Exploring Progress Guarantees in Multi-Version Software Transactional Memory Systems

Sweta Kumari

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.

Sweta kumari 13/2/2020

(Signature)

Sweta Kumari

(Name)

CS15RESCH01004

(Roll No.)

Approval Sheet

This Thesis entitled **Exploring Progress Guarantees in Multi-Version Software Transactional Memory Systems** by **Sweta Kumari** is approved for the degree of Doctor of Philosophy from IIT Hyderabad.

[illegible]

(Prof. Gautam Barua) Examiner 1
Dept. of CSE, IIIT Guwahati

[illegible]

(Prof. Kamakoti Veezhinathan) Examiner 2
Dept. of CSE, IIT Madras

[illegible]

(Dr. Sobhan Babu) Internal Examiner
Dept. of CSE, IITH

[illegible]

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

[illegible]

(Prof. Deepak John Matthew) Chairman
Dept. of Design, IITH

Acknowledgements

First and foremost, I would like to thanks lord Krishna for health, wisdom, strength to undertake this research work and enabled me to complete it. My gratitude to all the praises and being always there with me in every moment of life. The success and outcome of this doctoral dissertation is a lot of effort, guidance, and assistance from several people. I am extremely privileged to have those people around me and express my sincere gratitude to all of them. I would like to express my deepest gratitude to my research supervisor, Dr. Sathya Peri, for his guidance, persistent support, many insightful conversations, and encouraging my research throughout my Ph.D. His suggestions on the research and my career have been priceless and helped me a lot to improve myself.

My sincere gratitude also extends to Dr. M.V. Panduranga Rao, Dr. Manish Singh and Dr. Sumohana S. Channappayya for serving as my doctoral committee members and providing insightful comments and discussions to enhance the research. I would also like to express my gratefulness to Prof. Sandeep Kulkarni from Michigan State University, USA for the continuous research discussion whenever I required it. A special thanks to Ministry of Electronics and Information Technology, Government of India initiated "Visvesvaraya Ph.D. Scheme for Electronics and IT" for financial assistance during my Ph.D.

I appreciate all the faculties from the department of CSE and the Indian Institute of Technology, Hyderabad for their impactful lectures and precious time along with a good research infrastructure provided to me. I am thankful to the office staff for entertaining my administrative work. I would like to take this opportunity to thank all my friends and labmates especially, Archit Somani, Parwat Singh Anjana, Ved Prakash Chaudhary, Chirag Juyal, Sachin Rathor for the research discussions and memorable time. Along with them, I have been very fortunate to make some wonderful friendships with Anakana Sen, Aparajita Das, Kalpana, and Sakshi for sharing a beautiful time (the birthday celebrations, outings, fun, DJ, and adventures) at IITH.

A special thanks to my dear mother Sri Mati Nirmala Devi and loving father Sri Suresh Prasad whose sacrifices, blessings, and contentious moral support are the main reason for I being here. I am indebted to my elder brother Krishna Kant who blessed me with the solutions to all the problems of my life and made me happy. I am also grateful to everyone in my family, especially my sisters, Sharda, Shalini and sister-in-law Shyama Narayan for their continuous love and support. I want to convey the best regards to my loving husband, Archit Somani for his continuous support, understanding, caring in technical and non-technical aspects. I would also like to extend my heartfelt gratitude to my responsible in-laws family for their encouragement and motivation.

Eventually, I want to thank everyone who helped me directly or indirectly during my ph.D. Thank You, Lord Krishna!

Sweta Kumari

Dedication

This dissertation is wholeheartedly dedicated to almighty god, my loving parents, enthusiastic brother, encouraged sisters, supportive husband, caring in-laws family, and friends for their endless sacrifice, love, moral, and spiritual support throughout this endeavor.

Abstract

In the current era of multi-core processors, *Software Transactional Memory systems (STMs)* have garnered significant interest as an elegant alternative for addressing synchronization and concurrency issues with multi-threaded programming to utilize the cores properly. Client programs use STMs by issuing transactions. A transaction of STMs is a piece of code that performs reads and writes to the shared memory. Typical STMs work on read/write methods which maintain single-version corresponding to each *transactional-object or t-object* called as *Single-Version Read-Write STMs (SV-RWSTMs or RWSTMs).*

It has been shown in the literature that maintaining multiple versions corresponding to each *t-object* reduces the number of aborts and enhances performance. Several *Multi-Version RW-STMs (or MV-RWSTMs)* have been proposed in the literature that maintain multiple versions and provide increased concurrency along with better performance than *SV-RWSTMs*.

Some STMs work at higher-level operations and ensure greater concurrency than MV-RWSTMs and SV-RWSTMs. They include more semantically rich operations such as push/pop on stack objects, enqueue/dequeue on queue objects and insert/lookup/delete on sets, trees or hash table objects depending upon the underlying data structure used to implement higher-level systems. Such STMs are known as *Single-Version Object-based STMs (SV-OSTMs or OSTMs).*

To achieve the greater concurrency further, researchers have proposed *Multi-Version OSTM (or MV-OSTM)* which maintains multiple versions corresponding to each t-object in OSTMs. MV-OSTM system reduces the number of aborts and improves performance than SV-OSTMs, MV-RWSTMs, and SV-RWSTMs.

All the STMs defined above ensure that transaction either $commits$ or $aborts$. A transaction aborted due to conflicts (two transactions are said to be in conflict if both of them are accessing same *t-object* x and at least one of the transaction performs write/update on x) is typically re-issued with the expectation that it will complete successfully in a subsequent incarnation. However, many existing STMs fail to provide *starvation freedom*, i.e., in these systems, it is possible that concurrency conflicts may prevent an incarnated transaction from committing.

To overcome this limitation, we developed an efficient STM system which ensures *starvation-freedom* as a progress condition. An STM system is said to be *starvation-free* if a thread invoking a transaction T_i gets the opportunity to retry T_i on every abort (due to the presence of a fair underlying scheduler with bounded termination) and T_i is not *parasitic*, i.e., T_i will try to commit given a chance then T_i will eventually commit. *Wait-freedom* is another interesting progress condition for STMs in which every transaction commits regardless of the nature of concurrent transactions and the underlying scheduler. But it was shown in the literature that it is not possible to achieve *wait-freedom* in dynamic STMs in which t-objects of transactions are not known in advance. So in this thesis, we explore the weaker progress condition *starvation-freedom* for transactional memory systems (SV-RWSTMs, MV-RWSTMs, SV-OSTMs, and MV-OSTMs) while assuming that the t-objects of the transactions are *not* known in advance.

Some researchers have explored starvation-freedom in SV-RWSTMs while maintaining single-version corresponding to each t-object. We denote such an algorithm as *Starvation-Free Single-Version RWSTM (or SF-SV-RWSTM)*. Although SF-SV-RWSTM guarantees starvation-freedom, but it can still abort many transactions spuriously which brings down the efficiency and progress of the entire system.

To overcome this issue, we systematically developed a novel and efficient starvation free algorithm as *Starvation-free Multi-Version RWSTM (SF-MV-RWSTM)*. It maintains multiple versions corresponding to each *t-object* which reduces the number of aborts and enhances the performance than *SF-SV-RWSTMs*. Proposed SF-MV-RWSTM can be used either with the case where the number of versions is unbounded and *Garbage Collection (GC)* is used to delete unwanted versions as *SF-MV-RWSTM-GC* or where only the latest K-versions are maintained, as *Starvation-Free K-Version RWSTM (or SF-K-RWSTM)*. Our experimental analysis demonstrates that the proposed *SF-K-RWSTM* algorithm performs best among its variants (*SF-MV-RWSTM* and *SF-MV-RWSTM-GC*) along with state-of-the-art STMs under long-running transactions with high contention. *SF-K-RWSTM* satisfies the popular correctness-criteria local opacity and ensures the progress condition as starvation-freedom.

To achieve *starvation-freedom* along with higher concurrency, we proposed *Starvation-Freedom in SV-OSTM as SF-SV-OSTM* which assigns the priority to the transaction on abort. SF-SV-OSTM satisfies the correctness criteria *conflict-opacity* while ensuring the progress condition as starvation-freedom.

To achieve the greater concurrency further while ensuring the starvation-freedom, we maintained multiple versions corresponding to each t-object in starvation-free OSTMs and proposed a novel and efficient *Starvation-Freedom Multi-Version OSTM (or SF-MV-OSTM)*. The number of versions maintained by SF-MV-OSTM either be unbounded with Garbage Collection (GC) as *SF-MV-OSTM-GC* or bounded with the latest K-versions as *SF-K-OSTM*. SF-K-OSTM ensures starvation-freedom, satisfies the correctness criteria as local opacity and shows the performance benefits as compared with state-of-the-art STMs.

This thesis explores the progress guarantee *starvation-freedom* in single and multi-version RWSTMs, single and multi-version OSTMs while satisfying the correctness-criteria as *conflict-opacity* and *local opacity*. It shows that maintaining multiple versions improves the concurrency than single-version while reducing the number of aborts and increasing the throughput. This motivated us to use efficient multi-version STMs to improve the performance of smart contract executions in *blockchain* systems.

Blockchain platforms such as Ethereum and several others execute *complex transactions* in blocks through user-defined scripts known as *smart contracts*. Normally, a block of the chain consists of multiple transactions of smart contracts that are added by a *miner*. To append a correct block into the blockchain, miners execute these transactions of smart contracts sequentially. Later the *validators* serially re-execute the smart contract transactions of the block. If the validators agree with the final state of the block as recorded by the miner and reach the consensus, then the block is said to be validated and the respective miner gets an incentive on such a valid block successfully added to the blockchain.

Nowadays, multi-core processors are ubiquitous. By employing serial execution of the transactions, the miners and validators fail to utilize the cores properly and as a result, have poor throughput. Adding concurrency to smart contracts execution can result in better utilization of the cores and as a result higher throughput. In this thesis, we develop a framework to execute the smart contract transactions concurrently by miner using efficient *Multi-Version Software Transactional Memory systems (MVSTMs)*. The miner proposes a block which consists of a set of transactions, block graph, the hash of the previous block and final state of each shared t-object. The block graph captures the conflicting relations among the transactions. Later, the validators re-execute the same smart contract transactions concurrently and deterministically with the help of the block graph given by miner to verify the final state. If the validation is successful then the proposed block is appended into the blockchain as a part of the consensus protocol. In the case of the blockchain as a cryptocurrency like Ethereum, Bitcoin, the respective miner also gets a reward for producing the block. If validation is not successful, then validator discards the proposed block.

Concurrent execution of smart contract transactions by miner and validator achieve significant performance gain as compared to serial miner and validator. But concurrent execution of smart contracts poses some interesting challenges. So, in this thesis, we show how to overcome these challenges to improve the performance of smart contract execution by executing them concurrently using efficient MVSTM protocols.

Contents

List of Figures

List of Tables

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
SF-SV-RWSTMs	Starvation-Free Single-Version Read-Write 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
PMVTO	Priority-based Multi-Version Timestamp Ordering Protocol
PMVTO-GC	Garbage Collection in Priority-based Multi-Version Timestamp Ordering Protocol
PKTO	Priority-based K-Version Timestamp Ordering Protocol
SF-MV-TO	Starvation-Free Multi-Version Timestamp Ordering Protocol
SF-MV-RWSTMs	Starvation-Free Multi-Version Read-Write Software Transactional Memory systems
SF-MV-RWSTM-GC	Garbage Collection in Starvation-Free Multi-Version Read-Write Software Transactional Memory systems
SF-K-RWSTMs	Starvation-Free K-Version Read-Write Software Transactional Memory systems
SF-UV-RWSTMs	Starvation-Free Unbounded Version Read-Write Software Transactional Memory systems
SF-UV-RWSTM-GC	Garbage Collection in Starvation-Free Unbounded Version Read-Write Software Transactional Memory systems
OSTMs	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
SV-OSTMs	Single-Version Object-based Software Transactional Memory systems
SF-SV-OSTMs	Starvation-Free Single-Version Object-based Software Transactional Memory systems
HT-SF-SV-OSTMs	Hash Table based Starvation-Free Single-Version Object-based Software Transactional Memory systems
list-SF-SV-OSTMs	list based Starvation-Free Single-Version Object-based Software Transactional Memory systems
MV-OSTMs	Multi-Version Object-based Software Transactional Memory systems
HT-MV-OSTMs	Hash Table based Multi-Version Object-based Software Transactional Memory systems
list-MV-OSTMs	list based Multi-Version Object-based Software Transactional Memory systems
K-OSTMs	Finite K-Version Object-based Software Transactional Memory systems
HT-K-OSTMs	Hash Table based Finite K-Version Object-based Software Transactional Memory systems
list-K-OSTMs	list based Finite K-Version Object-based Software Transactional Memory systems
SF-MV-OSTMs	Starvation-Free Multi-Version Object-based Software Transactional Memory systems
HT-SF-MV-OSTMs	Hash Table based Starvation-Free Multi-Version Object-based Software Transactional Memory systems
list-SF-MV-OSTMs	list based Starvation-Free Multi-Version Object-based Software Transactional Memory systems
SF-K-OSTMs	Starvation-Free Finite K-Version Object-based Software Transactional Memory systems
HT-SF-K-OSTMs	Hash Table based Starvation-Free K-Version Object-based Software Transactional Memory systems
list-SF-K-OSTMs	list based Starvation-Free K-Version Object-based Software Transactional Memory systems

MV-OSTM-GC	Garbage Collection in Multi-Version Object-based Software Transactional Memory systems
HT-MV-OSTM-GC	Hash Table based Garbage Collection in Multi-Version Object-based Software Transactional Memory systems
list-MV-OSTM-GC	list based Garbage Collection in Multi-Version Object-based Software Transactional Memory systems
ESTM	Elastic Software Transactional Memory
NOrec STM	An Ownership-record-free Software Transactional Memory
HT	Hash Table
list	Linked-list
GC	Garbage Collection
RL	Red Link
BL	Blue Link
lazyrb-list	Lazy Red-Blue List
SCTs	Smart Contract Transactions

Chapter 1

Introduction

The performance of single processors has stagnated for more than a decade. So, industry emerges with the solution using multiple cores on a single chip system known as *multi-core or many-core systems*. Underlying multi-core systems are equipped with more than one cores/processors on a single chip and these cores are communicating to each other through shared memory. To exploit the underlying hardware of multi-core systems properly concurrent programming is needed. But developing a correct and efficient concurrent program using multi-threading while ensuring the correctness is difficult. Some of the challenges are as follows:

- Collaboration between threads which access the same data in main or secondary memory.
- Uncontrolled writes may lead to inconsistent data values.
- Synchronized memory access is required since processors cannot modify independent memory locations atomically.

Table 1.1: Concurrent execution of file transfer

Th1	**Time**	**Th2**
Find(F1, Dir1)	1	
	2	Find(F1, Dir1)
	3	Delete(F1, Dir1)
Modify (F1, Dir1)	4	
	5	Add(F1, Dir2)
Inconsistent Data		

Table 1.1 shows the difficulties of multi-threading while considering the example of file transfer from one directory to another directory. It illustrates the concurrent execution using two threads $Th1$ and $Th2$. Here, $Th1$ finds the file $F1$ in directory $Dir1$. After that thread

$Th2$ also finds the file $F1$, delete it from $Dir1$, and add it to $Dir2$. Then $Th1$ tried to modify $F1$ believing that it is still in $Dir1$. But $F1$ has already been deleted by $Th2$ form $Dir1$ and $Th1$ does not have this information. So, $Th1$ will see the inconsistent data of the $F1$. Hence, collaboration and synchronization between the threads are required during concurrent execution.

To address these issues of multi-threading/concurrent execution of the program, the programmer uses *locks, semaphores, monitors*, etc. Normally, thread acquires the lock on shared data-items, then access these data-items followed by releasing the respective locks.

Threads have two options while handling with multiple shared variables: (1) Coarse-grained locking: All the share variables are accessed through a single common lock (2) Fine-Grained locking: Each shared variable has an individual lock.

Coarse-grained locking is simple to use. But it is inefficient and can bring down the performance of the system. On the other hand, fine-grain locking is more efficient. However, fine-grained locking poses some engineering challenges to the programmer to develop correct programs while not compromising on efficiency.

Table 1.2: Improper locks in concurrent execution of file transfer

Th1	**Time**	**Th2**
Lock F1	1	
Find(F1, Dir1)	2	
Release F1	3	
	4	Lock F1
	5	Find(F1, Dir1)
	6	Delete(F1, Dir1)
	7	Release F1
Lock F1	8	
Modify (F1, Dir1)	9	
Release F1	10	
	11	Lock F1
	12	Add(F1, Dir2)
	13	Release F1
Inconsistent Data		

Table 1.2 demonstrates the issues created by improper use of locks with fine-grained locking. Although both the threads $Th1$ and $Th2$ are using locks but executing in an interleaving manner. Here, $Th1$ acquires the lock on file $F1$, finds the $F1$ in $Dir1$, and releases the lock from $F1$. Then $Th2$ acquires the lock on $F1$ and finds it in $Dir1$ after that deletes $F1$ from $Dir1$ followed by releasing the lock from $F1$. Later, $Th1$ again acquires the lock on $F1$ to

modify it. But $Th1$ is unaware of the latest state updated by $Th2$. Threads are designed assuming sequential execution. Hence, $Th1$ ends up with an inconsistent state due to improper use of locks. So, it can be seen that improper use of locks during concurrent execution may lead to deadlock, livelock, priority inversion, etc.

To handle this problem, one of the popular locking mechanism in the literature is two-phase locking (2PL) [1, 2]. As the name suggests it is having two phases. First is the locking phase where thread acquires the locks on all the shared data-items and works on it. In the second phase, the thread releases the locks on all the shared data-items 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 the system into deadlock. Consider an example where $Th1$ acquired the lock on file $F1$ and $Th2$ acquired the lock on file $F2$. After that, both the threads are waiting on each other to acquire the lock on the next shared data-item. Here, $Th1$ and $Th2$ are waiting for file $F2$ and $F1$ respectively. This situation leads the system into deadlock.

To address these problems, *Software Transactional Memory systems (STMs)* has emerged as an elegant alternative to locks in the past few years.

1.1 Software Transactional Memory systems (STMs)

In the era of multi-core processors, we can exploit the cores by concurrent programming. But developing an efficient concurrent program while ensuring the correctness is difficult. *Software Transactional Memory systems (STMs)* [3, 4] are a convenient programming interface for a programmer to access shared memory without worrying about consistency issues such as deadlock, livelock, priority inversion, etc. It provides a high-level abstraction to the programmer. STMs handle all the synchronization and concurrency issues associated with multi-threaded programming and help the programmers to utilize the cores effectively. Client programs use STMs by issuing transactions.

The notion of transaction in the STM system is inspired by database transactions. A transaction is a piece of code/instructions that must be executed atomically since it accesses shared variables. A transaction accesses and modifies the shared data-items called as *transactional-objects (t-objects)*.

Database transactions satisfy serializability [5] and ACID (Atomicity, Consistency, Isolation, and Durability) properties. Unlike databases, STM transactions provide atomicity, consistency and isolation properties. But, STM systems do not provide durability as by design the applications using the STM systems do not require durability.

As mentioned earlier, Software Transactional Memory systems (STMs) are a convenient programming interface that provides ease of multi-threading to the programmer [6]. With the emergence of multi-core systems, a need has arisen for a convenient programming interface to exploit all the cores of the system effectively. STMs provide such an interface that can be used with a modern programming language construct like atomic or some related API to handle the synchronization issues. The concept of a transactional memory system can be applied to hardware, software, and hybrid (a combination of hardware and software) to achieve better performance. The correct execution of file transfer using STM is shown below for threads Th1 and Th2. These code snippets demonstrate the ease of use of STMs to handle the synchronization issues.

```
Th1()
{
        initialization();
        atomic
        {
                Find(F1, Dir1)
                Modify(F1, Dir1)
        }
}
```

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

```
Th2()
{
        initialization();
        atomic
        {
                Find(F1, Dir1)
                Modify(F1, Dir1)
                Add(F1, Dir2)
        }
}
```

A transaction of STMs is a piece of code that performs reads and writes to the shared memory. In this thesis, we consider the optimistic execution of the STM system which ensures that transaction reads from the shared memory, but all write updates are performed on local memory. On completion, the STM system *validates* the reads and writes of the transaction. If any inconsistency is found, the transaction is *aborted*, and its local writes are discarded. Otherwise, the transaction is committed, and its local writes are transferred to the shared memory. A transaction that has begun but has not yet committed/aborted is referred to as *live*. One more advantage of the STMs is it provides *composability* which ensures the effect of the multiple methods of the transaction will be atomic.

Correctness: An important requirement of STMs is to precisely identify the criterion as to when a transaction should be *aborted/committed*, referred to as *correctness-criterion*. Some of the popular *correctness-criteria* of STMs are opacity [7], local opacity [8], and conflict-opacity (co-opacity) [9]. These *correctness-criteria* require that all the transactions including the aborted ones appear to execute sequentially in an order that agrees with the order of non-overlapping transactions. Unlike the correctness-criteria for traditional databases, such as serializability, strict-serializability [5], the correctness-criteria for STMs ensure that even aborted transactions see only correct values. This ensures that programmers do not see any undesirable side-effects due to the reads by the transaction that get aborted later such as divide-by-zero, infinite-loops, crashes, etc. in the application due to concurrent executions. This additional requirement on aborted transactions is a fundamental requirement of STMs which differentiates STMs from databases as observed by Guerraoui & Kapalka [7]. Thus in this thesis, we focus on optimistic executions with the *correctness-criteria* being *local opacity* [8] and co-opacity [9].

Methods of STMs: A typical STM system is a library which exports the following methods:

1. *STM_begin()*: begins a transaction T_i with unique id i.
2. *STM_read(x) or (r(x))*: T_i reads a shared data-item or *transactional object* (*t-object*) x from shared memory. Method $r(x)$ can return abort. In that case, we denote them as a failed method.
3. *STM_write(x, v) or (w(x, v))*: T_i writes to a *t-object* x with value v into its local memory.
4. *STM_tryC()*: tries to commit the transaction T_i. On successful validation, the effect of the transaction T_i will be visible to the shared memory and T_i returns commit. Method $STM\_tryC()$ can return abort. In that case, we denote them as a failed method.
5. *STM_tryA$_i$()*: T_i returns abort.

1.2 Efficiency Achieved in STMs

Typical STMs work on read/write methods and maintain single-version corresponding to each *t-object* called as *Single-Version Read-Write STMs (SV-RWSTMs or RWSTMs).*

1.2.1 Benefits of Multi-Version RWSTMs (MV-RWSTMs)

It has been shown in the literature that maintaining multiple versions corresponding to each *t-object* reduces the number of aborts and enhance the performance. Several *multi-version*

RWSTMs (MV-RWSTMs) have been proposed in the literature [10–13] that reduces the number of aborts and provide increased concurrency.

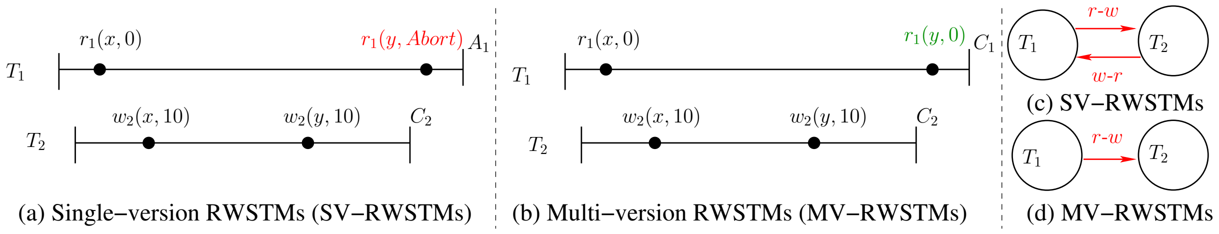


Figure 1.1: Benefits of multi-version over single-version RWSTMs

We illustrate the advantage of multiple versions over single-version with an interesting example. Consider Figure 1.1 (a) which demonstrates the execution of single-version history say $H = r_1(x,0)w_2(x,10)w_2(y,10)C_2r_1(y,A)A_1$. It consists of two transactions T_1 and T_2. Here, $r_1(x,0)$ represents read by T_1 on t-object x and returns value 0 implying that some other transaction T_0 had previously created 0 into x. $w_2(x,10)$ represents the write by T_2 on t-object x with value 10. T_2 performs write on t-object x and y and returns commit as C_2. After that T_1 wants to read y and returns abort A to make H as serializable [5]. This is reflected by a cycle in the corresponding conflict graph between T_1 and T_2, as shown in Figure 1.1 (c). Hence, to make the correct concurrent execution under SV-RWSTMs T_1 have to returns abort.

Now, consider the same history with multiple versions corresponding to each t-object as demonstrated in Figure 1.1 (b). Even after T_2 created a new version of y with value 10 the previous value of 0 is still retained. Thus, when T_1 invokes $r_1(y)$ then it returns value 0 (as previous value) and commits successfully with equivalent serial history T_1T_2. The corresponding conflict graph is shown in Figure 1.1 (d) does not have a cycle. Hence, we can conclude that multi-version RWSTMs reduce the number of aborts and improve the concurrency than SV-RWSTMs.

1.2.2 Benefits of Object-based STMs (OSTMs)

Few researchers Herlihy et al. [14], Hassan et al. [15], and Peri et al. [9] have shown that working at higher-level operations such as insert, delete and lookup on the linked-list and hash table gives better concurrency than RWSTMs. STMs which works on higher-level operations are known as *Object-based STMs (or OSTMs)* [9].

Methods of OSTMs: It exports the following methods:

1. *STM_begin()*: begins a transaction T_i with unique id i (same as RWSTMs).
2. *STM_lookup$_i(k)$* (or $l(k)$): T_i lookups t-object (or key) k from shared memory and returns the value. Method *STM_lookup$_i(k)$* can return abort. In that case, we denote them as a failed method.

3. *STM_insert*$_i(k, v)$ (or $i(k, v)$): T_i inserts a key k with value v into its local memory.

4. *STM_delete*$_i(k)$ (or $d(k)$): T_i deletes key k.

5. *STM_tryC*$_i$*()*: actual effect of *STM_insert()* and *STM_delete()* will be visible to the shared memory after successful validation and T_i returns commit. Method $STM\_tryC()$ can return abort. In that case, we denote them as a failed method.

6. *STM_tryA*$_i$*()*: Otherwise, T_i returns abort.

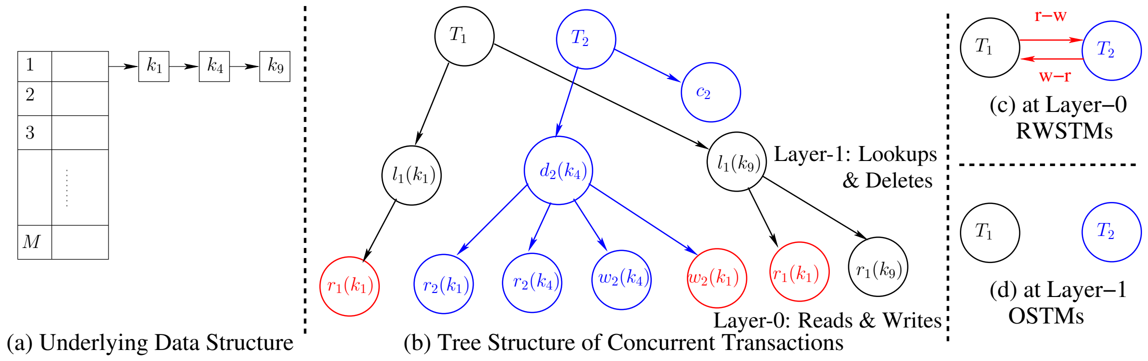


Figure 1.2: Advantage of OSTMs over RWTSMs

Motivation to work on OSTMs: Figure 1.2 represents the advantage of OSTMs over RW-STMs while achieving greater concurrency and reducing the number of aborts. Figure 1.2.(a) depicts the underlying data structure as hash table (or ht) with M buckets and bucket 1 stores three keys k_1, k_4 and k_9 in the form of a list. Thus, to access k_4, a thread has to access k_1 before it. Figure 1.2.(b) shows the tree structure of concurrent execution of two transactions T_1 and T_2 with RWSTMs at layer-0 and OSTMs at layer-1 respectively. Consider the execution at layer-0, T_1 and T_2 are in conflict. Two transactions are in conflict if both are accessing the same key k and at least one transaction performs write operation on k. Here, write operation of T_2 on key k_1 as $w_2(k_1)$ is occurring between two read operations of T_1 on k_1 as $r_1(k_1)$. So, this concurrent execution cannot be atomic as shown in Figure 1.2.(c). To make it atomic either T_1 or T_2 has to return abort. Whereas execution at layer-1 shows the higher-level operations $l_1(k_1)$, $d_2(k_4)$ and $l_1(k_9)$ on different keys k_1, k_4 and k_9 respectively. All the higher-level operations are isolated to each other so tree can be pruned [2, Chap 6] from layer-0 to layer-1 and both the transactions return commit with equivalent serial schedule T_1T_2 or T_2T_1 as shown in Figure 1.2.(d). Hence, some conflicts of RWSTMs do not matter at OSTMs which reduce the number of aborts and improve the concurrency using OSTMs.

1.2.3 Advantage of Appending Multiple Versions in OSTMs (MV-OSTMs)

It has been shown in the literature of databases and RWSTMs [10–13] that greater concurrency can be obtained by storing multiple versions for each *transactional-object (t-object)* or *key* as explained in SubSection 1.2.1. So, to achieve the greater concurrency further, Juyal et al. [16] proposed *Multi-Version OSTM (or MV-OSTM)* which maintains multiple versions corresponding to each t-object instead of maintaining single-version corresponding to each t-object as *Single-Version OSTM (SV-OSTM or OSTM)*. Specifically, maintaining multiple versions can ensure that more lookup operations succeed because the lookup operation will have an appropriate version to read.

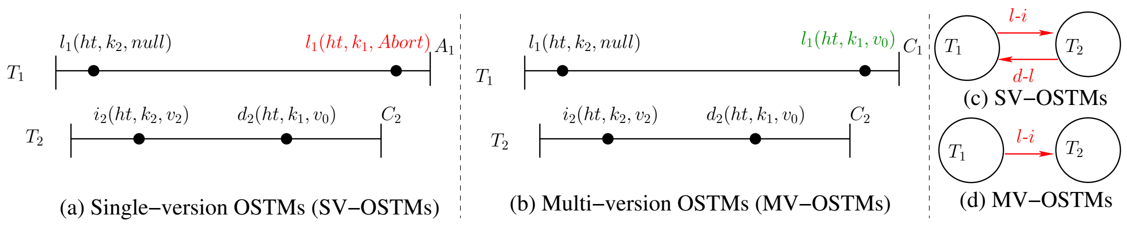


Figure 1.3: Advantages of MV-OSTM over SV-OSTM

Benefit of *MV-OSTM*s over *SV-OSTM*s and multi-version *RWSTM*s: We now illustrate the advantage of *MV-OSTM*s as compared to single-version *OSTM*s (*SV-OSTM*s) using hash table object having the same operations as discussed above in SubSection 1.2.2: *insert (or i), lookup (or l), delete (or d)*. Figure 1.3 (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 l 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 i 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 l on the key k_1. With this execution, any *SV-OSTM* system has to return abort for T_1's l operation to ensure correctness, i.e., opacity. Otherwise, if T_1 would have obtained a return value v_0 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 1.3 (c). Thus to ensure opacity, *SV-OSTM* system has to return abort for T_1's lookup on k_1.

In an *MV-OSTM* based on hash table, whenever a transaction inserts or deletes a key k, a new version is created. Consider the above example with a *MV-OSTM*, as shown in Figure 1.3 (b). Even after T_2 deletes k_1, the previous value of v_0 is still retained. Thus, when T_1 invokes l on k_1 after the delete on k_1 by T_2, *MV-OSTM* 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 is shown in Figure 1.3 (d) does not have a cycle.

Thus, *MV-OSTM* reduces the number of aborts and achieve greater concurrency than *SV-OSTM*s while ensuring the compositionality. We believe that the benefit of *MV-OSTM* over multi-version *RWSTM* is similar to *SV-OSTM* over single-version *RWSTM* as explained in SubSection 1.2.2. Hence, MV-OSTM reduces the number of aborts and improves performance than SV-OSTMs, MV-RWSTMs, and SV-RWSTMs.

1.3 Progress Conditions in STMs

All the STMs (SV-RWSTM, MV-RWSTM, SV-OSTM, and MV-OSTM) defined above ensure that transaction either $commits$ or $aborts$. A transaction aborted due to conflicts (two transactions are said to be in conflict, if both of them are accessing same *t-object* x concurrently and at least one of the transaction performs write/update on x) is typically re-issued with the expectation that it will complete successfully in a subsequent incarnation. However, many existing STMs fail to provide *starvation freedom*, i.e., in these systems, it is possible that concurrency conflicts may prevent any incarnation of a transaction from ever committing. To overcome this limitation, we develop an efficient STM system which ensures starvation-freedom.

