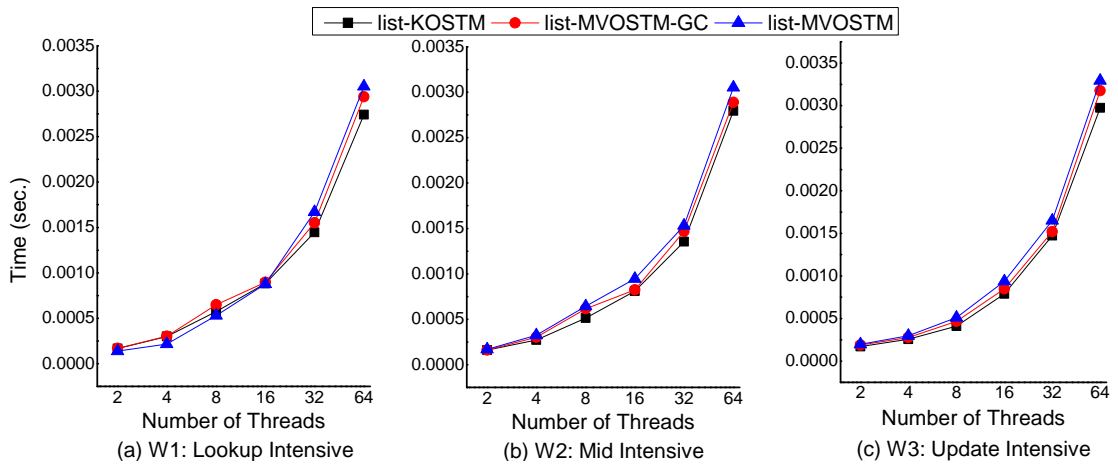


Figure 4.10: Performance of *list-KOSTM* and its variants (*list-MVOSTM-GC*, *list-MVOSTM*)

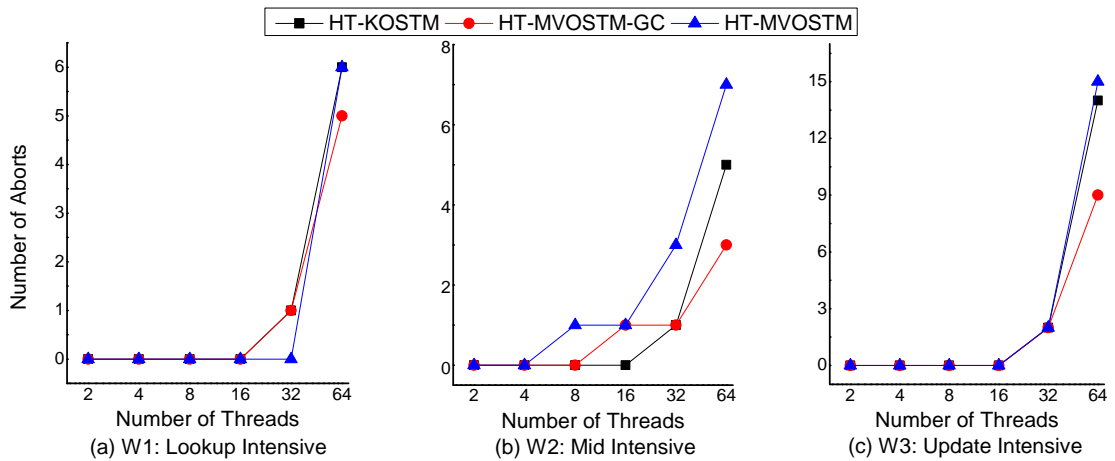


Figure 4.11: Aborts Count of *HT-KOSTM* and its variants

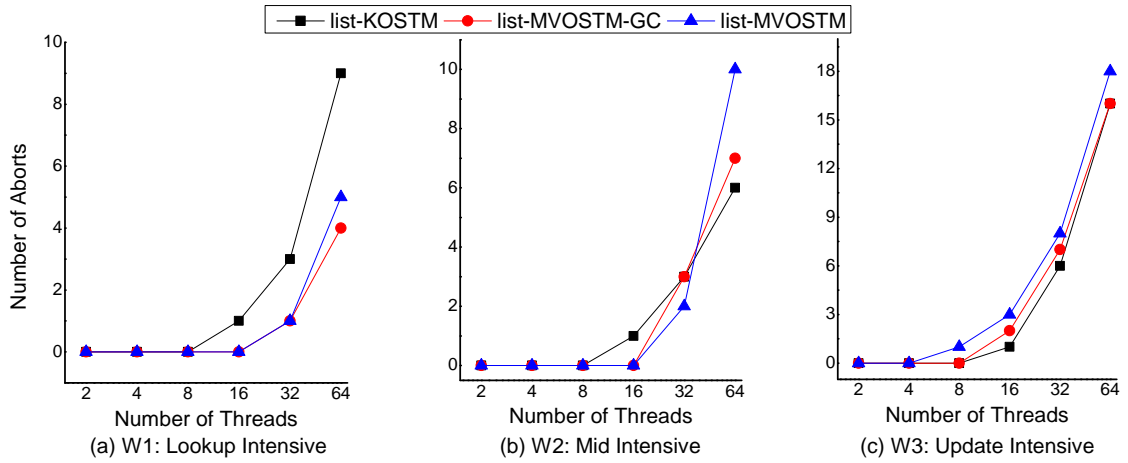


Figure 4.12: Aborts Count of *list-KOSTM* and its variants

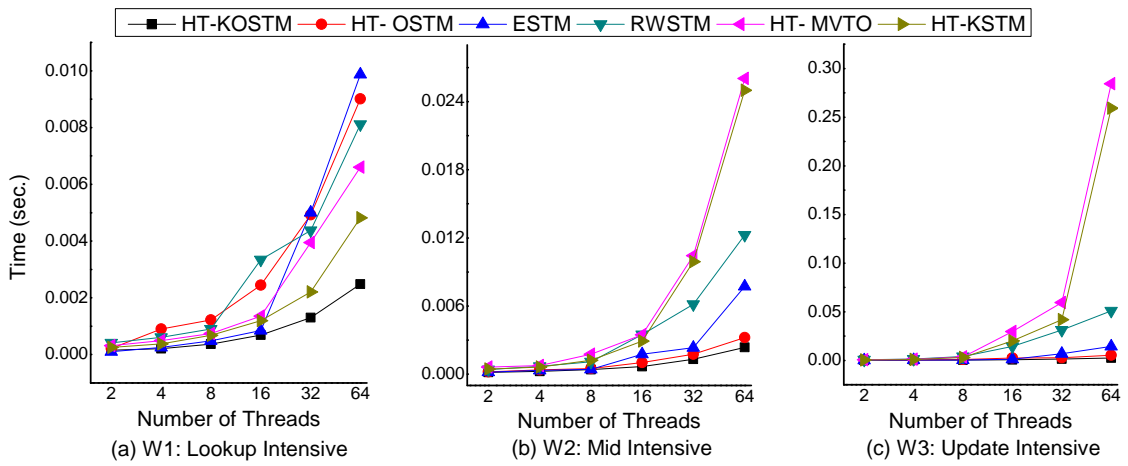


Figure 4.13: Performance of *HT-KOSTM* against State-of-the-art hash table based STMs

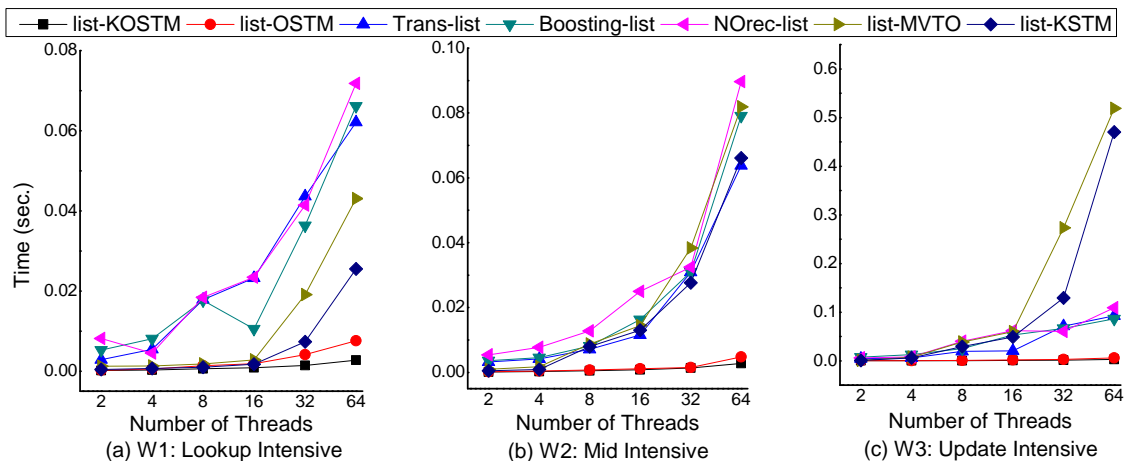


Figure 4.14: Performance of *list-KOSTM* against State-of-the-art list based STMs

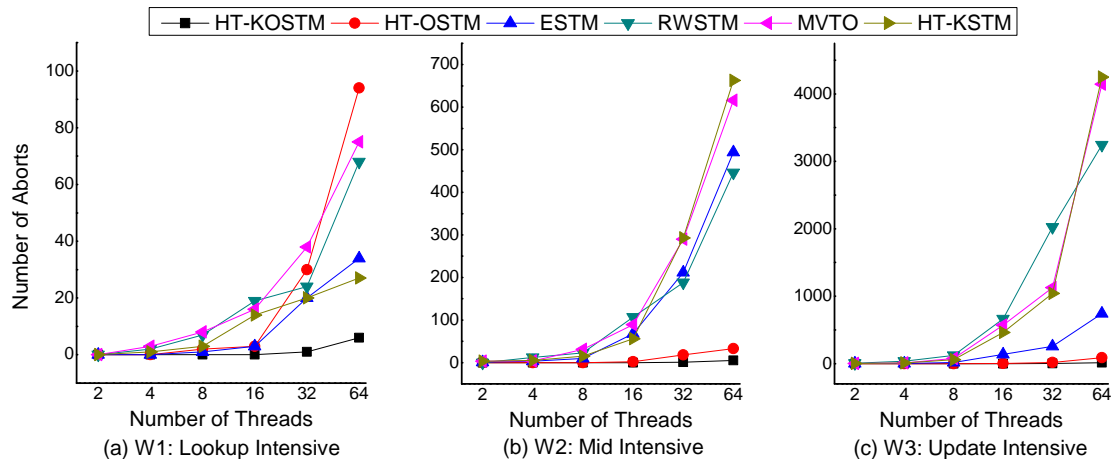


Figure 4.15: Aborts Count of *HT-KOSTM* against State-of-the-art hash table based STMs

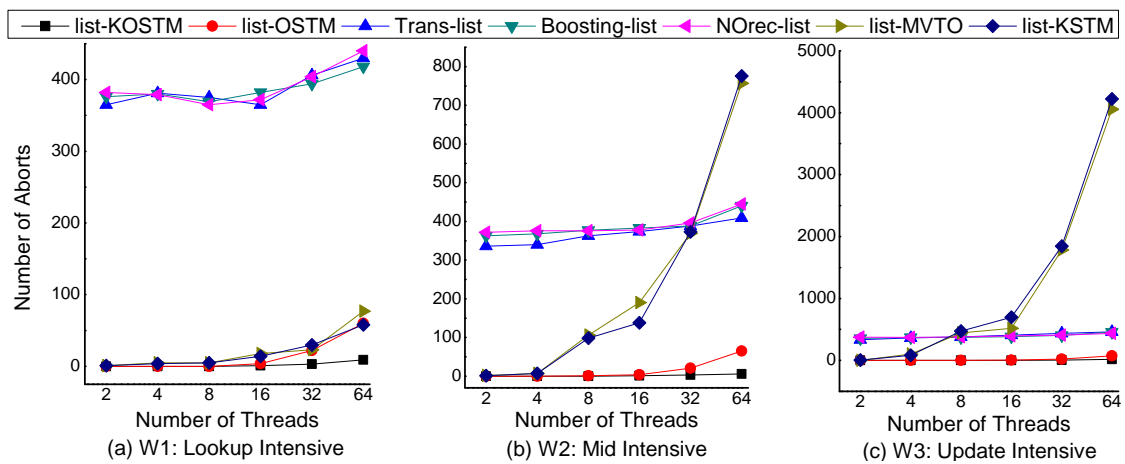


Figure 4.16: Aborts Count of *list-KOSTM* against State-of-the-art list based STMs

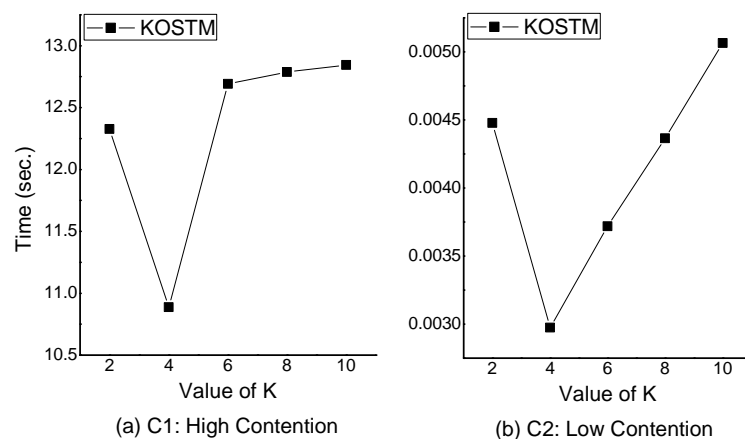


Figure 4.17: Optimal Value of K

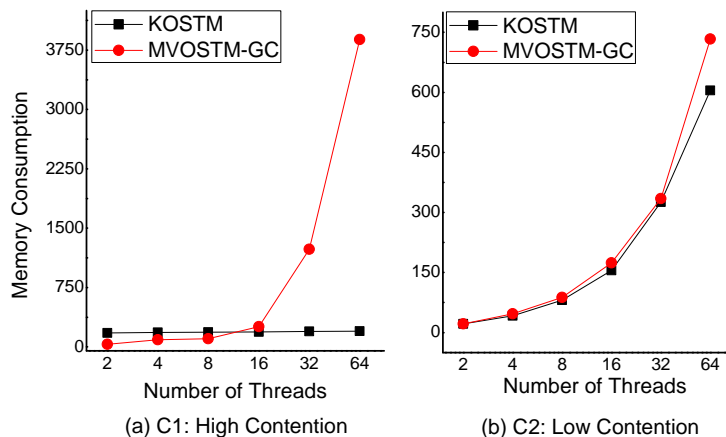


Figure 4.18: Memory Consumption

Garbage Collection in MVOSTMs (*MVOSTM-GC*): Providing multiple versions to increase the performance of OSTMs in MVOSTMs lead to more space requirements. As many unnecessary versions pertain in the memory a technique to remove these versions or to collect these garbage versions is required. Hence, we came up with the idea of garbage collection in MVOSTMs. We have implemented garbage collection for *MVOSTM* for both hash table and linked-list based approaches. Each transaction, in the beginning, logs its timestamp in a global list named as *All Live Transactions List (ALTL)*, which keeps track of all the live transactions in the system. Under the optimistic approach of STM, each transaction performs its updates in the shared memory in `STM.tryC()`. Each transaction in `STM.tryC()` performs some validations and if all validations are completed successfully a version of that key is created by that transaction. When a transaction goes to create a version of a key in the shared memory, it checks for the least timestamp live transaction present in the ALTL. If the current transaction is the one with least timestamp present in ALTL, then this transaction deletes all the older versions of the current key and create a version of its own.

If current transaction is not the least timestamp live transaction then it does not do any garbage collection. In this way, we ensure each transaction performs garbage collection on the keys it is going to create a version on. Once the transaction, changes its state to commit, it removes its entry from the ALTL. Figure 4.9 and Figure 4.10 demonstrate that *MVOSTM* with garbage collection (*HT-MVOSTM-GC* and *list-MVOSTM-GC*) performs better than *MVOSTM* without garbage collection.

Finite K-versions *MVOSTM (KOSTM)*: Another technique to efficiently use memory is to restrict the number of versions rather than using unbounded number of versions, without compromising on the benefits of multi-version. *KOSTM*, keeps

at most K -versions by deleting the oldest version when $(K + 1)^{th}$ version is created by a current transaction. That means, once a key reaches its maximum number of versions count K , no new version is created in a new memory location rather new version overrides the version with the oldest timestamp. To find the ideal value of K such that performance as compared to *MVOSTM-GC* does not degrade or can be increased, we perform experiments on two settings one on *High Contention Workload (C1)* and other on *Low Contention Workload (C2)*.

Under high contention *C1*, each thread spawns over 100 different transactions and each transaction performs 10% lookup, 45% insert, and 45% delete operations over 50 random keys. And under low contention *C2*, each thread spawns over one transaction and each transaction performs 10% lookup, 45% insert, and 45% delete operations over 1000 random keys. The best value of K is application dependent. Here, we used counter application (explained in SubSection 3.7.1) and get the best value of K is 4 as illustrated in Figure 4.17 under both contention settings.

Memory Consumption by *MVOSTM-GC* and *KOSTM*: As depicted above *KOSTM* performs better than *MVOSTM-GC*. Continuing the comparison between the two variations of *MVOSTM* we chose another parameter as memory consumption. Here, we test for the memory consumed by each variants of proposed algorithms in creating a version of a key. We count the total versions created, where creating a version increases the counter value by 1 and deleting a version decreases the counter value by 1. Our experiments, as shown in Figure 4.18, under the same contentions *C1* and *C2* show that *KOSTM* needs less memory space than *MVOSTM-GC*.

4.7 Summary

This chapter of the thesis presents the notion of *Multi-Version Object-based STMs* and compares their effectiveness with *Single-Version Object-based STMs* and *Multi-Version Read-Write STMs*. We find that multi-version object-based STM provides a significant benefit over both of these for different types of workloads. Specifically, we have evaluated the effectiveness of *MVOSTM* for two *Concurrent Data Structures CDS*, hash table and list as *HT-MVOSTM* and *list-MVOSTM* but its generic to other data structure as well. Initially, *HT-MVOSTM* and *list-MVOSTM* use unbounded number of versions for each key. To limit the number of versions, we developed two variants for both hash table and list data structures: (1) A *Garbage Collection (GC)* method in *MVOSTM* to delete the unwanted versions of a key, denoted as *MVOSTM-GC*. (2) Placing a limit of K on the number versions in *MVOSTM*, resulting in *KOSTM*. Both these variants gave a performance gain of over 15% over *MVOSTM*.

Experimental results of *HT-KOSTM* shows a significant performance gain of almost two to nineteen times better than existing state-of-the-art hash table based STMs (HT-OSTM [16] proposed by us in Chapter 3, ESTM [7], RWSTM [3], HT-MVTO [15], HT-KSTM [15]). Similarly, *list-KOSTM* provide almost two to twenty fold speedup over existing state-of-the-art list based STMs (list-OSTM [16] proposed by us in Chapter 3, Trans-list [12], Boosting-list [13], NOrec-list [6], list-MVTO [15], list-KSTM [15]).

Chapter 5

Optimized-Multi-Version OSTMs

5.1 Introduction

Nowadays, multi-core systems are in trend which necessitated the need for concurrent programming to exploit the cores appropriately. However, developing the correct and efficient concurrent programs is difficult. Software Transactional Memory Systems (STMs) are a convenient programming interface which assist the programmer to access the shared memory concurrently using multiple threads without worrying about consistency issues such as deadlock, livelock, priority-inversion, etc. STMs facilitate one more feature compositionality of concurrent programs with great ease which makes it more approachable. Different concurrent operations that need to be composed to form a single atomic unit is achieved by encapsulating them in a transaction.

In this chapter, we performed a few more optimizations on MVOSTM (proposed in Chapter 4) to further harness greater concurrency and propose the new notion of *Optimized Multi-Version OSTMs (or OPT-MVOSTMs)*. OPT-MVOSTM directly inherits all the benefits of proposed MVOSTM explained in Chapter 4. Our goal is to analyze the benefit of *OPT-MVOSTMs* over *Single and Multi-Version OSTMs*, *Single and Multi-Version RWSTMs* in this thesis. We did the following optimization on MVOSTM and propose OPT-MVOSTM:

- To reduce the *traversal time*, we have added max_{rvl} field corresponding to each version which contains the maximum timestamp of the transaction that looked up on this version explained in Section 5.2.
- To identify the *early abort* of the transaction, we are doing the validation in `STM_insert()` as well before `STM_tryC()` which prevents the work done by a

transactions that gone be abort in future. Hence, early validation in `STM_insert()` saves time and computational power consumed by aborted transactions as illustrated in Section 5.3.

- To make it *search efficient*, we applied the *Garbage Collection* on the *red-link (RL)* which contains the deleted node information represented in Section 5.5.

OPT-MVOSTM is a generic concept which can be applied to any data structure. In this chapter of the thesis, we have considered two *Concurrent Data Structures (CDS)*, hash table and list based *OPT-MVOSTMs* as *OPT-HT-MVOSTM* and *OPT-list-MVOSTM* respectively. If the bucket size B of hash table becomes *one* then hash table based *OPT-MVOSTMs* boils down to the list based *OPT-MVOSTMs*.

OPT-HT-MVOSTM and *OPT-list-MVOSTM* use an unbounded number of versions for each key. To address this issue, we developed two variants for both hash table and list data structures: (1) A *Garbage Collection (GC)* method in *OPT-MVOSTMs* to delete the unwanted versions of a key, denoted as *OPT-MVOSTM-GC*. Garbage collection gave an average performance gain of 16% over *OPT-MVOSTM* without garbage collection in the best case. Thus, the overhead of garbage collection scheme is less than the performance improvement due to improved memory usage. (2) Placing a limit of K on the number versions in *OPT-MVOSTM*, resulting in *OPT-KOSTM*. This gave an average performance gain of 24% over *OPT-MVOSTM* without garbage collection in the best case.

Contributions of this Chapter is as follows:

- We proposed a new notion of *Optimized Multi-Version Object-based STM* system as *OPT-MVOSTM* in Section 5.2. In this chapter of the thesis, we developed it for two CDS, hash table and list objects as *OPT-HT-MVOSTM* and *OPT-list-MVOSTM* respectively. *OPT-MVOSTM* is generic for other data structures as well.
- Section 5.4 shows that *OPT-HT-MVOSTM* and *OPT-list-MVOSTM* satisfy standard correctness-criterion of STMs, *opacity* [10].
- For efficient space utilization in *OPT-MVOSTMs* with unbounded versions, we developed *Garbage Collection* for *OPT-MVOSTM* (i.e. *OPT-MVOSTM-GC*) and bounded version *OPT-MVOSTM* (i.e. *OPT-KOSTM*) for both the hash table and list data structure.
- Experimental analysis of both hash table based *OPT-KOSTM (OPT-HT-KOSTM)* and list based *OPT-KOSTM (OPT-list-KOSTM)* with state-of-the-art STMs

are present in Section 5.5. Proposed *OPT-HT-KOSTM* and *OPT-list-KOSTM* provide greater concurrency and reduces the number of aborts as compared to single and multi-version OSTMs, single and multi-version *RWSTMs* while maintaining multiple versions corresponding to each key.

Roadmap. This chapter is organized as follows. Section 5.2 represents the *OPT-MVOSTMs* design and data structure. Section 5.3 shows the working of *OPT-HT-MVOSTMs* and its algorithms. We formally prove the correctness of *OPT-MVOSTMs* in Section 5.4. In Section 5.5 we show the experimental evaluation of *OPT-MVOSTMs* with state-of-art-STMs. Finally, we summaries this chapter in Section 5.6.

5.2 OPT-MVOSTM Design and Data Structure

This section describes the design and data structure of optimized *MVOSTMs* (or *OPT-MVOSTMs*). Here, we proposed hash table and list based *OPT-MVOSTMs* as *OPT-HT-MVOSTM* and *OPT-list-MVOSTM* respectively. *OPT-MVOSTMs* are generic for other data structure as well. *OPT-HT-MVOSTM* is a hash table based *OPT-MVOSTM* that explores the idea of multiple versions in *OSTMs* for hash table object to achieve greater concurrency. The design of *OPT-HT-MVOSTM* is extension to our proposed *HT-MVOSTM* [17] explained in Section 4.3. *OPT-HT-MVOSTM* comprises of hash table with B buckets. All the keys of the hash table in the range \mathcal{K} are statically allocated to one of these buckets.

Each bucket consists of linked-list of nodes along with two sentinel nodes *head* and *tail* with values $-\infty$ and $+\infty$ respectively. The structure of each node is as $\langle key, lock, marked, vl, RL, BL \rangle$. The *key* is a unique value from the set of all keys \mathcal{K} . All the nodes are stored in increasing order in each bucket as shown in Figure 5.1 (a), similar to any linked-list based concurrent set implementation [9, 28]. In the rest of the document, we use the terms key and node interchangeably. To perform any operation on a key, the corresponding *lock* is acquired. *marked* is a boolean field which represents whether the key is deleted or not. The deletion is performed in a lazy manner similar to the concurrent linked-lists structure [9]. If the *marked* field is true then key corresponding to the node has been logically deleted; otherwise, it is present. The *vl* field of the node points to the version list (shown in Figure 5.1 (b)) which stores multiple versions corresponding to the key. The last two fields of the node is red link (or **RL**) and blue link (or **BL**) which stores the address of the next node. The node which is not marked (or not deleted) are accessible from the head via **BL**. While all the nodes including the marked ones can be accessed from the head via **RL**. We denote it as *lazy red-blue list* or *lazyrb-list*. Further, for a bucket B , we

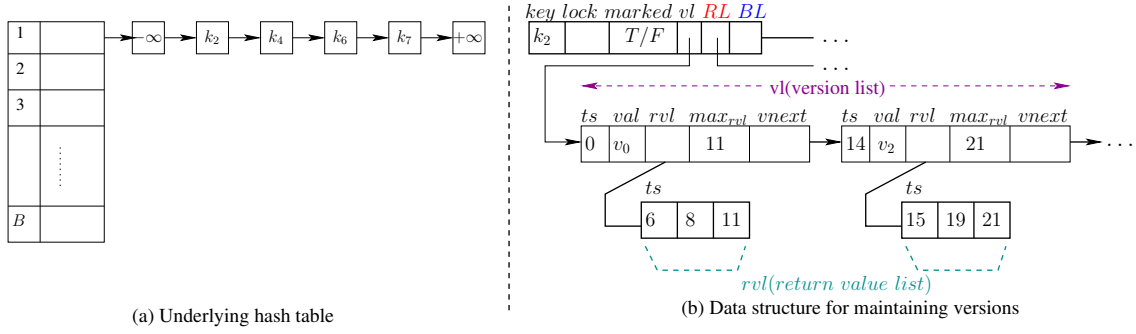


Figure 5.1: Optimized *HT-MVOSTM* design

denote its linked-list as $B.lazyrb-list$. Given a node n in the linked-list of bucket B with key k , we denote its fields as $n.key$ (or $k.key$), $n.lock$ (or $k.lock$), $n.marked$ (or $k.marked$), $n.vl$ (or $k.vl$), $n.RL$ (or $k.RL$), $n.BL$ (or $k.BL$).