Algorithm 1 Insert(LL, e): Invoked by a thread to insert an element e into a linked-list LL. This method is implemented using transactions.

```
1:  $retry$ = 0;
2:  while ($true$) do
3:      $id$ = $STM\_begin(retry)$;
4:      ...
5:      $v$ = $STM\_read(id, x)$; /*reads value of $x$ as $v$*/
6:      ...
7:      $STM\_write(id, x, v')$; /*writes a value $v'$ to $x$*/
8:      ...
9:      $ret$ = $STM\_tryC(id)$; /*$STM\_tryC()$ can return $commit$ or $abort$*/
10:     if ($ret == commit$) then break;
11:     else
12:         $retry$++;
13:     end if
14: end while
```

A typical code using STMs is as shown in Algorithm 1. It shows the overview of a thread safe concurrent *insert* method which inserts an element e into a linked-list LL. It consists of a loop where the thread creates a transaction. This transaction executes the code to insert an element e in a linked-list LL using $STM\_read()$ and $STM\_write()$ operations. (The result of $STM\_write()$ operation are stored locally.) At the end of the transaction, the thread calls *STM_tryC()*. At this point, the STM checks if the given transaction can be *committed* while satisfying the required safety properties (e.g., serializability [5], opacity [7]). If yes, then the transaction is *committed* and updates done by the transaction are reflected into the shared memory. Otherwise, it is aborted and all the updates made by the transaction are discarded. If the given transaction is aborted, then the invoking thread may retry that transaction again like Line 12 in Algorithm 1. The execution shown in Algorithm 1 has a possibility that the transaction which a thread tries to execute gets aborted again and again. Every time, it executes the transaction, say T_i, T_i conflicts with some other transaction and hence gets aborted. In other words, the thread is effectively starved because it is not able to commit T_i successfully.

Starvation-Freedom: A popular blocking progress condition associated with concurrent programming is starvation-freedom [17, chap 2], [18]. In the context of STMs, starvation-freedom ensures that every aborted transaction that is retried infinitely often eventually commits. It can be defined as: an STM system is said to be *starvation-free* if a thread invoking a transaction T_i gets the opportunity to retry T_i on every abort (due to the presence of a fair underlying scheduler with bounded termination) and T_i is not *parasitic*, i.e., T_i will try to commit given a chance then T_i will eventually commit. Parasitic transactions [19] will not commit even when given a chance to commit possibly because they are caught in an infinite loop or some other error.

Wait-freedom: It is another interesting progress condition for concurrent systems. In the context of STMs, it can be defined as one in which every transaction commits regardless of the nature of concurrent transactions and the underlying scheduler [18]. But it was shown by Guerraoui and Kapalka [19] that it is not possible to achieve *wait-freedom* in dynamic STMs in which data sets (or t-objects) of transactions are not known in advance. So in this thesis, we explore the weaker progress condition of *starvation-freedom* for transactional memory systems (SV-RWSTMs, MV-RWSTMs, SV-OSTMs, and MV-OSTMs) while assuming that the t-objects of the transactions are *not* known in advance.

1.4 Techniques to Achieve Starvation-Freedom

Few researchers in literature such as Gramoli et al. [20], Waliullah and Stenstrom [21], Spear et al. [22] have explored starvation-freedom in SV-RWSTMs while maintaining single-version corresponding to each t-object. Most of these systems work by assigning priorities to transactions. In case of a conflict between two transactions, the transaction with lower priority is aborted. They ensure that every aborted transaction, on being retried a sufficient number of times, will eventually have the highest priority and hence will commit. We denote such an algorithm as *Starvation-Free Single-Version RWSTM (or SF-SV-RWSTM).*

Although *SF-SV-RWSTM* guarantees starvation-freedom, it can still abort many transactions spuriously. Consider the case where a transaction T_i has the highest priority. Hence, as per *SF-SV-RWSTM*, T_i cannot be aborted. But if it is slow (for some reason), then it can cause several other conflicting transactions to abort and hence, bring down the efficiency and progress of the entire system.

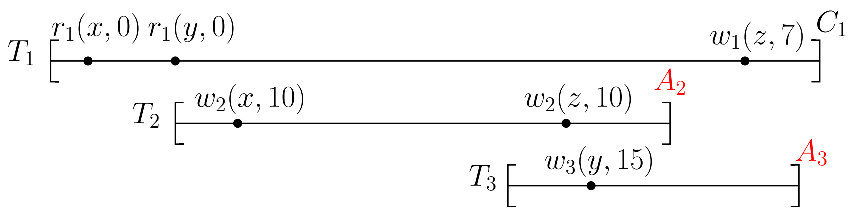


Figure 1.4: Limitation of Starvation-Free Single-Version Algorithm

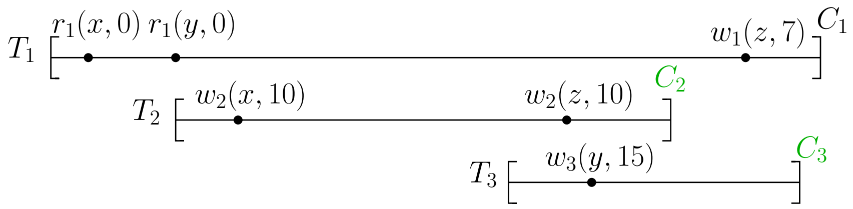


Figure 1.5: Advantage of Starvation-Free Multi-Version Algorithm

Figure 1.4 illustrates this problem. Consider the execution with: $r_1(x,0)r_1(y,0)w_2(x,10)$ $w_2(z,10)w_3(y,15)w_1(z,7)$. It has three transactions T_1, T_2 and T_3. Let T_1 have the highest priority and read t-object x and y as 0. After reading y, suppose T_1 becomes slow (for some reason). Next T_2 and T_3 want to write to x, z and y respectively and *commit*. But T_2 and T_3's write operations are in conflict with T_1's read operations. Since T_1 has higher priority and has not committed yet, T_2 and T_3 have to *abort*. If these transactions (T_2 and T_3) are retried and again conflict with T_1 (while it is still live), they will have to *abort* again. Thus, any transaction with priority lower than T_1 and conflicts with it has to abort. It is as if T_1 has locked the t-objects x, y and does not allow any other transaction, write to these t-objects and to *commit*.

1.4.1 Single-Version and Multi-Version RWSTMs

A key limitation of single-version RWSTMs is limited concurrency. As shown above in Figure 1.4, it is possible that one long transaction conflicts with several transactions causing them to abort. This limitation can be overcome by using multi-version RWSTMs where we store multiple versions of the data item (either unbounded versions with garbage collection, or bounded versions where the oldest version is replaced when the number of versions exceeds the bound).
Motivation towards Starvation-Free Multi-Version RWSTM (SF-MV-RWSTM): Several multi-version RWSTMs have been proposed in the literature [10–13] that provide increased concurrency. But none of them provide starvation-freedom. To overcome this issue of SF-SV-RWSTM, we systematically developed a novel and efficient starvation free algorithm as *Starvation-free Multi-Version RWSTM (SF-MV-RWSTM)*. It maintains multiple versions corresponding to each *t-object* which reduces the number of aborts and enhances the performance than *SF-SV-RWSTMs*.

Consider Figure 1.5 which illustrates the same execution as Figure 1.4 but maintains multiple version corresponding to each t-object. Due to maintaining the multiple versions, both T_2 and T_3 create a new version corresponding to each t-object x, z and y and return commit while not causing T_1 to abort as well. T_1 reads the initial value of z, and returns commit. So, by maintaining multiple versions all the transactions T_1, T_2, and T_3 can commit with equivalent serial history as $T_1T_2T_3$ or $T_1T_3T_2$. Thus, multiple versions can help with starvation-freedom without sacrificing on concurrency. This motivated us to developed a starvation-free multi-version RWSTM system.

Proposed SF-MV-RWSTM can be used with the case where the number of versions is unbounded and *Garbage Collection (GC)* is used to delete unwanted versions. We denote this algorithm as *SF-MV-RWSTM-GC*. Although SF-MV-RWSTMs provide greater concurrency, they suffer from the cost of garbage collection. One way to avoid this is to use bounded-version RWSTMs, where the number of versions is bounded to be at most K. Thus, when $(K + 1)^{th}$ version is created, the oldest version is removed. Furthermore, achieving starvation-freedom while using only bounded versions is especially challenging given that a transaction may rely on the oldest version that is removed to commit. In that case, it would be necessary to abort that transaction, making it harder to achieve starvation-freedom.

This thesis addresses this gap by developing a starvation-free algorithm for bounded-version RWSTMs as *Starvation-Free K-Version RWSTM (SF-K-RWSTM)* for a given parameter K. Here K is the number of versions of each t-object and can range from 1 to ∞ [1]. Our approach is different from the approach used in *SF-SV-RWSTM* to provide starvation-freedom in single-version RWSTMs (the policy of aborting lower priority transactions in case of conflict) as it does not work for MV-RWSTMs.

Our experimental analysis shows that the proposed *SF-K-RWSTM* algorithm performs best among its variants (without *garbage collection* as *SF-MV-RWSTM* and with *garbage collection* as *SF-MV-RWSTM-GC*) along with starvation-free and non-starvation-free state-of-the-art STMs under long-running transactions with high contention. *SF-K-RWSTM* gives an average speedup on the *max-time* (maximum time for a transaction to commit) by a factor of 1.22, 1.89, 23.26 and 13.12 times over *PKTO* [23], *SF-SV-RWSTM*, NOrec STM [24] and ESTM [25] respectively for counter application. *SF-K-RWSTM* performs 1.5 and 1.44 times better than *PKTO* and *SF-SV-RWSTM* but 1.09 times worse than NOrec for low contention KMEANS application of STAMP [26] benchmark whereas *SF-K-RWSTM* performs 1.14, 1.4 and 2.63 times better than *PKTO*, *SF-SV-RWSTM* and NOrec for LABYRINTH application of STAMP benchmark which has high contention with long-running transactions.

1.4.2 Single-Version and Multi-Version Object-based STMs

SV-OSTMs reduce the number of aborts and improve the performance than SV-RWSTMs as explained in SubSection 1.2.2. The transactions of SV-OSTMs can return commit or abort. Whenever a SV-OSTM transaction is aborted (due to conflicts), it is typically re-issued with the expectation that it will complete successfully in a subsequent incarnation. However, none of the existing SV-OSTMs provide *starvation freedom*, i.e., in these SV-OSTMs, it is possible that concurrency conflicts may prevent an incarnated transaction from ever committing. Figure 1.6.(a) illustrates the execution under SV-OSTMs on hash table *ht* as shown in Figure 1.2. It has same methods *insert (or i)*, *lookup (or l)*, and *delete (or d)* as discussed above in Sub-

[1]We use ∞ to denote the largest possible value that can be represented in a system.

Section 1.2.2. Figure 1.6.(a) demonstrates the execution in which transaction T_1 is starving. We represent that any transaction T_i has the timestamp i. Here, transaction T_2 with higher timestamp 2 than the timestamp of T_1 as 1 has already been committed so a conflicting lower timestamp transaction T_1 returns abort [9]. T_1 retries with an incarnation as T_3 but again conflicting transaction T_4 with higher timestamp than T_3 has been committed which causes T_3 (an incarnation of T_1) to abort again. This situation can occur again and again and leads to starve the transaction T_1.

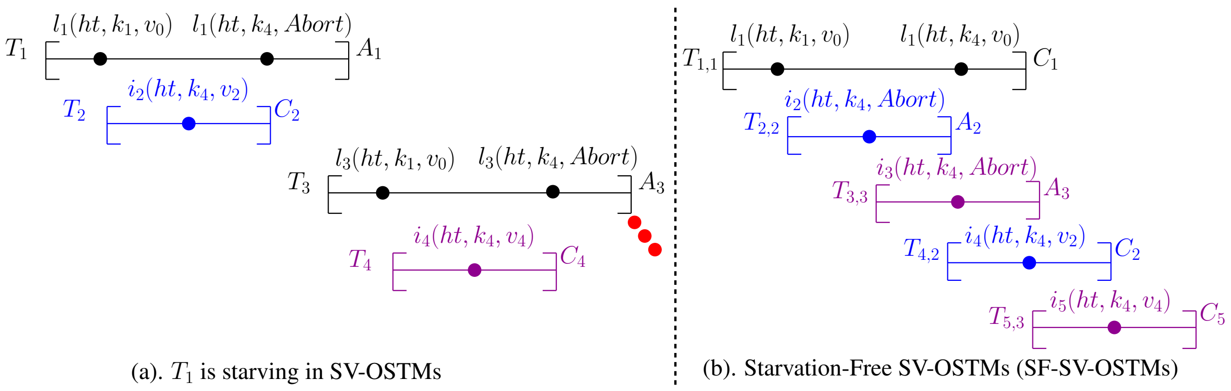
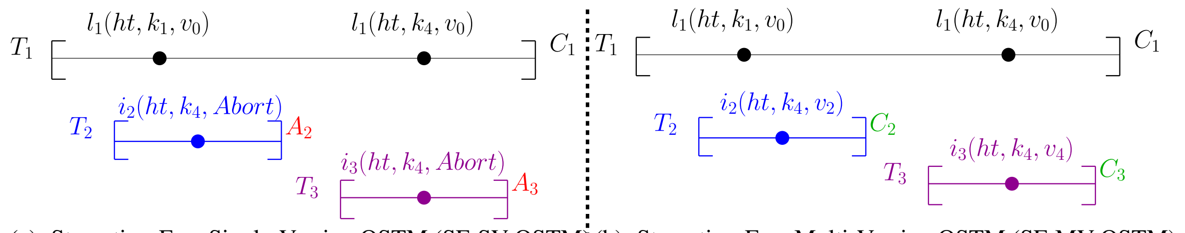


Figure 1.6: Advantage of SF-SV-OSTM over SV-OSTMs

Motivation to Propose Starvation-Freedom in Single-Version OSTM: To achieve starvation-freedom along within SV-OSTMs, we propose Starvation-Free algorithm for SV-OSTM as *SF-SV-OSTM*. It maintains single-version corresponding to each t-object. The main idea of this algorithm is similar to SF-SV-RWSTM. SF-SV-OSTM uses the notion of timestamps similar to SF-SV-RWSTM. To ensure starvation-freedom it assigns higher priority to the lowest timestamp transaction. Here, each transaction maintains two timestamps, *Initial Timestamp* (its) and *Current Timestamp* (cts). Whenever a transaction T_i starts using *STM_begin()*, it gets a unique timestamp which for simplicity we denote as i. When a transaction begins for the first time, it gets a its which is same as cts as well. On subsequent invocations, the cts changes but its remains the same. Figure 1.6.(b) demonstrates the execution under SF-SV-OSTM in which $T_{1,1}$ as $T_{\langle cts,its \rangle}$ represents the first incarnation of T_1 so, cts equals to its as 1. $T_{1,1}$ conflicts with $T_{2,2}$ and $T_{3,3}$. As mentioned earlier, SF-SV-OSTM gives preference to the transaction with lower timestamp. More precisely, it gives preference to the transaction having lower its. As $T_{1,1}$ have the lowest its so T_1 gets the priority to execute whereas $T_{2,2}$ and $T_{3,3}$ return aborts. On abort, $T_{2,2}$ and $T_{3,3}$ retries with new cts 4 and 5 but with same its 2 and 3 respectively. So, due to lowest its $T_{4,2}$ returns commit but $T_{5,3}$ returns abort and so on. Hence, none of the transactions starve. The key idea here is that by assigning priority to the transaction having lower its (i.e. longer running) in case of conflict ensures starvation-freedom. SF-SV-OSTM achieves starvation-freedom and satisfies the correctness criteria as *conflict-opacity* [9].

Motivation to Propose Starvation-Freedom in Multi-Version OSTM: In SF-SV-OSTM al-



(a). Starvation-Free Single-Version OSTM (SF-SV-OSTM) (b). Starvation-Free Multi-Version OSTM (SF-MV-OSTM)

Figure 1.7: Benefits of Starvation-Free MV-OSTM over SF-SV-OSTM

gorithm described above, if the highest priority transaction becomes slow (for some reason), then it may cause several other transactions to abort and bring down the progress of the system. This case is similar to the drawback of SF-SV-RWSTM. Figure 1.7.(a) demonstrates this in which the highest priority transaction T_1 became slow so, it is forcing the conflicting transactions T_2 and T_3 to abort again and again until T_1 commits with the same reasoning explained in SubSection 1.4.1 for SF-SV-RWSTMs. Database, RWSTMs [10–13] and OSTMs [16] literature show that maintaining multiple versions corresponding to each key reduces the number of aborts and improves throughput.

So, to achieve the greater concurrency further, this thesis proposes a novel and efficient *Starvation-Free Multi-Version OSTM (SF-MV-OSTM)* which maintains multiple versions corresponding to each key (or t-object). Figure 1.7.(b) shows the benefits of using SF-MV-OSTM in which T_1 lookups from the older version with value v_0 created by transaction T_0 (assuming as initial transaction) for key k_1 and k_4. Concurrently, T_2 and T_3 create the new versions for key k_4. So, all the three transactions commit with equivalent serial schedule $T_1T_2T_3$. So, SF-MV-OSTM improves the concurrency than SF-SV-OSTM while reducing the number of aborts and ensures the starvation-freedom.

We propose SF-SV-OSTM and SF-MV-OSTM for *hash table* and *linked-list* data structure but it can be generalized to other data structures as well. The number of versions maintains by SF-MV-OSTM either be unbounded with Garbage Collection (GC) denoted as *SF-MV-OSTM-GC* or bounded with latest K-versions denoted as *SF-K-OSTM*. SF-K-OSTM ensures starvation-freedom and satisfies the correctness criteria as local opacity [8].

Experimental analysis shows that SF-K-OSTM is best among all proposed Starvation-Free OSTMs (SF-SV-OSTM, SF-MV-OSTM, and SF-MV-OSTM-GC) for both hash table and linked-list data structure. Proposed hash table based SF-K-OSTM (HT-SF-K-OSTM) performs 3.9x, 32.18x, 22.67x, 10.8x and 17.1x average speedup on *max-time* (maximum time for a transaction to commit) than state-of-the-art STMs HT-K-OSTM [16], HT-SV-OSTM [9], ESTM [25], RWSTM [2, Chap. 4], and HT-MVTO [10] respectively. Proposed list based SF-K-OSTM (list-SF-K-OSTM) performs 2.4x, 10.6x, 7.37x, 36.7x, 9.05x, 14.47x, and 1.43x average speedup on *max-time* than state-of-the-art STMs list-K-OSTM [16], list-SV-OSTM [9], Trans-list [27], Boosting-list [14], NOrec-list [24], list-MVTO [10], and list-SF-K-RWSTM [23] respectively.

1.5 Application of Efficient Multi-Version STMs in Blockchain

Maintaining multiple versions corresponding to each shared t-object in Software Transactional Memory systems (STMs) increase the concurrency and improve the performance. So in this thesis, we use efficient multi-version STMs to improve the performance of *blockchain* systems. It is commonly believed that blockchain is a revolutionary technology for doing business on the Internet. Blockchain is a decentralized, distributed database or ledger of records. Cryptocurrencies such as Bitcoin [28] and Ethereum [29] were the first to popularize the blockchain technology. Blockchains ensure that the records are tamper-proof but publicly readable.

Basically, the blockchain network consists of multiple peers (or nodes) where the peers do not necessarily trust each other. Each node maintains a copy of the distributed ledger. *Clients*, users of the blockchain, send requests or *transactions* to the nodes of the blockchain called as *miners*. The miners collect multiple transactions from the clients, execute it sequentially, and form a *block*. Miners then propose these blocks to be added to the blockchain. They follow a global consensus protocol to agree on which blocks are chosen to be added and in what order. While adding a block to the blockchain, the miner incorporates the hash of the previous block into the current block. This makes it difficult to tamper with the distributed ledger. The resulting structure is a chain of blocks and hence the name blockchain.

The transactions sent by clients to miners are part of a larger code called as *smart contracts* that provide several complex services such as managing the system state, ensuring rules, or credentials checking of the parties involved [30]. Smart contracts are like a 'class' in programming languages that encapsulate data and methods which operate on the data. The data represents the state of the smart contract (as well as the blockchain) and the methods (or functions) are the transactions that possibly can change contract state. A transaction invoked by a client is typically such a method or a collection of methods of the smart contracts. Ethereum uses Solidity [31] while Hyperledger [32] supports language such as Java, Golang, Node.js etc.

Motivation for Concurrent Execution of Smart Contracts: As observed by Dickerson et al. [30], smart contract transactions are executed in two different contexts specifically in Ethereum. First, they are executed by miners while forming a block. A miner selects a sequence of client request transactions, executes the smart contract code of these transactions in sequence, transforming the state of the associated contract in this process. The miner then stores the sequence of transactions, the resulting final state of the contracts in the block along with the hash of the previous block. After creating the block, the miner proposes it to be added to the blockchain through the consensus protocol.

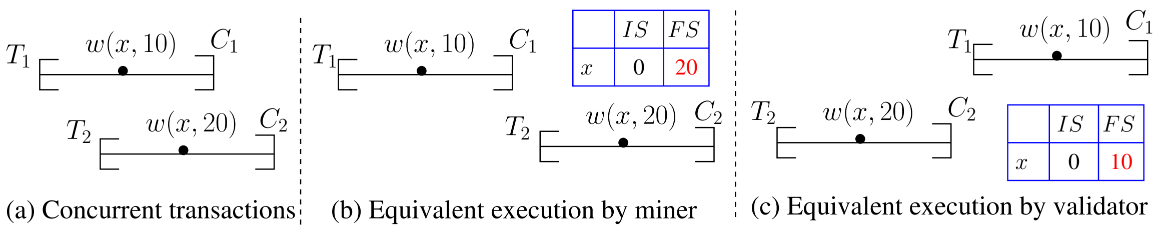
Once a block is added, the other peers in the system, referred to as *validators* in this context, validate the contents of the block. They re-execute the smart contract transactions of the block sequentially to verify the block's final states match or not. If final states match, then the block is accepted as valid and the miner who appended this block is rewarded. Otherwise, the block is discarded. Thus the transactions are executed by every peer in the system. In this setting, it turns out that the validation code runs several times more than miner code [30].

This design of smart contract execution is not very efficient as it does not allow any concurrency. Both the miner and the validator execute transactions serially one after another. In today's world of multi-core systems, the serial execution does not utilize all the cores and hence results in lower throughput. This limitation is not specific only to Ethereum but almost all the popular blockchains such as Bitcoin [28], EOSIO [33]. Higher throughput means more number of transactions executed per unit time by miners and validators which clearly will be desired by both of them.

But the concurrent execution of smart contract transactions is not an easy task. The various transactions requested by the clients could consist of conflicting access to the shared data-objects. Arbitrary execution of these transactions by the miners might result in the data-races leading to the inconsistent final state of the blockchain. Unfortunately, it is not possible to statically identify if two contract transactions are conflicting or not since they are developed in Turing-complete languages [30]. The common solution for correct execution of concurrent transactions is to ensure that the execution is *serializable* [5]. The standard correctness-criterion in databases, serializability ensures that the concurrent execution is equivalent to some serial execution of the same transactions. Thus the miners must ensure that their execution is serializable [30].

The concurrent execution of the smart contract transactions of a block by the validators, although highly desirable, can further complicate the situation. Suppose a miner ensures that the concurrent execution of the transactions in a block are serializable. Later a validator executes the same transactions concurrently. But during the concurrent execution, the validator may execute two conflicting transactions in an order different from what was executed by the miner. Thus the serialization order of the miner is different from the validator. Then this can result in the validator obtaining a final state different from what was obtained by the miner. Consequently, the validator may falsely reject the block although it is valid. We refer this problem as *False Block Rejection* (or *FBR*) error.

Figure 1.8 illustrates this in the following example. Figure 1.8 (a) consists of two concurrent conflicting transactions T_1 and T_2 working on same shared data-objects x which are part of a block. Figure 1.8 (b) represents the concurrent execution by miner with an equivalent serial schedule as T_1, T_2 and final state (or FS) as 20 from the initial state (or IS) 0. Whereas Figure 1.8 (c), shows the concurrent execution by a validator with an equivalent serial schedule as T_2, T_1, and final state as 10 from IS 0 which is different from the final state proposed by the miner. Such a situation leads to false rejection of the valid block by the validator which is undesirable. This can negate the benefits of concurrent executions.



T_1
$w(x,10)$
C_1
T_2
$w(x,20)$
C_2
(a) Concurrent transactions
(b) Equivalent execution by miner
IS
FS
x
0
20
(c) Equivalent execution by validator
10

Figure 1.8: Concurrent execution of transactions by miner and validator

These important issues were identified by Dickerson et al. [30] who proposed a solution of concurrent execution for both the miners and validators. In their solution, the miners concurrently execute the transactions of a block using abstract locks and inverse logs to generate a serializable execution. Then, to enable correct concurrent execution by the validators, the miners also provide a *happens-before* graph in the block. The happens-before graph is a directed acyclic graph over all the transaction of the block. If there is a path from a transaction T_i to T_j then the validator has to execute T_i before T_j. Transactions with no path between them can execute concurrently. The validator using the happens-before graph in the block executes all the transactions concurrently using the fork-join approach. This methodology ensures that the final state of the blockchain generated by the miners and the validators are the same for a valid block and hence not rejected by the validators.

The presence of tools such as a happens-before graph in the block provides greater enhancement to validators to consider such blocks as it helps them execute quickly by means of parallelization as opposed to a block which does not have any tools for parallelization. This, in turn, entices the miners to provide such tools in the block for concurrent execution by the validators.

Our Solution Approach - Optimistic Concurrent Execution and Lock-Free Graph: Dickerson et al. [30] developed a solution to the problem of concurrent miner and validators using locks and inverse logs. It is well known that locks are pessimistic in nature. So, in this thesis, we explore a novel and efficient framework for concurrent miners using optimistic Multi-Version Software Transactional Memory systems (MVSTMs).

The requirement of the miner, as explained above, is to concurrently execute the smart contract transactions correctly and output a graph capturing dependencies among the transactions of the block such as happens-before graph. We denote this graph as *Block Graph* (or *BG*). In the proposed solution, the miner uses the services of an optimistic STM system to concurrently execute the smart contract transactions. Since STMs also work with transactions, we differentiate between smart contract transactions and STM transactions. The STM transactions invoked by an STM system is a piece of code that it tries to execute atomically even in presence of other concurrent STM transactions. If the STM system is not able to execute it atomically, then the STM transaction is aborted.

The expectation of a smart contract transaction is that it will be executed serially. Thus, when it is executed in a concurrent setting, it is expected to be executed atomically (or serialized). Thus to differentiate between smart contract transaction from STM transaction, we denote smart contract transaction as *Atomic Unit* or *atomic-unit* and STM transaction as transaction in the rest of the document. Thus the miner uses the STM system to invoke a transaction for each atomic-unit. In case the transaction gets aborted, then the STM repeatedly invokes new transactions for the same atomic-unit until a transaction invocation eventually commits.

Among the various STMs available, we have chosen two timestamp based STMs in our design to execute the smart contract transactions concurrently by miner: (1) *Basic Timestamp Ordering* or *BTO* STM [2, Chap 4], maintains only one version for each t-object and proposed *BTO Miner*. (2) *Multi-Version Timestamp Ordering* or *MVTO* STM [10], maintains multiple versions corresponding to each t-object and proposed *MVTO Miner* which further reduces the number of aborts and improves the throughput.

The advantage of using timestamp based STM is that in these systems the equivalent serial history is ordered based on the timestamps of the transactions. Thus using the timestamps, the miner can generate the BG of the atomic-units. Dickerson et al. [30], developed the BG in a serial manner. In our approach, the graph is developed by the miner in concurrent and lock-free [18] manner.

The validator process creates multiple threads. Each of these threads parses the BG and re-execute the atomic-units for validation. The BG provided by concurrent miner shows dependency among the atomic-units. Each validator thread, claims a node which does not have any dependency, i.e. a node without any incoming edges by marking it. After that, it executes the corresponding atomic-units deterministically. Since the threads execute only those nodes that do not have any incoming edges, the concurrently executing atomic-units will not have any conflicts. Hence the validator threads need not to worry about synchronization issues. We denote this approach adopted by the validator as a decentralized approach (or Decentralized Validator) as the multiple threads are working on BG concurrently in the absence of master thread. So, we proposed two decentralized validators as *BTO Decentralized* and *MVTO Decentralized Validator*.

The approach adopted by Dickerson et al. [30], works on fork-join in which a master thread allocates different tasks to slave threads. The master thread will identify those atomic-units which do not have any dependencies from the BG and allocates them to different slave threads to work on. So, we proposed two fork-join validators as *BTO Fork-join Validator* and *MVTO Fork-join Validator*. In this thesis, we compare the performance of both these approaches with the serial validator.

We have executed the smart contract transactions concurrently using two efficient protocols of STMs. Initially, we used single-version RWSTM protocol as Basic Timestamp Ordering (BTO) [2, Chap 4] which maintains only one version corresponding to each t-object. Later, we employed Multi-Version RWSTM protocol as Multi-Version Timestamp Ordering (MVTO)

[10] which maintains multiple versions corresponding to each t-object. BTO and MVTO miner performed 3.6x and 3.7x average speedups over serial miner respectively. Along with, BTO and MVTO validator outperformed average 40.8x and 47.1x than serial validator respectively. So, this thesis addressed the bottleneck of blockchain by executing the smart contract transactions concurrently using efficient MVSTM protocols.

Contribution of the thesis is as follows:

- We proposed an efficient and novel starvation-free multi-version RWSTM system as *Starvation-Free K-version STM* or *SF-K-RWSTM* for a given parameter K in Chapter 3. Here K is the number of versions of each t-object and can range from 1 to ∞. To the best of our knowledge, this is the first starvation-free MV-RWSTM. We developed *SF-K-RWSTM* algorithm in a step-wise manner starting from MVTO [10] *(Multi-Version Timestamp Order)* as follows:
  - First, in SubSection 3.3.3, we used the standard idea to provide higher priority to older transactions. Specifically, we proposed priority-based K-version STM algorithm *Priority-based K-version MVTO* or *PKTO*. This algorithm guarantees the safety properties of strict-serializability [5] and local opacity [8]. However, it is not starvation-free.
  - We analyzed *PKTO* to identify the characteristics that will help us to achieve preventing a transaction from getting aborted forever. This analysis leads us to the development of *Starvation-Free K-version TO* or *SF-K-TO* (SubSection 3.3.5), a starvation-free multi-version STM obtained by revising *PKTO*. But SF-K-TO does not satisfy correctness, i.e., strict-serializability, and local opacity.
  - Finally, we extended SF-K-TO to develop *SF-K-RWSTM* (SubSection 3.3.6) that preserves the starvation-freedom, strict-serializability, and local opacity. Our algorithm works on the assumption that any transaction that is not deadlocked, terminates (commits or aborts) in a bounded time.
  - Our experiments (Section 3.6) show that *SF-K-RWSTM* gives an average speedup on the *max-time* for a transaction to commit by a factor of 1.22, 1.89, 23.26 and 13.12 times over *PKTO*, *SF-SV-RWSTM*, NOrec STM [24] and ESTM [25] respectively for counter application. *SF-K-RWSTM* performs 1.5 and 1.44 times better than *PKTO* and *SF-SV-RWSTM* but 1.09 times worse than NOrec for low contention KMEANS application of STAMP [26] benchmark whereas *SF-K-RWSTM* performs 1.14, 1.4 and 2.63 times better than *PKTO*, *SF-SV-RWSTM* and NOrec for LABYRINTH application of STAMP benchmark which has high contention with long-running transactions.
- We proposed two Starvation-Free OSTM system in Chapter 4 as follows:

  - Initially, we proposed Starvation-Freedom for Single-Version OSTM as SF-SV-OSTM in Section 4.3 which satisfies correctness criteria as *conflict-opacity (or co-opacity)* [9].
  - To achieve the greater concurrency further, we proposed Starvation-Freedom for Multi-Version OSTM as SF-MV-OSTM in Section 4.6 which maintains multiple versions corresponding to each key and satisfies the correctness as *local opacity* [8].
  - We proposed SF-MV-OSTM for *hash table* and *linked-list* data structure described in SubSection 4.6.1 but it is generic for other data structures as well.
  - SF-MV-OSTM works for unbounded versions with *Garbage Collection (GC)* as SF-MV-OSTM-GC which deletes the unwanted versions from version list of keys and for bounded/finite versions as SF-K-OSTM which stores finite say latest K number of versions corresponding to each key k. So, whenever any thread creates $(K+1)^{th}$ version of key, it replaces the oldest version of it. The most challenging task is achieving starvation-freedom in bounded version OSTM because say, the highest priority transaction relies on the oldest version that has been replaced. So, in this case, the highest priority transaction has to return abort and hence make it harder to achieve starvation-freedom unlike the approach follow in SF-SV-OSTM. Thus, we proposed a novel approach SF-K-OSTM which bridges the gap by developing starvation-free OSTM while maintaining bounded number of versions.
  - Section 4.10 shows that SF-K-OSTM is best among all proposed Starvation-Free OSTMs (SF-SV-OSTM, SF-MV-OSTM, and SF-MV-OSTM-GC) for both hash table and linked-list data structure. Proposed hash table based SF-K-OSTM (HT-SF-K-OSTM) performs 3.9x, 32.18x, 22.67x, 10.8x and 17.1x average speedup on *max-time* for a transaction to commit than state-of-the-art STMs HT-K-OSTM [16], HT-SV-OSTM [9], ESTM [25], RWSTM [2, Chap. 4], and HT-MVTO [10] respectively. Proposed list based SF-K-OSTM (list-SF-K-OSTM) performs 2.4x, 10.6x, 7.37x, 36.7x, 9.05x, 14.47x, and 1.43x average speedup on *max-time* for a transaction to commit than state-of-the-art STMs list-KOSTM [16], list-SV-OSTM [9], Trans-list [27], Boosting-list [14], NOrec-list [24], list-MVTO [10], and list-K-SFTM [23] respectively.

- We used efficient Multi-Version STMs as an application to improve the performance of *Blockchain* in Chapter 5.
  - We introduced a framework for the concurrent execution of SCTs by miner in SubSection 5.3.2.
    * We proposed a novel way to execute the smart contract transactions efficiently using BTO [2, Chap 4] by the miner while ensuring correctness criteria as *co-opacity* [9].

    * To achieve the greater concurrency further, we proposed a new way for the execution of SCTs by the miner using efficient MVTO [10] while satisfying the correctness criteria as opacity [7].
- We proposed the concurrent execution of smart contract transactions by validator which uses BG given by miner to avoid FBR error in SubSection 5.3.3. The validator executes the smart contract transactions using (a) fork-join and (b) decentralized approaches.
- We performed extensive simulations in Section 5.5.
    * The concurrent execution of smart contract transactions by BTO and MVTO miner provide an average speedup of 3.6x and 3.7x over serial miner.
    * BTO and MVTO based decentralized validator provide on average of 40.8x and 47.1x over serial validator.

1.6 Organization of the Thesis

This thesis explores the progress guarantee *starvation-freedom* in single and multi-version RW-STMs, single and multi-version OSTMs while satisfying the correctness-criteria as *co-opacity* and *local opacity*. It shows that maintaining multiple versions improves the concurrency than single-version while reducing the number of aborts and increases the throughput. STMs with unbounded versions ensure that read/lookup method of any transaction will always find a version to read/lookup from, hence never returns abort and enhance the performance. So, we use efficient multi-version STMs as an application to improve the performance of *blockchain*. The organization of the thesis is as follows:
Chapter 2 describes the system model and preliminaries of the thesis. It explains assumptions about the systems of n processes/threads followed by definitions of Events, Methods, Transactions, Histories, Real-Time order and Serial Histories, Conflict-order. It illustrates the correctness-criteria for databases and STMs in Section 2.1 and Section 2.2 respectively.
Chapter 3 explores the progress guarantees as starvation-freedom in single and multi-version RWSTMs. Section 3.1 describes the motivation towards multi-version RWSTMs over single-version RWSTMs along with the related work on starvation-free STMs. It includes the graph characterization of local opacity [8] in Section 3.2 which helps to prove the correctness of proposed algorithms. Section 3.3 introduces a novel, efficient, and starvation-free bounded-version RWSTM system as *Starvation-Free K-version RWSTM* or *SF-K-RWSTM* for a given parameter K. Here K is the number of versions of each t-object and can range from 1 to ∞. It includes design and data structures of proposed *SF-K-RWSTM* algorithm along with the detailed pseudocode. Section 3.4 and Section 3.5 explain the liveness and safety proof of proposed *SF-K-RWSTM* algorithm. Section 3.6 describes the experimental analysis of proposed *SF-K-RWSTM* with state-of-the-art STMs followed by summary of this chapter in Section 3.7.

Chapter 4 introduces the progress guarantees as starvation-freedom in single and multi-version OSTMs. Section 4.1 presents the motivation towards MV-OSTMs over SV-OSTMs, MV-RWSTMs, and SV-RWSTMs. Section 4.2 describes the graph characterization of conflict-opacity [9] which helps to prove the correctness of proposed algorithms. Initially, we propose Starvation-Freedom in Single-Version OSTM as SF-SV-OSTM for *hash table* and *linked-list* data structure describe in SubSection 4.3.2 but it is generic for other data structures as well. Section 4.4 and Section 4.5 shows the liveness and safety proof of SF-SV-OSTM. To achieve the greater concurrency further, we propose Starvation-Freedom for Multi-Version OSTM as SF-K-OSTM in Section 4.6 which maintains K number of versions corresponding to each key and satisfies the correctness criteria as *local opacity* [34]. We propose SF-K-OSTM for *hash table* and *linked-list* data structure describe in SubSection 4.6.1 but it is generic for other data structures as well. Section 4.7 describes the graph characterization of local opacity. Section 4.8 and Section 4.9 shows the liveness and safety proof of SF-K-OSTM. Section 4.10 shows that SF-K-OSTM is best among all propose Starvation-Free OSTMs (SF-SV-OSTM, SF-MV-OSTM, and SF-MV-OSTM-GC) and state-of-the-art STMs for both hash table and linked-list data structure. Section 4.11 demonstrates the summary of this chapter.

Chapter 5 describes the current design and bottleneck of the *blockchain*. After that, it uses efficient multi-version STMs as an application to improve the performance of blockchain while executing the smart contract transactions concurrently in Section 5.1. First, we studied and analyzed the requirements of concurrent miner, concurrent validator and BG in Section 5.2. We introduced a novel way to execute the smart contract transactions by concurrent miner using optimistic STMs in SubSection 5.3.2. Here, we implemented the concurrent miner with the help of BTO and MVTO protocol of STMs but it is generic to any STM protocol. To get rid of FBR error, concurrent miner proposes a lock-free graph library to generate the BG. After that, we proposes concurrent validator in SubSection 5.3.3 which re-executes the smart contract transactions deterministically and efficiently with the help of BG given by concurrent miner. We proved the correctness of BG, concurrent miner and concurrent validator in Section 5.4. Experimental analysis shown in Section 5.5, followed by summary of this chapter in Section 5.6.

Chapter 6 describes the contributions of this thesis followed by direction for future research. Section 6.1 presents the three main contributions of the thesis. It explored the weaker progress condition *starvation-freedom* for single and multi-version RWSTMs, single and multi-version OSTMs while satisfying the correctness-criteria as *conflict-opacity* and *local opacity*. It shows that maintaining multiple versions improves the concurrency than single-version while reducing the number of aborts and increases the throughput. So, we use efficient multi-version STMs as an application to improve the performance of blockchain. Section 6.2 explains the directions for future research have been proposed in the thesis. It includes identification of malicious miner by the smart concurrent validator, optimizing the size of the BG developed by the concurrent miner, the implementation of the proposed framework to actual blockchain such as EOS [33], and concurrent execution of smart contract transactions using object semantics to achieve the better performance. Finally, we summaries this chapter in Section 6.3.

Chapter 2

System Model and Preliminaries

The notions, definitions, and preliminaries of this thesis follows [8,35]. We assume a system of n processes/threads, $p_1, \ldots, p_n$ that run in a completely asynchronous manner and communicate through a collection of *transactional objects (t-object)* or *keys* $\mathscr{K}$ via atomic *transactions*. We also assume that none of the threads crash or fail abruptly. In this thesis, a thread executes the methods on t-object or $\mathscr{K}$ via atomic *transactions* $T_1, \ldots, T_n$ and receives the corresponding response. Each transaction has a unique identifier. Within a transaction, processes can perform *transactional operations or methods*.

Events and Methods: Transaction T_i of the RWSTM system works at read-write level which invokes multiple read-write (or lower-level) operations known as *events* (or $evts$). Mainly, it invokes *STM_begin()* that begins a transaction with unique identifier, *STM_write*(x, v) (or $w(x, v)$) operation that updates a t-object x with value v in its local memory, the *STM_read*(x, v) (or $r(x, v)$) operation tries to read x and returns value v, *STM_tryC*$(\mathscr{C})$ that tries to commit the transaction and returns $commit$ $\mathscr{C}$ if it succeeds. Otherwise, *STM_tryA*$(\mathscr{A})$ that aborts the transaction and returns $abort$ $\mathscr{A}$ if it fails as defined in Section 1.1. For the sake of presentation simplicity, we assume that the value v taken as argument by *STM_write*(x, v) are unique. For a transaction T_k, we denote all the t-objects accessed by its read operations as $rset_k$ and t-objects accessed by its write operations as $wset_k$. We denote all the operations of a transaction T_k as $T_k.evts$ or $evts_k$.

Operations *STM_read*$()$ and *STM_tryC*$()$ may return $\mathscr{A}$, in which case we say that the operations *forcefully abort*. Otherwise, we say that the operations have *successfully* executed. Each operation is equipped with a unique transaction identifier. A transaction T_i starts with the first operation and completes when any of its operations return $\mathscr{A}$ or $\mathscr{C}$. We denote any operation that returns $\mathscr{A}$ or $\mathscr{C}$ as *terminal operations*. Hence, operations *STM_tryC()* and *STM_tryA()* are terminal operations. A transaction does not invoke any further operations after terminal operations. The transaction which neither committed nor aborted is known as *live transactions*.

Threads execute the transactions with higher-level methods (or operations) on OSTM system internally that execute multiple read-write (or lower-level) operations as shown in Figure 1.2. A thread executes higher-level operations on $\mathscr{K}$ via transaction T_i are known as *methods* (or $mths$). T_i at object level (or higher-level) invokes *STM_begin()*, *STM_lookup$_i$(k) (or $l_i(k)$)*, *STM_insert$_i$(k, v) (or $i_i(k, v)$)*, *STM_delete$_i$(k) (or $d_i(k)$)*, *STM_tryC$_i$()*, and *STM_tryA$_i$()* methods described in SubSection 1.2.2. We denote a method m_{ij} as the j^{th} method of T_i. Method *invocation* (or inv) and *response* (or rsp) on higher-level methods are also considered as an event.

Here, *STM_lookup()*, and *STM_delete()* return the value from underlying data structure so, we called these methods as *return value method (or $rv\_method()$)*. Whereas, *STM_insert()*, and *STM_delete()* are updating the underlying data structure after successful *STM_tryC()* so, we called these methods as *update* method (or $upd\_method()$). *STM_delete()* behaves as a $rv\_method()$ as well as an $upd\_method()$. The $rv\_method()$ and $upd\_method()$ of OSTM system may return $\mathscr{A}$. Similar to the transaction of RWSTM system, a transaction T_i of OSTM system begins with unique timestamp i using *STM_begin()* and completes with any of its method which returns either commit as $\mathscr{C}$ or abort as $\mathscr{A}$. Hence, *STM_tryC()* and *STM_tryA()* are terminal operations represented as $Term(T_i)$. A transaction does not invoke any further methods after terminal operations. For a transaction T_i, we denote all the keys accessed by its $rv\_method_i()$ and $upd\_method_i()$ methods as $rvSet_i$ and $updSet_i$ respectively. Depending on the context, we ignore some of the parameters of the transactional methods.

Transactions: We follow multi-level transactions [2] model which consists of two layers. Layer 0 (or lower-level) is RWSTM system which composed of read-write operations. Whereas layer 1 (or higher-level) is OSTM system comprises of object-level methods which internally calls multiple read-write events. Formally, we define a transaction T_i at higher-level as the tuple $\langle T_i.evts, <_{T_i}\rangle$, here $<_{T_i}$ represents the total order among all the events of T_i.

History: A *history* is a sequence of *events*, i.e., a sequence of invocations and responses of transactional operations. The collection of events is denoted as $H.evts$. For simplicity, we only consider *sequential* histories here: the invocation of each transactional operation is immediately followed by a matching response. Therefore, we treat each transactional operation as one atomic event, and let $<_H$ denote the total order on the transactional operations incurred by H. With this assumption, the only relevant events of a transaction T_k of RWSTM system are $r_k(x, v)$, $r_k(x, \mathscr{A})$, $w_k(x, v)$, $STM\_tryC_k(\mathscr{C})$ (or c_k for short), $STM\_tryC_k(\mathscr{A})$, $STM\_tryA_k(\mathscr{A})$ (or a_k for short) and the only relevant methods of a transaction T_i of OSTM system are $l_i(k, v)$, $l_i(k, \mathscr{A})$, $i_i(k, v)$, $d_i(k, v)$, $STM\_tryC_i(\mathscr{C})$ (or c_i), $STM\_tryC_i(\mathscr{A})$, and $STM\_tryA_i(\mathscr{A})$ (or a_k). We identify a history H as tuple $\langle H.evts, <_H\rangle$.

Let $H|T$ denote the history consisting of events of T in H, and $H|p_i$ denote the history consisting of events of p_i in H. We only consider *well-formed* histories here, i.e., no transaction of a process begins before the previous transaction invocation has completed (either $commits$ or $aborts$) and a transaction can not invoke any other method after receiving the response as $\mathscr{C}$ or $\mathscr{A}$. We also assume that every history has an initial *committed* transaction T_0 that initializes all the t-objects with value 0 in RWSTMs.

The set of transactions that appear in H is denoted by $H.txns$. The set of *committed* (resp., *aborted*) transactions in H is denoted by $H.committed$ (resp., $H.aborted$). The set of *incomplete* or *live* transactions in H is denoted by $H.incomp = H.live = (H.txns - H.committed - H.aborted)$. For a history H, we construct *completion* of H, denoted as $\overline{H}$, by inserting $STM\_tryA_k(\mathscr{A})$ immediately after the last event of every transaction $T_k \in H.live$.

Real-Time Order and Serial History: Two complete method m_{ij} and m_{xy} are said to be in method real-time order (or MR), if the response of method m_{ij} happens before the invocation of m_{xy}. Formally, if $(rsp(m_{ij}) <_H inv(m_{xy})) \implies (m_{ij} \prec_H^{MR} m_{xy})$. Following [5], if transaction T_i terminates (either commits or aborts) before beginning of T_j then T_i and T_j follows transaction real-time order (or TR). Formally, if $(Term(T_i) <_H \textit{STM\_begin}_j()) \implies (T_i \prec_H^{TR} T_j)$. If all the transactions of a history H follow the real-time order, i.e. transactions are atomic then such history is known as *serial history* [5] or *t-sequential* [36]. Formally, $\langle (H \text{ is serial}) \implies (\forall T_i \in H.txns : (T_i \in Term(H)) \land (\forall T_i, T_j \in H.txns : (T_i \prec_H^{TR} T_j) \lor (T_j \prec_H^{TR} T_i)) \rangle$. In the serial history all the methods within a transaction are also ordered, it is also sequential.

Conflict order: We say that two transactions T_k, T_m are in conflict at RWSTMs, if T_k, T_m access same data-item and at least one of them performs write on it. The conflict order between T_k and T_m is denoted as $T_k \prec_H^{Conf} T_m$, if (1) $STM\_tryC_k() <_H STM\_tryC_m()$ and $wset(T_k) \cap wset(T_m) \neq \emptyset$; (2) $STM\_tryC_k() <_H r_m(x, v)$, $x \in wset(T_k)$ and $v \neq \mathscr{A}$; (3) $r_k(x, v) <_H STM\_tryC_m()$, $x \in wset(T_m)$ and $v \neq \mathscr{A}$.

We say that T_k, T_m are in conflict at OSTMs, denoted as $T_k \prec_H^{Conf} T_m$, if (1) $rv_k(x, v) <_H STM\_tryC_m()$, $x \in updSet(T_m)$ and $v \neq \mathscr{A}$; (2) $STM\_tryC_k() <_H rv_m(x, v)$, $x \in updSet(T_k)$ and $v \neq \mathscr{A}$; (3) $STM\_tryC_k() <_H STM\_tryC_m()$ and $(updSet(T_k) \cap updSet(T_m) \neq \emptyset)$. Thus, it can be seen that the conflict order is defined only on operations that have successfully executed. We denote the corresponding operations as conflicting.

2.1 Correctness Criteria for Databases

Serializability: It is one of the most popular correctness criteria of databases. The concurrent history H is correct if there exist an equivalent serial schedule. A history H is said to be *serializable* [5] if there exist a serial history S such that (1) S is equivalent to H where H consists of only committed transactions while removing all the aborted transactions of H and (2) each and every transaction of S is atomic, i.e., either all the operations of transaction T_i happen before all the operations of transaction T_j or vice versa. Serializability considers only committed transaction of H.

The commonly used notion of serializability in databases are *View Serializability (VSR)* [2, Chap. 3], *Multi-Version View Serializability (MVSR)* [2, Chap. 5], and *Conflict Serializability (CSR)* [2, Chap. 3].

Notions of Equivalence: Two histories H, H' are *view equivalent* [2, Chap. 3] or *VE* if (1) H, H' are legal histories and (2) H is equivalent to H'. By restricting to legal histories, view equivalence does not use multi-versions.

Two histories H and H' are *equivalent* if they have the same set of events. We say two histories H, H' are *multi-version view equivalent* [2, Chap. 5] or *MVVE* if (1) H, H' are valid histories and (2) H is equivalent to H'. *Multi-version view equivalence* uses the concept of multi-versions.

Two histories H, H' are *conflict equivalent* [2, Chap. 3] or *CE* if (1) H, H' are legal histories and (2) conflict in H, H' are the same, i.e., $conf(H) = conf(H')$. Conflict equivalence like view equivalence does not use multi-versions and restricts itself to legal histories.

VSR, MVSR, and CSR: A history H is said to VSR (or View Serializable) [2, Chap. 3], if there exist a serial history S such that S is view equivalent to H. But it maintains only one version corresponding to each t-object.

So, MVSR (or Multi-Version View Serializable) comes into picture when multiple version of each t-object is maintained. A history H is said to MVSR [2, Chap. 5], if there exist a serial history S such that S is multi-version view equivalent to H. It can be proved that verifying the membership of VSR as well as MVSR in databases is NP-Complete [5]. To circumvent this issue, researchers in databases have identified an efficient sub-class of VSR, called CSR based on the notion of conflicts. The membership of CSR can be verified in polynomial time using conflict graph characterization.

A history H is said to CSR (or Conflict Serializable) [2, Chap. 3], if there exists a serial history S such that S is conflict equivalent to H.

Strict Serializability: It is a subclass of serializability. It considers only committed transactions of history H while removing all the aborted transactions of H. A history H is said to be *strict serializable* [5] if there exist a serial history S such that (1) S respect the real-time as H i.e., $\prec_H^{RT} \subset \prec_S^{RT}$ and (2) H is view serializable.

2.2 Correctness Criteria for STMs

Sub-history: A *sub-history* (SH) of a history (H) denoted as the tuple $\langle SH.evts, <_{SH}\rangle$ and is defined as: (1) $<_{SH} \subseteq <_H$; (2) $SH.evts \subseteq H.evts$; (3) If an event of a transaction $T_k \in H.txns$ is in SH then all the events of T_k in H should also be in SH.

For a history H, let R be a subset of transactions of $H.txns$. Then $H.subhist(R)$ denotes the sub-history of H that is formed from the operations in R.

Valid and legal history: A successful read $r_k(x, v)$ (i.e., $v \neq \mathscr{A}$) in a history H is said to be *valid* if there exists a transaction T_j that writes v to x and *commits* before $r_k(x, v)$. Formally, $\langle r_k(x, v)$ is valid $\Leftrightarrow \exists T_j : (c_j <_H r_k(x, v)) \wedge (w_j(x, v) \in T_j.evts) \wedge (v \neq \mathscr{A})\rangle$. The history H is valid if all its successful read operations are valid.

We define $r_k(x, v)$'s *lastWrite* as the latest commit event c_i preceding $r_k(x, v)$ in H such that $x \in wset_i$ (T_i can also be T_0). A successful read operation $r_k(x, v)$, is said to be *legal* if the transaction containing r_k's lastWrite also writes v onto x: $\langle r_k(x, v)$ is legal $\Leftrightarrow$ $(v \neq \mathscr{A}) \wedge (H.lastWrite(r_k(x, v)) = c_i) \wedge (w_i(x, v) \in T_i.evts)\rangle$. The history H is legal if all its successful read operations are legal. From the definitions we get that if H is legal then it is also valid.

Opacity: We say that two histories H and H' are *equivalent* if they have the same set of events. Now a history H is said to be *opaque* [7] if it is valid and there exists a t-sequential legal history S such that (1) S is equivalent to $\overline{H}$ and (2) S respects $\prec_H^{RT}$, i.e., $\prec_H^{RT} \subset \prec_S^{RT}$. By requiring S being equivalent to $\overline{H}$, opacity treats all the incomplete transactions as aborted. We call S an (opaque) *serialization* of H.

Along same lines, a valid history H is said to be *strictly serializable* if $H.subhist(H.committed)$ is opaque. Unlike opacity, strict serializability does not include aborted or incomplete transactions in the global serialization order. An opaque history H is also strictly serializable: a serialization of *H.subhist(H.committed)* is simply the sub-sequence of a serialization of H that only contains transactions in $H.committed$.

Serializability is commonly used criterion in databases. But it is not suitable for STMs as it does not consider the correctness of *aborted* transactions as shown by Guerraoui & Kapalka [7]. Opacity, on the other hand, considers the correctness of *aborted* transactions as well. Similarly, local opacity (described below) is another correctness-criterion for STMs but is not as restrictive as opacity.

Co-Opacity: It is a well known correctness criteria of STMs that is polynomial time verifiable. Co-Opacity [9] is a subclass of opacity. A history H is said to be *co-opaque* [9] if there exists a t-sequential legal history S such that (1) S is equivalent to $\overline{H}$, i.e. $S.evts = \overline{H}.evts$. (2) S should be legal. (3) S respect the real-time as H i.e., $\prec_H^{RT} \subset \prec_S^{RT}$. (4) S preserves conflict-order of H, i.e. $\prec_S^{Conf} \subseteq \prec_H^{Conf}$.

Local opacity: For a history H, we define a set of sub-histories, denoted as $H.subhistSet$ as follows: (1) For each aborted transaction T_i, we consider a $subhist$ consisting of operations from all previously *committed* transactions and including all successful operations of T_i (i.e., operations which did not return $\mathscr{A}$) while immediately putting commit after last successful operation of T_i; (2) for last *committed* transaction T_l considers all the previously *committed* transactions including T_l.

A history H is said to be *locally-opaque* [8, 34] if all the sub-histories in $H.subhistSet$ are opaque. It must be seen that in the construction of sub-history of an aborted transaction T_i, the $subhist$ will contain operations from only one aborted transaction which is T_i itself and no other live/aborted transactions. Similarly, the sub-history of *committed* transaction T_l has no operations of aborted and live transactions. Thus in local opacity, no aborted or live transaction can cause another transaction to abort. It was shown that local opacity [8, 34] allows greater concurrency than opacity. Any history that is opaque is also locally-opaque but not necessarily the vice-versa. On the other hand, a history that is locally-opaque is also strict-serializable, but the vice-versa need not be true.

Linearizability: A linearizable [37] 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.

Lock Freedom: It is a non-blocking progress property in which if multiple threads are running for a sufficiently long time then at least one of the thread will always make progress. Lock-free [18] guarantees system-wide progress but individual threads may starve.

Chapter 3

Exploring Starvation-Freedom in Single-Version and Multi-Version RWSTMs

3.1 Introduction

Software Transactional Memory systems (STMs) [3, 4] have garnered significant interest as an elegant alternative for addressing synchronization and concurrency issues with multi-threaded programming in multi-core systems. Client programs use STMs by issuing transactions (a piece of code invoked by a thread). STMs often use an optimistic approach for concurrent execution of *transactions*. In optimistic execution, each transaction reads from the shared memory, but all write updates are performed on local memory. On completion, the STM system *validates* the reads and writes of the transaction. If any inconsistency is found, the transaction is *aborted*, and its local writes are discarded. Otherwise, the transaction is committed, and its local writes are transferred to the shared memory. Such STMs which work on read/write methods and maintain single-version corresponding to each *transactional-object or t-object* are called as *Single-Version Read-Write STMs (SV-RWSTMs or RWSTMs).*

A typical RWSTM system is a library which exports the following methods: (1) *STM_begin()*: begins a transaction T_i with unique timestamp i. (2) *STM_read(x) or (r(x))*: T_i reads a shared data-item or *transactional object* (*t-object*) x from shared memory. (3) *STM_write(x, v) or (w(x, v))*: T_i writes to a *t-object* x with value v into its local memory. (4) *STM_tryC()*: tries to commit the transaction T_i. On successful validation, the effect of the transaction T_i will be visible to the shared memory and T_i returns commit, otherwise, T_i returns abort using *STM_tryA$_i$()* as explained in Section 1.1.

Requirement of Progress of Transactions: A transaction aborted due to conflicts (two transactions are said to be in conflict, if both of them are accessing the same t-object x and at least one of the transaction performs a write on x) is typically re-issued with the expectation that it will complete successfully in a subsequent incarnation. There is a possibility that the transaction which a thread tries to execute gets aborted again and again. Every time, it executes the transaction, say T_i, T_i conflicts with some other transaction and hence gets aborted. In other words, the thread is effectively starved because it is not able to commit T_i successfully. However, many existing RWSTMs fail to provide *starvation freedom*, i.e., in these systems, it is possible that concurrency conflicts may prevent an incarnated transaction from committing. To address this issue, we develop an efficient RWSTM system which ensures *starvation-freedom* as a progress condition.

Starvation-freedom: An STM system is said to be *starvation-free* if a thread invoking a transaction T_i gets the opportunity to retry T_i on every abort (due to the presence of a fair underlying scheduler with bounded termination) and T_i is not *parasitic* [19], i.e., T_i will try to commit given a chance then T_i will eventually commit.

Wait-freedom is another interesting progress condition for STMs in which every transaction commits regardless of the nature of concurrent transactions and the underlying scheduler [18]. But it was shown by Guerraoui and Kapalka [19] that it is not possible to achieve *wait-freedom* in dynamic STMs in which data sets of transactions are not known in advance. So in this thesis, we explore the weaker progress condition *starvation-freedom* for transactional memories while assuming that the data sets of the transactions are *not* known in advance.

Related work on starvation-free RWSTMs: Starvation-freedom in RWSTMs has been explored by a few researchers in literature such as Gramoli et al. [20], Waliullah and Stenstrom [21], Spear et al. [22]. Most of these systems work by assigning priorities to transactions. In case of a conflict between two transactions, the transaction with lower priority is aborted. They ensure that every aborted transaction, on being retried a sufficient number of times, will eventually have the highest priority and hence will commit. We denote such an algorithm as *Single-Version Starvation-Free RWSTM* or *SF-SV-RWSTM*.

Although *SF-SV-RWSTM* guarantees starvation-freedom, it can still abort many transactions spuriously. Consider the case where a transaction T_i has the highest priority. Hence, as per *SF-SV-RWSTM*, T_i cannot be aborted. But if it is slow (for some reason), then it can cause several other conflicting transactions to abort and hence, bring down the efficiency and progress of the entire system. Figure 1.4 of Section 1.4 illustrates the limitation of SF-SV-RWSTMs with an example.

Motivation towards Starvation-Free Multi-Version RWSTMs (SF-MV-RWSTMs): A key limitation of single-version RWSTMs is limited concurrency. As shown above in Figure 1.4 of Section 1.4, it is possible that one long transaction conflicts with several transactions causing them to abort. This limitation can be overcome by using multi-version RWSTMs where we store multiple versions of the data-item (either unbounded versions with garbage collection, or bounded versions where the oldest version is replaced when the number of versions exceeds the bound).

Several multi-version RWSTMs have been proposed in the literature [10–13] that provide increased concurrency. But none of them provide starvation-freedom. To overcome this issue of SF-SV-RWSTM, we systematically develop a novel and efficient starvation free algorithm as *Starvation-free Multi-Version RWSTM (SF-MV-RWSTM)*. It maintains multiple versions corresponding to each *t-object* which reduces the number of aborts and which enhances the performance compared to *SF-SV-RWSTMs*. Figure 1.5 of SubSection 1.4.1 illustrates the benefits of SF-MV-RWSTMs over SF-SV-RWSTMs with an example. Thus multiple versions can help with starvation-freedom without sacrificing on concurrency. This motivated us to develop a starvation-free multi-version RWSTM system. Proposed SF-MV-RWSTM can be used with the case where the number of versions is unbounded and Garbage Collection (GC) is used to delete unwanted versions as SF-MV-RWSTM-GC.

Motivation towards Starvation-Free K-version RWSTMs (SF-K-RWSTMs): Although multi-version STMs provide greater concurrency, they suffer from the cost of garbage collection. One way to avoid this is to use bounded multi-version STMs, where the number of versions is bounded to be at most K. Thus, when $(K+1)^{th}$ version is created, the oldest version is removed. Achieving starvation-freedom while using only bounded K-versions is especially challenging given that a transaction may rely on the oldest version that is removed. In that case, it would be necessary to abort that transaction, making it harder to achieve starvation-freedom.

This chapter addresses this gap by developing a starvation-free algorithm for bounded MV-RWSTMs as SF-K-RWSTMs. Our approach is different from the approach used in *SF-SV-RWSTM* to provide starvation-freedom in single-version RWSTMs (the policy of aborting lower priority transactions in case of conflict) as it does not work for SF-K-RWSTMs.

Our experimental analysis shows that the proposed *SF-K-RWSTM* algorithm performs best among its variants (*SF-MV-RWSTM* and *SF-MV-RWSTM-GC*) along with starvation-free and non-starvation-free state-of-the-art STMs under long-running transactions with high contention. *SF-K-RWSTM* gives an average speedup on the *max-time* (maximum time for a transaction to commit) by a factor of 1.22, 1.89, 23.26 and 13.12 times over *PKTO*, *SF-SV-RWSTM*, NOrec [24] and ESTM [25] respectively for counter application. *SF-K-RWSTM* performs 1.5 and 1.44 times better than *PKTO* and *SF-SV-RWSTM* but 1.09 times worse than NOrec for low contention KMEANS application of STAMP [26] benchmark. On the other hand, *SF-K-RWSTM* performs 1.14, 1.4 and 2.63 times better than *PKTO*, *SF-SV-RWSTM* and NOrec [24] for LABYRINTH application of STAMP benchmark which has high contention with long-running transactions.

Roadmap: We explore the progress guarantees as starvation-freedom in single and multi-version RWSTMs. It includes the graph characterization of local opacity [8] in Section 3.2 which helps to prove the correctness of proposed algorithms. Section 3.3 introduces a novel, efficient, and starvation-free bounded-version RWSTM system as *Starvation-Free K-version RWSTM* or *SF-K-RWSTM* for a given parameter K. Here K is the number of versions of each t-object and can range from 1 to ∞. It includes design and data structures of proposed *SF-K-RWSTM* algorithm along with the detailed pseudocode. Section 3.4 and Section 3.5 explain the liveness and safety proof of proposed *SF-K-RWSTM* algorithm. Section 3.6 describes the experimental analysis of proposed *SF-K-RWSTM* with state-of-the-art STMs followed by summary of this chapter in Section 3.7.

3.2 Graph Characterization of Local Opacity

To prove correctness of RWSTM systems, it is useful to consider graph characterization of histories. In this section, we describe the graph characterization developed by Kumar et al. [10] for proving opacity which is based on characterization by Bernstein and Goodman [38]. We extend this characterization for local opacity (or LO).

Consider a history H which consists of multiple versions for each t-object. The graph characterization uses the notion of *version order*. Given H and a t-object x, we define a version order for x as any (non-reflexive) total order on all the versions of x 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 version of x 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 $H2$ with: $r_1(x,0)r_2(x,0)r_1(y,0)r_3(z,0)w_1(x,5)w_3(y,15)w_2(y,10)$ $w_1(z,10)c_1c_2r_4(x,5)r_4(y,10)w_3(z,15)c_3r_4(z,10)$. Using the notation that a committed transaction T_i writing to x creates a version x_i, a possible version order for $H2$ $\ll_{H2}$ is: $\langle x_0 \ll x_1\rangle, \langle y_0 \ll y_2 \ll y_3\rangle, \langle z_0 \ll z_1 \ll z_3\rangle$.

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. *real-time*(real-time) edges: If T_i commits before T_j starts in H, then there is an edge from v_i to v_j. This set of edges are referred to as $rt(H)$.

2. *rf*(reads-from) edges: If T_j reads x from T_i in H, then there is an edge from v_i to v_j. Note that in order for this to happen, T_i must have committed before T_j and $c_i <_H r_j(x)$. This set of edges are referred to as $rf(H)$.

3. *mv*(multiversion) edges: The mv edges capture the multiversion relations and is based on the version order. Consider a successful read operation $r_k(x,v)$ and the write operation $w_j(x,v)$ belonging to transaction T_j such that $r_k(x,v)$ reads x from $w_j(x,v)$ (it must be noted T_j is a committed transaction and $c_j <_H r_k$). Consider a committed transaction T_i which writes to x, $w_i(x,u)$ where $u \neq v$. Thus the versions created x_i, x_j are related by

$\ll$. Then, if $x_i \ll x_j$ we add an edge from v_i to v_j. Otherwise ($x_j \ll x_i$), we add an edge from v_k to v_i. This set of edges are referred to as $mv(H, \ll)$.

Using the construction, the $OPG(H2, \ll_{H2})$ for history $H2$ and $\ll_{H2}$ is shown in Figure 3.1. The edges are annotated. The only mv edge from $T4$ to $T3$ is because of t-objects y, z. $T4$ reads value 5 for z from $T1$ whereas $T3$ also writes 15 to z and commits before $r_4(z)$.

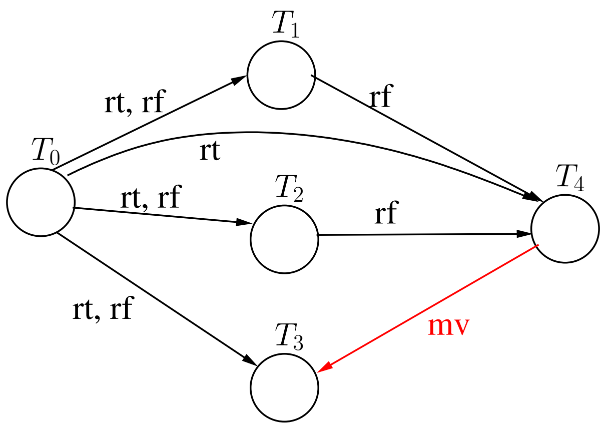


Figure 3.1: $OPG(H2, \ll_{H2})$

Kumar et al. [10] showed that if a version order $\ll$ exists for a history H such that $OPG(H, \ll_H)$ is acyclic, then H is opaque. This is captured in the following result.