The structure of each version in the vl of a key k is $\langle ts, val, rvl, max_{rvl}, vnext \rangle$ as shown in Figure 5.1 (b). The field ts denotes the unique timestamp of the version. In our algorithm, every transaction is assigned a unique timestamp when it begins which is also its id . Thus ts of this version is the timestamp of the transaction that created it. All the versions in the vl of k are sorted by ts . Since the timestamps are unique, we denote a version, ver of a node n with key k having ts j as $n.vl[j].ver$ or $k.vl[j].ver$. The corresponding fields in the version as $k.vl[j].ts$, $k.vl[j].val$, $k.vl[j].rvl$, $k.vl[j].max_{rvl}$, $k.vl[j].vnext$.

The field val contains the value updated by an update transaction. If this version is created by an insert method $STM_insert_i(ht, k, v)$ by transaction T_i , then val will be v . On the other hand, if the method is $STM_delete_i(ht, k, v)$ then val will be $null$. In this case, as per the algorithm, the node of key k will also be marked. *OPT-HT-MVOSTM* algorithm does not immediately physically remove deleted keys from the hash table to ensures the correctness criteria as opacity explained in Section 4.3 of Chapter 4. Thus an rv_method ($STM_delete()$ or $STM_lookup()$) on key k can return $null$ when it does not find the key or encounters a $null$ value for k .

The rvl field stands for *return value list* which is a list of all the transactions that executed rv_method on this version, i.e., those transactions which returned val . **The first optimization in *OPT-HT-MVOSTM* to reduce the traversal time of rvl , we have used max_{rvl} which contains the maximum ts of the transaction that executed rv_method on this version.** This optimization makes *OPT-HT-MVOSTM* to be search efficient as compared to proposed *MVOSTM* described in Chapter 4. The field $vnext$ points to the next available version of that key.

In order to increase the efficiency and utilize the memory properly, we proposed two variants of *OPT-HT-MVOSTM* as follows: First, we apply *Garbage Collection (or GC)* on the versions and proposed *OPT-HT-MVOSTM-GC*. It maintains unbounded

versions in vl (the length of the list) while deleting the unwanted versions using garbage collection scheme. Second, we proposed *OPT-HT-KOSTM* which maintains the bounded number of versions such as K and improves the efficiency further. Whenever a new version ver is created and is about to be added to vl , the length of vl is checked. If the length becomes greater than K , the version with lowest ts (i.e., the oldest) is replaced with the new version ver and thus maintaining the length back to K .

We proposed *OPT-list-MVOSTMs* while considering the bucket size as 1 in *OPT-HT-MVOSTM*. Along with this, we proposed two variants of *OPT-list-MVOSTM* as *OPT-list-MVOSTM-GC* and *OPT-list-KOSTM* which applies the garbage collection scheme in unbounded versions and bounded K versions for list based object respectively similar to *OPT-HT-MVOSTM*.

5.3 The Working of OPT-MVOSTM

OPT-HT-MVOSTM exports *STM_begin()*, *STM_insert()*, *STM_delete()*, *STM_lookup()*, and *STM_tryC()* methods same as our proposed MVOSTM explained in Section 4.1. Among them *STM_delete()*, *STM_lookup()* are return-value methods (or *rv_methods*) while *STM_insert()*, *STM_delete()* are update methods (or *upd_methods*). We treat *STM_delete()* as both *rv_method* as well as *upd_method*. The *rv_methods* return the current value of the key. The *upd_methods*, update to the keys are first noted down in the local log, *txLog*. Then in the *STM_tryC()* method after successful validations of these updates are transferred to the shared memory. Now, we explain the working of each method as follows:

***STM_begin()*:** It is same as *STM_begin()* of MVOSTM in Section 4.4 of Chapter 4.
***rv_methods*:** It can be either *STM_delete(ht, k, v)* or *STM_lookup(ht, k, v)*. Both these methods return the current value of key k . Algorithm 30 gives the high level overview of these methods. First, the algorithm checks to see if the given key is already in the local log, *txLog_i* of T_i (Line 9). If the key is already there then the current *rv_method* is not the first method on k and is a subsequent method of T_i on k . So, we can return the value of k from the *txLog_i*.

If the key is not present in the *txLog_i*, then *OPT-HT-MVOSTM* searches into shared memory. Specifically, it searches the bucket to which k belongs to. Every key in the range \mathcal{K} is statically allocated to one of the B buckets. So the algorithms search for k in the corresponding bucket, say B_k to identify the appropriate location, i.e., identify the correct *predecessor* or *pred* and *current* or *curr* keys in the lazyrb-list of B_k without acquiring any locks similar to the search in lazy-list [9]. Since each key

has two links, **RL** and **BL**, the algorithm identifies four node references: two *pred* and two *curr* according to red and blue links. They are stored in the form of an array with *preds*[0] and *currs*[1] corresponding to blue links; *preds*[1] and *currs*[0] corresponding to red links. If both *preds*[1] and *currs*[0] nodes are unmarked then the *pred*, *curr* nodes of both red and blue links will be the same, i.e., *preds*[0] = *preds*[1] and *currs*[0] = *currs*[1]. Thus depending on the marking of *pred*, *curr* nodes, a total of two, three or four different nodes will be identified. Here, the search ensures that *preds*[0].key ≤ *preds*[1].key < k ≤ *currs*[0].key ≤ *currs*[1].key.

Algorithm 30 *rv_method*(*ht*, *k*, *v*): It can be either *STM_delete_i*(*ht*, *k*, *v*) or *STM_lookup_i*(*ht*, *k*, *v*) on key *k* that maps to bucket *B_k* of hash table *ht*.

```

8: procedure rv_methodi(ht, k, v)
9:   if (k ∈ txLogi) then
10:     Update the local log and return val.
11:   else
12:     Search in lazyrb-list to identify the preds[] and currs[] for k using BL and
RL in bucket Bk.
13:     Acquire the locks on preds[] and currs[] in increasing order.
14:     if (!rv_Validation(preds[], currs[])) then
15:       Release the locks and goto Line 12.
16:     end if
17:     if (k ∉ Bk.lazyrb-list) then
18:       Create a new node n with key k as: ⟨ key = k, lock = false, marked =
true, vl = ver, RL =  $\phi$ , BL =  $\phi$  ⟩.
19:       /*The vl consists of a single element ver with ts as 0*/
20:       Create the version ver as: ⟨ts = 0, val = null, rvl = i, maxrvl = i, vnext =  $\phi$ ⟩.
21:       Insert n into Bk.lazyrb-list such that it is accessible only via RLs. /*n
is marked*/
22:       Release the locks; update the txLogi with k.
23:       return ⟨null⟩.
24:     end if
25:     Identify the version verj with ts = j such that j is the largest timestamp
smaller than i.
26:     Add i into the rvl of verj.
27:     if (verj.maxrvl < i) then
28:       Set verj.maxrvl to i.
29:     end if
30:     retVal = verj.val.
31:     Release the locks; update the txLogi with k and retVal.
32:   end if
33:   return ⟨retVal⟩.
34: end procedure

```

Next, the re-entrant locks on all the *pred*, *curr* keys are acquired in increasing order to avoid the deadlock. Then all the *pred* and *curr* keys are validated by *rv-Validation()* in Line 14 as follows: (1) If *pred* and *curr* nodes of blue links are not marked, i.e., ($\neg pred[0].marked$) && ($\neg curr[1].marked$). (2) If the next links of both blue and red *pred* nodes point to the correct *curr* nodes: ($preds[0].BL = curr[1]$) && ($preds[1].RL = curr[0]$).

If any of these checks fail, then the algorithm retries to find the correct *pred* and *curr* keys. It can be seen that the validation check is similar to the validation in concurrent lazy-list [9].

Next, we check if *k* is in $B_k.lazyrb-list$. If *k* is not in B_k , then we create a new node *n* for *k* as: $\langle key = k, lock = false, marked = true, vl = ver, RL = \phi, BL = \phi \rangle$ and insert it into $B_k.lazyrb-list$ such that it is accessible only via **RL**. This node will have a single version *ver* as $\langle ts = 0, val = null, rvl = i, max_{rvl} = i, vnext = \phi \rangle$. Here invoking transaction T_i is creating a version with timestamp 0 to ensure that *rv-*methods of other transactions will never abort as explained in Section 4.3 of Chapter 4 for proposed MVOSTM. Such that, each *rv_method* will find a version to read while maintaining the infinite version corresponding to each key *k*. *marked* field sets to true because it access by **RL** only. In *rvl* and *max_{rvl}*, T_i adds the timestamp as *i* in it and *vnext* is initialized to empty value. Since *val* is null and the *n*, this version and the node are not technically inserted into $B_k.lazyrb-list$.

If *k* is in $B_k.lazyrb-list$ then, *k* is the same as $curr[0]$ or $curr[1]$ or both. Let *n* be the node of *k* in $B_k.lazyrb-list$. We then find the version of *n*, *ver_j* which has the timestamp *j* such that *j* has the largest timestamp smaller than *i* (timestamp of T_i). Add *i* to *ver_j*'s *rvl* (Line 26). *max_{rvl}* maintains the maximum timestamp among all *rv_methods* read from this version at Line 28. Then release the locks, update the local log *txLog_i* in Line 31 and return the value stored in *ver_j.val* in Line 33.

STM_insert(): This is **another optimization** done in *OPT-HT-MVOSTMs* to identify the *early abort* of the transaction, we are doing the validation in *STM_insert()* as well before *STM_tryC()* which prevents the work done by a transactions that gone be abort in future. Hence, early validation in *STM_insert()* saves time and computation power consumed by aborted transactions.

The actual effect of the *STM_insert()* comes after the successful *STM_tryC()* method. First, *STM_insert()* searches the key *k* in the local log, *txLog_i* of T_i at Line 36. If *k* does not exist in the *txLog_i* then it identifies the appropriate location (*pred* and *curr*) of key *k* using **BL** and **RL** (Line 37) in the lazyrb-list of B_k without acquiring any locks similar to *rv_method* explained above.

Next, it acquires the re-entrant locks on all the *pred* and *curr* keys in increasing

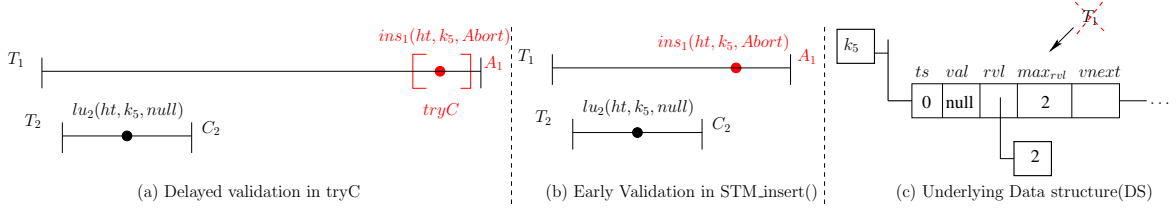


Figure 5.2: Advantage of early validation in `STM_insert()`

order. After that, all the *pred* and *curr* keys are validated by `tryC_Validation()` in Line 39 as follows: (1) It does the `rv_Validation()` as explained above in the `rv_` method. (2) If key k exists in the $B_k.lazyrb-list$ and let n as a node of k . Then algorithm identifies the version of n , ver_j which has the timestamp j such that j has the largest timestamp smaller than i (timestamp of T_i) at Line 87. If max_{rvl} of ver_j is greater than timestamp i at Line 88 then it returns `Abort` in Line 40.

`tryC_Validation()` in `STM_insert()` identifies the early abort of invalid transaction. The advantage of doing the early validation to save the significant computation of long running transaction which will abort in the future. Consider Figure 5.2 where two transaction T_1 and T_2 working on key k_5 . In Figure 5.2 (a), T_1 aborts in `STM_tryC()` (delayed validation) because higher timestamp T_2 committed. But in Figure 5.2 (b), T_1 validates the `STM_insert()` instantly by looking into the max_{rvl} of k_5 as shown in Figure 5.2 (c) and save its computation and returns abort.

Algorithm 31 `STM_insert()`: Actual insertion happens in the `STM_tryC()`.

```

35: procedure STM_insert()
36:   if ( $k \notin txLog_i$ ) then
37:     Search in lazyrb-list to identify the preds[] and currs[] for  $k$  using BL and
     RL in bucket  $B_k$ .
38:     Acquire the locks on preds[] and currs[] in increasing order.
39:     if (!tryC_Validation()) then
40:       return Abort. /*Release the locks*/
41:     end if
42:     Release the locks.
43:   else
44:     Update the local log.
45:   end if
46: end procedure

```

upd_methods: It can be either `STM_insert(ht, k, v)` or `STM_delete(ht, k, v)`. Both the methods create a version corresponding to the key k . The actual effect of `STM_insert()` and `STM_delete()` in shared memory will take place in `STM_tryC()`. Algorithm 32 represents the high-level overview of `STM_tryC()`.

Initially, to avoid deadlocks, the algorithm sorts all the *keys* in increasing order

which are present in the local log, $txLog_i$. In $STM_tryC()$, $txLog_i$ consists of upd-methods ($STM_insert()$ or $STM_delete()$) only. For all the upd-methods (opn_i) it searches the key k in the shared memory corresponding to the bucket B_k . It identifies the appropriate location ($pred$ and $curr$) of key k using **BL** and **RL** (Line 52) in the lazyrb-list of B_k without acquiring any locks similar to rv_method explained above.

Next, it acquires the re-entrant locks on all the $pred$ and $curr$ keys in increasing order. After that, all the $pred$ and $curr$ keys are validated by $tryC_Validation()$ in Line 54 as explained in $STM_insert()$.

If $tryC_Validation()$ is successful then each upd-methods exist in $txLog_i$ will take the effect in the shared memory after doing the $intraTransValidation()$ in Line 60. If two $upd_methods$ of the same transaction have at least one common shared node among its recorded $pred$ and $curr$ keys, then the previous upd_method effect may overwrite if the current upd_method of $pred$ and $curr$ keys are not updated according to the updates are done by the previous upd_method . Thus to solve this we have $intraTransValidation()$ that modifies the $pred$ and $curr$ keys of current operation based on the previous operation in Line 60.

Next, we check if upd-method is $STM_insert()$ and k is in $B_k.lazyrb-list$. If k is not in B_k , then create a new node n for k as $\langle key = k, lock = false, marked = false, vl = ver, RL = \phi, BL = \phi \rangle$. This node will have two versions ver as $\langle ts = 0, val = null, rvl = \phi, max_{rvl} = \phi, vnext = i \rangle$ for T_0 and $\langle ts = i, val = v, rvl = \phi, max_{rvl} = \phi, vnext = \phi \rangle$ for T_i . T_i is creating a version with timestamp 0 to ensure that $rv_methods$ of other transactions will never abort. For second version, i is the timestamp of the transaction T_i invoking this method; $marked$ field sets to false because the node is inserted in the **BL**. rvl , max_{rvl} , and $vnext$ are initialized to empty values. We set the val as v and insert n into $B_k.lazyrb-list$ such that it is accessible via **RL** as well as **BL** and set the lock field to be $true$ (Line 64).

If k is in $B_k.lazyrb-list$ then, k is the same as $currs[0]$ or $currs[1]$ or both. Let n be the node of k in $B_k.lazyrb-list$. Then, we create the version ver as: $\langle ts = i, val = v, rvl = \phi, max_{rvl} = \phi, vnext = \phi \rangle$ and insert the version into $B_k.lazyrb-list$ such that it is accessible via **RL** as well as **BL** (Line 66).

Subsequently, we check if upd-method is $STM_delete()$ and k is in $B_k.lazyrb-list$. Let n be the node of k in $B_k.lazyrb-list$. Then create the version ver as $\langle ts = i, val = null, rvl = \phi, max_{rvl} = \phi, vnext = \phi \rangle$ and insert the version into $B_k.lazyrb-list$ such that it is accessible only via **RL** (Line 69).

Finally, at Line 71 it updates the $pred$ and $curr$ of opn_i in local log, $txLog_i$. At Line 73 releases the locks on all the $pred$ and $curr$ in increasing order of keys to avoid deadlocks and return $Commit$.

Algorithm 32 *STM_tryC(T_i)*: Validate the upd_methods of the transaction and then commit.

```

47: procedure STM_tryC(Ti)
48:   /*Operation name (opn) could be either STM_insert() or STM_delete()*/
49:   /*Sort the keys of txLogi in increasing order.*/
50:   for all (opni ∈ txLogi) do
51:     if ((opni == STM_insert()) || (opni == STM_delete())) then
52:       Search in lazyrb-list to identify the preds[] and currs[] for k using BL
       and RL in bucket Bk.
53:       Acquire the locks on preds[] and currs[] in increasing order.
54:       if (!tryC_Validation()) then
55:         return ⟨Abort⟩. /*Release the locks*/
56:       end if
57:     end if
58:   end for
59:   for all (opni ∈ txLogi) do
60:     intraTransValidation() modifies the preds[] and currs[] of current
       operation which would have been updated by the previous operation of
       the same transaction.
61:     if ((opni == STM_insert()) && (k ∉ Bk.lazyrb-list)) then
62:       Create new node n with k as: ⟨key = k, lock = false, marked = false,
       vl = ver, RL = φ, BL = φ⟩.
63:       Create two versions ver as: ⟨ts=0, val=null, rvl=φ, maxrvl = φ,
       vnext=i⟩ for T0 and ⟨ts=i, val=v, rvl=φ, maxrvl = φ, vnext=φ⟩
       for Ti.
64:       Insert node n into Bk.lazyrb-list such that it is accessible via RL as
       well as BL /*lock sets true*/.
65:     else if (opni == STM_insert()) then
66:       Add the version ver as: ⟨ts=i, val=v, rvl=φ, maxrvl=φ, vnext=φ⟩
       into Bk.lazyrb-list such that it is accessible via RL as well as BL.
67:     end if
68:     if (opni == STM_delete()) then
69:       Add the version ver as: ⟨ts=i, val=null, rvl=φ, maxrvl=φ, vnext=φ⟩
       into Bk.lazyrb-list such that it is accessible only via RL.
70:     end if
71:     Update the preds[] and currs[] of opni in txLogi.
72:   end for
73:   Release the locks; return ⟨Commit⟩.
74: end procedure

```

We illustrate the helping methods of `rv_method`, `STM_insert()`, and `upd_method` in detail as follows:

rv_Validation(): It is same as `rv_Validation()` of MVOSTM in Section 4.4 of Chapter 4.

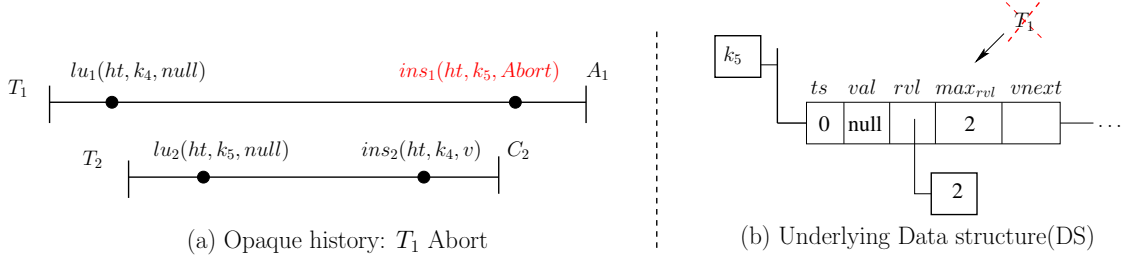


Figure 5.3: Illustration of `tryC_Validation()`

tryC_Validation(): It is called by `STM_insert()`, and `upd_method` in `STM_tryC()`. First, it does the `rv_Validation()` in Line 83. If its successful and key k exists in the $B_k.lazyrb-list$ and let n as a node of k . Then algorithm identifies the version of n , ver_j which has the timestamp j such that j has the largest timestamp smaller than i (timestamp of T_i) at Line 87. If max_{rvl} of ver_j is greater than the timestamp of i then the algorithm returns false (in Line 89) and eventually, returns `Abort` in Line 40 or Line 55. Consider an example as shown in Figure 5.3 (a), where second method $ins_1(ht, k_5)$ of transaction T_1 returns `Abort` because higher timestamp of transaction T_2 is already present in the max_{rvl} of version T_0 identified by T_1 in Figure 5.3 (b).

Algorithm 33 `tryC_Validation()`: It maintains the order among the transactions.

```

82: procedure tryC_validation()
83:   if (!rv_Validation(preds[], currs[])) then
84:     Release the locks and retry.
85:   end if
86:   if (k ∈ B_k.lazyrb-list) then
87:     Identify the version ver_j with ts = j such that j is the largest timestamp
      smaller than i.
88:     if (ver_j.max_rvl > i) then
89:       return ⟨false⟩.
90:     end if
91:   end if
92:   return ⟨true⟩.
93: end procedure

```

intraTransValidation(): It is same as *intraTransValidation()* of MVOSTM in Section 4.4 of Chapter 4.

5.4 Correctness of OPT-MVOSTM

This section describes about the correctness of *OPT-MVOSTM*. Since, *OPT-MVOSTM* satisfies all the properties of proposed *MVOSTM* (explained in Section 4.2) along with graph $OPG(H, \ll)$ construction of history H for a given version order \ll . Even, the vertices and the edges of graph $OPG(H, \ll)$ generated by *OPT-MVOSTM* is same as the vertices and the edges of graph defined for the *MVOSTM*. So, the correctness of *OPT-MVOSTM* is same as the correctness of proposed *MVOSTM* explained in Section 4.5. The main theorems for *OPT-MVOSTM* is as follows:

Theorem 58 *Any history H_{to} generated by *OPT-MVOSTM* is opaque.*

Proof: To prove this theorem, we need to show that the graph generated by *OPT-MVOSTM*, $G_{to} = OPG(H_{to}, \ll_{to})$ is acyclic. The proof of this statement is directly implies from the Lemma 54 of Section 4.5. It ensures that all the edges of the graph G_{to} is following the increasing order of their timestamp. So, if there is an edge from T_j to T_i then the $j < i$. Hence, graph G_{to} is acyclic. Thus from Theorem 47, we get that any history H_{to} generated by *OPT-MVOSTM* is opaque.

Theorem 59 *OPT-MVOSTM with unbounded versions ensures that *rv_methods* do not abort.*

Proof: This is self-explanatory with the help of *OPT-MVOSTM* algorithm because each *key* is maintaining multiple versions in the case of unbounded versions. So *rv_method* always finds a correct version to read it from. Thus, *rv_methods* of *OPT-MVOSTM* with unbounded versions do not *abort*.

5.5 Experimental Evaluations

This section describes the experimental analysis of proposed *OPT-MVOSTMs* with state-of-the-art STMs. We have three main goals in this section: (1) Analyze the performance benefits of the *Optimized Multi-Version Object-based STMs* (or *OPT-MVOSTMs*) over proposed *Multi-Version Object-based STMs* (or *MVOSTMs*) explained in Chapter 4. (2) Evaluate the benefits of *OPT-MVOSTMs* over the *Single-Version Object-based STMs* (or *OSTMs*), and (3) Analyze the benefits of *OPT-*

MVOSTMs over *Single and Multi-Version Read-Write STMs*. We implement hash table object and list object as *OPT-HT-MVOSTM* and *OPT-list-MVOSTM* described in Section 5.3.

We also consider the extension of this optimized multi-version object-based STMs to reduce memory usage. Specifically, we consider a variant that implements garbage collection with unbounded versions and another variant where the number of versions never exceeds a given threshold K for both *OPT-HT-MVOSTMs* and *OPT-list-MVOSTMs*.

STM implementations: To show the performance of proposed OPT-MVOSTM against state-of-the-art STMs, we have taken the implementation of NOrec-list [6], Boosting-list [13], Trans-list [12], ESTM [7], and RWSTM directly from the TLDS framework¹. Along with this, we have considered the implementation of MVOSTM [17], OSTM [16] proposed by us in Section 4.6 and Section 3.7 respectively and MVTO [15] defined in PDCRL library². We implemented our algorithms in C++. We used counter application (defined in SubSection 3.7.1) where each STM algorithm first creates N-threads, each thread, in turn, spawns a transaction. Each transaction exports *STM_begin()*, *STM_insert()*, *STM_lookup()*, *STM_delete()* and *STM_tryC()* methods as described in Section 5.3.

Methodology:³ We have considered three types of workloads: (W1) Lookup Intensive (90% lookup, 5% insert, and 5% delete), (W2) Mid Intensive (50% lookup, 25% insert, and 25% delete), and (W3) Update Intensive (10% lookup, 45% insert, and 45% delete). The experiments are conducted by varying number of threads from 2 to 64 in power of 2, with 1000 keys randomly chosen. We assume that the hash table of *OPT-HT-MVOSTM* has five buckets and each of the bucket (or list in case of *OPT-list-MVOSTM*) can have a maximum size of 1000 keys. Each transaction, in turn, executes 10 operations which include *STM_lookup()*, *STM_delete()*, and *STM-insert()* operations. For accuracy, we take an average over 10 results for the final result in which the first run is discarded and considered as a warm-up result for each experiment.

5.5.1 Result Analysis

Observations of *OPT-KOSTM* is same as observations of *KOSTM* defined in Section 4.6 of Chapter 4. The performance benefit of proposed optimized hash table

¹TLDS Framework: <https://ucf-cs.github.io/tlds/>

²PDCRL Library: <https://github.com/PDCRL>

³Proposed OPT-MVOSTM code is available here: <https://github.com/PDCRL/MVOSTM/tree/master/OPT-MVOSTM>

based *MVOSTM* (OPT-HT-MVOSTM) and its variants (OPT-HT-MVOSTM-GC, OPT-HT-KOSTM) for hash table objects is demonstrated in Figure 5.4. It shows *OPT-HT-KOSTM* performs best among all its variants (OPT-HT-MVOSTM-GC, OPT-HT-MVOSTM) by a factor of 1.02, 1.11 for workload W1, by a factor of 1.06, 1.09 for workload W2, and by a factor of 1.01, 1.03 for workload W3 respectively. Along with this, Figure 5.5 shows the abort count for respective algorithms on workload W1, W2, and W3. It represents the number of aborts are almost same for lesser number of threads in all the algorithms. But while increasing the number of threads, the number of aborts are least in *OPT-HT-KOSTM* as compare to its variants.

So, we compared the performance of *OPT-HT-KOSTM* with the state-of-the-art hash table based STMs as shown in Figure 5.6. *OPT-HT-KOSTM* outperforms all the state-of-the-art hash table based STMs (HT-KOSTM [17] proposed in Chapter 4, HT-OSTM [16] proposed in Chapter 3, ESTM [7], RWSTM [3], HT-MVTO [15], HT-KSTM [15]) by a factor of 1.05, 3.62, 3.95, 3.44, 2.75, 1.85 for workload W1, by a factor of 1.07, 1.44, 3.36, 5.45, 10.84, 8.42 for workload W2, and by a factor of 1.07, 2.11, 5.1, 19.8, 70.3, 60.23 for workload W3 respectively. The corresponding number of aborts are represented in Figure 5.7. Number of aborts are minimum for *OPT-HT-KOSTM* as compare to other state-of-the-art STMs. Especially, the number of aborts for *OPT-HT-KOSTM* is almost negligible as compared to HT-OSTM on lookup-intensive workload (W1) because *OPT-HT-KOSTM* finds a correct version to looks up and does not return abort as shown in Figure 5.7 (a).