Result 1 *A valid history H is opaque iff there exists a version order $\ll_H$ such that $OPG(H, \ll_H)$ is acyclic.*

This result can be easily extended to prove LO as follows:

Theorem 2 *A valid history H is locally-opaque iff for each sub-history sh in $H.subhistSet$ there exists a version order $\ll_{sh}$ such that $OPG(sh, \ll_{sh})$ is acyclic.*
Formally, $\langle (H \text{ is locally-opaque}) \Leftrightarrow (\forall sh \in H.subhistSet, \exists \ll_{sh}: OPG(sh, \ll_{sh}) \text{ is acyclic}) \rangle$.

Proof. To prove this theorem, we have to show that each sub-history sh in $H.subhistSet$ is valid. Then the rest follows from Result 1. Now consider a sub-history sh. Consider any read operation $r_i(x, v)$ of a transaction T_i. It is clear that T_i must have read a version of x created by a previously committed transaction. From the construction of sh, we get that all the transaction that committed before r_i are also in sh. Hence sh is also valid.

Now, proving sh to be opaque iff there exists a version order $\ll_{sh}$ such that $OPG(sh, \ll_{sh})$ is acyclic follows from Result 1. Hence, valid history H is locally-opaque iff $OPG(sh, \ll_{sh})$ is acyclic.

3.3 The Working of SF-K-RWSTM Algorithm

In this section, we start with the definition of *starvation-freedom*. Then we describe the invocation of transactions by the application. Next, we describe the data structures used by the algorithms.

Here, we propose *K-version Starvation-Free RWSTM* or *SF-K-RWSTM* for a given parameter K. Here K is the number of versions of each t-object and can range from 1 to ∞. When K is 1, it boils down to single-version *starvation-free* RWSTM. If K is ∞, then *SF-K-RWSTM* uses unbounded versions and needs a separate garbage collection mechanism to delete old versions like other MV-RWSTMs proposed in the literature [10, 11]. We denote *SF-K-RWSTM* using unbounded versions as *SF-UV-RWSTM* and *SF-UV-RWSTM* with garbage collection as *SF-UV-RWSTM-GC*.

Next, we describe some *starvation-freedom* preliminaries in SubSection 3.3.1 to explain the working of *SF-K-RWSTM* algorithm. To explain the intuition behind the *SF-K-RWSTM* algorithm, we start with the modification of MVTO [10,38] algorithm in SubSection 3.3.3. We then make a sequence of modifications to it to arrive at *SF-K-RWSTM* algorithm in SubSection 3.3.6.

3.3.1 Starvation-Freedom Explanation

In this subsection, we described the definition of *starvation-freedom* along with the assumptions that helps to achieve starvation-freedom.

Definition 1 ***Starvation-Freedom:*** *A STM system is said to be starvation-free if a thread invoking a non-parasitic transaction T_i gets the opportunity to retry T_i on every abort, due to the presence of a fair scheduler, then T_i will eventually commit.*

As explained by Herlihy & Shavit [18], a fair scheduler implies that no thread is forever delayed or crashed. Hence with a fair scheduler, we get that if a thread acquires locks then it will eventually release the locks. Thus a thread cannot block out other threads from progressing.

Assumption about Scheduler: In order for starvation-free algorithm *SF-K-RWSTM* (described in SubSection 3.3.6) to work correctly, we make the following assumption about the fair scheduler:

Assumption 1 ***Bounded-Termination****: For any transaction T_i, invoked by a thread Th_x, the fair system scheduler ensures, in the absence of deadlocks, Th_x is given sufficient time on a CPU (and memory etc.) such that T_i terminates (either commits or aborts) in bounded time.*

While the bound for each transaction may be different, we use L to denote the maximum bound. In other words, in time L, every transaction will either abort or commit due to the absence of deadlocks.

There are different ways to satisfy the scheduler requirement. For example, a round-robin scheduler which provides each thread equal amount of time in any window satisfies this requirement as long as the number of threads is bounded. In a system with two threads, even if a scheduler provides one thread 1% of CPU and another thread 99% of the CPU, it satisfies the above requirement. On the other hand, a scheduler that schedules the threads as '$T_1, T_2, T_1, T_2, T_2, T_1, T_2, T_2, T_2, T_2, T_1, T_2, T_2, T_2, T_2, T_2, T_2, T_2, T_2, T_1, T_2(16times)$' does not satisfy the above requirement. This is due to the fact that over time, thread 1 gets infinitesimally smaller portion of the CPU and, hence, the time required for it to complete (commit or abort) will continue to increase over time.

In our algorithm, we will ensure that it is deadlock free using standard techniques from the literature. In other words, each thread is in a position to make progress. We assume that the scheduler provides sufficient CPU time to complete (either commit or abort) within a bounded time.

3.3.2 Algorithm Preliminaries

In this subsection, we describe the invocation of transactions by the application. Next, we describe the data structures used by the algorithms.

Transaction Invocation: Transactions are invoked by threads. Suppose a thread Th_x invokes a transaction T_i. If this transaction T_i gets *aborted*, Th_x will reissue it, as a new incarnation of T_i, say T_j. The thread Th_x will continue to invoke new incarnations of T_i until an incarnation commits.

When the thread Th_x invokes a transaction, say T_i, for the first time then the STM system assigns T_i a unique timestamp called *current timestamp or cts*. If it aborts and retries again as T_j, then its *cts* will change. However, in this case, the thread Th_x will also pass the *cts* value of the first incarnation (T_i) to T_j. By this, Th_x informs the STM system that, T_j is not a new invocation but is an incarnation of T_i.

We denote the *cts* of T_i (first incarnation) as *Initial Timestamp or its* for all the incarnations of T_i. Thus, the invoking thread Th_x passes cts_i to all the incarnations of T_i (including T_j). Thus for T_j, $its_j = cts_i$. The transaction T_j is associated with the timestamps: $\langle its_j, cts_j \rangle$. For T_i, which is the initial incarnation, its *its* and *cts* are the same, i.e., $its_i = cts_i$. For simplicity, we use the notation that for transaction T_j, j is its *cts*, i.e., $cts_j = j$.
We now state our assumptions about transactions in the system.

Assumption 2 *We assume that in the absence of other concurrent conflicting transactions, every transaction will commit. In other words, (a) if a transaction T_i is executing in a system where other concurrent conflicting transactions are not present then T_i will not self-abort. (b) Transactions are not parasitic (explained in Section 3.1) [19].*

If transactions self-abort or behave in parasitic manner then providing starvation-freedom is impossible.

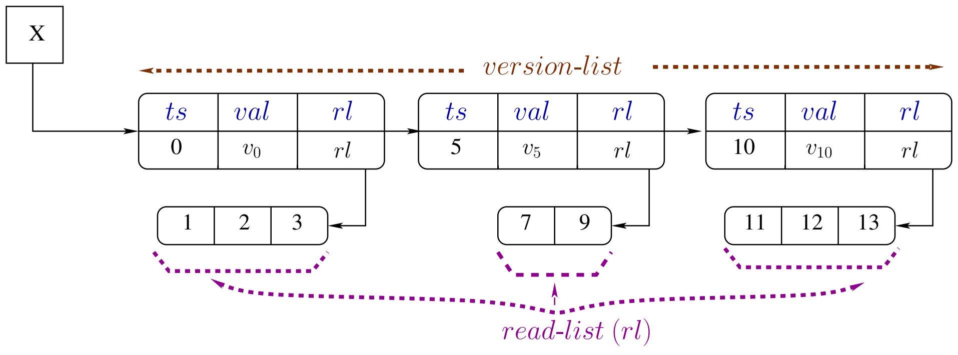


Figure 3.2: Data Structures for Maintaining Versions

Common Data Structures and STM Methods: Here, we describe the common data structures used by all the algorithms proposed in this chapter.

In all our algorithms, for each t-object, the algorithms maintain multiple versions in form of *version-list* (or vlist). Similar to MVTO [10], each version of a t-object is a tuple denoted as *vTuple* and consists of three fields: (1) timestamp characterizing the transaction that created the version, (2) value, and (3) a list, *read-list* (or rl) consisting of transaction ids (or *cts* of transactions) that read from this version.

Figure 3.2 illustrates this structure. For a t-object x, we use the notation $x[t]$ to access the version with timestamp t. Depending on the algorithm considered, the fields of this structure change.

We assume that the STM system exports the following methods for a transaction T_i: (1) $STM\_begin(t)$ where t is provided by the invoking thread, Th_x. From our earlier assumption, it is the *cts* of the first incarnation or $null$ if Th_x is invoking this transaction for the first time. This method returns a unique timestamp to Th_x which is the *cts*/id of the transaction. (2) $STM\_read_i(x)$ tries to read t-object x. It returns either value v or $\mathscr{A}$. (3) $STM\_write_i(x, v)$ operation that updates a t-object x with value v locally. It returns ok. (4) $STM\_tryC_i()$ tries to commit the transaction and returns $\mathscr{C}$ if it succeeds. Otherwise, it returns $\mathscr{A}$.
Correctness Criteria: For ease of exposition, we initially consider strict-serializability as *correctness-criterion* to illustrate the correctness of the algorithms. Subsequently, we consider a stronger property, local opacity that is more suitable for STMs.

3.3.3 Priority-based MVTO Algorithm

In this subsection, we describe a modification to the Multi-Version Timestamp Ordering (MVTO) algorithm [10, 38] to ensure that it provides preference to transactions that have low *its*, i.e., transactions that have been in the system for a longer time. We denote the basic algorithm which maintains unbounded versions as *Priority-based MVTO* or *PMVTO* (akin to the original MVTO). We denote the variant of *PMVTO* that maintains K versions as *PKTO* and the unbounded versions variant with garbage collection as *PMVTO-GC*.

While providing higher priority to older transactions suffices to provide starvation-freedom in *SF-SV-RWSTM*, we note that *PKTO* is not starvation free. The reason that demonstrates why *PKTO* is not starvation free forms our basis of designing SF-MV-TO that provides starvation-freedom (described in SubSection 3.3.5).

We now describe *PKTO*. This description can be trivially extended to *PMVTO* and *PMVTO-GC* as well.

$STM\_begin(t)$**:** A unique timestamp ts is allocated to T_i which is its *cts* (i from our assumption). The timestamp ts is generated by atomically incrementing the global counter G_tCntr. If the input t is null, then $cts_i = its_i = ts$ as this is the first incarnation of this transaction. Otherwise, the non-null value of t is assigned as its_i.

$STM\_read(x)$**:** The timestamp of transaction T_i is denoted by i. T_i reads from a version of x in the shared memory (if x does not exist in T_i's local buffer) with timestamp j such that j is the largest timestamp less than i (among the versions of x), i.e., there exists no version of x with timestamp k such that $j < k < i$. After reading this version of x, T_i is stored in $x[j]$'s read-list. If no such version exists then T_i is *aborted*.

$STM\_write(x, v)$**:** T_i stores this write to value x locally in its $wset_i$. If T_i ever reads x again, this value will be returned.

$STM\_tryC()$: This operation consists of three steps. In Step 1, it checks whether T_i can be *committed*. In Step 2, it performs the necessary tasks to mark T_i as a *committed* transaction and in Step 3, T_i returns commit.

1. Before T_i can commit, it needs to verify that any version it creates does not violate consistency. Suppose T_i creates a new version of x with timestamp i. Let j be the largest timestamp smaller than i for which version of x exists. Let this version be $x[j]$. Now, T_i needs to make sure that any transaction that has read $x[j]$ is not affected by the new version created by T_i. There are two possibilities of concern:

    (a) Let T_k be some transaction that has read $x[j]$ and $k > i$ (k = *cts* of T_k). In this scenario, the value read by T_k would be incorrect (w.r.t strict-serializability) if T_i is allowed to create a new version. In this case, we say that the transactions T_i and T_k are in *conflict*. So, we do the following: (i) if T_k has already *committed* then T_i is *aborted*; (ii) Suppose T_k is live and its_k is less than its_i. Then again T_i is *aborted*; (iii) If T_k is still live with its_i less than its_k then T_k is *aborted*.

    (b) The previous version $x[j]$ does not exist. This happens when the previous version $x[j]$ has been overwritten due to a limited number K, of versions (see below). In this case, T_i is *aborted* since *PKTO* does not know if T_i conflicts with any other transaction T_k that has read the previous version.

2. After Step 1, we have verified that it is ok for T_i to commit. Now, we have to create a version of each t-object x in the $wset$ of T_i. This is achieved as follows:
   (a) T_i creates a $vTuple$ $\langle i, wset_i.x.v, null\rangle$. In this tuple, i (*cts* of T_i) is the timestamp of the new version; $wset_i.x.v$ is the value of x is in T_i's $wset$, and the read-list of the $vTuple$ is $null$.
   (b) If the total number of versions already existing for x is K. Then among all the versions of x, T_i replaces the version with the smallest timestamp with $vTuple$ $\langle i, wset_i.x.v, null\rangle$. Otherwise, the $vTuple$ is added to x's *vlist* at its end.
3. Transaction T_i is then *committed*.

The algorithm described here is only the main idea of *PKTO*. The actual implementation will use locks to ensure that each of these methods are linearizable [37]. It can be seen that *PKTO* gives preference to the transaction having lower *its* in Step 1a. Transactions having lower *its* have been in the system for a longer time. Hence, *PKTO* gives preference to them.

Correctness of *PKTO*: We have the following property on the correctness of *PKTO*.

Property 3 *Any history generated by PKTO is strict-serializable.*

Consider a history H generated by *PKTO*. Let the *committed* sub-history of H be $CSH = H.subhist(H.committed)$. It can be shown that CSH is opaque with the equivalent serialized history SH' is one in which all the transactions of CSH are ordered by their *cts*. Hence, H is strict-serializable.

While *PKTO* (and *PMVTO*) satisfies strict-serializability, it fails to prevent starvation. The key reason is that if transaction T_j conflicts with T_k and T_k has already committed, then T_j must be aborted. This is true even if T_j is the oldest transaction in the system. Furthermore, next incarnation of T_j may have to be aborted by another transaction T'_k. This cannot be prevented as conflict between T_j and T'_k may not be detected before T'_k has committed.

Possibility of Starvation in *PKTO*: As discussed above, *PKTO* gives priority to transactions having lower *its*. But a transaction T_i having the lowest *its* could still abort due to one of the following reasons: (1) Upon executing $STM\_read(x)$ method if it does not find any other version of x to read from. This can happen if all the versions of x present have a timestamp greater than cts_i. (2) While executing Step 1a(i), of the $STM\_tryC()$ method, if T_i wishes to create a version of x with timestamp i, but some other transaction, say T_k has read from a version with timestamp j and $j < i < k$. In this case, T_i has to abort if T_k has already *committed*. This issue is not restricted only to *PKTO*. It can occur in *PMVTO* and *PMVTO-GC* as well.

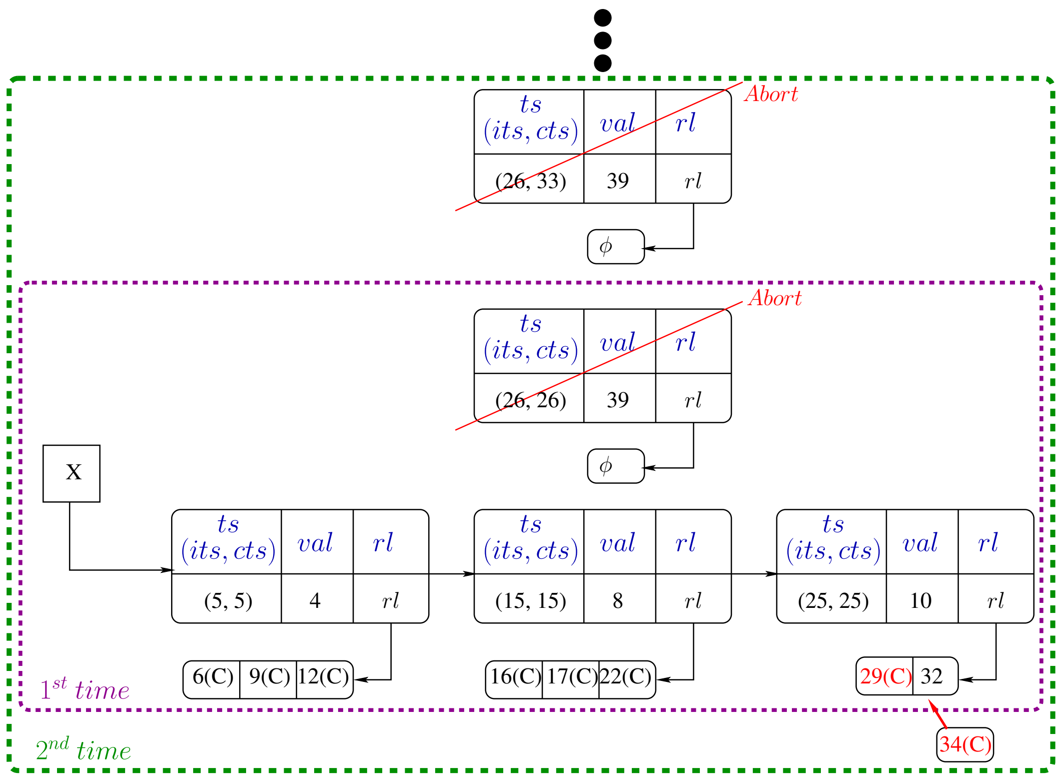


Figure 3.3: Pictorial representation of execution under *PKTO*

We illustrate this problem in *PKTO* with Figure 3.3. Here transaction T_{26}, with *its* 26 is the lowest among all the live transactions, starves due to Step 1a.(i) of the $STM\_tryC()$. *First time*, T_{26} gets *aborted* due to higher timestamp transaction T_{29} in the read-list of $x[25]$ has *committed*. We have denoted it by a '(C)' next to the version. The *second time*, T_{26} retries with same *its* 26 but new *cts* 33. Now when T_{33} comes for commit, suppose another transaction T_{34} in the read-list of $x[25]$ has already *committed*. So this will cause T_{33} (another incarnation of T_{26}) to abort again. Such scenario can possibly repeat again and again and thus causing no incarnation of T_{26} to ever commit leading to its starvation.

Garbage Collection in *SF-UV-RWSTM-GC* and *PMVTO-GC*: Having multiple versions to increase the performance and to decrease the number of aborts, leads to creating too many versions which are not of any use and occupies space. So, such garbage versions need to be taken care of. Hence we come up with a garbage collection over these unwanted versions. This technique helps to conserve memory space and increases the performance in turn as no more unnecessary traversing of garbage versions by transactions is necessary. We have used a global, i.e., across all transactions a list that keeps track of all the live transactions in the system. We call this list as *live-list*. Each transaction at the beginning of its life cycle creates its entry in this *live-list*. Under the optimistic approach of STM, each transaction in the shared memory performs its updates in the $STM\_tryC()$. In this phase, each transaction performs some validations, and if all the validations are successful then the transaction make changes or in simple terms creates versions of the corresponding t-object in the shared memory. While creating a version every transaction, check if it is the least timestamp live transaction present in the system by using *live-list* data structure, if yes then the current transaction deletes all the version of that t-object and create one of its own. Else the transaction does not do any garbage collection or delete any version and look for creating a new version of next t-object in the write set, if at all. Experimental analysis shows that both ***SF-UV-RWSTM-GC*** and ***PMVTO-GC*** perform better than ***SF-UV-RWSTM*** and ***PMVTO*** respectively across all workloads.

3.3.4 Data structure and Detailed Pseudocode of PKTO

This subsection describes the data structures in detail and gives the pseudocode of ***PKTO***. We start with data structures that are local to each transaction. For each transaction T_i:

- $rset_i$(read-set): It is a list of data tuples (d_tuples) of the form $\langle x, val \rangle$, where x is the t-object and val is the value read by the transaction T_i. We refer to a tuple in T_i's read-set by $rset_i[x]$.
- $wset_i$(write-set): It is a list of (d_tuples) of the form $\langle x, val \rangle$, where x is the t-object to which transaction T_i writes the value val. Similarly, we refer to a tuple in T_i's write-set by $wset_i[x]$.

In addition to these local structures, the following shared global structures are maintained that are shared across transactions (and hence, threads). All shared variables start with 'G'.

- G_tCntr (counter): This a numerical valued counter that is incremented when a transaction begins.

For each transaction T_i we maintain the following shared timestamps:

- G_lock_i: A lock for accessing all the shared variables of T_i.
- G_its_i (initial timestamp): It is a timestamp assigned to T_i when it was invoked for the first time.
- G_cts_i (current timestamp): It is a timestamp when T_i is invoked again at a later time. When T_i is created for the first time, then its G_cts is same as its its.
- G_valid_i: This is a boolean variable which is initially true (T). If it becomes false (F) then T_i has to be aborted.
- G_state_i: This is a variable which states the current value of T_i. It has three states: `live`, `commit` or `abort`.

For each data item x in history H, we maintain:

- $x.val$ (value): It is the successful previous closest value written by any transaction.
- $x.rl$ (readList): It is the read list consists of all the transactions that have read x.

Algorithm 2 $STM\_init()$: Invoked at the start of the STM system and initializes all the t-objects used by the STM system.

```
1: $G\_tCntr$ = 1;
2: **for all** $x$ in $\mathscr{T}$ **do** /*All the t-objects used by the STM System*/
3:     add $\langle 0, 0, nil\rangle$ to $x.\texttt{vl}$; /*$T_0$ is initializing $x$*/
4: **end for**;
```

Algorithm 3 $STM\_begin(its)$: Invoked by a thread to start a new transaction T_i. Thread can pass a parameter its which is the initial timestamp when this transaction was invoked for the first time. If this is the first invocation then its is nil. It returns the tuple $\langle id, G\_cts\rangle$.

```
5: $i$ = unique-id; /*An unique id to identify this transaction. It could be same as G_cts */
6: /*Initialize transaction specific local and global variables*/
7: **if** ($its == nil$) **then**
8:     /*$G\_tCntr.get\&Inc()$ returns the current value of G_tCntr and atomically increments
       it*/
9:     $G\_its_i = G\_cts_i = G\_tCntr.get\&Inc()$;
10: **else**
11:     $G\_its_i = its$;
12:     $G\_cts_i = G\_tCntr.get\&Inc()$;
13: **end if**
14: $rset_i = wset_i = null$;
15: $G\_state_i$ = live;
16: $G\_valid_i = T$;
17: return $\langle i, G\_cts_i\rangle$
```

Algorithm 4 $STM\_read(i, x)$: Invoked by a transaction T_i to read t-object x. It returns either the value of x or $\mathscr{A}$.

```
18: **if** ($x \in rset_i$) **then** /*Check if the t-object $x$ is in $rset_i$*/
19:     return $rset_i[x].val$;
20: **else if** ($x \in wset_i$) **then** /*Check if the t-object $x$ is in $wset_i$*/
21:     return $wset_i[x].val$;
```

```
22: else/*t-object $x$ is not in $rset_i$ and $wset_i$*/
23:     lock $x$; lock $G\_lock_i$;
24:     if ($G\_valid_i == F$) then return abort(i);
25:     end if
26:     /*findLTS: From $x$.vl, returns the largest ts value less than $G\_cts_i$. If no such version
        exists, it returns $nil$ */
27:     $curVer = findLTS(G\_cts_i, x)$;
28:     if ($curVer == nil$) then return abort(i); /*Proceed only if $curVer$ is not nil*/
29:     end if
30:     $val = x[curVer].v$; add $\langle x, val \rangle$ to $rset_i$;
31:     add $T_i$ to $x[curVer].rl$;
32:     unlock $G\_lock_i$; unlock $x$;
33:     return $val$;
34: end if
```

Algorithm 5 $STM\_write(x, val)$: A Transaction T_i writes into local memory.

```
35: Append the $d\_tuple\langle x, val \rangle$ to $wset_i$.
36: return $ok$;
```

Algorithm 6 $STM\_tryC()$: Returns ok on commit else return Abort.

```
37: /*The following check is an optimization which needs to be performed again later*/
38: lock $G\_lock_i$;
39: if ($G\_valid_i == F$) then
40:     return abort(i);
41: end if
42: unlock $G\_lock_i$;
43: $largeRL = allRL = nil$; /*Initialize larger read list (largeRL), all read list (allRL) to nil*/
44: for all $x \in wset_i$ do
45:     lock $x$ in pre-defined order;
46:     /*findLTS: returns the version with the largest ts value less than $G\_cts_i$. If no such
        version exists, it returns $nil$.*/
47:     $prevVer = findLTS(G\_cts_i, x)$; /*prevVer: largest version smaller than $G\_cts_i$*/
48:     if ($prevVer == nil$) then /*There exists no version with ts value less than $G\_cts_i$*/
49:         lock $G\_lock_i$; return abort(i);
50:     end if
51:     /*getLar: obtain the list of reading transactions of $x[prevVer].rl$ whose $G\_cts$ is
        greater than $G\_cts_i$*/
```

```
52:     $largeRL = largeRL \cup getLar(G\_cts_i, x[prevVer].rl)$;
53: **end for**/*$x \in wset_i$*/
54: $relLL = largeRL \cup T_i$; /*Initialize relevant Lock List (relLL)*/
55: **for all** $(T_k \in relLL)$ **do**
56:     lock $G\_lock_k$ in pre-defined order; /*Note: Since $T_i$ is also in $relLL$, $G\_lock_i$ is also
        locked*/
57: **end for**
58: /*Verify if $G\_valid_i$ is false*/
59: **if** $(G\_valid_i == F)$ **then**
60:     return abort(i);
61: **end if**
62: $abortRL = nil$ /*Initialize abort read list (abortRL)*/
63: /*Among the transactions in $T_k$ in $largeRL$, either $T_k$ or $T_i$ has to be aborted*/
64: **for all** $(T_k \in largeRL)$ **do**
65:     **if** $(isAborted(T_k))$ **then** /*Transaction $T_k$ can be ignored since it is already aborted or
        about to be aborted*/
66:         continue;
67:     **end if**
68:     **if** $(G\_its_i < G\_its_k) \wedge (G\_state_k == \texttt{live})$ **then**
69:         /*Transaction $T_k$ has lower priority and is not yet committed. So it needs to be
            aborted*/
70:         $abortRL = abortRL \cup T_k$; /*Store $T_k$ in abortRL */
71:     **else**/*Transaction $T_i$ has to be aborted*/
72:         return abort(i);
73:     **end if**
74: **end for**
75: /*Store the current value of the global counter as commit time and increment it*/
76: $comTime = G\_tCntr.get\&Inc()$;
77: **for all** $T_k \in abortRL$ **do** /*Abort all the transactions in abortRL */
78:     $G\_valid_k = F$;
79: **end for**
80: /*Having completed all the checks, $T_i$ can be committed*/
81: **for all** $(x \in wset_i)$ **do**
82:     $newTuple = \langle G\_cts_i, wset_i[x].val, nil \rangle$; /*Create new v_tuple: G_cts, val, `rl` for $x$*/
83:     **if** $(|x.vl| > k)$ **then**
84:         replace the oldest tuple in $x.$`vl` with $newTuple$; /*$x.$`vl` is ordered by timestamp*/
85:     **else**
86:         add a $newTuple$ to $x.vl$ in sorted order;
87:     **end if**
```

```
88: end for/*$x \in wset_i$*/
89: $G\_state_i$ = commit;
90: unlock all variables;
91: return $\mathscr{C}$;
```

Algorithm 7 $isAborted(T_k)$: Verifies if T_i is already aborted or its G_valid flag is set to false implying that T_i will be aborted soon.

```
92: if $(G\_valid_k == F) \vee (G\_state_k == \texttt{abort}) \vee (T_k \in abortRL)$ then
93:     return $T$;
94: else
95:     return $F$;
96: end if
```

Algorithm 8 $abort(i)$: Invoked by various STM methods to abort transaction T_i and returns $\mathscr{A}$.

```
97: $G\_valid_i = F$; $G\_state_i$ = abort;
98: unlock all variables locked by $T_i$;
99: return $\mathscr{A}$;
```

3.3.5 Modifying PKTO to Obtain SF-K-TO: Trading the Correctness for Starvation-Freedom

Our goal is to revise *PKTO* algorithm to ensure that *starvation-freedom* is satisfied. Specifically, we want the transaction with the lowest *its* to eventually commit. Once this happens, the next non-committed transaction with the lowest *its* will commit. Thus, from induction, we can see that every transaction will eventually commit.

Key Insights For Eliminating Starvation in *PKTO*: To identify the necessary revision, we first focus on the effect of this algorithm on two transactions, say T_{50} and T_{60} with their *cts* values being 50 and 60 respectively. Furthermore, for the sake of discussion, assume that these transactions only read and write t-object x. Also, assume that the latest version for x is with ts 40. Each transaction first reads x and then writes x (as part of the $STM\_tryC()$ operation). We use r_{50} and r_{60} to denote their read operations while w_{50} and w_{60} to denote their $STM\_tryC()$ operations. Here, a read operation will not fail as there is a previous version present.

Now, there are six possible permutations of these statements. We identify these permutations and the action that should be taken for that permutation in Table 3.1. In all these permutations, the read operations of a transaction come before the write operations as the writes to the shared memory occurs only in the $STM\_tryC()$ operation (due to optimistic execution) which is the final operation of a transaction.

S. No	Sequence	Action
1.	$r_{50}, w_{50}, r_{60}, w_{60}$	T_{60} reads the version written by T_{50}. No conflict.
2.	$r_{50}, r_{60}, w_{50}, w_{60}$	Conflict detected at w_{50}. Either abort T_{50} or T_{60}.
3.	$r_{50}, r_{60}, w_{60}, w_{50}$	Conflict detected at w_{50}. Hence, abort T_{50}.
4.	$r_{60}, r_{50}, w_{60}, w_{50}$	Conflict detected at w_{50}. Hence, abort T_{50}.
5.	$r_{60}, r_{50}, w_{50}, w_{60}$	Conflict detected at w_{50}. Either abort T_{50} or T_{60}.
6.	$r_{60}, w_{60}, r_{50}, w_{50}$	Conflict detected at w_{50}. Hence, abort T_{50}.

Table 3.1: Permutations of operations

From this table, it can be seen that when a conflict is detected, in some cases, algorithm *PKTO must* abort T_{50}. In case both the transactions are live, *PKTO* has the option of aborting either transaction depending on their *its*. If T_{60} has lower *its* then in no case, *PKTO* is required to abort T_{60}. In other words, it is possible to ensure that the transaction with lowest *its* and the highest *cts* is never aborted. Although in this example, we considered only one t-object, this logic can be extended to cases having multiple operations and t-objects.

Next, consider Step 1b of *PKTO* algorithm from SubSection 3.3.3. Suppose a transaction T_i wants to read a t-object but does not find a version with a timestamp smaller than i. In this case, T_i has to abort. But if T_i has the highest *cts*, then it will certainly find a version to read from. This is because the timestamp of a version corresponds to the timestamp of the transaction that created it. If T_i has the highest *cts* value then it implies that all versions of all the t-objects have a timestamp smaller than *cts* of T_i. This reinforces the above observation that a transaction with lowest *its* and highest *cts* is not aborted.

To summarize the discussion, algorithm *PKTO* has an in-built mechanism to protect transactions with lowest *its* and highest *cts* value. However, this is different from what we need. Specifically, we want to protect a transaction T_i, with lowest *its* value. One way to ensure this: if transaction T_i with lowest *its* keeps getting aborted, eventually it will achieve the highest *cts*. Once this happens, *PKTO* ensures that T_i cannot be further aborted. In this way, we can ensure the liveness of all transactions.

The working of *starvation-free* algorithm: To realize this idea and achieve *starvation-freedom*, we consider another variation of MVTO, *Starvation-Free MVTO* or *SF-MV-TO*. We specifically consider SF-MV-TO with K versions, denoted as *SF-K-TO*.

A transaction T_i instead of using the current time as cts_i, uses a potentially higher timestamp, *Working Timestamp - wts* or wts_i. Specifically, it adds $C * (cts_i - its_i)$ to cts_i, i.e.,

$$wts_i = cts_i + C * (cts_i - its_i); \tag{3.1}$$

where, C is any constant greater than 0. In other words, when the transaction T_i is issued for the first time, wts_i is same as $cts_i (= its_i)$. However, as transaction keeps getting aborted, the drift between cts_i and wts_i increases. The value of wts_i increases with each retry.

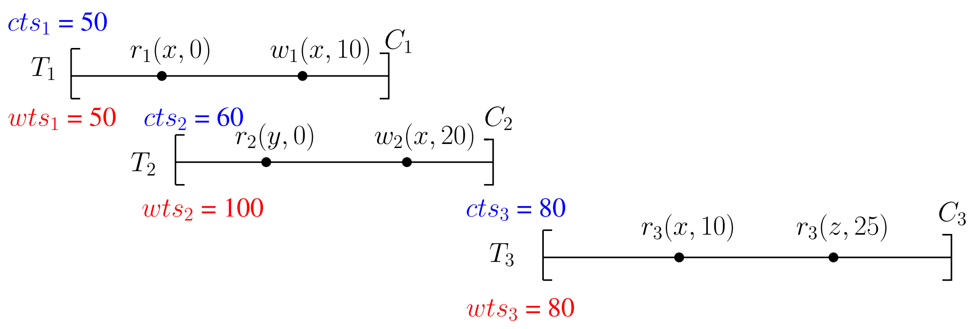


Figure 3.4: Correctness of SF-K-TO Algorithm

Furthermore, in SF-K-TO algorithm, *cts* is replaced with *wts* for $STM\_read()$, $STM\_write()$ and $STM\_tryC()$ operations of *PKTO* defined in SubSection 3.3.3. In SF-K-TO, a transaction T_i uses wts_i to read a version in $STM\_read()$. Similarly, T_i uses wts_i in $STM\_tryC()$ to find the appropriate previous version (in Step 1b of SubSection 3.3.3) and to verify if T_i has to be aborted (in Step 1a of SubSection 3.3.3). Along the same lines, once T_i decides to commit and create new versions of x, the timestamp of x will be same as its wts_i (in Step 3 of SubSection 3.3.3). Thus the timestamp of all the versions in *vlist* will be *wts* of the transactions that created them.
Now, we have the following property of the SF-K-TO algorithm.