The observation of optimized list based *MVOSTM* is similar as optimized hash table based *MVOSTM*. Figure 5.8 represents the performance benefit of proposed optimized list based *MVOSTM* (OPT-list-MVOSTM) with all its variants (OPT-list-MVOSTM-GC, OPT-list-KOSTM) for list objects. It shows *OPT-list-KOSTM* performs best among its variants (OPT-list-MVOSTM-GC, OPT-list-MVOSTM) by a factor of 1.14, 1.24 for W1, by a factor of 1.06, 1.07 for W2, and by a factor of 1.09, 1.19 for W3 respectively. Along with this, Figure 5.9 shows the minimum abort count by *OPT-list-KOSTM* as compare to its variants on workload W1, W2, and W3. Hence, we choose the best-proposed algorithm *OPT-list-KOSTM* and compare with the state-of-the-art list based STMs.

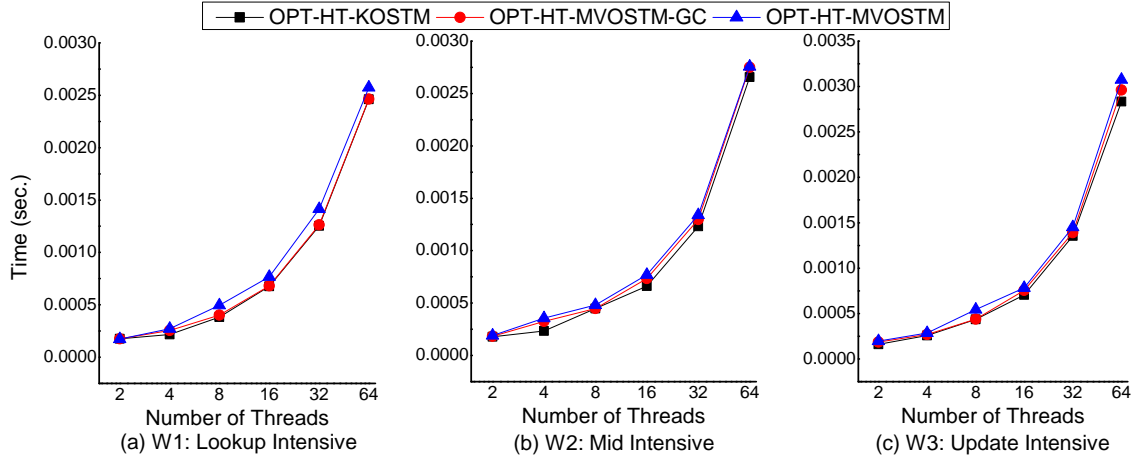


Figure 5.4: Performance of *OPT-HT-KOSTM* and its variants (*OPT-HT-MVOSTM-GC*, *OPT-HT-MVOSTM*)

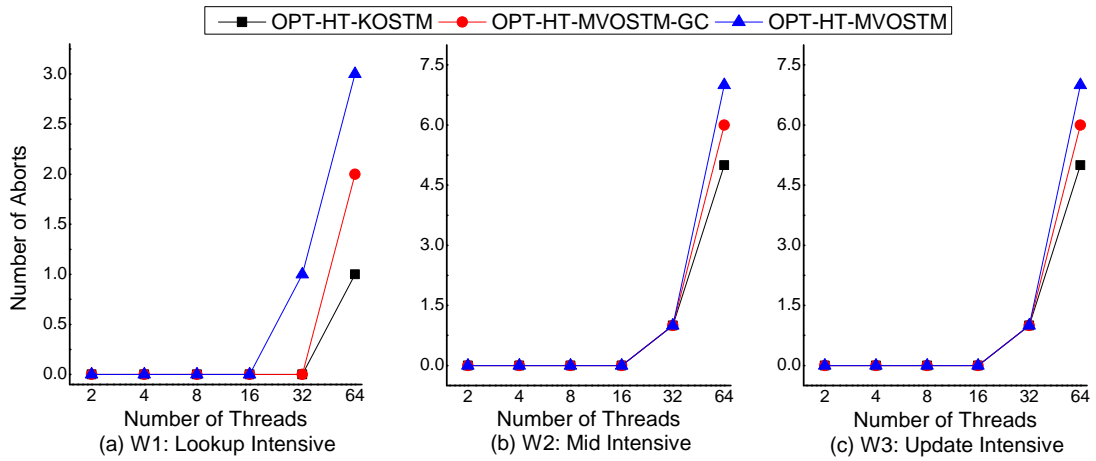


Figure 5.5: Abort Count of *OPT-HT-KOSTM* and its variants (*OPT-HT-MVOSTM-GC*, *OPT-HT-MVOSTM*)

Figure 5.10 represents *OPT-list-KOSTM* outperforms state-of-the-art list based STMs (*list-KOSTM* [17] proposed in Chapter 4, *list-OSTM* [16] proposed in Chapter 3, *Trans-list* [12], *Boosting-list* [13], *NOrec-list* [6], *list-MVTO* [15], *list-KSTM* [15]) by a factor of 1.2, 2.56, 25.38, 23.57, 27.44, 13.1, 6.8 for W1, by a factor of 1.12, 2.11, 21.54, 26.27, 30.1, 27.89, 20.1 for W2, and by a factor of 1.11, 2.91, 36.1, 42.2, 48.89, 149.92, 114.89 for W3 respectively. Similarly, Figure 5.11 depicts that *OPT-list-KOSTM* obtained the least number of aborts as compare to state-of-the-art STMs for all the workloads.

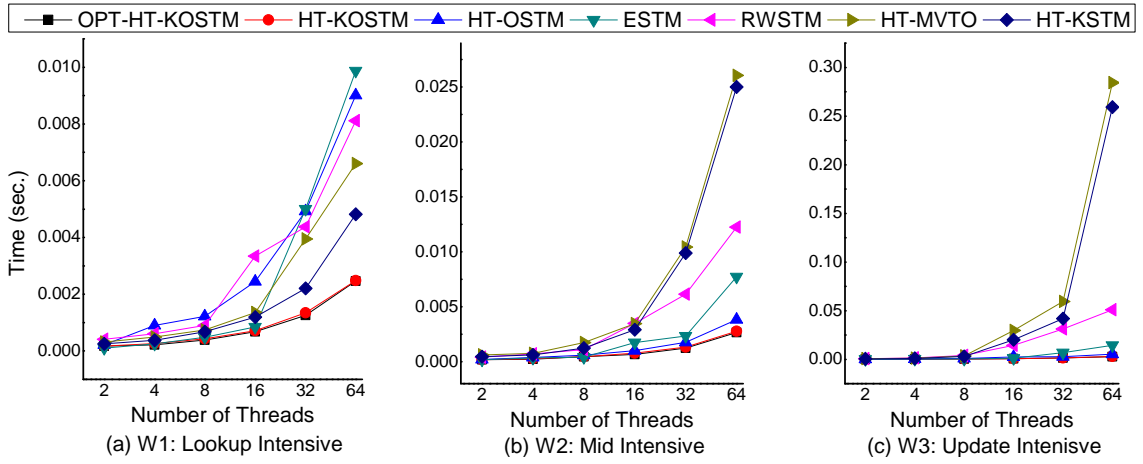


Figure 5.6: Performance of *OPT-HT-KOSTM* against State-of-the-art hash table based STMs

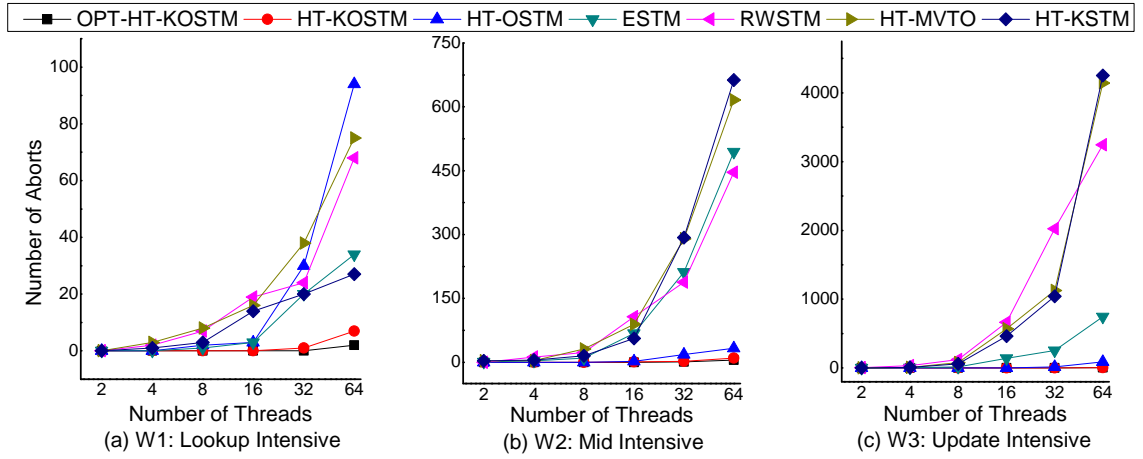


Figure 5.7: Abort Count of *OPT-HT-KOSTM* against State-of-the-art hash table based STMs

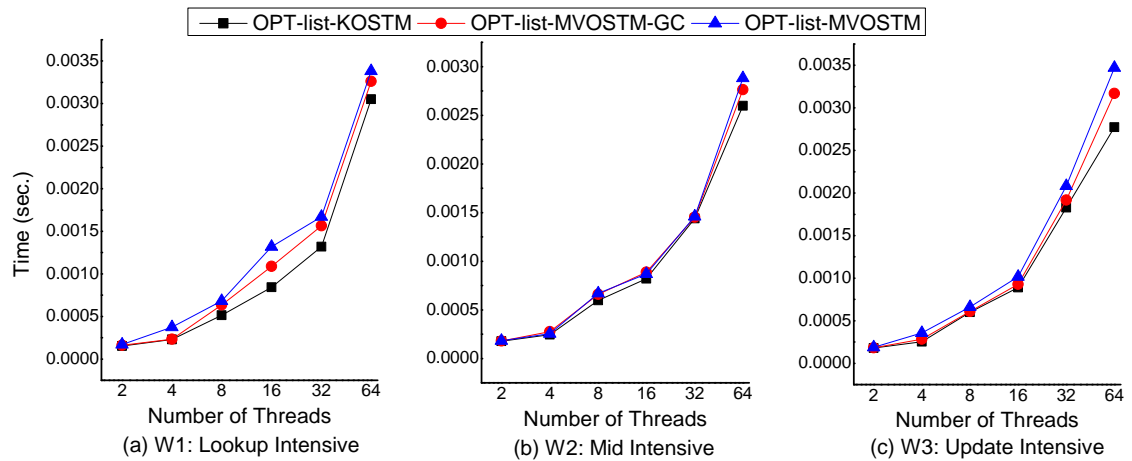


Figure 5.8: Performance of *OPT-list-KOSTM* and its variants (*OPT-list-MVOSTM-GC*, *OPT-list-MVOSTM*)

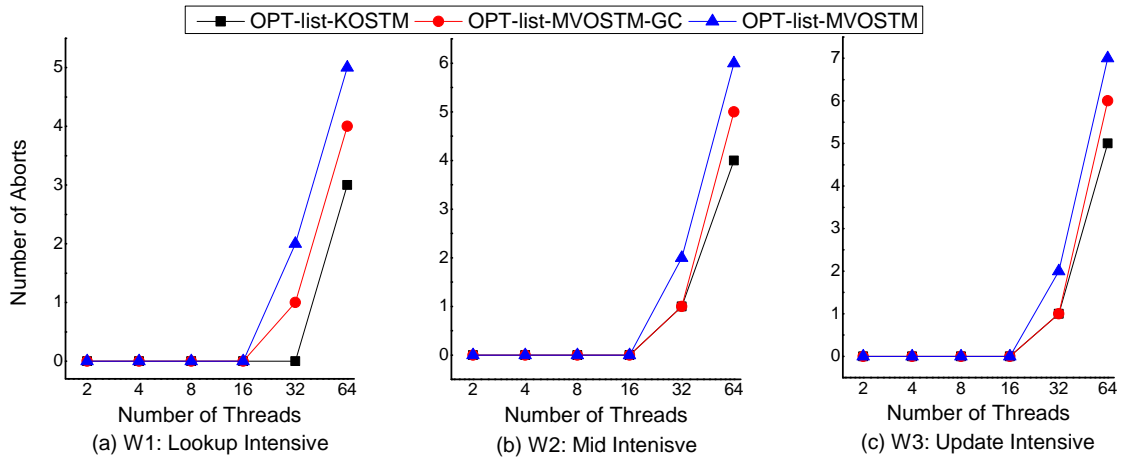


Figure 5.9: Abort Count of *OPT-list-KOSTM* and its variants (*OPT-list-MVOSTM-GC*, *OPT-list-MVOSTM*)