Property 4 *SF-K-TO algorithm ensures starvation-freedom.*

The proof of this property is somewhat involved, the key idea is that the transaction with lowest *its* value, say T_{low}, will eventually have highest *wts* value than all the other transactions in the system. Moreover, after a certain duration, any *new* transaction arriving in the system (i.e., whose *its* value sufficiently higher than that of T_{low}) will have a lower *wts* value than T_{low}. This will ensure that T_{low} will not be aborted. Using a similar argument, the property can be shown to hold for SF-MV-TO as well (the version with no bound K).
The drawback of SF-K-TO: Although SF-K-TO satisfies starvation-freedom, it, unfortunately, does not satisfy strict-serializability. Specifically, it violates the real-time requirement. *PKTO* uses *cts* for its working while SF-K-TO uses *wts*. It can be seen that *cts* is close to the real-time execution of transactions whereas *wts* of a transaction T_i is artificially inflated based on its *its* and might be much larger than its *cts*. We illustrate this with an example. Consider the history $H1$ as shown in Figure 3.4: $r_1(x,0)r_2(y,0)w_1(x,10)C_1w_2(x,20)C_2r_3(x,10)r_3(z,25)C_3$ with *cts* as 50, 60 and 80 and *wts* as 50, 100 and 80 for T_1, T_2, T_3 respectively. Here T_1, T_2 are ordered before T_3 in real-time with $T_1 \prec_{H1}^{RT} T_3$ and $T_2 \prec_{H1}^{RT} T_3$ although T_2 has a higher *wts* than T_3.

Here, as per SF-K-TO algorithm, T_3 reads x from T_1 since T_1 has the largest *wts* (50) smaller than T_3's *wts* (80). It can be verified that it is possible for SF-K-TO to generate such a history. But this history is not strict-serializable. The only possible serial order equivalent to

$H1$ and legal is $T_1T_3T_2$. But this violates real-time order as T_3 is serialized before T_2 but in $H1$, T_2 completes before T_3 has begun. Since $H1$ is not strict-serializable, it is not locally-opaque as well. Naturally, this drawback extends to SF-MV-TO as well.

3.3.6 Design of SF-K-RWSTM: Regaining Correctness while Preserving Starvation-Freedom

In this subsection, we discuss how principles of *PKTO* and SF-K-TO can be combined to obtain *SF-K-RWSTM* that provides both correctness (strict-serializability and locally-opaque) as well as starvation-freedom. To achieve this, we first understand why the initial algorithm, *PKTO* satisfies strict-serializability. This is because *cts* was used to create the ordering among committed transactions. *cts* is based on real-time ordering. In contrast, SF-K-TO uses *wts* which may not correspond to the real-time, as *wts* may be significantly larger than *cts* as shown by history $H1$ in Figure 3.4.

One straightforward way to modify SF-K-TO is to delay a committing transaction, say T_i with *wts* value wts_i until the real-time (G_tCntr) catches up to wts_i. This will ensure that the value of *wts* will also become the same as the real-time thereby guaranteeing strict-serializability. However, this is unacceptable, as in practice, it would require transaction T_i locking all the variables it plans to update and wait. This will adversely affect the performance of the STM system.

We can allow the transaction T_i to commit before its wts_i has caught up with the actual time if it does not violate the real-time ordering. Thus, to ensure that the notion of real-time order is respected by transactions in the course of their execution in SF-K-TO, we add extra time constraints. We use the idea of timestamp ranges. This notion of timestamp ranges was first used by Riegel et al. [39] in the context of multi-version RWSTMs. Several other researchers have used this idea since then such as Guerraoui et al. [6], Crain et al. [40], etc.

Thus, in addition to *its*, *cts*, and *wts*, each transaction T_i maintains a timestamp range: *Transaction Lower Timestamp Limit* or $tltl_i$ and *Transaction Upper Timestamp Limit* or $tutl_i$. When a transaction T_i begins, $tltl_i$ is assigned cts_i and $tutl_i$ is assigned the largest possible value which we denote as infinity. When T_i executes a method m in which it reads a version of a t-object x or creates a new version of x in $STM\_tryC()$, $tltl_i$ is incremented while $tutl_i$ gets decremented [1].

We require that all the transactions are serialized based on their *wts* while maintaining their real-time order. On executing a method m, T_i is ordered w.r.t to other transactions that have created a version of x based on increasing order of *wts*. For all transactions T_j which also have created a version of x and whose wts_j is less than wts_i, $tltl_i$ is incremented such that $tutl_j$ is less than $tltl_i$. Note that all such T_j are serialized before T_i. Similarly, for any transaction T_k

[1]Technically ∞, which is assigned to $tutl_i$, cannot be decremented. But here as mentioned earlier, we use ∞ to denote the largest possible value that can be represented in a system.

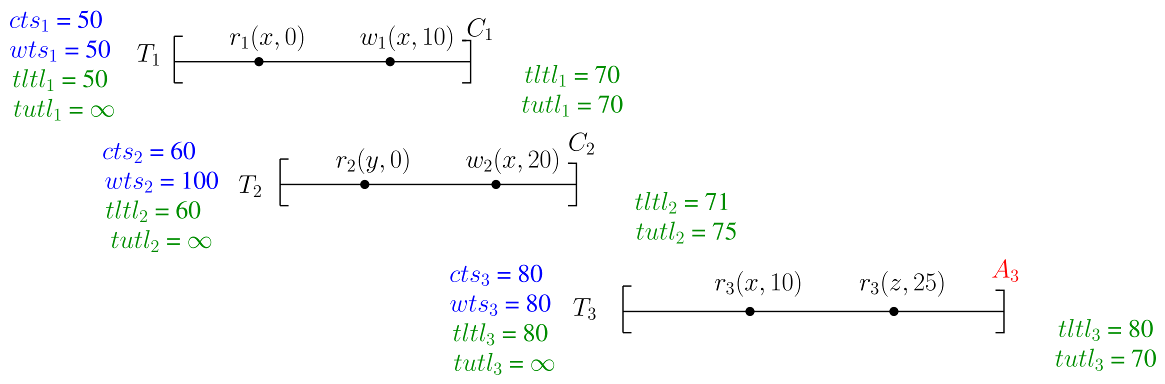


Figure 3.5: Execution under *SF-K-RWSTM* using $tltl$ and $tutl$

which has created a version of x and whose wts_k is greater than wts_i, $tutl_i$ is decremented such that it becomes less than $tltl_k$. Again, note that all such T_k are serialized after T_i.

If T_i reads a version x created by T_j then T_i is serialized after T_j and before any other T_k that also created a version of x such that $wts_j < wts_k$. The algorithm ensures that $wts_j < wts_i < wts_k$. For correctness, we again increment $tltl_i$ and decrement $tutl_i$ as above. After the increments of $tltl_i$ and the decrements of $tutl_i$, if $tltl_i$ turns out to be greater than $tutl_i$ then T_i is aborted. Intuitively, this implies that T_i's *wts* and real-time orders are out of *synchrony* and cannot be reconciled.

Finally, when a transaction T_i commits: T_i records its commit time (or $comTime_i$) by getting the current value of G_tCntr and incrementing it by $incrVal$ which is any value greater than or equal to 1. Then $tutl_i$ is set to $comTime_i$ if it is not already less than it. Now suppose T_i occurs in real-time before some other transaction, T_k but does not have any conflict with it. This step ensures that $tutl_i$ remains less than $tltl_k$ (which is initialized with cts_k).

We illustrate this technique with the history $H1$ shown in Figure 3.5. When T_1 starts its $cts_1 = 50, tltl_1 = 50, tutl_1 = \infty$. Now when T_1 commits, suppose G_tCntr is 70. Hence, $tutl_1$ reduces to 70. Next, when T_2 commits, suppose $tutl_2$ reduces to 75 (the current value of G_tCntr). As T_1, T_2 have accessed a common t-object x in a conflicting manner, $tltl_2$ is incremented to a value greater than $tutl_1$, say 71. Next, when T_3 begins, $tltl_3$ is assigned cts_3 which is 80 and $tutl_3$ is initialized to ∞. When T_3 reads 10 from T_1, which is $r_3(x, 10)$, $tutl_3$ is reduced to a value less than $tltl_2(= 71)$, say 70. But $tltl_3$ is already at 80. Hence, the limits of T_3 have crossed and thus causing T_3 to abort. The resulting history consisting of only committed transactions T_1T_2 is strict-serializable.

Based on this idea, we next develop a variation of SF-K-TO, *K-version Starvation-Free RWSTM System* or *SF-K-RWSTM*. To explain this algorithm, we first describe the structure of the version of a t-object used. It is a slight variation of the t-object used in *PKTO* algorithm explained in SubSection 3.3.2. It consists of following fields: (1) timestamp, ts which is the *wts* of the transaction that created this version (and not *cts* like *PKTO*); (2) the value of the version; (3) a list, called read-list, consisting of transactions ids (could be *cts* as well) that read from this version; (4) version real-time timestamp or `vrt` which is the tutl of the transaction that created this version. Thus a version has information of *wts* and tutl of the transaction that created it.

Now, we describe the main idea behind the $STM\_begin()$, $STM\_read()$, $STM\_write()$, and $STM\_tryC()$ methods of a transaction T_i which is an extension of ***PKTO***. Note that as per our notation i represents the *cts* of T_i.

$STM\_begin(t)$**:** A unique timestamp ts is allocated to T_i which is its *cts* (i from our assumption) which is generated by atomically incrementing the global counter G_tCntr. If the input t is null then $cts_i = its_i = ts$ as this is the first incarnation of this transaction. Otherwise, the non-null value of t is assigned to its_i. Then, *wts* is computed by Eq.(3.1). Finally, tltl and tutl are initialized as: $tltl_i = cts_i$, $tutl_i = \infty$.

$STM\_read(x)$**:** Transaction T_i reads from a version of x with timestamp j such that j is the largest timestamp less than wts_i (among the versions x), i.e. there exists no version k such that $j < k < wts_i$ is true. If no such j exists then T_i is aborted. Otherwise, after reading this version of x, T_i is stored in j's rl. Then we modify tltl, tutl as follows:

1. The version $x[j]$ is created by a transaction with wts_j which is less than wts_i. Hence, $tltl_i = max(tltl_i, x[j].\texttt{vrt} + 1)$.

2. Let p be the timestamp of smallest version larger than i. Then $tutl_i = min(tutl_i, x[p].\texttt{vrt} - 1)$.

3. After these steps, abort T_i if tltl and tutl have crossed, i.e., $tltl_i > tutl_i$.

$STM\_write(x, v)$**:** T_i stores this write to value x locally in its $wset_i$.

$STM\_tryC()$: This operation consists of multiple steps:

1. Before T_i can commit, we need to verify that any version it creates is updated consistently. T_i creates a new version with timestamp wts_i. Hence, we must ensure that any transaction that read a previous version is unaffected by this new version. Additionally, creating this version would require an update of tltl and tutl of T_i and other transactions whose read-write set overlaps with that of T_i. Thus, T_i first validates each t-object x in its $wset$ as follows:

   (a) T_i finds a version of x with timestamp j such that j is the largest timestamp less than wts_i (like in $STM\_read()$). If there exists no version of x with a timestamp less than wts_i then T_i is aborted. This is similar to Step 1b of the $STM\_tryC()$ of ***PKTO*** algorithm explained in SubSection 3.3.3.

   (b) Among all the transactions that have previously read from j suppose there is a transaction T_k such that $j < wts_i < wts_k$. Then (i) if T_k has already committed then T_i is aborted; (ii) Suppose T_k is live, and its_k is less than its_i. Then again T_i is aborted; (iii) If T_k is still live with its_i less than its_k then T_k is aborted.

This step is similar to Step 1a of the $STM\_tryC()$ of ***PKTO*** algorithm explained in SubSection 3.3.3.

(c) Next, we must ensure that T_i's tltl and tutl are updated correctly w.r.t to other concurrently executing transactions. To achieve this, we adjust tltl, tutl as follows: (i) Let j be the ts of the largest version smaller than wts_i. Then $tltl_i = max(tltl_i, x[j].\texttt{vrt} + 1)$. Next, for each reading transaction, T_r in $x[j].read\text{-}list$, we again set, $tltl_i = max(tltl_i, tutl_r + 1)$. (ii) Similarly, let p be the ts of the smallest version larger than wts_i. Then, $tutl_i = min(tutl_i, x[p].\texttt{vrt} - 1)$. (Note that we don't have to check for the transactions in the read-list of $x[p]$ as those transactions will have tltl higher than $x[p].\texttt{vrt}$ due to $STM\_read()$.) (iii) Finally, we get the commit time of this transaction from G_tCntr: $comTime_i = G\_tCntr.add\&Get(incrVal)$ where $incrVal$ is any constant ≥ 1. Then, $tutl_i = min(tutl_i, comTime_i)$. After performing these updates, abort T_i if tltl and tutl have crossed, i.e., $tltl_i > tutl_i$.

2. After performing the tests of Step 1 over each t-objects x in T_i's $wset$, if T_i has not yet been aborted, we proceed as follows: for each x in $wset_i$ create a vTuple $\langle wts_i, wset_i.x.v, null, tutl_i\rangle$. In this tuple, wts_i is the timestamp of the new version; $wset_i.x.v$ is the value of x is in T_i's $wset$; the read-list of the $vTuple$ is $null$; $\texttt{vrt}$ is $tutl_i$ (actually it can be any value between $tltl_i$ and $tutl_i$). Update the *vlist* of each t-object x similar to Step 2 of $STM\_tryC()$ of ***PKTO*** explained in SubSection 3.3.3.

3. Transaction T_i is then committed.

Step 1c.(iii) of $STM\_tryC()$ ensures that real-time order between transactions that are not in conflict. It can be seen that locks have to be used to ensure that all these methods to execute in a linearizable manner (i.e., atomically).

3.3.7 Data Structure and Detailed Pseudocode of SF-K-RWSTM

STM system consists of the following methods: $STM\_init(), STM\_begin(), STM\_read(x)$, $STM\_write(x, v)$, and $STM\_tryC()$. We assume that all the t-objects are ordered as $x_1, x_2, ..x_n$ and belong to the set $\mathscr{T}$. We describe the data structures used by the algorithm.

We start with structures that local to each transaction. Each transaction T_i maintains a $rset_i$ and $wset_i$. In addition it maintains the following structures (1) $comTime_i$: This is value given to T_i when it terminates which is assigned a value in STM_tryC() method. (2) A series of lists: smallRL, largeRL, allRL, prevVL, nextVL, relLL, abortRL. The meaning of these lists will be clear with the description of the pseudocode. In addition to these local structures, the following shared global structures are maintained that are shared across transactions (and hence, threads). We name all the shared variable starting with 'G'.

- G_tCntr (counter): This a numerical valued counter that is incremented when a transaction begins and terminates.

For each transaction T_i we maintain the following shared timestamps:

- G_lock_i: A lock for accessing all the shared variables of T_i.
- G_its_i (initial timestamp): It is a timestamp assigned to T_i when it was invoked for the first time without any aborts. The current value of G_tCntr is atomically assigned to it and then incremented. If T_i is aborted and restarts later then the application assigns it the same G_its.
- G_cts_i (current timestamp): It is a timestamp when T_i is invoked again at a later time after an abort. Like G_its, the current value of G_tCntr is atomically assigned to it and then incremented. When T_i is created for the first time, then its G_cts is same as its G_its.
- G_wts_i (working timestamp): It is the timestamp that T_i works with. It is either greater than or equal to T_i's G_cts. It is computed as follows: $G\_wts_i = G\_cts_i + C * (G\_cts_i - G\_its_i)$.
- G_valid_i: This is a boolean variable which is initially true. If it becomes false then T_i has to be aborted.
- G_state_i: This is a variable which states the current value of T_i. It has three states: `live`, `committed` or `aborted`.
- G_tltl_i, G_tutl_i (transaction lower and upper time limits): These are the time-limits described in the previous section used to keep the transaction *wts* and real-time orders in sync. G_tltl_i is G_cts of T_i when transaction begins and is a non-decreasing value. It continues to increase (or remains same) as T_i reads t-objects and later terminates. G_tutl_i on the other hand is a non-increasing value starting with ∞ when the T_i is created. It reduces (or remains same) as T_i reads t-objects and later terminates. If T_i commits then both G_tltl_i and G_tutl_i are made equal.

Two transactions having the same *its* are said to be incarnations. No two transaction can have the same *cts*. For simplicity, we assume that no two transactions have the same *wts* as well. In case, two transactions have the same *wts*, one can use the tuple ⟨*wts*, *cts* ⟩ instead of *wts*. But we ignore such cases. For each t-object x in $\mathscr{T}$, we maintain:

- x.`vl` (version list): It is a list consisting of version tuples or *vTuple* of the form ⟨`ts`, val, `rl`, `vrt`⟩. The details of the tuple are explained below.

- `ts` (timestmp): Here `ts` is the G_wts_i of a committed transaction T_i that has created this version.

- val: The value of this version.

- `rl` (readList): rl is the read list consists of all the transactions that have read this version. Each entry in this list is of the form $\langle rts \rangle$ where rts is the G_wts_j of a transaction T_j that read this version.

- `vrt` (version real-time timestamp): It is the G_tutl value (which is same as G_tltl) of the transaction T_i that created this version at the time of commit of T_i.

Algorithm 9 $STM\_init()$: Invoked at the start of the STM system. Initializes all the t-objects used by the STM system.

```
100: $G\_tCntr$ = 1; /*Global Transaction Counter*/
101: for all $x$ in $\mathscr{T}$ do /*All the t-objects used by the STM System*/
102:     /* $T_0$ is creating the first version of $x$: $\texttt{ts} = 0, val = 0, \texttt{rl} = nil, \texttt{vrt} = 0$ */
103:     add $\langle 0, 0, nil, 0 \rangle$ to $x.\texttt{vl}$;
104: end for;
```

Algorithm 10 $STM\_begin(its)$: Invoked by a thread to start a new transaction T_i. Thread can pass a parameter its which is the initial timestamp when this transaction was invoked for the first time. If this is the first invocation then its is nil. It returns the tuple $\langle id, G\_wts, G\_cts \rangle$.

```
105: $i$ = unique-id; /*An unique id to identify this transaction. It could be same as G_cts */
106: /*Initialize transaction specific local & global variables*/
107: if ($its == nil$) then
108:     $G\_its_i = G\_wts_i = G\_cts_i = G\_tCntr.get\&Inc()$; /*$G\_tCntr.get\&Inc()$ returns
     the     current value of G_tCntr and atomically increments it*/
109: else
110:     $G\_its_i = its$;
111:     $G\_cts_i = G\_tCntr.get\&Inc()$;
112:     $G\_wts_i = G\_cts_i + C * (G\_cts_i - G\_its_i)$; /*$C$ is any constant greater or equal to than
         1*/
113: end if
114: $G\_tltl_i = G\_cts_i$; $G\_tutl_i = comTime_i = \infty$;
115: $G\_state_i$ = live; $G\_valid_i = T$;
116: $rset_i = wset_i = nil$;
117: return $\langle i, G\_wts_i, G\_cts_i \rangle$
```

Algorithm 11 $STM\_read(i, x)$: Invoked by a transaction T_i to read t-object x. It returns either the value of x or $\mathscr{A}$.

```
118: if ($x \in wset_i$) then /*Check if the t-object $x$ is in $wset_i$*/
119:     return $wset_i[x].val$;
120: else if ($x \in rset_i$) then /*Check if the t-object $x$ is in $rset_i$*/
121:     return $rset_i[x].val$;
122: else/*t-object $x$ is not in $rset_i$ and $wset_i$*/
123:     lock $x$; lock $G\_lock_i$;
124:     if ($G\_valid_i == F$) then return abort(i);
125:     end if
126:     /* findLTS: From $x.\texttt{vl}$, returns the largest $\texttt{ts}$ value less than $G\_wts_i$. If no such
         version exists, it returns $nil$ */
127:     $curVer = findLTS(G\_wts_i, x)$;
128:     if ($curVer == nil$) then return abort(i); /*Proceed only if $curVer$ is not nil*/
129:     end if
130:     /* findSTL: From $x.\texttt{vl}$, returns the smallest $\texttt{ts}$ value greater than $G\_wts_i$. If no such
         version exists, it returns $nil$ */
131:     $nextVer = findSTL(G\_wts_i, x)$;
132:     if ($nextVer \neq nil$) then
133:         /*Ensure that $G\_tutl_i$ remains smaller than $nextVer$'s $\texttt{vrt}$ */
134:         $G\_tutl_i = min(G\_tutl_i, x[nextVer].\texttt{vrt} - 1)$;
135:     end if
136:     /*$G\_tltl_i$ should be greater than $x[curVer].\texttt{vrt}$*/
137:     $G\_tltl_i = max(G\_tltl_i, x[curVer].\texttt{vrt} + 1)$;
138:     if ($G\_tltl_i > G\_tutl_i$) then /*If the limits have crossed each other, then $T_i$ is aborted*/
139:         return abort(i);
140:     end if
141:     $val = x[curVer].v$; add $\langle x, val \rangle$ to $rset_i$;
142:     add $T_i$ to $x[curVer].rl$;
143:     unlock $G\_lock_i$; unlock $x$;
144:     return $val$;
145: end if
```

Algorithm 12 $STM\_write_i(x, val)$: A Transaction T_i writes into local memory.

```
146: Append the $d\_tuple\langle x, val \rangle$ to $wset_i$.
147: return $ok$;
```

Algorithm 13 $STM\_tryC()$: Returns ok on commit else return Abort.

```
148: /*The following check is an optimization which needs to be performed again later*/
149: lock $G\_lock_i$;
150: if ($G\_valid_i == F$) then return abort(i);
151: end if
152: unlock $G\_lock_i$;
153: /*Initialize smaller read list (smallRL), larger read list (largeRL), all read list (allRL) to
     nil*/
154: $smallRL = largeRL = allRL = nil$;
155: /*Initialize previous version list (prevVL), next version list (nextVL) to nil*/
156: $prevVL = nextVL = nil$;
157: for all $x \in wset_i$ do
158:     lock $x$ in pre-defined order;
159:     /* findLTS: returns the version of $x$ with the largest ts less than $G\_wts_i$. If no such
         version exists, it returns $nil$. */
160:     $prevVer = findLTS(G\_wts_i, x)$; /*prevVer: largest version smaller than $G\_wts_i$*/
161:     if ($prevVer == nil$) then /*There exists no version with ts value less than $G\_wts_i$*/
162:         lock $G\_lock_i$; return abort(i);
163:     end if
164:     $prevVL = prevVL \cup prevVer$; /*prevVL stores the previous version in sorted order
         */
165:     $allRL = allRL \cup x[prevVer].rl$; /*Store the read-list of the previous version*/
166:     /*getLar: obtain the list of reading transactions of $x[prevVer].rl$ whose $G\_wts$ is
         greater than $G\_wts_i$*/
167:     $largeRL = largeRL \cup getLar(G\_wts_i, x[prevVer].rl)$;
168:     /*getSm: obtain the list of reading transactions of $x[prevVer].rl$ whose $G\_wts$ is
         smaller than $G\_wts_i$*/
169:     $smallRL = smallRL \cup getSm(G\_wts_i, x[prevVer].rl)$;
170:     /* findSTL: returns the version with the smallest ts value greater than $G\_wts_i$. If no
         such version exists, it returns $nil$. */
171:     $nextVer = findSTL(G\_wts_i, x)$; /*nextVer: smallest version larger than $G\_wts_i$*/
172:     if ($nextVer \neq nil$)) then
173:         $nextVL = nextVL \cup nextVer$; /*nextVL stores next version in sorted order*/
174:     end if
175: end for/*$x \in wset_i$*/
176: $relLL = allRL \cup T_i$; /*Initialize relevant Lock List (relLL)*/
177: for all ($T_k \in relLL$) do
178:     lock $G\_lock_k$ in pre-defined order; /*Note: Since $T_i$ is also in $relLL$, $G\_lock_i$ is also
         locked*/
```

```
179: end for
180: /*Verify if $G\_valid_i$ is false*/
181: if $(G\_valid_i == F)$ then return abort(i);
182: end if
183: $abortRL = nil$ /*Initialize abort read list (abortRL)*/
184: /*Among the transactions in $T_k$ in $largeRL$, either $T_k$ or $T_i$ has to be aborted*/
185: for all $(T_k \in largeRL)$ do
186:     if $(isAborted(T_k))$ then
187:         /*Transaction $T_k$ can be ignored since it is already aborted or about to be aborted*/
188:         continue;
189:     end if
190:     if $(G\_its_i < G\_its_k) \wedge (G\_state_k == \texttt{live})$ then
191:         /*Transaction $T_k$ has lower priority and is not yet committed. So it needs to be
            aborted*/
192:         $abortRL = abortRL \cup T_k$; /*Store $T_k$ in abortRL */
193:     else/*Transaction $T_i$ has to be aborted*/
194:         return abort(i);
195:     end if
196: end for
197: /*Ensure that $G\_tltl_i$ is greater than $\texttt{vrt}$ of the versions in $prevVL$*/
198: for all $(ver \in prevVL)$ do
199:     $x$ = t-object of $ver$;
200:     $G\_tltl_i = max(G\_tltl_i, x[ver].\texttt{vrt} + 1)$;
201: end for
202: /*Ensure that $\texttt{vutl}_i$ is less than $\texttt{vrt}$ of versions in $nextVL$*/
203: for all $(ver \in nextVL)$ do
204:     $x$ = t-object of $ver$;
205:     $G\_tutl_i = min(G\_tutl_i, x[ver].\texttt{vrt} - 1)$;
206: end for
207: /*Store the current value of the global counter as commit time and increment it*/
208: $comTime_i = G\_tCntr.add\&Get(incrVal)$; /*$incrVal$ can be any constant $\geq$ 1*/
209: $G\_tutl_i = min(G\_tutl_i, comTime_i)$; /*Ensure that $G\_tutl_i$ is less than or equal to
     $comTime$*/
210: /*Abort $T_i$ if its limits have crossed*/
211: if $(G\_tltl_i > G\_tutl_i)$ then return abort(i);
212: end if
213: for all $(T_k \in smallRL)$ do
214:     if $(isAborted(T_k))$ then
215:         continue;
```

```
216:     **end if**
217:     **if** $(G\_tltl_k \geq G\_tutl_i)$ **then** /*Ensure that the limits do not cross for both $T_i$ & $T_k$*/
218:         **if** $(G\_state_k == live)$ **then** /*Check if $T_k$ is live*/
219:             **if** $(G\_its_i < G\_its_k)$ **then**
220:                 /*Transaction $T_k$ has lower priority and is not yet committed. So it needs
                    to be aborted*/
221:                 $abortRL = abortRL \cup T_k$; /*Store $T_k$ in abortRL */
222:             **else**/*Transaction $T_i$ has to be aborted*/
223:                 return abort(i);
224:             **end if**/*$(G\_its_i < G\_its_k)$*/
225:         **else**/*($T_k$ is committed. Hence, $T_i$ has to be aborted)*/
226:             return abort(i);
227:         **end if**/*$(G\_state_k == live)$*/
228:     **end if**/*$(G\_tltl_k \geq G\_tutl_i)$*/
229: **end for**$(T_k \in smallRL)$
230: /*After this point $T_i$ can't abort.*/
231: $G\_tltl_i = G\_tutl_i$;
232: /*Since $T_i$ can't abort, we can update $T_k$'s G_tutl */
233: **for all** $(T_k \in smallRL)$ **do**
234:     **if** $(isAborted(T_k))$ **then**
235:         continue;
236:     **end if**
237:     /* The following line ensure that $G\_tltl_k \leq G\_tutl_k < G\_tltl_i$. Note that this does not
       cause the limits of $T_k$ to cross each other because of the check in Line 217.*/
238:     $G\_tutl_k = min(G\_tutl_k, G\_tltl_i - 1)$;
239: **end for**
240: **for all** $T_k \in abortRL$ **do** /*Abort all the transactions in abortRL since $T_i$ can't abort*/
241:     $G\_valid_k = F$;
242: **end for**
243: /*Having completed all the checks, $T_i$ can be committed*/
244: **for all** $(x \in wset_i)$ **do**
245:     /* Create new v_tuple: ts, $val$, rl, vrt for $x$ */
246:     $newTuple = \langle G\_wts_i, wset_i[x].val, nil, G\_tltl_i \rangle$;
247:     **if** $(|x.vl| > k)$ **then**
248:         replace the oldest tuple in $x$.vl with $newTuple$; /*$x$.vl is ordered by ts*/
249:     **else**
250:         add a $newTuple$ to $x.vl$ in sorted order;
251:     **end if**
252: **end for**/*$x \in wset_i$*/
```

```
253: $G\_state_i$ = commit;
254: unlock all variables;
255: return $\mathscr{C}$;
```

Algorithm 14 $isAborted(T_k)$: Verifies if T_i is already aborted or its G_valid flag is set to false implying that T_i will be aborted soon.

```
256: if $(G\_valid_k == F) \vee (G\_state_k ==$ abort$) \vee (T_k \in abortRL)$ then
257:     return $T$;
258: else
259:     return $F$;
260: end if
```

Algorithm 15 $abort(i)$: Invoked by various STM methods to abort transaction T_i and returns $\mathscr{A}$.

```
261: $G\_valid_i = F$; $G\_state_i$ = abort;
262: unlock all variables locked by $T_i$;
263: return $\mathscr{A}$;
```

Garbage Collection: Having described the *starvation-free* algorithm, we now describe how garbage collection can be performed on the unbounded variant, *SF-UV-RWSTM* to achieve *SF-UV-RWSTM-GC*. This is achieved by deleting non-latest version (i.e., there exists a version with greater ts) of each t-object whose timestamp, ts is less than the *cts* of smallest live transaction. It must be noted that *SF-UV-RWSTM* (*SF-K-RWSTM*) works with *wts* which is greater or equal to *cts* for any transaction. Interestingly, the same garbage collection principle can be applicable here as well that has been applied for *PMVTO* to achieve *PMVTO-GC* explained in SubSection 3.3.3.

To identify the transaction with the smallest *cts* among live transactions, we maintain a set of all the live transactions, *live-list*. When a transaction T_i begins, its *cts* is added to this *live-list* and when T_i terminates (either commits or aborts), T_i is deleted from this *live-list*.

3.4 Liveness Proof of SF-K-RWSTM Algorithm

Proof Notations: Let $gen($*SF-K-RWSTM*$)$ consist of all the histories accepted by *SF-K-RWSTM* algorithm. In the follow sub-section, we only consider histories that are generated by *SF-K-RWSTM* unless explicitly stated otherwise. For simplicity, we only consider sequential histories in our discussion below.

Consider a transaction T_i in a history H generated by *SF-K-RWSTM*. Once a transaction executes STM_begin () then *its*, *cts*, and *wts* values of it do not change. Thus, we denote them as

its_i, cts_i, wts_i respectively for T_i. In case the context of the history H in which the transaction executing is important, we denote these variables as $H.its_i, H.cts_i, H.wts_i$ respectively.

The other variables that a transaction maintains are: tltl, tutl, lock, valid, state. These values change as the execution proceeds. Hence, we denote them as: $H.tltl_i, H.tutl_i, H.lock_i, H.valid_i$, $H.state_i$. These represent the values of tltl, tutl, lock, valid, state after the execution of last event in H. Depending on the context, we sometimes ignore H and denote them only as: $lock_i, valid_i, state_i, tltl_i, tutl_i$.

We approximate the system time with the value of $tCntr$. We denote the sys-time of history H as the value of $tCntr$ immediately after the last event of H. Further, we also assume that the value of C is 1 in our arguments. But, it can be seen that the proof will work for any value greater than 0 as well.

The application invokes transactions in such a way that if the current T_i transaction aborts, it invokes a new transaction T_j with the same *its*. We say that T_i is an *incarnation* of T_j in a history H if $H.its_i = H.its_j$. Thus the multiple incarnations of a transaction T_i get invoked by the application until an incarnation finally commits.

To capture this notion of multiple transactions with the same *its*, we define *incarSet* (incarnation set) of T_i in H as the set of all the transactions in H which have the same *its* as T_i and includes T_i as well. Formally,

$$H.incarSet(T_i) = \{T_j | (T_i = T_j) \vee (H.its_i = H.its_j)\}$$

Note that from this definition of incarSet, we implicitly get that T_i and all the transactions in its incarSet of H also belong to H. Formally, $H.incarSet(T_i) \in H.txns$.

The application invokes different incarnations of a transaction T_i in such a way that as long as an incarnation is live, it does not invoke the next incarnation. It invokes the next incarnation after the current incarnation has got aborted. Once an incarnation of T_i has committed, it can't have any future incarnations. Thus, the application views all the incarnations of a transaction as a single *application-transaction*.

We assign *incNums* to all the transactions that have the same *its*. We say that a transaction T_i starts *afresh*, if $T_i.incNum$ is 1. We say that T_i is the nextInc of T_i if T_j and T_i have the same *its* and T_i's incNum is T_j's incNum + 1. Formally, $\langle (T_i.nextInc = T_j) \equiv (its_i = its_j) \wedge (T_i.incNum = T_j.incNum + 1) \rangle$

As mentioned the objective of the application is to ensure that every application-transaction eventually commits. Thus, the applications views the entire incarSet as a single application-transaction (with all the transactions in the incarSet having the same *its*). We can say that an application-transaction has committed if in the corresponding incarSet a transaction in eventually commits. For T_i in a history H, we denote this by a boolean value incarCt (incarnation set committed) which implies that either T_i or an incarnation of T_i has committed. Formally, we define it as $H.incarCt(T_i)$

$$H.incarCt(T_i) = \begin{cases} True & (\exists T_j : (T_j \in H.incarSet(T_i)) \land (T_j \in H.committed)) \\ False & \text{otherwise} \end{cases}$$

From the definition of incarCt we get the following observations & lemmas about a transaction T_i

Observation 5 *Consider a transaction T_i in a history H with its incarCt being true in H. Then T_i is terminated (either committed or aborted) in H. Formally, $\langle H, T_i : (T_i \in H.txns) \land (H.incarCt(T_i)) \implies (T_i \in H.terminated) \rangle$.*

Observation 6 *Consider a transaction T_i in a history H with its incarCt being true in $H1$. Let $H2$ be a extension of $H1$ with a transaction T_j in it. Suppose T_j is an incarnation of T_i. Then T_j's incarCt is true in $H2$. Formally, $\langle H1, H2, T_i, T_j : (H1 \sqsubseteq H2) \land (H1.incarCt(T_i)) \land (T_j \in H2.txns) \land (T_i \in H2.incarSet(T_j)) \implies (H2.incarCt(T_j)) \rangle$.*

Lemma 7 *Consider a history $H1$ with a strict extension $H2$. Let T_i & T_j be two transactions in $H1$ & $H2$ respectively. Let T_j not be in $H1$. Suppose T_i's incarCt is true. Then its of T_i cannot be the same as its of T_j. Formally, $\langle H1, H2, T_i, T_j : (H1 \sqsubset H2) \land (H1.incarCt(T_i)) \land (T_j \in H2.txns) \land (T_j \notin H1.txns) \implies (H1.its_i \neq H2.its_j) \rangle$.*

Proof. Here, we have that T_i's incarCt is true in $H1$. Suppose T_j is an incarnation of T_i, i.e., their *its*s are the same. We are given that T_j is not in $H1$. This implies that T_j must have started after the last event of $H1$.

We are also given that T_i's incarCt is true in $H1$. This implies that an incarnation of T_i or T_i itself has committed in $H1$. After this commit, the application will not invoke another transaction with the same *its* as T_i. Thus, there cannot be a transaction after the last event of $H1$ and in any extension of $H1$ with the same *its* of T_1. Hence, $H1.its_i$ cannot be same as $H2.its_j$.

Now we show the liveness with the following observations, lemmas & theorems. We start with two observations about that histories of which one is an extension of the other. The following states that for any history, there exists an extension. In other words, we assume that the STM system runs forever and does not terminate. This is required for showing that every transaction eventually commits.

Observation 8 *Consider a history $H1$ generated by gen(SF-K-RWSTM). Then there is a history $H2$ in gen(SF-K-RWSTM) such that $H2$ is a strict extension of $H1$. Formally, $\langle \forall H1 : (H1 \in gen(ksftm)) \implies (\exists H2 : (H2 \in gen(ksftm)) \land (H1 \sqsubset H2) \rangle$.*

The follow observation is about the transaction in a history and any of its extensions.

Observation 9 *Given two histories $H1$ & $H2$ such that $H2$ is an extension of $H1$. Then, the set of transactions in $H1$ are a subset equal to the set of transaction in $H2$. Formally, $\langle \forall H1, H2 : (H1 \sqsubseteq H2) \implies (H1.txns \subseteq H2.txns) \rangle$.*

In order for a transaction T_i to commit in a history H, it has to compete with all the live transactions and all the aborted that can become live again as a different incarnation. Once a transaction T_j aborts, another incarnation of T_j can start and become live again. Thus T_i will have to compete with this incarnation of T_j later. Thus, we have the following observation about aborted & committed transactions.

Observation 10 *Consider an aborted transaction T_i in a history $H1$. Then there is an extension of $H1$, $H2$ in which an incarnation of T_i, T_j is live and has cts_j is greater than cts_i. Formally, $\langle H1, T_i : (T_i \in H1.aborted) \implies (\exists T_j, H2 : (H1 \sqsubseteq H2) \land (T_j \in H2.live) \land (H2.its_i = H2.its_j) \land (H2.cts_i < H2.cts_j))\rangle$.*

Observation 11 *Consider an committed transaction T_i in a history $H1$. Then there is no extension of $H1$, in which an incarnation of T_i, T_j is live. Formally, $\langle H1, T_i : (T_i \in H1.committed) \implies (\nexists T_j, H2 : (H1 \sqsubseteq H2) \land (T_j \in H2.live) \land (H2.its_i = H2.its_j))\rangle$.*

Lemma 12 *Consider a history $H1$ and its extension $H2$. Let T_i, T_j be in $H1, H2$ respectively such that they are incarnations of each other. If wts of T_i is less than wts of T_j then cts of T_i is less than cts T_j. Formally, $\langle H1, H2, T_i, T_j : (H1 \sqsubset H2) \land (T_i \in H1.txns) \land (T_j \in H2.txns) \land (T_i \in H2.incarSet(T_j)) \land (H1.wts_i < H2.wts_j) \implies (H1.cts_i < H2.cts_j)\rangle$*

Proof. Here we are given that

$$H1.wts_i < H2.wts_j \tag{3.2}$$

The definition of *wts* of T_i is: $H1.wts_i = H1.cts_i + C * (H1.cts_i - H1.its_i)$. Combining this Eq.(3.2), we get that

$(C+1) * H1.cts_i - C * H1.its_i < (C+1) * H2.cts_j - C * H2.its_j \xrightarrow[H1.its_i = H2.its_j]{T_i \in H2.incarSet(T_j)}$ $H1.cts_i < H2.cts_j$.

Lemma 13 *Consider a live transaction T_i in a history $H1$ with its wts_i less than a constant α. Then there is a strict extension of $H1$, $H2$ in which an incarnation of T_i, T_j is live with wts greater than α. Formally, $\langle H1, T_i : (T_i \in H1.live) \land (H1.wts_i < \alpha) \implies (\exists T_j, H2 : (H1 \sqsubseteq H2) \land (T_i \in H2.incarSet(T_j)) \land ((T_j \in H2.committed) \lor ((T_j \in H2.live) \land (H2.wts_j > \alpha))))\rangle$.*

Proof. The proof comes the behavior of an application-transaction. The application keeps invoking a transaction with the same *its* until it commits. Thus the transaction T_i which is live in $H1$ will eventually terminate with an abort or commit. If it commits, $H2$ could be any history after the commit of T_2.

On the other hand if T_i is aborted, as seen in Observation 10 it will be invoked again or reincarnated with another *cts* and *wts*. It can be seen that *cts* is always increasing. As a result, the *wts* is also increasing. Thus eventually the *wts* will become greater α. Hence, we have that either an incarnation of T_i will get committed or will eventually have *wts* greater than or equal to α.

Next we have a lemma about *cts* of a transaction and the sys-time of a history.

Lemma 14 *Consider a transaction T_i in a history H. Then, we have that cts of T_i will be less than or equal to sys-time of H. Formally, $\langle T_i, H1 : (T_i \in H.txns) \implies (H.cts_i \leq H.sys\text{-}time)\rangle$.*

Proof. We get this lemma by observing the methods of the STM System that increment the tCntr which are STM_begin and STM_tryC. It can be seen that *cts* of T_i gets assigned in the STM_begin method. So if the last method of H is the STM_begin of T_i then we get that *cts* of T_i is same as sys-time of H. On the other hand if some other method got executed in H after STM_begin of T_i then we have that *cts* of T_i is less than sys-time of H. Thus combining both the cases, we get that *cts* of T_i is less than or equal to as sys-time of H, i.e., $(H.cts_i \leq H.sys\text{-}time)$
From this lemma, we get the following corollary which is the converse of the lemma statement

Corollary 15 *Consider a transaction T_i which is not in a history $H1$ but in an strict extension of $H1$, $H2$. Then, we have that cts of T_i is greater than the sys-time of H. Formally, $\langle T_i, H1, H2 : (H1 \sqsubset H2) \land (T_i \notin H1.txns) \land (T_i \in H2.txns) \implies (H2.cts_i > H1.sys\text{-}time)\rangle$.*

Now, we have lemma about the methods of *SF-K-RWSTM* completing in finite time.

Lemma 16 *If all the locks are fair and the underlying system scheduler is fair then all the methods of SF-K-RWSTM will eventually complete.*

Proof. It can be seen that in any method, whenever a transaction T_i obtains multiple locks, it obtains locks in the same order: first lock relevant t-objects in a pre-defined order and then lock relevant G_locks again in a predefined order. Since all the locks are obtained in the same order, it can be seen that the methods of *SF-K-RWSTM* will not deadlock.

It can also be seen that none of the methods have any unbounded while loops. All the loops in STM_tryC method iterate through all the t-objects in the write-set of T_i. Moreover, since we assume that the underlying scheduler is fair, we can see that no thread gets swapped out infinitely. Finally, since we assume that all the locks are fair, it can be seen all the methods terminate in finite time.

Theorem 17 *Every transaction either commits or aborts in finite time.*

Proof. This theorem comes directly from the Lemma 16. Since every method of *SF-K-RWSTM* will eventually complete, all the transactions will either commit or abort in finite time.
From this theorem, we get the following corollary which states that the maximum *lifetime* of any transaction is L.

Corollary 18 *Any transaction T_i in a history H will either commit or abort before the sys-time of H crosses $cts_i + L$.*

The following lemma connects *wts* and *its* of two transactions, T_i, T_j.

Lemma 19 *Consider a history $H1$ with two transactions T_i, T_j. Let T_i be in $H1.live$. Suppose T_j's wts is greater or equal to T_i' s wts. Then its of T_j is less than $its_i + 2 * L$. Formally, $\langle H, T_i, T_j : (\{T_i, T_j\} \subseteq H.txns) \wedge (T_i \in H.live) \wedge (H.wts_j \geq H.wts_i) \implies (H.its_i + 2L \geq H.its_j)\rangle$.*

Proof. Since T_i is live in $H1$, from Corollary 18, we get that it terminates before the system time, $tCntr$ becomes $cts_i + L$. Thus, sys-time of history $H1$ did not progress beyond $cts_i + L$. Hence, for any other transaction T_j (which is either live or terminated) in $H1$, it must have started before sys-time has crossed $cts_i + L$. Formally $\langle cts_j \leq cts_i + L\rangle$.

Note that we have defined *wts* of a transaction T_j as: $wts_j = (cts_j + C * (cts_j - its_j))$. Now, let us consider the difference of the *wts*s of both the transactions.
$wts_j - wts_i = (cts_j + C * (cts_j - its_j)) - (cts_i + C * (cts_i - its_i))$
$= (C + 1)(cts_j - cts_i) - C(its_j - its_i)$
$\leq (C + 1)L - C(its_j - its_i) \qquad [\because cts_j \leq cts_i + L]$
$= 2 * L + its_i - its_j \qquad [\because C = 1]$

Thus, we have that: $\langle (its_i + 2L - its_j) \geq (wts_j - wts_i)\rangle$. This gives us that
$((wts_j - wts_i) \geq 0) \implies ((its_i + 2L - its_j) \geq 0)$.
From the above implication we get that, $(wts_j \geq wts_i) \implies (its_i + 2L \geq its_j)$.

It can be seen that *SF-K-RWSTM* algorithm gives preference to transactions with lower *its* to commit. To understand this notion of preference, we define a few notions of enablement of a transaction T_i in a history H. We start with the definition of *itsEnabled* as:

Definition 2 *We say T_i is* itsEnabled *in H if for all transactions T_j with its lower than its of T_i in H have incarCt to be true. Formally,*

$$H.itsEnabled(T_i) = \begin{cases} True & (T_i \in H.live) \wedge (\forall T_j \in H.txns : (H.its_j < H.its_i) \\ & \implies (H.incarCt(T_j))) \\ False & otherwise \end{cases}$$

The follow lemma states that once a transaction T_i becomes itsEnabled it continues to remain so until it terminates.

Lemma 20 *Consider two histories $H1$ and $H2$ with $H2$ being a extension of $H1$. Let a transaction T_i being live in both of them. Suppose T_i is itsEnabled in $H1$. Then T_i is itsEnabled in $H2$ as well. Formally, $\langle H1, H2, T_i : (H1 \sqsubseteq H2) \wedge (T_i \in H1.live) \wedge (T_i \in H2.live) \wedge (H1.itsEnabled(T_i)) \implies (H2.itsEnabled(T_i))\rangle$.*

Proof. When T_i begins in a history $H3$ let the set of transactions with *its* less than its_i be $smIts$. Then in any extension of $H3$, $H4$ the set of transactions with *its* less than its_i remains as $smIts$.

Suppose $H1, H2$ are extensions of $H3$. Thus in $H1, H2$ the set of transactions with *its* less than its_i will be $smIts$. Hence, if T_i is itsEnabled in $H1$ then all the transactions T_j in $smIts$ are $H1.incarCt(T_j)$. It can be seen that this continues to remain true in $H2$. Hence in $H2$, T_i is also itsEnabled which proves the lemma.

The following lemma deals with a committed transaction T_i and any transaction T_j that terminates later. In the following lemma, $incrVal$ is any constant greater than or equal to 1.

Lemma 21 *Consider a history H with two transactions T_i, T_j in it. Suppose transaction T_i commits before T_j terminates (either by commit or abort) in H. Then $comTime_i$ is less than $comTime_j$ by at least $incrVal$. Formally, $\langle H, \{T_i, T_j\} \in H.txns : (STM\_tryC_i <_H term\text{-}op_j) \implies (comTime_i + incrVal \leq comTime_j)\rangle$.*

Proof. When T_i commits, let the value of the global $tCntr$ be α. It can be seen that in STM_begin method, $comTime_j$ get initialized to ∞. The only place where $comTime_j$ gets modified is at Line 208 of STM_tryC. Thus if T_j gets aborted before executing STM_tryC method or before this line of STM_tryC we have that $comTime_j$ remains at ∞. Hence in this case we have that $\langle comTime_i + incrVal < comTime_j\rangle$.

If T_j terminates after executing Line 208 of STM_tryC method then $comTime_j$ is assigned a value, say β. It can be seen that β will be greater than α by at least $incrVal$ due to the execution of this line. Thus, we have that $\langle \alpha + incrVal \leq \beta\rangle$

The following lemma connects the G_tltl and comTime of a transaction T_i.

Lemma 22 *Consider a history H with a transaction T_i in it. Then in H, $tltl_i$ will be less than or equal to $comTime_i$. Formally, $\langle H, \{T_i\} \in H.txns : (H.tltl_i \leq H.comTime_i)\rangle$.*

Proof. Consider the transaction T_i. In STM_begin method, $comTime_i$ get initialized to ∞. The only place where $comTime_i$ gets modified is at Line 208 of STM_tryC. Thus if T_i gets aborted before this line or if T_i is live we have that $(tltl_i \leq comTime_i)$. On executing Line 208, $comTime_i$ gets assigned to some finite value and it does not change after that.

It can be seen that $tltl_i$ gets initialized to cts_i in Line 108 of STM_begin method. In that line, cts_i reads $tCntr$ and increments it atomically. Then in Line 208, $comTime_i$ gets assigned the value of $tCntr$ after incrementing it. Thus, we clearly get that $cts_i (= tltl_i$ initially$) < comTime_i$. Then $tltl_i$ gets updated on Line 137 of read, Line 200 and Line 231 of STM_tryC methods. Let us analyze them case by case assuming that $tltl_i$ was last updated in each of these methods before the termination of T_i:

1. Line 137 of read method: Suppose this is the last line where $tltl_i$ updated. Here $tltl_i$ gets assigned to 1 + `vrt` of the previously committed version which say was created by a transaction T_j. Thus, we have the following equation,

$$tltl_i = 1 + x[j].\texttt{vrt} \tag{3.3}$$

   It can be seen that $x[j].\texttt{vrt}$ is same as $tltl_j$ when T_j executed Line 246 of STM_tryC. Further, $tltl_j$ in turn is same as $tutl_j$ due to Line 231 of STM_tryC. From Line 209, it can be seen that $tutl_j$ is less than or equal to $comTime_j$ when T_j committed. Thus we have that

$$x[j].\texttt{vrt} = tltl_j = tutl_j \leq comTime_j \tag{3.4}$$

   It is clear that from the above discussion that T_j executed STM_tryC method before T_i terminated (i.e. $STM\_tryC_j <_{H1} term\text{-}op_i$). From Eq.(3.3) and Eq.(3.4), we get $tltl_i \leq 1 + comTime_j \xrightarrow{incrVal \geq 1} tltl_i \leq incrVal + comTime_j \xrightarrow{Lemma\ 21} tltl_i \leq comTime_i$

2. Line 200 of STM_tryC method: The reasoning in this case is very similar to the above case.

3. Line 231 of STM_tryC method: In this line, $tltl_i$ is made equal to $tutl_i$. Further, in Line 209, $tutl_i$ is made lesser than or equal to $comTime_i$. Thus combing these, we get that $tltl_i \leq comTime_i$. It can be seen that the reasoning here is similar in part to Case 1.

Hence, in all the three cases we get that $\langle tltl_i \leq comTime_i \rangle$.

The following lemma connects the G_tutl,comTime of a transaction T_i with *wts* of a transaction T_j that has already committed.

Lemma 23 *Consider a history H with a transaction T_i in it. Suppose $tutl_i$ is less than $comTime_i$. Then, there is a committed transaction T_j in H such that wts_j is greater than wts_i. Formally, $\langle H \in gen(\textbf{SF-K-RWSTM}), \{T_i\} \in H.txns : (H.tutl_i < H.comTime_i) \implies (\exists T_j \in H.committed : H.wts_j > H.wts_i)\rangle$.*

Proof. It can be seen that G_tutl_i initialized in STM_begin method to ∞. $tutl_i$ is updated in Line 134 of read method, Line 205 & Line 209 of STM_tryC method. If T_i executes Line 134 of read method and/or Line 205 of STM_tryC method then $tutl_i$ gets decremented to some value less than ∞, say α. Further, it can be seen that in both these lines the value of $tutl_i$ is possibly decremented from ∞ because of $nextVer$ (or ver), a version of x whose `ts` is greater than T_i's *wts*. This implies that some transaction T_j, which is committed in H, must have created $nextVer$ (or ver) and $wts_j > wts_i$.

Next, let us analyze the value of α. It can be seen that $\alpha = x[nextVer/ver].vrt - 1$ where $nextVer/ver$ was created by T_j. Further, we can see when T_j executed STM_tryC, we have that $x[nextVer].vrt = tltl_j$ (from Line 246). From Lemma 22, we get that $tltl_j \leq comTime_j$. This implies that $\alpha < comTime_j$. Now, we have that T_j has already committed before the termination of T_i. Thus from Lemma 21, we get that $comTime_j < comTime_i$. Hence, we have that,

$$\alpha < comTime_i \tag{3.5}$$

Now let us consider Line 209 executed by T_i which causes $tutl_i$ to change. This line will get executed only after both Line 134 of read method, Line 205 of STM_tryC method. This is because every transaction executes STM_tryC method only after read method. Further within STM_tryC method, Line 209 follows Line 205.

There are two sub-cases depending on the value of $tutl_i$ before the execution of Line 209: (i) If $tutl_i$ was ∞ and then get decremented to $comTime_i$ upon executing this line, then we get $comTime_i = tutl_i$. From Eq.(3.5), we can ignore this case. (ii) Suppose the value of $tutl_i$ before executing Line 209 was α. Then from Eq.(3.5) we get that $tutl_i$ remains at α on execution of Line 209. This implies that a transaction T_j committed such that $wts_j > wts_i$.
The following lemma connects the G_tltl of a committed transaction T_j and comTime of a transaction T_i that commits later.

Lemma 24 *Consider a history $H1$ with transactions T_i, T_j in it. Suppose T_j is committed and T_i is live in $H1$. Then in any extension of $H1$, say $H2$, $tltl_j$ is less than or equal to $comTime_i$. Formally, $\langle H1, H2 \in gen(\textit{SF-K-RWSTM}), \{T_i, T_j\} \subseteq H1, H2.txns : (H1 \sqsubseteq H2) \wedge (T_j \in H1.committed) \wedge (T_i \in H1.live) \implies (H2.tltl_j < H2.comTime_i)\rangle$.*

Proof. As observed in the previous proof of Lemma 22, if T_i is live or aborted in $H2$, then its comTime is ∞. In both these cases, the result follows.

If T_i is committed in $H2$ then, one can see that comTime of T_i is not ∞. In this case, it can be seen that T_j committed before T_i. Hence, we have that $comTime_j < comTime_i$. From Lemma 22, we get that $tltl_j \leq comTime_j$. This implies that $tltl_j < comTime_i$.
In the following sequence of lemmas, we identify the condition by when a transaction will commit.

Lemma 25 *Consider two histories $H1, H3$ such that $H3$ is a strict extension of $H1$. Let T_i be a transaction in $H1.live$ such that T_i itsEnabled in $H1$ and G_valid_i flag is true in $H1$. Suppose T_i is aborted in $H3$. Then there is a history $H2$ which is an extension of $H1$ (and could be same as $H1$) such that (1) Transaction T_i is live in $H2$; (2) there is a transaction T_j that is live in $H2$; (3) $H2.wts_j$ is greater than $H2.wts_i$; (4) T_j is committed in $H3$. Formally, $\langle H1, H3, T_i : (H1 \sqsubset H3) \wedge (T_i \in H1.live) \wedge (H1.valid_i = True) \wedge (H1.itsEnabled(T_i)) \wedge (T_i \in H3.aborted)) \implies (\exists H2, T_j : (H1 \sqsubseteq H2 \sqsubset H3) \wedge (T_i \in H2.live) \wedge (T_j \in H2.txns) \wedge (H2.wts_i < H2.wts_j) \wedge (T_j \in H3.committed))\rangle$.*

Proof. To show this lemma, w.l.o.g we assume that T_i on executing either read or STM_tryC in $H2$ (which could be same as $H1$) gets aborted resulting in $H3$. Thus, we have that T_i is live in $H2$. Here T_i is itsEnabled in $H1$. From Lemma 20, we get that T_i is itsEnabled in $H2$ as well.

Let us sequentially consider all the lines where a T_i could abort. In $H2$, T_i executes one of the following lines and is aborted in $H3$. We start with STM_tryC method.

1. STM_tryC():

   (a) Line 150 : This line invokes abort() method on T_i which releases all the locks and returns $\mathscr{A}$ to the invoking thread. Here T_i is aborted because its valid flag, is set to false by some other transaction, say T_j, in its STM_tryC algorithm. This can occur in Lines: 192, 221 where T_i is added to T_j's abortRL set. Later in Line 241, T_i's valid flag is set to false. Note that T_i's valid is true (after the execution of the last event) in $H1$. Thus, T_i's valid flag must have been set to false in an extension of $H1$, which we again denote as $H2$.

   This can happen only if in both the above cases, T_j is live in $H2$ and its *its* is less than T_i's *its*. But we have that T_i's itsEnabled in $H2$. As a result, it has the smallest among all live and aborted transactions of $H2$. Hence, there cannot exist such a T_j which is live and $H2.its_j < H2.its_i$. Thus, this case is not possible.

   (b) Line 162: This line is executed in $H2$ if there exists no version of x whose `ts` is less than T_i's *wts*. This implies that all the versions of x have `ts`s greater than wts_i. Thus the transactions that created these versions have *wts* greater than wts_i and have already committed in $H2$. Let T_j create one such version. Hence, we have that $\langle (T_j \in H2.committed) \implies (T_j \in H3.committed)\rangle$ since $H3$ is an extension of $H2$.

   (c) Line 181 : This case is similar to Case 1a, i.e., Line 150.

   (d) Line 194 : In this line, T_i is aborted as some other transaction T_j in T_i's largeRL has committed. Any transaction in T_i's largeRL has *wts* greater than T_i's *wts*. This implies that T_j is already committed in $H2$ and hence committed in $H3$ as well.

(e) Line 211 : In this line, T_i is aborted because its lower limit has crossed its upper limit. First, let us consider $tutl_i$. It is initialized in STM_begin method to ∞. As long as it is ∞, these limits cannot cross each other. Later, $tutl_i$ is updated in Line 134 of read method, Line 205 & Line 209 of STM_tryC method. Suppose $tutl_i$ gets decremented to some value α by one of these lines.

Now there are two cases here: (1) Suppose $tutl_i$ gets decremented to $comTime_i$ due to Line 209 of STM_tryC method. Then from Lemma 22, we have $tltl_i \leq comTime_i = tutl_i$. Thus in this case, T_i will not abort. (2) $tutl_i$ gets decremented to α which is less than $comTime_i$. Then from Lemma 23, we get that there is a committed transaction T_j in $H2.committed$ such that $wts_j > wts_i$. This implies that T_j is in $H3.committed$.

(f) Line 223: This case is similar to Case 1a, i.e., Line 150.

(g) Line 226 : In this case, T_k is in T_i's smallRL and is committed in $H1$. And, from this case, we have that

$$H2.tutl_i \leq H2.tltl_k \tag{3.6}$$

From the assumption of this case, we have that T_k commits before T_i. Thus, from Lemma 24, we get that $comTime_k < comTime_i$. From Lemma 22, we have that $tltl_k \leq comTime_k$. Thus, we get that $tltl_k < comTime_i$. Combining this with the inequality of this case Eq.(3.6), we get that $tutl_i < comTime_i$.

Combining this inequality with Lemma 23, we get that there is a transaction T_j in $H2.committed$ and $H2.wts_j > H2.wts_i$. This implies that T_j is in $H3.committed$ as well.

2. STM_read():

   (a) Line 124: This case is similar to Case 1a, i.e., Line 150

   (b) Line 139: The reasoning here is similar to Case 1e, i.e., Line 211.

The interesting aspect of the above lemma is that it gives us a insight as to when a T_i will get commit. If an itsEnabled transaction T_i aborts then it is because of another transaction T_j with *wts* higher than T_i has committed. To precisely capture this, we define two more notions of a transaction being enabled *cdsEnabled* and *finEnabled*. To define these notions of enabled, we in turn define a few other auxiliary notions. We start with *affectSet*,

$$H.affectSet(T_i) = \{T_j | (T_j \in H.txns) \wedge (H.its_j < H.its_i + 2 * L)\}$$

From the description of *SF-K-RWSTM* algorithm and Lemma 19, it can be seen that a transaction T_i's commit can depend on committing of transactions (or their incarnations) which have their *its* less than *its* of T_i + 2 $*$ L, which is T_i's affectSet. We capture this notion of dependency for a transaction T_i in a history H as *commit dependent set* or *cds* as: the set of all transactions T_j in T_i's affectSet that do not any incarnation that is committed yet, i.e., not yet have their incarCt flag set as true. Formally,

$$H.cds(T_i) = \{T_j | (T_j \in H.affectSet(T_i)) \wedge (\neg H.incarCt(T_j))\}$$

Based on this definition of cds, we next define the notion of cdsEnabled.

Definition 3 *We say that transaction T_i is* cdsEnabled *if the following conditions hold true (1) T_i is live in H; (2) cts of T_i is greater than or equal to its of T_i + 2 $*$ L; (3) cds of T_i is empty, i.e., for all transactions T_j in H with its lower than its of T_i + 2 $*$ L in H have their incarCt to be true. Formally,*

$$H.cdsEnabled(T_i) = \begin{cases} True & (T_i \in H.live) \wedge (H.cts_i \geq H.its_i + 2 * L) \wedge (H.cds(T_i) = \phi) \\ False & \textit{otherwise} \end{cases}$$

The meaning and usefulness of these definitions will become clear in the course of the proof. In fact, we later show that once the transaction T_i is cdsEnabled, it will eventually commit. We will start with a few lemmas about these definitions.

Lemma 26 *Consider a transaction T_i in a history H. If T_i is cdsEnabled then T_i is also itsEnabled. Formally, $\langle H, T_i : (T_i \in H.txns) \wedge (H.cdsEnabled(T_i)) \implies (H.itsEnabled(T_i))\rangle$.*

Proof. If T_i is cdsEnabled in H then it implies that T_i is live in H. From the definition of cdsEnabled, we get that $H.cds(T_i)$ is ϕ implying that any transaction T_j with its_k less than $its_i + 2 * L$ has its incarCt flag as true in H. Hence, for any transaction T_k having its_k less than its_i, $H.incarCt(T_k)$ is also true. This shows that T_i is itsEnabled in H.

Lemma 27 *Consider a transaction T_i which is cdsEnabled in a history $H1$. Consider an extension of $H1$, $H2$ with a transaction T_j in it such that T_i is an incarnation of T_j. Let T_k be a transaction in the affectSet of T_j in $H2$ Then T_k is also in the set of transaction of $H1$. Formally, $\langle H1, H2, T_i, T_j, T_k : (H1 \sqsubseteq H2) \wedge (H1.cdsEnabled(T_i)) \wedge (T_i \in H2.incarSet(T_j)) \wedge (T_k \in H2.affectSet(T_j)) \implies (T_k \in H1.txns)\rangle$*

Proof. Since T_i is cdsEnabled in $H1$, we get (from the definition of cdsEnabled) that

$$H1.cts_i \geq H1.its_i + 2 * L \tag{3.7}$$

Here, we have that T_k is in $H2.affectSet(T_j)$. Thus from the definition of affectSet, we get that

$$H2.its_k < H2.its_j + 2 * L \tag{3.8}$$

Since T_i and T_j are incarnations of each other, their *its* are the same. Combining this with Eq.(3.8), we get that

$$H2.its_k < H1.its_i + 2 * L \tag{3.9}$$

We now show this proof through contradiction. Suppose T_k is not in $H1.txns$. Then there are two cases:

- No incarnation of T_k is in $H1$: This implies that T_k starts afresh after $H1$. Since T_k is not in $H1$, from Corollary 15 we get that

  $H2.cts_k > H1.sys\text{-}time \xrightarrow[H2.cts_k = H2.its_k]{T_k \text{ starts afresh}} H2.its_k > H1.sys\text{-}time \xrightarrow[H1.sys\text{-}time \geq H1.cts_i]{(T_i \in H1) \land Lemma\ 14} H2.its_k > H1.cts_i \xrightarrow{Eq.(3.7)} H2.its_k > H1.its_i + 2 * L \xrightarrow{H1.its_i = H2.its_j} H2.its_k > H2.its_j + 2 * L$

  But this result contradicts with Eq.(3.8). Hence, this case is not possible.

- There is an incarnation of T_k, T_l in $H1$: In this case, we have that

  $$H1.its_l = H2.its_k \tag{3.10}$$

  Now combing this result with Eq.(3.9), we get that $H1.its_l < H1.its_i + 2 * L$. This implies that T_l is in affectSet of T_i in $H1$. Since T_i is cdsEnabled, we get that T_l's incarCt must be true.

  We also have that T_k is not in $H1$ but in $H2$ where $H2$ is an extension of $H1$. Since $H2$ has some events more than $H1$, we get that $H2$ is a strict extension of $H1$.

  Thus, we have that, $(H1 \sqsubset H2) \land (H1.incarCt(T_l)) \land (T_k \in H2.txns) \land (T_k \notin H1.txns)$. Combining these with Lemma 7, we get that $(H1.its_l \neq H2.its_k)$. But this result contradicts Eq.(3.10). Hence, this case is also not possible.

Thus from both the cases we get that T_k should be in $H1$. Hence proved.

Lemma 28 *Consider two histories $H1, H2$ where $H2$ is an extension of $H1$. Let T_i, T_j, T_k be three transactions such that T_i is in $H1.txns$ while T_j, T_k are in $H2.txns$. Suppose we have that (1) cts_i is greater than its_i+2*L in $H1$; (2) T_i is an incarnation of T_j; (3) T_k is in affectSet of T_j in $H2$. Then an incarnation of T_k, say T_l (which could be same as T_k) is in $H1.txns$. Formally, $\langle H1, H2, T_i, T_j, T_k : (H1 \sqsubseteq H2) \land (T_i \in H1.txns) \land (\{T_j, T_k\} \in H2.txns) \land (H1.cts_i > H1.its_i + 2 * L) \land (T_i \in H2.incarSet(T_j)) \land (T_k \in H2.affectSet(T_j)) \implies (\exists T_l : (T_l \in H2.incarSet(T_k)) \land (T_l \in H1.txns))\rangle$*

Proof. This proof is similar to the proof of Lemma 27. We are given that

$$H1.cts_i \geq H1.its_i + 2 * L \tag{3.11}$$

We now show this proof through contradiction. Suppose no incarnation of T_k is in $H1.txns$. This implies that T_k must have started afresh in some history $H3$ which is an extension of $H1$. Also note that $H3$ could be same as $H2$ or a prefix of it, i.e., $H3 \sqsubseteq H2$. Thus, we have that $H3.its_k > H1.sys\text{-}time \xrightarrow{Lemma\ 14} H3.its_k > H1.cts_i \xrightarrow{Eq.(3.11)} H3.its_k > H1.its_i + 2 * L \xrightarrow{H1.its_i = H2.its_j} H3.its_k > H2.its_j + 2 * L \xrightarrow[Observation\ 9]{H3 \sqsubseteq H2} H2.its_k > H2.its_j + 2 * L \xrightarrow[definition]{affectSet} T_k \notin H2.affectSet(T_j)$

But we are given that T_k is in affectSet of T_j in $H2$. Hence, it is not possible that T_k started afresh after $H1$. Thus, T_k must have a incarnation in $H1$.