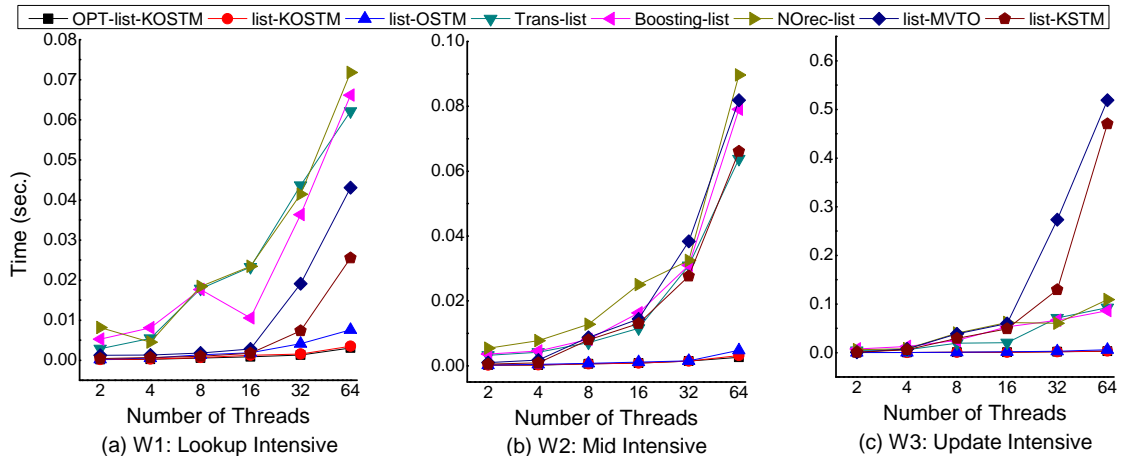


Figure 5.10: Performance of *OPT-list-KOSTM* against State-of-the-art list based STMs

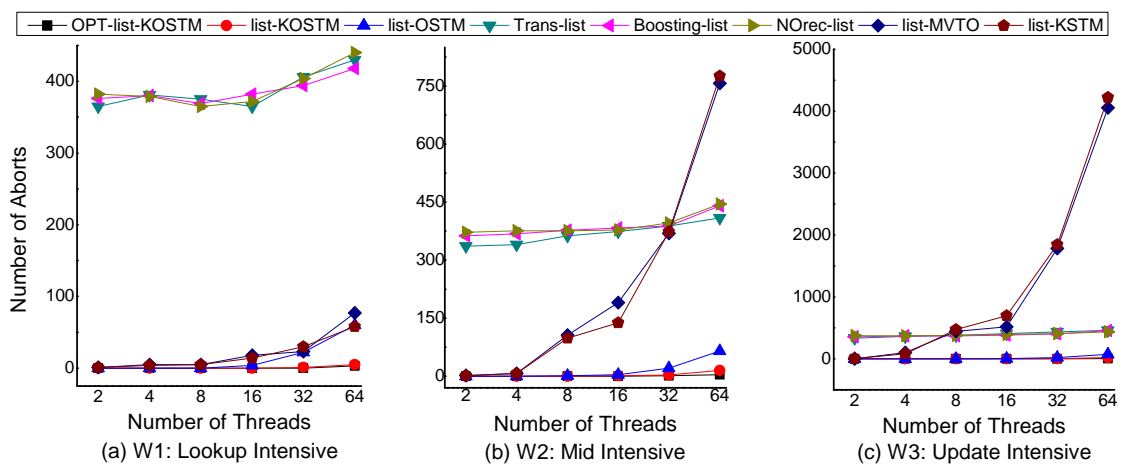


Figure 5.11: Abort Count of *OPT-list-KOSTM* against State-of-the-art list based STMs

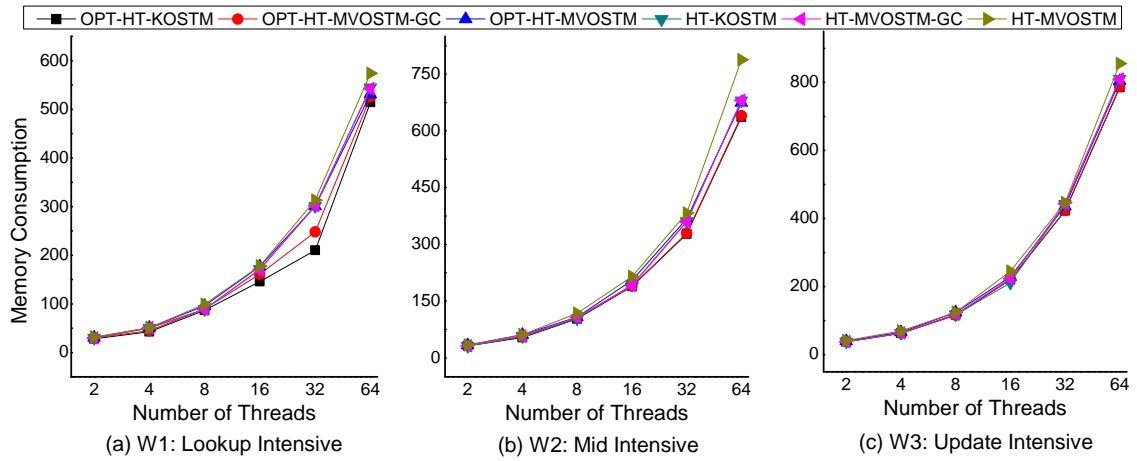


Figure 5.12: Memory consumption among variants of *OPT-HT-KOSTMs* and *HT-KOSTMs* on hash table

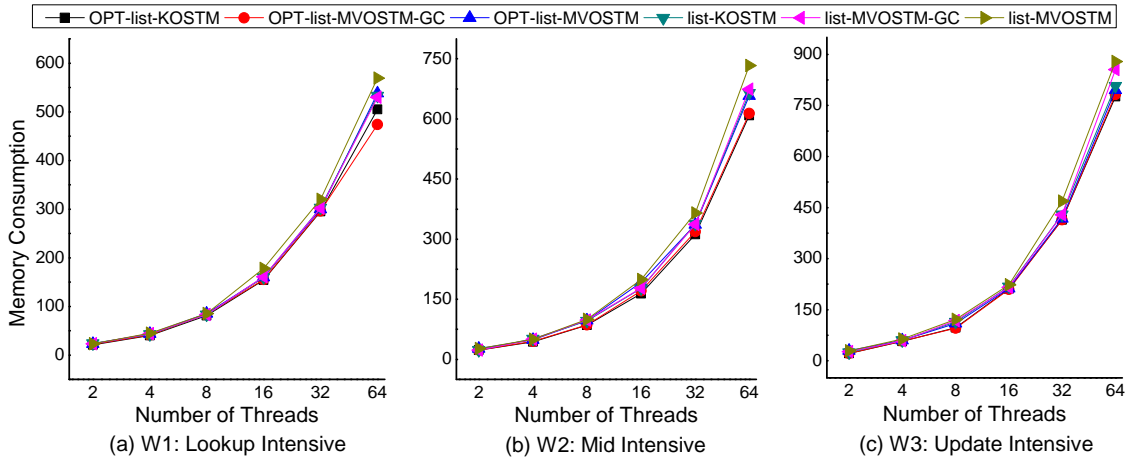


Figure 5.13: Memory consumption among variants of *OPT-list-KOSTMs* and *list-KOSTMs* on list

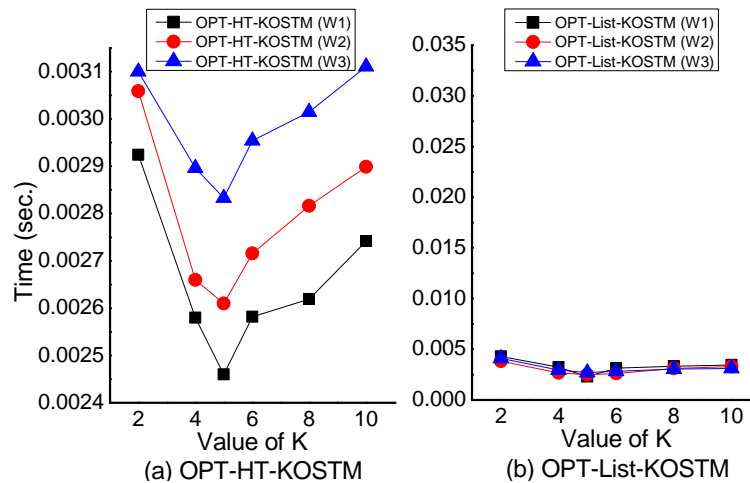


Figure 5.14: Optimal Value of K for *OPT-HT-KOSTM* and *OPT-list-KOSTM*

Garbage Collection in OPT-MVOSTM (OPT-MVOSTM-GC): As explained in Section 5.3, for efficient memory utilization, we developed two variations of *OPT-MVOSTM*. The first, *OPT-MVOSTM-GC*, uses unbounded versions but performs garbage collection. This is achieved by deleting non-latest versions whose timestamp is less than the timestamp of the least live transaction. *OPT-MVOSTM-GC* gave a performance gain of 16% over *OPT-MVOSTM* without garbage collection in the best case which is on workload W1 with 64 number of threads. We did one more optimization in *OPT-MVOSTM-GC* on the marked node exist in the **RL** to make it search efficiently. **This is achieved by deleting a marked node from **RL** (i.e. apply the garbage collection on **RL**) whose max_{rvl} of the last version is less than the timestamp of the least live transaction.**

The second, *OPT-KOSTM*, keeps at most K -versions by replacing the oldest version when $(K + 1)^{th}$ version is created by a current transaction as explained in Section 5.3. *OPT-KOSTM* shows a performance gain of 24% over *OPT-MVOSTM* without garbage collection in the best case which is on workload W1 with 64 number of threads. As *OPT-KOSTM* has a limited number of versions while *OPT-MVOSTM-GC* can have infinite versions, the memory consumed by *OPT-KOSTM* is also less than *OPT-MVOSTM-GC*. We have integrated these variants in both hash table based (*OPT-HT-MVOSTM-GC*, *OPT-HT-KOSTM*) and linked-list based MVOSTMs (*OPT-list-MVOSTM-GC*, *OPT-list-KOSTM*), we observed that these two variants increase the performance, concurrency and reduce the number of aborts as compared to *OPT-MVOSTM* which does not perform garbage collection.

Memory Consumption by OPT-MVOSTM-GC and OPT-KOSTM: As depicted above *OPT-KOSTM* performs better than *OPT-MVOSTM-GC*. Continuing the comparison between the two variants of *OPT-MVOSTM* we chose another parameter as memory consumption. Here, we test for the memory consumed by each algorithm in creating a version of a key. We count the total versions created, where creating a version increases the counter value by 1 and deleting a version decreases the counter value by 1. Figure 5.12 depicts the comparison of memory consumption by all the variants of proposed optimized *MVOSTM* with all the variants of *MVOSTM* (proposed in Chapter 4) for hash table objects. *OPT-HT-KOSTM* consumes minimum memory among all the proposed algorithms (*OPT-HT-MVOSTM-GC*, *OPT-HT-MVOSTM*, *HT-KOSTM*, *HT-MVOSTM-GC*, *HT-MVOSTM*) by a factor of 1.07, 1.16, 1.15, 1.15, 1.21 for W1, by a factor of 1.01, 1.08, 1.06, 1.07, 1.19 for W2, and by a factor of 1.01, 1.03, 1.02, 1.03, 1.08 for W3 respectively.

Similarly, Figure 5.13 depicts the comparison of memory consumption by all the

variants of proposed optimized *MVOSTM* with all the variants of *MVOSTM* (proposed in Chapter 4) for list objects. *OPT-list-KOSTM* consumes minimum memory among all the proposed algorithms (*OPT-list-MVOSTM-GC*, *OPT-list-MVOSTM*, *list-KOSTM*, *list-MVOSTM-GC*, *list-MVOSTM*) by a factor of 1.01, 1.05, 1.05, 1.04, 1.11 for W1, by a factor of 1.02, 1.1, 1.1, 1.11 1.19 for W2, and by a factor of 1.01, 1.03, 1.05, 1.08, 1.13 for W3 respectively.

Finite K-versions *OPT-MVOSTM (OPT-KOSTM)*: To find the ideal value of K such that performance as compared to *OPT-MVOSTM-GC* does not degrade or can be increased, we perform experiments on all the workloads W1, W2, and W3 for both *OPT-HT-KOSTM* and *OPT-list-KOSTM*. The best value of K is application dependent. Here, we used counter application (explained in SubSection 3.7.1) and get the best value of K is 5 for *OPT-HT-KOSTM* and *OPT-list-KOSTM* on all the workloads for both hash table and list objects demonstrated in Figure 5.14 (a) and (b).

5.6 Summary

This chapter of the thesis proposed the notion of the *Optimized Multi-Version Object-based STMs (OPT-MVOSTMs)* and compares their effectiveness with single and multi-version object-based STMs, single and multi-version read-write STMs. We find that *OPT-MVOSTM* provides a significant benefit over above-mentioned state-of-the-art STMs for different types of workloads. Specifically, we have evaluated the effectiveness of *OPT-MVOSTM* for two Concurrent Data Structures (CDS), hash table and list as *OPT-HT-MVOSTM* and *OPT-list-MVOSTM* respectively but its generic to other data structures as well..

OPT-HT-MVOSTM and *OPT-list-MVOSTM* use the unbounded number of versions for each key. To utilize the memory efficiently, we limit the number of versions and develop two variants for both hash table and list data structures: (1) A Garbage Collection (GC) method in *OPT-MVOSTM* to delete the unwanted versions of a key, denoted as *OPT-MVOSTM-GC*. (2) Placing a limit of K on the number of versions in *OPT-MVOSTM*, resulting in *OPT-KOSTM*. Both these variants (*OPT-MVOSTM-GC* and *OPT-KOSTM*) gave a performance gain of over 16% and 24% over *OPT-MVOSTM* in the best case. *OPT-KOSTM* consumes minimum memory among all the variants of it. We represent *OPT-MVOSTM-GC* in hash table and list as *OPT-HT-MVOSTM-GC* and *OPT-list-MVOSTM-GC* respectively. Similarly, We represent *OPT-KOSTM* in hash table and list as *OPT-HT-KOSTM* and *OPT-list-KOSTM* respectively.

Experimental analysis shows that *OPT-HT-KOSTM* performs best among its variants (*OPT-HT-MVOSTM* and *OPT-HT-MVOSTM-GC*) and outperforms all the state-of-the-art hash table based STMs (*HT-KOSTM* [17] proposed by us in Chapter 4, *HT-OSTM* [16] proposed by us in Chapter 3, *ESTM* [7], *RWSTM* [3], *HT-MVTO* [15], *HT-KSTM* [15]) by a factor of 1.05, 3.62, 3.95, 3.44, 2.75, 1.85 for workload W1, by a factor of 1.07, 1.44, 3.36, 5.45, 10.84, 8.42 for workload W2, and by a factor of 1.07, 2.11, 5.1, 19.8, 70.3, 60.23 for workload W3 respectively.

Similarly, *OPT-list-KOSTM* performs best among its variants (*OPT-list-MVOSTM* and *OPT-list-MVOSTM-GC*) and outperforms state-of-the-art list based STMs (*list-KOSTM* [17] proposed by us in Chapter 4, *list-OSTM* [16] proposed by us in Chapter 3, *Trans-list* [12], *Boosting-list* [13], *NOrec-list* [6], *list-MVTO* [15], *list-KSTM* [15]) by a factor of 1.2, 2.56, 25.38, 23.57, 27.44, 13.1, 6.8 for W1, by a factor of 1.12, 2.11, 21.54, 26.27, 30.1, 27.89, 20.1 for W2, and by a factor of 1.11, 2.91, 36.1, 42.2, 48.89, 149.92, 114.89 for W3 respectively.

Chapter 6

Conclusion and Future Work

This chapter describes the contributions of the thesis followed by directions for future research. Multi-core systems have become very common nowadays. Concurrent programming using multiple threads has become necessary to utilize all the cores present in the system effectively. But concurrent programming is usually challenging due to synchronization issues between the threads. To overcome these issues with concurrent programming, researchers have developed the paradigm of *Software Transactional Memory systems (STMs)* which helps programmers to develop the correct concurrent programs without compromising on the efficiency of the multi-core systems.

In this thesis, we designed and implemented a novel and efficient *Object-based Software Transactional Memory systems (OSTMs)* for multi-core systems. The proposed OSTMs provide greater concurrency and performance over Read-Write STMs (RW-STMs) while removing the synchronization burden from the developers. In this thesis, we proposed three efficient variants of OSTMs: Single-Version OSTMs (SVOSTM or OSTM), Multi-Version OSTMs (MVOSTMs), and OPT-MVOSTMs which are proved to be correct.

SVOSTMs achieve greater concurrency over RWSTMs using object semantics while maintaining single versions. MVOSTMs achieve the greater concurrency over SVOSTMs by using multiple versions which is a novel idea and has not been explored so far (even in databases). We further made a few optimizations to MVOSTMs, to achieve OPT-MVOSTMs, which enabled us to obtain even greater concurrency. We then discuss future research directions in Section 6.2.

6.1 Conclusion of the Thesis

This section explains the main research contributions of the thesis. It mainly proposed three Object-based STM systems (OSTMs, MVOSTMs, OPT-MVOSTMs) and proved their correctness followed by the performance evaluation with state-of-the-art STMs as follows:

- *Object-based STMs (OSTMs)*
- *Multi-Version Object-based STMs (MVOSTMs)*
- *Optimized Multi-Version OSTMs (OPT-MVOSTMs)*

The detailed description is as follows:

- **Object-based STMs (OSTMs):** In the past few years, several STMs have been proposed which address the synchronization issues and provide greater concurrency while ensuring the correctness. Another advantage of STMs is that they facilitate the compositionality of concurrent programs with great ease. Different concurrent operations that need to be composed to form a single atomic unit is achieved by encapsulating them in a single transaction.

In literature, most of the STMs are *RWSTMs* (such as NOrec [6], ESTM [7]) which export read and write operations. In this thesis, we build a model for building highly concurrent and composable data structures with object-level transactions called OSTMs. We showed that higher concurrency can be obtained by considering OSTMs as compared to traditional *RWSTM* by leveraging richer object-level semantics. We proposed a comprehensive theoretical model based on legality semantics and conflict notions for two *Concurrent Data Structures (CDS)*, hash table and list based OSTMs as *HT-OSTM* and *list-OSTM* but it is generic for other data structures as well. Using these notions we extended the definition of opacity [10] and co-opacity [18] for *OSTMs*. Then, based on this model, we developed a practical implementation of *HT-OSTM* and *list-OSTM* to verify the gains achieved. Further, we proved that the proposed model is co-opaque thus composable.

- **Multi-Version Object-based STMs (MVOSTMs):** It has been shown in the literature of databases and STMs [14, 15] that storing multiple versions for each *transactional object (or t-object)* or *key*, greater concurrency can be obtained. So, to improve the performance further we combine the multiple version with OSTM and proposed a novel and efficient *Multi-Version OSTMs*

(*MVOSTMs*). We find that multi-version object-based STM provides a significant benefit over single-version object-based STMs, single and multi-version read-write STMs for different types of workloads. Specifically, we have evaluated the effectiveness of MVOSTM for two CDS, hash table and list object as HT-MVOSTM and list-MVOSTM but it is generic for other data structures as well. Initially, HT-MVOSTM and list-MVOSTM use the unbounded number of versions for each key. To limit the number of versions, we developed two variants for both hash table and list data structures: (1) A *Garbage Collection (GC)* method in MVOSTM used to delete the unwanted versions of a key, denoted as MVOSTM-GC. (2) Placing a limit of K on the number versions in MVOSTM, resulting in KOSTM. The experimental analysis showed that KOSTM performs best among its variants (MVOSTM, MVOSTM-GC) and state-of-the-art STMs. We proved that *MVOSTMs* satisfy opacity [10] and ensure that the transaction with lookup only methods does not abort if unbounded versions are used. To the best of our knowledge, this is the first work to explore the idea of using multiple versions in OSTMs to achieve greater concurrency.

- **Optimized Multi-Version OSTMs (OPT-MVOSTMs):** To harness the greater concurrency further, we performed a few optimizations on the MVOSTM and proposed a new notion of *Optimized Multi-Version Object-based STM system* as *OPT-MVOSTM*. Here, we developed it for two CDS, hash table and list objects as *OPT-HT-MVOSTM* and *OPT-list-MVOSTM* respectively. But *OPT-MVOSTM* is generic for other data structures as well. Initially, OPT-HT-MVOSTM and OPT-list-MVOSTM use an unbounded number of versions for each key. To limit the number of versions, we developed two variants for both hash table and list data structures: (1) A *Garbage Collection (GC)* method in OPT-MVOSTM used to delete the unwanted versions of a key, denoted as OPT-MVOSTM-GC. (2) Placing a limit of K on the number versions in OPT-MVOSTM, resulting in OPT-KOSTM. The experimental analysis showed that OPT-KOSTM performs best among its variants (OPT-MVOSTM, OPT-MVOSTM-GC) and state-of-the-art STMs. We proved that *OPT-KOSTM* satisfies standard correctness-criterion of STMs, *opacity* [10].

6.2 Directions for Future Research

This thesis opens several directions for future research as follows:

- Nesting for Object-based STM systems

- Object-based STM systems as an Application to Blockchain
- Distributed Object-based STMs (DOSTMs)

The detailed description is as follows:

6.2.1 Nesting for Object-based STM systems

OSTMs facilitate one interesting property compositionality of concurrent programs with great ease. Different concurrent operations that need to be composed to form a single atomic unit is achieved by encapsulating them in a single transaction.

But, the composition of different simple transactions into a large single transaction is a very useful property in modular programming or many real-time applications [29–31]. Nesting [32–34] is one of the ways to achieve the compositionality of different transactions in STMs. So, nesting of a transaction T_i implies that T_i invokes another transaction T_j inside it. There are two types of nesting available: (1) *Closed Nesting* [35]: any transaction T_i invokes another transaction T_j then T_i is called as parent transaction whereas T_j is known as child transaction. So, in the closed nesting, whenever any child transaction T_j commits, the updates made by T_j is gone be visible to only its parent T_i and siblings but not to the other transactions available in the system. (2) *Open Nesting* [36–38]: whenever any child transaction T_j commits, the updates made by T_j is gone be visible immediately to all the transactions of the system. Even, it is not waiting till commit of its parent T_i but if the parent transaction T_i returns abort with some reason then the child transaction T_j also needs to return abort which is a bit difficult because T_j has already been committed and showed its effect to other transactions. A few researchers have explored nesting at read-write level [32–34] in which a transaction with read-write operation invokes the another transactions with read-write operations only. But, none of them explored nesting at object-level.

Exploring nesting of transactions for OSTMs in which one object-based transaction invokes another transaction - either read/write or object-based is one of the interesting direction of future research. Initially, closed nesting of the object-based transaction can be explored to enhance the scalability of the application while ensuring the correctness as closed nested opacity [32]. After designing an efficient and correct closed nesting for objects, measuring the cost of their implementation will be useful in which the algorithms developed will be implemented on various benchmarks and the performance compared against the various state-of-the-art STMs. It can also extend to the closed nesting of transaction in which one object-based transaction invokes other object-based transaction.

Having seen the benefits of multi-version OSTMs [17], closed nesting can be explored for MVOSTMs which will reduce the number of aborts and improves the concurrency further.

6.2.2 Object-based STM systems as an Application to Blockchain

Several popular blockchains such as Ethereum [29] executes *complex transactions* in blocks through user-defined scripts known as *smart contracts* [39]. Normally, a block of the chain consists of multiple *Smart Contract Transactions (SCTs)* which are added by a *miner*. To append a correct block into the blockchain, miner executes these SCTs **sequentially** and store the final state in the block. The remaining peers on receiving this block act as *validators*. The validators again sequentially re-execute the SCTs of the block as a part of the consensus protocol. If the validators agree with the final state of the block as recorded by the miner, then the block is said to be validated and is added to the blockchain. In Ethereum and other blockchains such as Bitcoin [30], EOS [31] that support cryptocurrencies, a miner gets an incentive every time such a valid block is successfully added to the blockchain.

In the current era of multi-core processors, the miners and validators fail to harness the power of multiple cores by serially executing the SCTs which can result in poor throughput. By adding concurrency to the execution of SCTs, we can achieve better efficiency and higher throughput.

However, there are challenges with concurrent execution. Figure 6.1 illustrates the difficulty. Let us consider two accounts A, B having \$10 as the *Initial State (or IS)*. Suppose there are two smart contract transactions, T_1, T_2 where T_1 is transferring \$10 from account A to B while T_2 is transferring \$20 from B to A . Since both the smart contract transactions are accessing common accounts (A and B) to transfer the amount, the order of SCTs execution becomes important. Suppose the miner executes them concurrently with the equivalent effect being T_1 followed by T_2 as shown in Figure 6.1(b). In this case, the *Final State (or FS)* of A will have \$20 while B will have \$0. On the other hand, suppose the validators execute in a different concurrent order which is equivalent to T_2 followed by T_1 as shown in Figure 6.1(c). T_2 executes first; but due to insufficient balance in B 's account, a validator, say v rejects this SCT. Then after executing T_1 , v transfers \$10 from A to B . With this execution, the final state of A will have \$0 while B will have \$20. Thus on receiving such a block, a validator will see that the final state in the block given by the miner is different from what it obtained and hence, falsely reject the block. We refer this problem as *False Block Rejection (or FBR)* error. This can negate the benefits of concurrent executions.

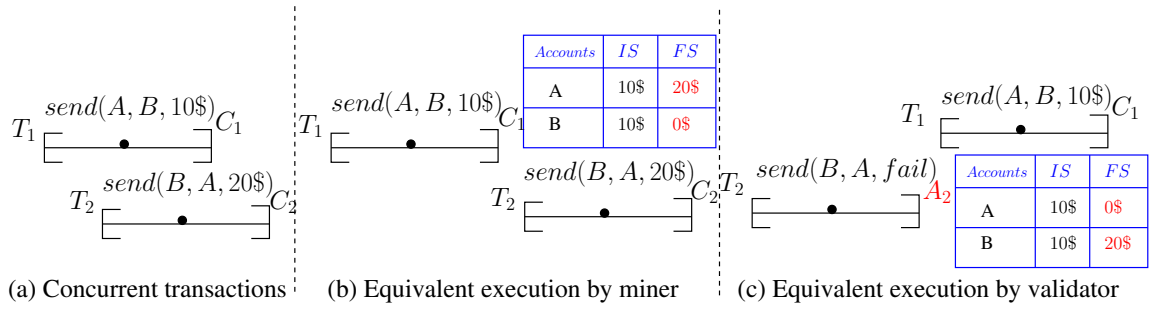


Figure 6.1: Challenges in Concurrent execution of SCTs

Recently, researchers have used *Read-Write Software Transactional Memory systems (RWSTMs)* [6, 40–42] for the concurrent execution of SCTs. But Chapter 3 of this thesis showed that proposed *Object-based STMs (OSTMs)* achieves greater concurrency, better throughput as compared to RWSTMs. Further, Chapter 4 of the thesis observed that greater concurrency can be obtained using *Multi-Version OSTMs (MVOSTMs)* by maintaining multiple versions for each shared data-item as opposed to traditional OSTMs and RWSTMs. So, another interesting direction for future research can be developing an efficient framework to execute the SCTs concurrently based on object semantics by miner using proposed OSTMs and MVOSTMs.

6.2.3 Distributed Object-based STMs (DOSTMs)

All the OSTMs proposed in this thesis work on the shared memory. It allows multiple threads to access the shared memory without worrying about concurrency issues such as priority-inversion, deadlock, livelock, etc.