Lemma 29 *Consider a transaction T_i which is cdsEnabled in a history $H1$. Consider an extension of $H1$, $H2$ with a transaction T_j in it such that T_j is an incarnation of T_i in $H2$. Then affectSet of T_i in $H1$ is same as the affectSet of T_j in $H2$. Formally,* $\langle H1, H2, T_i, T_j : (H1 \sqsubseteq H2) \land (H1.cdsEnabled(T_i)) \land (T_j \in H2.txns) \land (T_i \in H2.incarSet(T_j)) \implies ((H1.affectSet(T_i) = H2.affectSet(T_j)))\rangle$

Proof. From the definition of cdsEnabled, we get that T_i is in $H1.txns$. Now to prove that affectSets are the same, we have to show that $(H1.affectSet(T_i) \subseteq H2.affectSet(T_j))$ and $(H1.affectSet(T_j) \subseteq H2.affectSet(T_i))$. We show them one by one:

$(H1.affectSet(T_i) \subseteq H2.affectSet(T_j))$**:** Consider a transaction T_k in $H1.affectSet(T_i)$. We have to show that T_k is also in $H2.affectSet(T_j)$. From the definition of affectSet, we get that

$$T_k \in H1.txns \tag{3.12}$$

Combining Eq.(3.12) with Observation 9, we get that

$$T_k \in H2.txns \tag{3.13}$$

From the definition of *its*, we get that

$$H1.its_k = H2.its_k \tag{3.14}$$

Since T_i, T_j are incarnations we have that .

$$H1.its_i = H2.its_j \tag{3.15}$$

From the definition of affectSet, we get that,
$H1.its_k < H1.its_i + 2 * L \xrightarrow{Eq.(3.14)} H2.its_k < H1.its_i + 2 * L \xrightarrow{Eq.(3.15)} H2.its_k < H2.its_j + 2 * L$
Combining this result with Eq.(3.13), we get that $T_k \in H2.affectSet(T_j)$.

$(H1.affectSet(T_i) \subseteq H2.affectSet(T_j))$**:** Consider a transaction T_k in $H2.affectSet(T_j)$. We have to show that T_k is also in $H1.affectSet(T_i)$. From the definition of affectSet, we get that $T_k \in H2.txns$.

Here, we have that $(H1 \sqsubseteq H2) \wedge (H1.cdsEnabled(T_i)) \wedge (T_i \in H2.incarSet(T_j)) \wedge (T_k \in H2.affectSet(T_j))$. Thus from Lemma 27, we get that $T_k \in H1.txns$. Now, this case is similar to the above case. It can be seen that Equations 3.12, 3.13, 3.14, 3.15 hold good in this case as well.

Since T_k is in $H2.affectSet(T_j)$, we get that
$H2.its_k < H2.its_i + 2 * L \xrightarrow{Eq.(3.14)} H1.its_k < H2.its_j + 2 * L \xrightarrow{Eq.(3.15)} H1.its_k < H1.its_i + 2 * L$
Combining this result with Eq.(3.12), we get that $T_k \in H1.affectSet(T_i)$.
Next we explore how a cdsEnabled transaction remains cdsEnabled in the future histories once it becomes true.

Lemma 30 *Consider two histories $H1$ and $H2$ with $H2$ being an extension of $H1$. Let T_i and T_j be two transactions which are live in $H1$ and $H2$ respectively. Let T_i be an incarnation of T_j and cts_i is less than cts_j. Suppose T_i is cdsEnabled in $H1$. Then T_j is cdsEnabled in $H2$ as well. Formally, $\langle H1, H2, T_i, T_j : (H1 \sqsubseteq H2) \wedge (T_i \in H1.live) \wedge (T_j \in H2.live) \wedge (T_i \in H2.incarSet(T_j)) \wedge (H1.cts_i < H2.cts_j) \wedge (H1.cdsEnabled(T_i)) \implies (H2.cdsEnabled(T_j))\rangle$.*

Proof. We have that T_i is live in $H1$ and T_j is live in $H2$. Since T_i is cdsEnabled in $H1$, we get (from the definition of cdsEnabled) that

$$H1.cts_i \geq H2.its_i + 2 * L \tag{3.16}$$

We are given that cts_i is less than cts_j and T_i, T_j are incarnations of each other. Hence, we have that

$$\begin{aligned} H2.cts_j &> H1.cts_i \\ &> H1.its_i + 2 * L && \text{[From Eq.(3.16)]} \\ &> H2.its_j + 2 * L && [its_i = its_j] \end{aligned}$$

Thus we get that $cts_j > its_j + 2 * L$. We have that T_j is live in $H2$. In order to show that T_j is cdsEnabled in $H2$, it only remains to show that cds of T_j in $H2$ is empty, i.e., $H2.cds(T_j) = \phi$. The cds becomes empty when all the transactions of T_j's affectSet in $H2$ have their incarCt as true in $H2$.

Since T_j is live in $H2$, we get that T_j is in $H2.txns$. Here, we have that $(H1 \sqsubseteq H2) \wedge (T_j \in H2.txns) \wedge (T_i \in H2.incarSet(T_j)) \wedge (H1.cdsEnabled(T_i))$. Combining this with Lemma 29, we get that $H1.affectSet(T_i) = H2.affectSet(T_j)$.

Now, consider a transaction T_k in $H2.affectSet(T_j)$. From the above result, we get that T_k is also in $H1.affectSet(T_i)$. Since T_i is cdsEnabled in $H1$, i.e., $H1.cdsEnabled(T_i)$ is true, we get that $H1.incarCt(T_k)$ is true. Combining this with Observation 6, we get that T_k must have its incarCt as true in $H2$ as well, i.e. $H2.incarCt(T_k)$. This implies that all the transactions in T_j's affectSet have their incarCt flags as true in $H2$. Hence the $H2.cds(T_j)$ is empty. As a result, T_j is cdsEnabled in $H2$, i.e., $H2.cdsEnabled(T_j)$.

Having defined the properties related to cdsEnabled, we start defining notions for finEnabled. Next, we define *maxWTS* for a transaction T_i in H which is the transaction T_j with the largest *wts* in T_i's incarSet. Formally,

$$H.maxWTS(T_i) = max\{H.wts_j | (T_j \in H.incarSet(T_i))\}$$

From this definition of maxWTS, we get the following simple observation.

Observation 31 *For any transaction T_i in H, we have that wts_i is less than or equal to $H.maxWTS(T_i)$. Formally, $H.wts_i \leq H.maxWTS(T_i)$.*

Next, we combine the notions of affectSet and maxWTS to define *affWTS*. It is the maximum of maxWTS of all the transactions in its affectSet. Formally,

$$H.affWTS(T_i) = max\{H.maxWTS(T_j) | (T_j \in H.affectSet(T_i))\}$$

Having defined the notion of affWTS, we get the following lemma relating the affectSet and affWTS of two transactions.

Lemma 32 *Consider two histories $H1$ and $H2$ with $H2$ being an extension of $H1$. Let T_i and T_j be two transactions which are live in $H1$ and $H2$ respectively. Suppose the affectSet of T_i in $H1$ is same as affectSet of T_j in $H2$. Then the affWTS of T_i in $H1$ is same as affWTS of T_j in $H2$. Formally, $\langle H1, H2, T_i, T_j : (H1 \sqsubseteq H2) \wedge (T_i \in H1.txns) \wedge (T_j \in H2.txns) \wedge (H1.affectSet(T_i) = H2.affectSet(T_j)) \implies (H1.affWTS(T_i) = H2.affWTS(T_j))\rangle$.*

Proof.

From the definition of affWTS, we get the following equations

$$H.affWTS(T_i) = max\{H.maxWTS(T_k) | (T_k \in H1.affectSet(T_i))\} \quad (3.17)$$

$$H.affWTS(T_j) = max\{H.maxWTS(T_l)|(T_l \in H2.affectSet(T_j))\} \tag{3.18}$$

From these definitions, let us suppose that $H1.affWTS(T_i)$ is $H1.maxWTS(T_p)$ for some transaction T_p in $H1.affectSet(T_i)$. Similarly, suppose that $H2.affWTS(T_j)$ is $H2.maxWTS(T_q)$ for some transaction T_q in $H2.affectSet(T_j)$.

Here, we are given that $H1.affectSet(T_i) = H2.affectSet(T_j))$. Hence, we get that T_p is also in $H1.affectSet(T_i)$. Similarly, T_q is in $H2.affectSet(T_j)$ as well. Thus from Equations (3.17) & (3.18), we get that

$$H1.maxWTS(T_p) \geq H2.maxWTS(T_q) \tag{3.19}$$

$$H2.maxWTS(T_q) \geq H1.maxWTS(T_p) \tag{3.20}$$

Combining these both equations, we get that $H1.maxWTS(T_p) = H2.maxWTS(T_q)$ which in turn implies that $H1.affWTS(T_i) = H2.affWTS(T_j)$.
Finally, using the notion of affWTS and cdsEnabled, we define the notion of *finEnabled*

Definition 4 *We say that transaction T_i is* finEnabled *if the following conditions hold true (1) T_i is live in H; (2) T_i is cdsEnabled is H; (3) $H.wts_j$ is greater than $H.affWTS(T_i)$. Formally,*

$$H.finEnabled(T_i) = \begin{cases} True & (T_i \in H.live) \land (H.cdsEnabled(T_i)) \\ & \land (H.wts_j > H.affWTS(T_i)) \\ False & otherwise \end{cases}$$

It can be seen from this definition, a transaction that is finEnabled is also cdsEnabled. We now show that just like itsEnabled and cdsEnabled, once a transaction is finEnabled, it remains finEnabled until it terminates. The following lemma captures it.

Lemma 33 *Consider two histories $H1$ and $H2$ with $H2$ being an extension of $H1$. Let T_i and T_j be two transactions which are live in $H1$ and $H2$ respectively. Suppose T_i is finEnabled in $H1$. Let T_i be an incarnation of T_j and cts_i is less than cts_j. Then T_j is finEnabled in $H2$ as well. Formally, $\langle H1, H2, T_i, T_j : (H1 \sqsubseteq H2) \land (T_i \in H1.live) \land (T_j \in H2.live) \land (T_i \in H2.incarSet(T_j)) \land (H1.cts_i < H2.cts_j) \land (H1.finEnabled(T_i)) \implies (H2.finEnabled(T_j))\rangle$.*

Proof. Here we are given that T_j is live in $H2$. Since T_i is finEnabled in $H1$, we get that it is cdsEnabled in $H1$ as well. Combining this with the conditions given in the lemma statement, we have that,

$$\langle (H1 \sqsubseteq H2) \wedge (T_i \in H1.live) \wedge (T_j \in H2.live) \wedge (T_i \in H2.incarSet(T_j)) \\ \wedge (H1.cts_i < H2.cts_j) \wedge (H1.cdsEnabled(T_i)) \rangle \tag{3.21}$$

Combining Eq.(3.21) with Lemma 30, we get that T_j is cdsEnabled in $H2$, i.e., $H2.cdsEnabled(T_j)$. Now, in order to show that T_j is finEnabled in $H2$ it remains for us to show that $H2.wts_j > H2.affWTS(T_j)$.

We are given that T_j is live in $H2$ which in turn implies that T_j is in $H2.txns$. Thus changing this in Eq.(3.21), we get the following

$$\langle (H1 \sqsubseteq H2) \wedge (T_j \in H2.txns) \wedge (T_i \in H2.incarSet(T_j)) \wedge (H1.cts_i < H2.cts_j) \\ \wedge (H1.cdsEnabled(T_i)) \rangle \tag{3.22}$$

Combining Eq.(3.22) with Lemma 29 we get that

$$H1.affWTS(T_i) = H2.affWTS(T_j) \tag{3.23}$$

We are given that $H1.cts_i < H2.cts_j$. Combining this with the definition of *wts*, we get

$$H1.wts_i < H2.wts_j \tag{3.24}$$

Since T_i is finEnabled in $H1$, we have that
$H1.wts_i > H1.affWTS(T_i) \xrightarrow{Eq.(3.24)} H2.wts_j > H1.affWTS(T_i) \xrightarrow{Eq.(3.23)} H2.wts_j > H2.affWTS(T_j)$
Now, we show that a transaction that is finEnabled will eventually commit.

Lemma 34 *Consider a live transaction T_i in a history $H1$. Suppose T_i is finEnabled in $H1$ and $valid_i$ is true in $H1$. Then there exists an extension of $H1$, $H3$ in which T_i is committed. Formally, $\langle H1, T_i : (T_i \in H1.live) \wedge (H1.valid_i) \wedge (H1.finEnabled(T_i)) \implies (\exists H3 : (H1 \sqsubset H3) \wedge (T_i \in H3.committed)) \rangle$.*

Proof. Consider a history $H3$ such that its sys-time being greater than $cts_i + L$. We will prove this lemma using contradiction. Suppose T_i is aborted in $H3$.

Now consider T_i in $H1$: T_i is live; its valid flag is true; and is finEnabled. From the definition of finEnabled, we get that it is also cdsEnabled. From Lemma 26, we get that T_i is itsEnabled in $H1$. Thus from Lemma 25, we get that there exists an extension of $H1$, $H2$ such that (1) Transaction T_i is live in $H2$; (2) there is a transaction T_j in $H2$; (3) $H2.wts_j$ is greater than $H2.wts_i$; (4) T_j is committed in $H3$. Formally,

$$\langle(\exists H2, T_j : (H1 \sqsubseteq H2 \sqsubset H3) \land (T_i \in H2.live) \land (T_j \in H2.txns) \land (H2.wts_i < H2.wts_j) \\ \land(T_j \in H3.committed))\rangle \tag{3.25}$$

Here, we have that $H2$ is an extension of $H1$ with T_i being live in both of them and T_i is finEnabled in $H1$. Thus from Lemma 33, we get that T_i is finEnabled in $H2$ as well. Now, let us consider T_j in $H2$. From Eq.(3.25), we get that $(H2.wts_i < H2.wts_j)$. Combining this with the observation that T_i being live in $H2$, Lemma 19 we get that $(H2.its_j \leq H2.its_i + 2 * L)$.

This implies that T_j is in affectSet of T_i in $H2$, i.e., $(T_j \in H2.affectSet(T_i))$. From the definition of affWTS, we get that

$$(H2.affWTS(T_i) \geq H2.maxWTS(T_j)) \tag{3.26}$$

Since T_i is finEnabled in $H2$, we get that wts_i is greater than affWTS of T_i in $H2$.

$$(H2.wts_i > H2.affWTS(T_i)) \tag{3.27}$$

Now combining Equations 3.26, 3.27 we get,

$$\begin{aligned} H2.wts_i &> H2.affWTS(T_i) \geq H2.maxWTS(T_j) \\ &> H2.affWTS(T_i) \geq H2.maxWTS(T_j) \geq H2.wts_j \quad \text{[From Observation 31]} \\ &> H2.wts_j \end{aligned}$$

But this equation contradicts with Eq.(3.25). Hence our assumption that T_i will get aborted in $H3$ after getting finEnabled is not possible. Thus T_i has to commit in $H3$.

Next we show that once a transaction T_i becomes itsEnabled, it will eventually become finEnabled as well and then committed. We show this change happens in a sequence of steps. We first show that Transaction T_i which is itsEnabled first becomes cdsEnabled (or gets committed). We next show that T_i which is cdsEnabled becomes finEnabled or get committed. On becoming finEnabled, we have already shown that T_i will eventually commit.

Now, we show that a transaction that is itsEnabled will become cdsEnabled or committed. To show this, we introduce a few more notations and definitions. We start with the notion of *depIts (dependent-its)* which is the set of *its*s that a transaction T_i depends on to commit. It is the set of *its* of all the transactions in T_i's cds in a history H. Formally,

$$H.depIts(T_i) = \{H.its_j | T_j \in H.cds(T_i)\}$$

We have the following lemma on the depIts of a transaction T_i and its future incarnation T_j

which states that depIts of a T_i either reduces or remains the same.

Lemma 35 *Consider two histories $H1$ and $H2$ with $H2$ being an extension of $H1$. Let T_i and T_j be two transactions which are live in $H1$ and $H2$ respectively and T_i is an incarnation of T_j. In addition, we also have that cts_i is greater than $its_i + 2 * L$ in $H1$. Then, we get that $H2.depIts(T_j)$ is a subset of $H1.depIts(T_i)$. Formally, $\langle H1, H2, T_i, T_j : (H1 \sqsubseteq H2) \wedge (T_i \in H1.live) \wedge (T_j \in H2.live) \wedge (T_i \in H2.incarSet(T_j)) \wedge (H1.cts_i \geq H1.its_i + 2 * L) \implies (H2.depIts(T_j) \subseteq H1.depIts(T_i))\rangle$.*

Proof. Suppose $H2.depIts(T_j)$ is not a subset of $H1.depIts(T_i)$. This implies that there is a transaction T_k such that $H2.its_k \in H2.depIts(T_j)$ but $H1.its_k \notin H1.depIts(T_j)$. This implies that T_k starts afresh after $H1$ in some history say $H3$ such that $H1 \sqsubset H3 \sqsubseteq H2$. Hence, from Corollary 15 we get the following
$H3.its_k > H1.sys\text{-}time \xrightarrow{Lemma\ 14} H3.its_k > H1.cts_i \implies H3.its_k > H1.its_i + 2 * L \xrightarrow{H1.its_i = H2.its_j} H3.its_k > H2.its_j + 2 * L \xrightarrow[definitions]{affectSet, depIts} H2.its_k \notin H2.depIts(T_j)$

We started with its_k in $H2.depIts(T_j)$ and ended with its_k not in $H2.depIts(T_j)$. Thus, we have a contradiction. Hence, the lemma follows.
Next we denote the set of committed transactions in T_i's affectSet in H as *cis (commit independent set)*. Formally,

$$H.cis(T_i) = \{T_j | (T_j \in H.affectSet(T_i)) \wedge (H.incarCt(T_j))\}$$

In other words, we have that $H.cis(T_i) = H.affectSet(T_i) - H.cds(T_i)$. Finally, using the notion of cis we denote the maximum of maxWTS of all the transactions in T_i's cis as *partAffWTS* (partly affecting *wts*). It turns out that the value of partAffWTS affects the commit of T_i which we show in the course of the proof. Formally, partAffWTS is defined as

$$H.partAffWTS(T_i) = max\{H.maxWTS(T_j) | (T_j \in H.cis(T_i))\}$$

Having defined the required notations, we are now ready to show that a itsEnabled transaction will eventually become cdsEnabled.

Lemma 36 *Consider a transaction T_i which is live in a history $H1$ and cts_i is greater than or equal to $its_i + 2 * L$. If T_i is itsEnabled in $H1$ then there is an extension of $H1$, $H2$ in which an incarnation T_i, T_j (which could be same as T_i), is either committed or cdsEnabled. Formally, $\langle H1, T_i : (T_i \in H1.live) \wedge (H1.cts_i \geq H1.its_i + 2 * L) \wedge (H1.itsEnabled(T_i)) \implies (\exists H2, T_j : (H1 \sqsubset H2) \wedge (T_j \in H2.incarSet(T_i)) \wedge ((T_j \in H2.committed)$
$\vee (H2.cdsEnabled(T_j)))))\rangle$.*

Proof. We prove this by inducting on the size of $H1.depIts(T_i)$, n. For showing this, we define a boolean function $P(k)$ as follows:

$$P(k) = \begin{cases} True & \langle H1, T_i : (T_i \in H1.live) \wedge (H1.cts_i \geq H1.its_i + 2 * L) \\ & \wedge (H1.itsEnabled(T_i)) \wedge (k \geq |H1.depIts(T_i)|) \implies \\ & (\exists H2, T_j : (H1 \sqsubset H2) \wedge (T_j \in H2.incarSet(T_i)) \\ & \wedge ((T_j \in H2.committed) \vee (H2.cdsEnabled(T_j))))\rangle \\ False & \text{otherwise} \end{cases}$$

As can be seen, here $P(k)$ means that if (1) T_i is live in $H1$; (2) cts_i is greater than or equal to $its_i + 2 * L$; (3) T_i is itsEnabled in $H1$ (4) the size of $H1.depIts(T_i)$ is less than or equal to k; then there exists a history $H2$ with a transaction T_j in it which is an incarnation of T_i such that T_j is either committed or cdsEnabled in $H2$. We show $P(k)$ is true for all (integer) values of k using induction.

Base Case - $P(0)$**:** Here, from the definition of $P(0)$, we get that $|H1.depIts(T_i)| = 0$. This in turn implies that $H1.cds(T_i)$ is null. Further, we are already given that T_i is live in $H1$ and $H1.cts_i \geq H1.its_i + 2 * L$. Hence, all these imply that T_i is cdsEnabled in $H1$.

Induction case - To prove $P(k+1)$ **given that** $P(k)$ **is true:** If $|H1.depIts(T_i)| \leq k$, from the induction hypothesis $P(k)$, we get that T_j is either committed or cdsEnabled in $H2$. Hence, we consider the case when

$$|H1.depIts(T_i)| = k + 1 \tag{3.28}$$

Let α be $H1.partAffWTS(T_i)$. Suppose $H1.wts_i < \alpha$. Then from Lemma 13, we get that there is an extension of $H1$, say $H3$ in which an incarnation of T_i, T_l (which could be same as T_i) is committed or is live in $H3$ and has *wts* greater than α. If T_l is committed then $P(k+1)$ is trivially true. So we consider the latter case in which T_l is live in $H3$. In case $H1.wts_i \geq \alpha$, then in the analysis below follow where we can replace T_l with T_i.

Next, suppose T_l is aborted in an extension of $H3$, $H5$. Then from Lemma 25, we get that there exists an extension of $H3$, $H4$ in which (1) T_l is live; (2) there is a transaction T_m in $H4.txns$; (3) $H4.wts_m > H4.wts_l$ (4) T_m is committed in $H5$.

Combining the above derived conditions (1), (2), (3) with Lemma 22 we get that in $H4$,

$$H4.its_m \leq H4.its_l + 2 * L \tag{3.29}$$

Eq.(3.29) implies that T_m is in T_l's affectSet. Here, we have that T_l is an incarnation of T_i and we are given that $H1.cts_i \geq H1.its_i + 2 * L$. Thus from Lemma 28, we get that there exists an incarnation of T_m, T_n in $H1$.

Combining Eq.(3.29) with the observations (a) T_n, T_m are incarnations; (b) T_l, T_i are incarnations; (c) T_i, T_n are in $H1.txns$, we get that $H1.its_n \leq H1.its_i + 2 * L$. This implies that T_n is in $H1.affectSet(T_i)$. Since T_n is not committed in $H1$ (otherwise, it is not possible for T_m to be an incarnation of T_n), we get that T_n is in $H1.cds(T_i)$. Hence, we get that

$H4.its_m = H1.its_n$ is in $H1.depIts(T_i)$.

From Eq.(3.28), we have that $H1.depIts(T_i)$ is $k + 1$. From Lemma 35, we get that $H4.depIts(T_i)$ is a subset of $H1.depIts(T_i)$. Further, we have that transaction T_m has committed. Thus $H4.its_m$ which was in $H1.depIts(T_i)$ is no longer in $H4.depIts(T_i)$. This implies that $H4.depIts(T_i)$ is a strict subset of $H1.depIts(T_i)$ and hence $|H4.depIts(T_i)| \leq k$.
Since T_i and T_l are incarnations, we get that $H4.depIts(T_i) = H1.depIts(T_l)$. Thus, we get that

$$|H4.depIts(T_i)| \leq k \implies |H4.depIts(T_l)| \leq k \tag{3.30}$$

Further, we have that T_l is a later incarnation of T_i. So, we get that

$$H4.cts_l > H4.cts_i \xrightarrow{given} H4.cts_l > H4.its_i + 2 * L \xrightarrow{H4.its_i = H4.its_l} H4.cts_l > H4.its_l + 2 * L \tag{3.31}$$

We also have that T_l is live in $H4$. Combining this with Equations 3.30, 3.31 and given the induction hypothesis that $P(k)$ is true, we get that there exists a history extension of $H4$, $H6$ in which an incarnation of T_l (also T_i), T_p is either committed or cdsEnabled. This proves the lemma.

Lemma 37 *Consider a transaction T_i in a history $H1$. If T_i is cdsEnabled in $H1$ then there is an extension of $H1$, $H2$ in which an incarnation T_i, T_j (which could be same as T_i), is either committed or finEnabled. Formally,* $\langle H1, T_i : (T_i \in H.live) \wedge (H1.cdsEnabled(T_i)) \implies (\exists H2, T_j : (H1 \sqsubset H2) \wedge (T_j \in H2.incarSet(T_i)) \wedge ((T_j \in H2.committed) \vee (H2.finEnabled(T_j)))\rangle$.

Proof. In $H1$, suppose $H1.affWTS(T_i)$ is α. From Lemma 13, we get that there is a extension of $H1$, $H2$ with a transaction T_j which is an incarnation of T_i. Here there are two cases: (1) Either T_j is committed in $H2$. This trivially proves the lemma; (2) Otherwise, wts_j is greater than α.
In the second case, we get that

$$(T_i \in H1.live) \wedge (T_j \in H2.live) \wedge (H.cdsEnabled(T_i)) \wedge (T_j \in H2.incarSet(T_i)) \wedge (H1.wts_i < H2.wts_j) \tag{3.32}$$

Combining the above result with Lemma 12, we get that $H1.cts_i < H2.cts_j$. Thus the modified equation is

$$(T_i \in H1.live) \wedge (T_j \in H2.live) \wedge (H1.cdsEnabled(T_i)) \wedge (T_j \in H2.incarSet(T_i)) \wedge (H1.cts_i < H2.cts_j) \tag{3.33}$$

Next combining Eq.(3.33) with Lemma 29, we get that

$$H1.affectSet(T_i) = H2.affectSet(T_j) \tag{3.34}$$

Similarly, combining Eq.(3.33) with Lemma 30 we get that T_j is cdsEnabled in $H2$ as well. Formally,

$$H2.cdsEnabled(T_j) \tag{3.35}$$

Now combining Eq.(3.34) with Lemma 32, we get that

$$H1.affWTS(T_i) = H2.affWTS(T_j) \tag{3.36}$$

From our initial assumption we have that $H1.affWTS(T_i)$ is α. From Eq.(3.36), we get that $H2.affWTS(T_j) = \alpha$. Further, we had earlier also seen that $H2.wts_j$ is greater than α. Hence, we have that $H2.wts_j > H2.affWTS(T_j)$.
Combining the above result with Eq.(3.35), $H2.cdsEnabled(T_j)$, we get that T_j is finEnabled, i.e., $H2.finEnabled(T_j)$.
Next, we show that every live transaction eventually become itsEnabled.

Lemma 38 *Consider a history $H1$ with T_i be a transaction in $H1.live$. Then there is an extension of $H1$, $H2$ in which an incarnation of T_i, T_j (which could be same as T_i) is either committed or is itsEnabled. Formally, $\langle H1, T_i : (T_i \in H.live) \implies (\exists T_j, H2 : (H1 \sqsubset H2) \wedge (T_j \in H2.incarSet(T_i)) \wedge (T_j \in H2.committed) \vee (H.itsEnabled(T_i)))\rangle$.*

Proof. We prove this lemma by inducting on *its*.

Base Case - $its_i = 1$: In this case, T_i is the first transaction to be created. There are no transactions with smaller *its*. Thus T_i is trivially itsEnabled.

Induction Case: Here we assume that for any transaction $its_i \leq k$ the lemma is true.

Combining these lemmas gives us the result that for every live transaction T_i there is an incarnation T_j (which could be the same as T_i) that will commit. This implies that every application-transaction eventually commits. The follow lemma captures this notion.

Theorem 39 *Consider a history $H1$ with T_i be a transaction in $H1.live$. Then there is an extension of $H1$, $H2$ in which an incarnation of T_i, T_j is committed. Formally, $\langle H1, T_i : (T_i \in H.live) \implies (\exists T_j, H2 : (H1 \sqsubset H2) \wedge (T_j \in H2.incarSet(T_i)) \wedge (T_j \in H2.committed))\rangle$.*

Proof. Here we show the states that a transaction T_i (or one of it its incarnations) undergoes before it commits. In all these transitions, it is possible that an incarnation of T_i can commit. But to show the worst case, we assume that no incarnation of T_i commits. Continuing with this argument, we show that finally an incarnation of T_i commits.

Consider a live transaction T_i in $H1$. Then from Lemma 38, we get that there is a history $H2$, which is an extension of $H1$, in which T_j an incarnation of T_i is either committed or itsEnabled. If T_j is itsEnabled in $H2$, then from Lemma 36, we get that T_k, an incarnation of T_j, will be cdsEnabled in a extension of $H2$, $H3$ (assuming that T_k is not committed in $H3$).

From Lemma 37, we get that there is an extension of $H3$, $H4$ in which an incarnation of T_k, T_l will be finEnabled assuming that it is not committed in $H4$. Finally, from Lemma 34, we get that there is an extension of $H4$ in which T_m, an incarnation of T_l, will be committed. This proves our theorem.
From this theorem, we get the following corollary which states that any history generated by *SF-K-RWSTM* is $starvation\text{-}freedom$.

Corollary 40 *SF-K-RWSTM algorithm ensures starvation-freedom.*

3.5 Safety Proof of SF-K-RWSTM Algorithm

This section describes the correctness of *SF-K-RWSTM* algorithm with the help of graph characterization. It shows any history generated by *SF-K-RWSTM* algorithm satisfies the correctness criteria as local opacity [8].

Lemma 41 *Consider a history H in $gen($SF-K-RWSTM$)$ with two transactions T_i and T_j such that both their G_valid flags are true. If there is an edge from $T_i \rightarrow T_j$ then $G\_tltl_i < G\_tltl_j$.*

Proof. There are three types of possible edges in $OPG(H, \ll)$ (opacity graph defined in Section 3.2) as follows:

1. Real-time edge: Since, transaction T_i and T_j are in real time order so $comTime_i < G\_cts_j$. Lemma 22 of *SF-K-RWSTM* algorithm ensures that $(G\_tltl_i \leq comTime_i)$. So, $(G\_tltl_i \leq cts_j)$. We know from $STM\_begin(its)$ method, $G\_tltl_j = G\_cts_j$.
Eventually, $G\_tltl_i < G\_tltl_j$.

2. Read-from edge: Since, transaction T_i has been committed and T_j is reading from T_i so, from Line 246 of $STM\_tryC(T_i)$, $G\_tltl_i = \texttt{vrt}_i$ and from Line 137 of $STM\_read(j, x)$, $G\_tltl_j = max(G\_tltl_j, x[curVer].\texttt{vrt}+1) \Rightarrow (G\_tltl_j > \texttt{vrt}_i) \Rightarrow (G\_tltl_j > G\_tltl_i)$
Hence, $G\_tltl_i < G\_tltl_j$.

3. Version-order edge: Consider a triplet $w_j(x_j)r_k(x_j)w_i(x_i)$. It has two possibilities of version order as follows:

(a) i $\ll$ j $\Longrightarrow G\_wts_i < G\_wts_j$

There are two possibilities of commit order:

i. $comTime_i <_H comTime_j$: Since, T_i has been committed before T_j so $G\_tltl_i = \texttt{vrt}_i$. From Line 200 of $STM\_tryC(T_j)$, $\texttt{vrt}_i < G\_tltl(j)$.
Hence, $G\_tltl_i < G\_tltl_j$.

ii. $comTime_j <_H comTime_i$: Since, T_j has been committed before T_i so $G\_tltl_j = \texttt{vrt}_j$. From Line 205 of $STM\_tryC(T_i)$, $G\_tutl_i < \texttt{vrt}_j$. As we have assumed G_valid_i is true so definitely it will execute the Line 231 $STM\_tryC(T_i)$ i.e. $G\_tltl_i = G\_tutl_i$.
Hence, $G\_tltl_i < G\_tltl_j$.

(b) j $\ll$ i $\Longrightarrow G\_wts_j < G\_wts_i$

Again, there are two possibilities of commit order:

i. $comTime_j <_H comTime_i$: Since, T_j has been committed before T_i and T_k read from T_j. There can be two possibilities G_wts_k.

A. $G\_wts_k > G\_wts_i$: That means T_k is in largeRL of T_i. From Line 192 to Line 194 of $STM\_tryC(i)$, either transaction T_k or T_i, G_valid flag is set to be false. If T_i returns abort then this case will not be considered in Lemma 62. Otherwise, as T_j has already been committed and later T_i will execute the Line 246 of $STM\_tryC(T_i)$, Hence, $G\_tltl_j < G\_tltl_i$.