Nowadays, distributed systems are becoming very appealing to design a secure network while removing the responsibilities from the central authority. Few researchers have explored distributed STMs at read-write level [43, 44]. But, none of them explored it at object-level. As we have seen the benefit of object-level over the read-write level in Chapter 3, we can exploit the distributed systems by developing the distributed concurrent algorithms using object-based STM systems. Hence, *Distributed Object-based Software Transactional Memory systems (DOSTMs)* will be a useful research direction of this thesis which will help to improve the performance of distributed networks such as blockchain.

6.3 Summary

This chapter described the contributions of this thesis followed by direction for future research. We proposed mainly three Object-based STM systems (OSTMs, MVOSTMs, OPT-MVOSTMs) and proved their correctness followed by the performance evalua-

tion with state-of-the-art STMs. The direction for future research of this thesis can be (1) Nesting for Object-based STM systems, (2) Object-based STM systems as an Application to Blockchain, and (3) Distributed Object-based STMs (DOSTMs).

Bibliography

- [1] S. Peyton Jones, “Beautiful concurrency,” in *Beautiful code*. O’Reilly, January 2007. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/beautiful-concurrency/>
- [2] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.
- [3] G. Weikum and G. Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.
- [4] M. Herlihy and J. E. B. Moss, “Transactional memory: Architectural Support for Lock-Free Data Structures,” *SIGARCH Comput. Archit. News*, vol. 21, no. 2, pp. 289–300, 1993.
- [5] C. H. Papadimitriou, “The serializability of concurrent database updates,” *J. ACM*, vol. 26, no. 4, pp. 631–653, Oct. 1979. [Online]. Available: <http://doi.acm.org/10.1145/322154.322158>
- [6] L. Dalessandro, M. F. Spear, and M. L. Scott, “Norec: Streamlining stm by abolishing ownership records,” in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’10. New York, NY, USA: ACM, 2010, pp. 67–78. [Online]. Available: <http://doi.acm.org/10.1145/1693453.1693464>
- [7] P. Felber, V. Gramoli, and R. Guerraoui, “Elastic Transactions,” *J. Parallel Distrib. Comput.*, vol. 100, no. C, pp. 103–127, Feb. 2017. [Online]. Available: <https://doi.org/10.1016/j.jpdc.2016.10.010>
- [8] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Elsevier Science, 2012.

- [9] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. S. III, and N. Shavit, “A Lazy Concurrent List-Based Set Algorithm,” *Parallel Processing Letters*, vol. 17, no. 4, pp. 411–424, 2007.
- [10] R. Guerraoui and M. Kapalka, “On the correctness of transactional memory,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’08. New York, NY, USA: ACM, 2008, pp. 175–184. [Online]. Available: <http://doi.acm.org/10.1145/1345206.1345233>
- [11] T. Harris *et al.*, “Abstract nested transactions,” *TRANSACT*, 2007.
- [12] D. Zhang and D. Dechev, “Lock-free transactions without rollbacks for linked data structures,” in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA ’16. New York, NY, USA: ACM, 2016, pp. 325–336. [Online]. Available: <http://doi.acm.org/10.1145/2935764.2935780>
- [13] M. Herlihy and E. Koskinen, “Transactional boosting: a methodology for highly-concurrent transactional objects,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, 2008, pp. 207–216. [Online]. Available: <http://doi.acm.org/10.1145/1345206.1345237>
- [14] D. Perelman, R. Fan, and I. Keidar, “On maintaining multiple versions in stm,” in *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, ser. PODC ’10. New York, NY, USA: ACM, 2010, pp. 16–25. [Online]. Available: <http://doi.acm.org/10.1145/1835698.1835704>
- [15] P. Kumar, S. Peri, and K. Vidyasankar, “A timestamp based multi-version stm algorithm,” in *Proceedings of the 15th International Conference on Distributed Computing and Networking - Volume 8314*, ser. ICDCN 2014. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 212–226. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-45249-9_14
- [16] S. Peri, A. Singh, and A. Somani, “Efficient means of achieving composability using object based semantics in transactional memory systems,” in *Networked Systems - 6th International Conference, NETYS 2018, Essaouira, Morocco, May 9-11, 2018, Revised Selected Papers*, 2018, pp. 157–174. [Online]. Available: https://doi.org/10.1007/978-3-030-05529-5_11
- [17] C. Juyal, S. S. Kulkarni, S. Kumari, S. Peri, and A. Somani, “An innovative approach to achieve compositionality efficiently using multi-version

- object based transactional systems,” in *Stabilization, Safety, and Security of Distributed Systems - 20th International Symposium, SSS 2018, Tokyo, Japan, November 4-7, 2018, Proceedings*, 2018, pp. 284–300. [Online]. Available: https://doi.org/10.1007/978-3-030-03232-6_19
- [18] P. Kuznetsov and S. Peri, “Non-interference and local correctness in transactional memory,” *Theor. Comput. Sci.*, vol. 688, pp. 103–116, 2017.
- [19] P. Kuznetsov and S. Ravi, “On the cost of concurrency in transactional memory,” in *Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*, 2011, pp. 112–127. [Online]. Available: https://doi.org/10.1007/978-3-642-25873-2_9
- [20] M. P. Herlihy and J. M. Wing, “Linearizability: a correctness condition for concurrent objects,” *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.
- [21] N. Shavit and D. Touitou, “Software transactional memory,” in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC ’95. New York, NY, USA: ACM, 1995, pp. 204–213. [Online]. Available: <http://doi.acm.org/10.1145/224964.224987>
- [22] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy, “Composable memory transactions,” in *PPoPP’05: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Chicago, Illinois, June 2005. [Online]. Available: <http://simonmar.github.io/bib/papers/stm.pdf>
- [23] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman, “Open nesting in software transactional memory,” in *PPoPP*. ACM, 2007.
- [24] A. Hassan, R. Palmieri, and B. Ravindran, “Optimistic transactional boosting,” in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’14. New York, NY, USA: ACM, 2014, pp. 387–388. [Online]. Available: <http://doi.acm.org/10.1145/2555243.2555283>
- [25] K. Fraser and T. Harris, “Concurrent programming without locks,” *ACM Trans. Comput. Syst.*, vol. 25, no. 2, May 2007. [Online]. Available: <http://doi.acm.org/10.1145/1233307.1233309>

- [26] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro, “Proving Correctness of Highly-concurrent Linearisable Objects,” in *PPoPP*, 2006, pp. 129–136.
- [27] R. Guerraoui and M. Kapalka, *Principles of Transactional Memory*, ser. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010. [Online]. Available: <https://doi.org/10.2200/S00253ED1V01Y201009DCT004>
- [28] T. L. Harris, “A pragmatic implementation of non-blocking linked-lists,” in *Proceedings of the 15th International Conference on Distributed Computing*, ser. DISC ’01. London, UK, UK: Springer-Verlag, 2001, pp. 300–314. [Online]. Available: <http://dl.acm.org/citation.cfm?id=645958.676105>
- [29] “Ethereum,” <http://github.com/ethereum>, [Accessed 26-3-2019].
- [30] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 2009.
- [31] “EOS,” <https://eos.io/>, [Accessed 26-3-2019].
- [32] S. Peri and K. Vidyasankar, “Correctness of concurrent executions of closed nested transactions in transactional memory systems,” in *Distributed Computing and Networking*, M. K. Aguilera, H. Yu, N. H. Vaidya, V. Srinivasan, and R. R. Choudhury, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 95–106.
- [33] K. Agrawal, J. T. Fineman, and J. Sukha, “Nested parallelism in transactional memory,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’08. New York, NY, USA: ACM, 2008, pp. 163–174. [Online]. Available: <http://doi.acm.org/10.1145/1345206.1345232>
- [34] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood, “Supporting nested transactional memory in logtm,” *SIGPLAN Not.*, vol. 41, no. 11, pp. 359–370, Oct. 2006. [Online]. Available: <http://doi.acm.org/10.1145/1168918.1168902>
- [35] J. E. B. Moss and A. L. Hosking, “Nested transactional memory: Model and architecture sketches,” *Sci. Comput. Program.*, vol. 63, pp. 186–201, 2006.
- [36] M. Saheb, R. Karoui, and S. Sédillot, “Open nested transaction: A support for increasing performance and multi-tier applications,” in *Selected Papers from the Eight International Workshop on Foundations of Models and*

- Languages for Data and Objects, Transactions and Database Dynamics*. London, UK, UK: Springer-Verlag, 2000, pp. 167–192. [Online]. Available: <http://dl.acm.org/citation.cfm?id=646202.681990>
- [37] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman, “Open nesting in software transactional memory,” in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’07. New York, NY, USA: ACM, 2007, pp. 68–78. [Online]. Available: <http://doi.acm.org/10.1145/1229428.1229442>
- [38] K. Agrawal, C. E. Leiserson, and J. Sukha, “Memory models for open-nested transactions,” in *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, ser. MSPC ’06. New York, NY, USA: ACM, 2006, pp. 70–81. [Online]. Available: <http://doi.acm.org/10.1145/1178597.1178610>
- [39] N. Szabo, “Formalizing and securing relationships on public networks,” *First Monday*, vol. 2, no. 9, 1997.
- [40] P. S. Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani, “An efficient framework for optimistic concurrent execution of smart contracts,” in *PDP*, 2019, pp. 83–92.
- [41] A. Zhang and K. Zhang, “Enabling concurrency on smart contracts using multiversion ordering,” in *Web and Big Data*, Y. Cai, Y. Ishikawa, and J. Xu, Eds., Cham, 2018, pp. 425–439.
- [42] I. Sergey and A. Hobor, “A concurrent perspective on smart contracts,” in *Financial Cryptography Workshops*, 2017.
- [43] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson, “Distm: A software transactional memory framework for clusters,” in *2008 37th International Conference on Parallel Processing*, Sep. 2008, pp. 51–58.
- [44] M. M. Saad and B. Ravindran, “Snake: Control flow distributed software transactional memory,” in *Stabilization, Safety, and Security of Distributed Systems*, X. Défago, F. Petit, and V. Villain, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 238–252.

List of Publications ¹

Publications part of the thesis:

- Journal Paper:
 1. “An Efficient Approach to Achieve Compositionality using Optimized Multi-Version Object Based Transactional Systems”, Chirag Juyal, Sandeep Kulkarni, Sweta Kumari, Sathya Peri, and Archit Somani in Information and Computation Journal by Elsevier. [Under Review: Extension of SSS’18 Conference Paper]
- Conference Papers:
 1. “Efficient means of Achieving Composability using Object based Semantics in Transactional Memory”, Sathya Peri, Ajay Singh, and Archit Somani in 6th International Conference On Networked Systems(NETYS 2018), Essaouira, Morocco.
 2. “An Innovative Approach to Achieve Compositionality Efficiently using Multi-version Object Based Transactional Systems”, Chirag Juyal, Sandeep Kulkarni, Sweta Kumari, Sathya Peri, and Archit Somani in 20th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2018), Tokyo, Japan (**Best Student Paper Award**).
- Accepted as Work-in-Progress Papers:
 1. “An Innovative Approach to Achieve Compositionality Efficiently using Multi-Version Object Based Transactional Systems”, Chirag Juyal, Sandeep Kulkarni, Sweta Kumari, Sathya Peri, and Archit Somani in 2nd International Workshop on Algorithms & Architectures for Distributed Data Analytics (AADDA) held in conjunction with ICDCN 2018 at IIT BHU, Varanasi, India.
 2. “Achieving greater Concurrency in Transaction Memory Systems using Object Semantics”, Sathya Peri, Ajay Singh, Archit Somani in 1st International Workshop on Algorithms & Architectures for Distributed Data Analytics (AADDA) held in conjunction with ICDCN 2017 at Hyderabad, India.
- Posters:

¹Author sequence follows lexical order of last names.

1. “An Innovative Approach to Achieve Compositionality Efficiently using Multi- Version Object Based Transactional Systems”, Chirag Juyal, Sandeep Kulkarni, Sweta Kumari, Sathya Peri, and Archit Somani in 6th International Conference On Networked Systems, 2018, Essaouira, Morocco (**Best Poster Award**).
2. “Achieving greater Concurrency using Object based Software Transaction Memory System”, Sathya Peri, Ajay Singh, and Archit Somani in 30th IEEE International Parallel and Distributed Processing Symposium, 2016, Chicago, Illinois, USA. (**Microsoft Research Travel Grant**)

Publications not part of the thesis:

- Conference Papers:
 1. “An Efficient Framework for Optimistic Concurrent Execution of Smart Contracts”, Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani in 27th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP 2019) at Pavai, Italy.
 2. “Achieving Progress Guarantee and Greater Concurrency in Multi-Version Object Semantics”, Chirag Juyal, Sandeep Kulkarni, Sweta Kumari, Sathya Peri, and Archit Somani in 21st International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2019), Pisa, Italy.
 3. “Efficient Concurrent Execution of Smart Contracts in Blockchains using Object-based Transactional Memory”, Parwat Singh Anjana, Sweta Kumari, Sathya Peri, and Archit Somani. [Submitted in OPODIS 2019]
- Accepted as Work-in-Progress Papers:
 1. “Proving Correctness of Concurrent Objects by Validating Linearization Points”, Sathya Peri, Mukhtikanta Sa, Ajay Singh, Nandini Singhal, Archit Somani in 2nd International Workshop on Algorithms & Architectures for Distributed Data Analytics (AADDA) held in conjunction with ICDCN 2018 at IIT BHU, Varanasi, India.
- Posters:
 1. “Entitling Concurrency to Smart Contracts Using Optimistic Transactional Memory”, Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani in Doctoral Symposium, ICDCN 2019 at Indian Institute of Science, Bangalore, India (**Best Poster Award**).