B. $G\_wts_k < G\_wts_i$: That means T_k is in smallRL of T_i. From Line 134 of $read(k, x)$, $G\_tutl_k < \texttt{vrt}_i$ and from Line 137 of $read(k, x)$, $G\_tltl_k > \texttt{vrt}_j$. Here, T_j has already been committed so, $G\_tltl_j = \texttt{vrt}_j$. As we have assumed G_valid_i is true so definitely it will execute the Line 246 $STM\_tryC(T_i)$, $G\_tltl_i = \texttt{vrt}_i$. So, $G\_tutl_k < G\_tltl_i$ and $G\_tltl_k > G\_tltl_j$. While considering G_valid_k flag is true $\rightarrow G\_tltl_k < G\_tutl_k$.
Hence, $G\_tltl_j < G\_tltl_k < G\_tutl_k < G\_tltl_i$.
Therefore, $G\_tltl_j < G\_tltl_k < G\_tltl_i$.

ii. $comTime_i <_H comTime_j$: Since, T_i has been committed before T_j so, $G\_tltl_i = \texttt{vrt}_i$. From Line 205 of $STM\_tryC(T_j)$, $G\_tutl_j < \texttt{vrt}_i$ i.e. $G\_tutl_j < G\_tltl_i$. Here, T_k read from T_j. So, From Line 134 of $read(k, x)$, $G\_tutl_k < \texttt{vrt}_i \rightarrow G\_tutl_k < G\_tltl_i$ from Line 137 of $read(k, x)$, $G\_tltl_k > \texttt{vrt}_j$. As we have assumed G_valid_j is true so definitely it will execute the Line 246 $STM\_tryC(T_j)$, $G\_tltl_j = \texttt{vrt}_j$. Hence, $G\_tltl_j < G\_tltl_k < G\_tutl_k < G\_tltl_i$.
Therefore, $G\_tltl_j < G\_tltl_k < G\_tltl_i$.

Theorem 42 *Any history H, generated by SF-K-RWSTM as gen(SF-K-RWSTM) is local opaque iff for a given version order $\ll$ H, $OPG(H, \ll)$ is acyclic.*

Proof. We are proving it by contradiction, so Assuming $OPG(H, \ll)$ has cycle. From Lemma 62, For any two transactions T_i and T_j such that both their G_valid flags are true and if there is an edge from $T_i \rightarrow T_j$ then $G\_tltl_i < G\_tltl_j$. While considering transitive case for k transactions $T_1, T_2, T_3 ... T_k$ such that G_valid flags of all the transactions are true. if there is an edge from T_1 $\rightarrow T_2 \rightarrow T_3 \rightarrow .... \rightarrow T_k$ then $G\_tltl_1 < G\_tltl_2 < G\_tltl_3 < .... < G\_tltl_k$.
Now, considering our assumption, $OPG(H, \ll)$ has cycle so, $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow .... \rightarrow T_k \rightarrow T_1$ that implies $G\_tltl_1 < G\_tltl_2 < G\_tltl_3 < .... < G\_tltl_k < G\_tltl_1$.
Hence from above assumption, $G\_tltl_1 < G\_tltl_1$ but this is impossible. So, our assumption is wrong.
Therefore, $OPG(H, \ll)$ produced by ***SF-K-RWSTM*** is acyclic.

M_Order$_H$: It stands for method order of history H in which methods of transactions are interval (consists of invocation and response of a method) instead of dot (atomic). Because of having method as an interval, methods of different transactions can overlap. To prove the correctness *(local opacity)* of our algorithm, we need to order the overlapping methods.

Let say, there are two transactions T_i and T_j either accessing common (t-objects/G_lock) or G_tCntr through operations op_i and op_j respectively. If res(op_i) $<_H$ inv(op_j) then op_i and op_j are in real-time order in H. So, the ***M_Order$_H$*** is $op_i \rightarrow op_j$.

If operations are overlapping and either accessing common t-objects or sharing G_lock:

1. $read_i(x)$ and $read_j(x)$: If $read_i(x)$ acquires the lock on x before $read_j(x)$ then the ***M_Order$_H$*** is $op_i \rightarrow op_j$.

2. $read_i(x)$ and $STM\_tryC_j()$: If they are accessing common t-objects then, let say $read_i(x)$ acquires the lock on x before $STM\_tryC_j()$ then the ***M_Order$_H$*** is $op_i \rightarrow op_j$. Now if they are not accessing common t-objects but sharing G_lock then, let say $read_i(x)$ acquires the lock on G_lock_i before $STM\_tryC_j()$ acquires the lock on $relLL$ (which consists of G_lock_i and G_lock_j) then the ***M_Order$_H$*** is $op_i \rightarrow op_j$.

3. $STM\_tryC_i()$ and $STM\_tryC_j()$: If they are accessing common t-objects then, let say $STM\_tryC_i()$ acquires the lock on x before $STM\_tryC_j()$ then the ***M_Order$_H$*** is $op_i \rightarrow op_j$. Now if they are not accessing common t-objects but sharing G_lock then, let say $STM\_tryC_i()$ acquires the lock on $relLL_i$ before $STM\_tryC_j()$ then the ***M_Order$_H$*** is $op_i \rightarrow op_j$.

If operations are overlapping and accessing different t-objects but sharing G_tCntr counter:

1. STM_begin_i and STM_begin_j: Both the STM_begin are accessing shared counter variable G_tCntr. If STM_begin_i executes $G\_tCntr.get\&Inc()$ before STM_begin_j then the ***M_Order$_H$*** is $op_i \rightarrow op_j$.

2. STM_begin_i and $STM\_tryC(j)$: If STM_begin_i executes $G\_tCntr.get\&Inc()$ before $STM\_tryC(j)$ then the ***M_Order$_H$*** is $op_i \rightarrow op_j$.

Linearization [37]: The history generated by STMs are generally not sequintial because operations of the transactions are overlapping. The correctness of STMs is defined on sequintial history, inorder to show history generated by our algorithm is correct we have to consider sequintial history. We have enough information to order the overlapping methods, after ordering the operations will have equivalent sequintial history, the total order of the operation is called linearization of the history.

Operation graph (OpnG): Consider each operation as a vertex and edges as below:

1. Real time edge: If response of operation op_i happen before the invocation of operation op_j i.e. rsp(op_i) $<_H$ inv(op_j) then there exist real time edge between $op_i \rightarrow op_j$.
2. Conflict edge: It is based on L_Order_H which depends on three conflicts:
   (a) Common *t-object*: If two operations op_i and op_j are overlapping and accessing common *t-object x*. Let say op_i acquire lock first on x then $L\_Order.op_i$(x) $<_H$ $L\_Order.op_j$(x) so, conflict edge is $op_i \rightarrow op_j$.
   (b) Common G_valid flag: If two operation op_i and op_j are overlapping but accessing common G_valid flag instead of *t-object*. Let say op_i acquire lock first on G_valid_i then $L\_Order.op_i$(x) $<_H$ $L\_Order.op_j$(x) so, conflict edge is $op_i \rightarrow op_j$.
3. Common G_tCntr counter: If two operation op_i and op_j are overlapping but accessing common G_tCntr counter instead of *t-object*. Let say op_i access G_tCntr counter before op_j then $L\_Order.op_i$(x) $<_H$ $L\_Order.op_j$(x) so, conflict edge is $op_i \rightarrow op_j$.

Lemma 43 *All the locks in history H (L_Order_H) gen(SF-K-RWSTM) follows strict partial order. So, operation graph (OpnG(H)) is acyclic. If ($op_i \rightarrow op_j$) in OpnG, then atleast one of them will definitely true: ($Fpu_i(\alpha)$ < $Lpl\_op_j(\alpha)$) ∪ ($access.G\_tCntr_i$ < $access.G\_tCntr_j$) ∪ ($Fpu\_op_i(\alpha)$ < $access.G\_tCntr_j$) ∪ ($access.G\_tCntr_i$ < $Lpl\_op_j(\alpha)$). Here, α can either be t-object or G_valid.*

Proof. we consider proof by induction, So we assummed there exist a path from op_1 to op_n and there is an edge between op_n to op_{n+1}. As we described, while constructing OpnG(H) we need to consider three types of edges. We are considering one by one:

1. Real time edge between op_n to op_{n+1}:
   (a) op_{n+1} is a locking method: In this we are considering all the possible path between op_1 to op_n:
      i. $(Fu\_op_1(\alpha) < Ll\_op_n(\alpha))$: Here, $(Fu\_op_n(\alpha) < Ll\_op_{n+1}(\alpha))$.
      So, $(Fu\_op_1(\alpha) < Ll\_op_n(\alpha)) < (Fu\_op_n(\alpha) < Ll\_op_{n+1}(\alpha))$
      Hence, $(Fu\_op_1(\alpha) < Ll\_op_{n+1}(\alpha))$

ii. $(Fu\_op_1(\alpha) < Ll\_op_n(\alpha))$: Here, $(access.G\_tCntr_n < Ll\_op_{n+1}(\alpha))$. As we know if any method is locking as well as accessing common counter then locking tobject first then accessing the counter after that unlocking tobject i.e.
So, $(Ll\_op_n(\alpha)) < (access.G\_tCntr_n) < (Fu\_op_n(\alpha))$.
Hence, $(Fu\_op_1(\alpha) < Ll\_op_{n+1}(\alpha))$

iii. $(access.G\_tCntr_1) < (access.G\_tCntr_n)$:
Here, $(access.G\_tCntr_n) < Ll\_op_{n+1}(\alpha))$.
So, $(access.G\_tCntr_1) < (access.G\_tCntr_n) < Ll\_op_{n+1}(\alpha))$.
Hence, $(access.G\_tCntr_1) < Ll\_op_{n+1}(\alpha))$.

iv. $(Fu\_op_1(\alpha) < (access.G\_tCntr_n)$: Here, $(access.G\_tCntr_n) < Ll\_op_{n+1}(\alpha))$.
So, $(Fu\_op_1(\alpha) < (access.G\_tCntr_n) < Ll\_op_{n+1}(\alpha))$.
Hence, $(Fu\_op_1(\alpha) < Ll\_op_{n+1}(\alpha))$

v. $(access.G\_tCntr_1) < Ll\_op_n(\alpha))$: Here, $(Fu\_op_n(\alpha) < Ll\_op_{n+1}(\alpha))$.
So, $(access.G\_tCntr_1) < Ll\_op_n(\alpha)) < (Fu\_op_n(\alpha) < Ll\_op_{n+1}(\alpha))$.
Hence, $(access.G\_tCntr_1) < Ll\_op_{n+1}(\alpha))$.

vi. $(access.G\_tCntr_1) < Ll\_op_n(\alpha))$: Here, $(access.G\_tCntr_n < Ll\_op_{n+1}(\alpha))$. As we know if any method is locking as well as accessing common counter then locking tobject first then accessing the counter after that unlocking tobject i.e. So, $(Ll\_op_n(\alpha)) < (access.G\_tCntr_n) < (Fu\_op_n(\alpha))$.
Hence, $(access.G\_tCntr_1) < Ll\_op_{n+1}(\alpha))$.

(b) op_{n+1} is a non-locking method: Again, we are considering all the possible path between op_1 to op_n:

i. $(Fu\_op_1(\alpha) < Ll\_op_n(\alpha))$: Here, $(access.G\_tCntr_n) < (access.G\_tCntr_{n+1})$. As we know if any method is locking as well as accessing common counter then locking tobject first then accessing the counter after that unlocking tobject i.e. So, $(Ll\_op_n(\alpha)) < (access.G\_tCntr_n) < (Fu\_op_n(\alpha))$.
Hence, $(Fu\_op_1(\alpha) < (access.G\_tCntr_{n+1})$

ii. $(Fu\_op_1(\alpha) < Ll\_op_n(\alpha))$: Here, $(Fu\_op_n(\alpha) < (access.G\_tCntr_{n+1})$.
So, $(Fu\_op_1(\alpha) < Ll\_op_n(\alpha)) < (Fu\_op_n(\alpha) < (access.G\_tCntr_{n+1})$
Hence, $(Fu\_op_1(\alpha) < (access.G\_tCntr_{n+1}))$

iii. $(access.G\_tCntr_1) < (access.G\_tCntr_n)$: Here,
$(access.G\_tCntr_n) < (access.G\_tCntr_{n+1})$.
So, $(access.G\_tCntr_1) < (access.G\_tCntr_n) < (access.G\_tCntr_{n+1})$.
Hence, $(access.G\_tCntr_1) < (access.G\_tCntr_{n+1})$.

iv. $(Fu\_op_1(\alpha) < (access.G\_tCntr_n)$: Here,
$(access.G\_tCntr_n) < (access.G\_tCntr_{n+1})$.
So, $(Fu\_op_1(\alpha) < (access.G\_tCntr_n) < (access.G\_tCntr_{n+1})$.

Hence, $(Fu\_op_1(\alpha) < (access.G\_tCntr_{n+1})$

v. $(access.G\_tCntr_1) < Ll\_op_n(\alpha))$:
Here, $(access.G\_tCntr_n) < (access.G\_tCntr_{n+1})$.
As we know if any method is locking as well as accessing common counter then locking tobject first then accessing the counter after that unlocking tobject i.e.
So, $(Ll\_op_n(\alpha)) < (access.G\_tCntr_n) < (Fu\_op_n(\alpha))$.
Hence, $(access.G\_tCntr_1) < (access.G\_tCntr_{n+1})$.

vi. $(access.G\_tCntr_1) < Ll\_op_n(\alpha))$: Here, $(Fu\_op_n(\alpha) < (access.G\_tCntr_{n+1})$.
So, $(access.G\_tCntr_1) < Ll\_op_n(\alpha)) < (Fu\_op_n(\alpha) < (access.G\_tCntr_{n+1})$.
Hence, $(access.G\_tCntr_1) < (access.G\_tCntr_{n+1})$.

2. Conflict edge between op_n to op_{n+1}:

   (a) $(Fu\_op_1(\alpha) < Ll\_op_n(\alpha))$: Here, $(Fu\_op_n(\alpha) < Ll\_op_{n+1}(\alpha))$. Ref 1.(a).i.

   (b) $(access.G\_tCntr_1) < (access.G\_tCntr_n)$: Here, $(Fu\_op_n(\alpha) < Ll\_op_{n+1}(\alpha))$. As we know if any method is locking as well as accessing common counter then locking tobject first then accessing the counter after that unlocking tobject i.e.
   So, $(Ll\_op_n(\alpha)) < (access.G\_tCntr_n) < (Fu\_op_n(\alpha))$.
   Hence, $(access.G\_tCntr_1) < Ll\_op_{n+1}(\alpha))$.

   (c) $(Fu\_op_1(\alpha) < (access.G\_tCntr_n)$: Here, $(Fu\_op_n(\alpha) < Ll\_op_{n+1}(\alpha))$. As we know if any method is locking as well as accessing common counter then locking tobject first then accessing the counter after that unlocking tobject i.e.
   So, $(Ll\_op_n(\alpha)) < (access.G\_tCntr_n) < (Fu\_op_n(\alpha))$.
   Hence, $(Fu\_op_1(\alpha) < Ll\_op_{n+1}(\alpha))$.

   (d) $(access.G\_tCntr_1) < Ll\_op_n(\alpha))$: Here, $(Fu\_op_n(\alpha) < Ll\_op_{n+1}(\alpha))$.
   Ref 1.(a).v.

3. Common counter edge between op_n to op_{n+1}:

   (a) $(Fu\_op_1(\alpha) < Ll\_op_n(\alpha))$: Here, $(access.G\_tCntr_n) < (access.G\_tCntr_{n+1})$. As we know if any method is locking as well as accessing common counter then locking tobject first then accessing the counter after that unlocking tobject i.e.
   So, $(Ll\_op_n(\alpha)) < (access.G\_tCntr_n) < (Fu\_op_n(\alpha))$.
   Hence, $(Fu\_op_1(\alpha) < (access.G\_tCntr_{n+1})$.

   (b) $(access.G\_tCntr_1) < (access.G\_tCntr_n)$:
   Here, $(access.G\_tCntr_n) < (access.G\_tCntr_{n+1})$. Ref 1.(b).iii.

   (c) $(Fu\_op_1(\alpha) < (access.G\_tCntr_n)$:
   Here, $(access.G\_tCntr_n) < (access.G\_tCntr_{n+1})$. Ref 1.(b).iv.

(d) $(access.G\_tCntr_1) < Ll\_op_n(\alpha))$:
Here, $(access.G\_tCntr_n) < (access.G\_tCntr_{n+1})$. Ref 1.(b).v

Therefore, OpnG(H, M_Order) produced by *SF-K-RWSTM* is acyclic.

Lemma 44 *Any history H, gen(SF-K-RWSTM) with α linearization point such that it respects M_Order_H then (H, α) is valid.*

Proof. From the definition of *valid history*: If all the read operations of H is reading from the previously committed transaction T_j then H is valid.
In order to prove H is valid, we are analyzing the read(i,x). so, from Line 127, it returns the largest `ts` value less than G_wts_i that has already been committed and return the value successfully. If such version created by transaction T_j found then T_i read from T_j. Otherwise, if there is no version whose *wts* is less than T_i's *wts*, then T_i returns abort.
Now, consider the base case read(i,x) is the first transaction T_1 and none of the transactions has been created a version then as we have assummed, there always exist T_0 by default that has been created a version for all t-objects. Hence, T_1 reads from committed transaction T_0.
So, all the reads are reading from largest `ts` value less than G_wts_i that has already been committed. Hence, (H, α) is valid.

Lemma 45 *Any history H gen(SF-K-RWSTM) with α and β linearization such that both respects M_Order_H i.e. $M\_Order_H \subseteq \alpha$ and $M\_Order_H \subseteq \beta$ then $\prec^{RT}_{(H,\alpha)} = \prec^{RT}_{(H,\beta)}$.*

Proof. Consider a history H gen(*SF-K-RWSTM*) such that two transactions T_i and T_j are in real time order which respects M_Order_H i.e. $STM\_tryC_i < STM\_begin_j$. As α and β are linearizations of H so, $STM\_tryC_i <_{(H,\alpha)} STM\_begin_j$ and $STM\_tryC_i <_{(H,\beta)} STM\_begin_j$. Hence in both the cases of linearizations, T_i committed before begin of T_j. So, $\prec^{RT}_{(H,\alpha)} = \prec^{RT}_{(H,\beta)}$.

Lemma 46 *Any history H gen(SF-K-RWSTM) with α and β linearization such that both respects M_Order_H i.e. $M\_Order_H \subseteq \alpha$ and $M\_Order_H \subseteq \beta$ then (H, α) is local opaque iff (H, β) is local opaque.*

Proof. As α and β are linearizations of history H gen(*SF-K-RWSTM*) so, from Lemma 44 (H, α) and (H, β) are valid histories.

Now assuming (H, α) is local opaque so we need to show (H, β) is also local opaque. Since (H, α) is local opaque so there exists legal t-sequential history S (with respect to each aborted transactions and last committed transaction while considering only committed transactions) which is equivalent to ($\overline{H}$, α). As we know β is a linearization of H so ($\overline{H}$, β) is equivalent to some legal t-sequential history S. From the definition of local opacity $\prec^{RT}_{(H,\alpha)} \subseteq \prec^{RT}_{S}$. From Lemma 45, $\prec^{RT}_{(H,\alpha)} = \prec^{RT}_{(H,\beta)}$ that implies $\prec^{RT}_{(H,\beta)} \subseteq \prec^{RT}_{S}$. Hence, (H, β) is local opaque.

Now consider the other way in which (H, β) is local opaque and we need to show (H, α) is also local opaque. We can prove it while giving the same argument as above, by exchanging α and β.

Hence, (H, α) is local opaque iff (H, β) is local opaque.

Theorem 47 *Any history generated by SF-K-RWSTM is locally-opaque.*

Proof. For proving this, we consider a sequential history H generated by *SF-K-RWSTM*. We define the version order $\ll_{\texttt{vrt}}$: for two versions v_i, v_j it is defined as $(v_i \ll_{\texttt{vrt}} v_j) \equiv (v_i.\texttt{vrt} < v_j.\texttt{vrt})$.
Using this version order $\ll_{\texttt{vrt}}$, we can show that all the sub-histories in $H.subhistSet$ are acyclic.
Since the histories generated by *SF-K-RWSTM* are locally-opaque, we get that they are also strict-serializable.

Corollary 48 *Any history generated by SF-K-RWSTM is strict-serializable.*

3.6 Experimental Evaluations

For performance evaluation of *SF-K-RWSTM* with the state-of-the-art STMs, we implemented the algorithms *PKTO*, *SF-SV-RWSTM* along with *SF-K-RWSTM* in C++ [2]. We used the available implementations of NOrec STM [24], and ESTM [25] developed in C++ from TLDS framework[3]. Although, only *SF-K-RWSTM* and *SF-SV-RWSTM* provide starvation-freedom, we compared with other RWSTMs as well, to see its performance in practice.
Experimental system: The experimental system is a 2-socket Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz with 14 cores per socket and 2 hyper-threads (HTs) per core, for a total of 56 threads. Each core has a private 32KB L1 cache and 256 KB L2 cache. The machine has 32GB of RAM and runs Ubuntu 16.04.2 LTS. In our implementation, all threads have the same base priority and we use the default Linux scheduling algorithm. This satisfies the Assumption 1 (bounded-termination) about the scheduler. We ensured that there is no parasitic transactions [41] in our experiments.
Methodology: Here we have considered two different applications:**(1)** Counter application (described in SubSection 3.6.1) - In this, each thread invokes a single transaction which performs 10 reads/writes operations on randomly chosen t-objects. A thread continues to invoke a transaction until it successfully commits. To obtain high contention, we have taken large number of threads ranging from 50-250 where each thread performs its read/write operation over a set

[2]Code is available here: https://github.com/PDCRL/KSFTM
[3]TLDS Framework: https://ucf-cs.github.io/tlds/

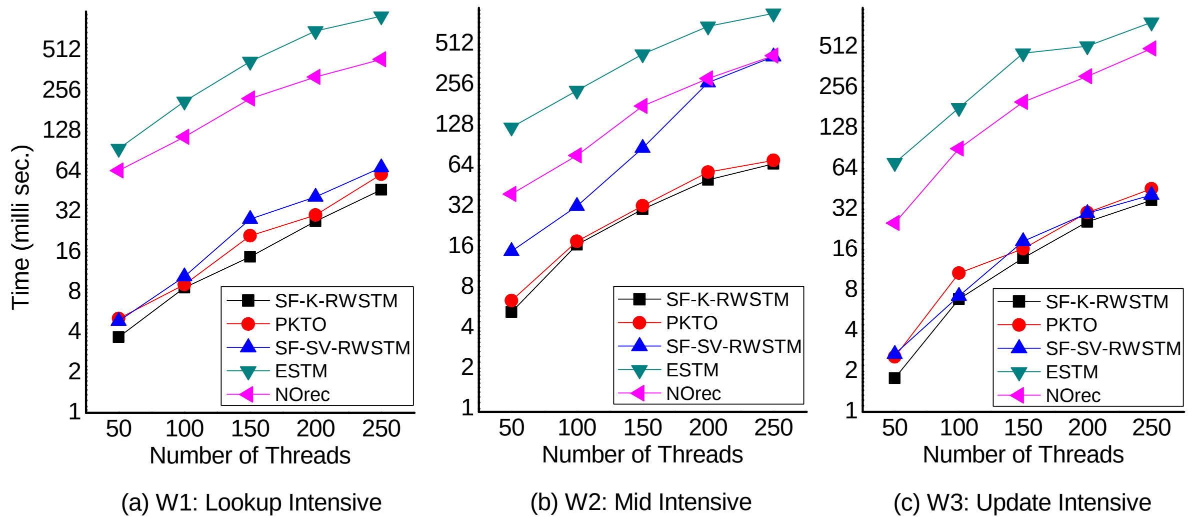


Figure 3.6: Performance analysis on workload $W1$, $W2$, and $W3$

of 5 t-objects. We have performed our tests on three workloads stated as: (W1) Li - Lookup intensive: 90% read, 10% write, (W2) Mi - Mid intensive: 50% read, 50% write, and (W3) Ui - Update intensive: 10% read, 90% write. Counter application is undoubtedly very flexible as it allows us to examine performance by tweaking different parameters. **(2)** Two benchmarks from STAMP suite [26] - (a) We considered KMEANS which has low contention with short running transactions. The number of data points as 2048 with 16 dimensions and total clusters as 5. (b) We then considered LABYRINTH which has high contention with long running transactions. We considered the grid size as 64x64x3 and paths to route as 48.

To study starvation in the various algorithms, we considered *max-time*, which is the maximum time taken by a transaction among all the transactions in a given experiment to commit from its first invocation. This includes time taken by all the aborted incarnations of the transaction to execute as well. For accuracy, all the experiments are averaged over 11 runs in which the first run is discarded and considered as a warm-up run.

3.6.1 Pseudocode of Counter Application

To analyze the absolute benefit of starvation-freedom, we use a *Counter Application* which provides us the flexibility to create a high contention environment where the probability of transactions undergoing starvation on an average is very high. In this subsection we described the detailed functionality of *Counter Application* though pseudocode as follows:

Algorithm 16 *main()*: The main function invoked by *Counter Application*.

```
264: /*Each thread $th_i$ log abort counts, average time taken by each transaction to commit and worst case time (maximum time to commit the transaction) in $abortCount_{th_i}$, $timeTaken_{th_i}$ and $worstTime_{th_i}$ respectively;*/
```

```
for all (numOfThreads) do /*Multiple threads call the helper function*/
    helperFun();
end for
for all (numOfThreads) do
    /*Join all the threads*/
end for
for all (numOfThreads) do
    if ($maxWorstTime < worstTime_{th_i}$) then
        /*Calculate the Maximum Worst Case Time*/
        $maxWorstTime$ = $worstTime_{th_i}$;
    end if
    /*Calculate the Total Abort Count*/
    $totalAbortCount$ += $abortCount_{th_i}$;
    /*Calculate the Average Time Taken*/
    $AvgTimeTaken$ /= $TimeTaken_{th_i}$;
end for
```

Algorithm 17 *helperFun()*:Multiple threads invoke this function.

```
Initialize the Transaction Count $txCount_i$ of $T_i$ as 0;
while (numOfTransactions) do /*Execute until number of transactions are non zero*/
    $startTime_{th_i}$ = timeRequest(); /*get the start time of thread $th_i$*/
    /*Execute the transactions $T_i$ by invoking testSTM functions;*/
    $abortCount_{th_i}$ = $testSTM_i()$;
    Increment the $txCount_i$ of $T_i$ by one.
    $endTime_{th_i}$ = timeRequest(); /*get the end time of thread $th_i$*/
    /*Calculate the Total Time Taken by each thread $th_i$*/
    $timeTaken_{th_i}$ += ($endTime_{th_i}$ - $startTime_{th_i}$);
    /*Calculate the Worst Case Time taken by each thread $th_i$*/
    if ($worstTime_{th_i} <$ ($endTime_{th_i}$ - $startTime_{th_i}$)) then
        $worstTime_{th_i}$ = ($endTime_{th_i}$ - $startTime_{th_i}$);
    end if
    Atomically, decrement the numOfTransactions;
end while
/*Calculate the Average Time taken by each thread $th_i$*/
$TimeTaken_{th_i}$ /= $txCount_i$;
```

Algorithm 18 $testSTM_i()$: Main function which executes the methods of the transaction T_i (or i) by thread th_i.

```
298: while (true) do
299:     if (i.its != nil) then
300:         STM_begin(i.its); /*If $T_i$ is an incarnation*/
301:     else
302:         STM_begin(nil); /*If $T_i$ is first invocation*/
303:     end if
304:     for all (numOfMethods) do
305:         $k_i$ = rand()%totalKeys;/*Select the key randomly*/
306:         $m_i$ = rand()%100;/*Select the method randomly*/
307:         switch ($m_i$) do
308:             case ($m_i \leq$ STM_read()):
309:                 $v \leftarrow$ STM_read($k_i$); /*Read key $k$ from a shared memory*/
310:                 if ($v$ == $abort$) then
311:                     $txAbortCount_i++$; /*Increment the transaction abort count*/
312:                     goto Line 282;
313:                 end if
314:             case ($m_i \leq$ STM_write()):
315:                 /*Write key $k_i$ into $T_i$ local memory with value $v$*/
316:                 /*Actual write happens after successful STM_tryC ()*/
317:                 STM_write ($k_i, v$);
318:             case default:
319:                 /*Neither lookup nor insert/delete on shared memory*/
320:             $v$ = STM_tryC(); /*Validate all the methods of $T_i$ in tryC*/
321:             if ($v$ == abort) then
322:                 $txAbortCount_i++$;
323:                 goto Line 282;
324:             end if
325:     end for
326:     return $\langle txAbortCount_i \rangle$;
327: end while
```

3.6.2 Result Analysis

This subsection represents the result analysis of proposed *SF-K-RWSTM* with state-of-the-art STMs on various workloads of counter application (described in SubSection 3.6.1) and STAMP benchmark [26].

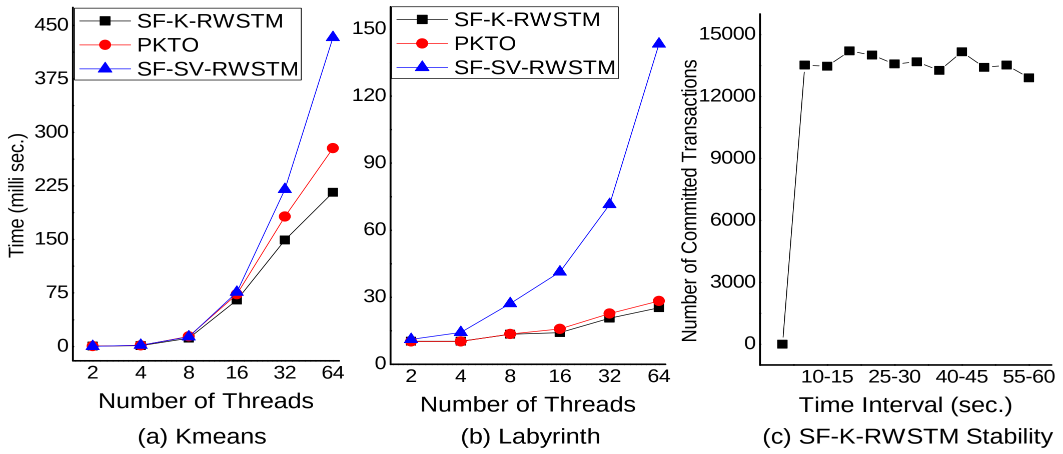


Figure 3.7: Performance analysis on KMEANS, LABYRINTH and *SF-K-RWSTM* Stability

Results Analysis for Counter Application: Figure 3.6 illustrates max-time analysis of *SF-K-RWSTM* over the above mentioned STMs for the counters application under the workloads $W1$, $W2$ and $W3$ while varying the number of threads from 50 to 250. For *SF-K-RWSTM* and *PKTO*, we chose the value of K as 5 and C as 0.1 as the best results were obtained with these parameters. The value of K is application dependent. We can see that *SF-K-RWSTM* performs the best for all the three workloads. *SF-K-RWSTM* gives an average speedup on max-time by a factor of 1.22, 1.89, 23.26 and 13.12 over *PKTO*, *SF-SV-RWSTM*, NOrec STM [24], and ESTM [25] respectively.

Results Analysis for STAMP Benchmark: Figure 3.7(a) shows the experimental analysis of max-time for KMEANS while Figure 3.7(b) shows for LABYRINTH applications from STAMP benchmark [26]. In this analysis we have not considered ESTM as the integrated STAMP code for ESTM is not publicly available. For KMEANS, *SF-K-RWSTM* performs 1.5 and 1.44 times better than *PKTO* and *SF-SV-RWSTM*. But, NOrec is performing 1.09 times better than *SF-K-RWSTM*. This is because KMEANS has short running transactions with low contention. As a result, the commit time of the transactions is also low.

On the other hand for LABYRINTH, *SF-K-RWSTM* again performs the best. It performs 1.14, 1.4 and 2.63 times better than *PKTO*, *SF-SV-RWSTM*, and NOrec [24] respectively. This is because LABYRINTH has high contention with long running transactions. This result in longer commit times for transactions.

Figure 3.7(c) shows the stability of *SF-K-RWSTM* algorithm over time for the counter application. Here we fixed the number of threads to 32, K as 5, C as 0.1, and t-objects as 1000 on $W1$ workload. Each thread invokes transactions until its time-bound of 60 seconds expires. We performed the experiments on number of transactions committed over time in the increments 5 seconds. The experiment shows that over time *SF-K-RWSTM* is stable which helps to hold the claim that the performance of *SF-K-RWSTM* will continue in same manner if time is increased to higher orders.

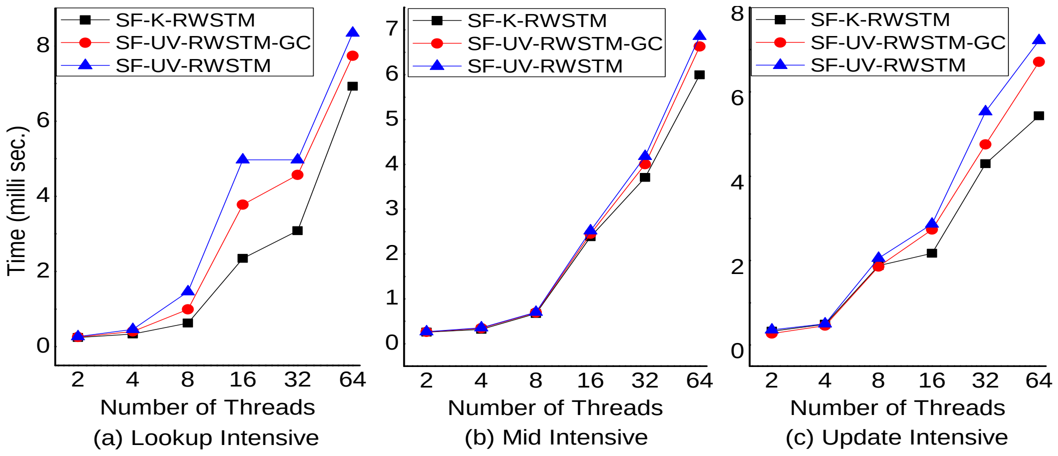


Figure 3.8: Time comparison among variants of *SF-K-RWSTM*

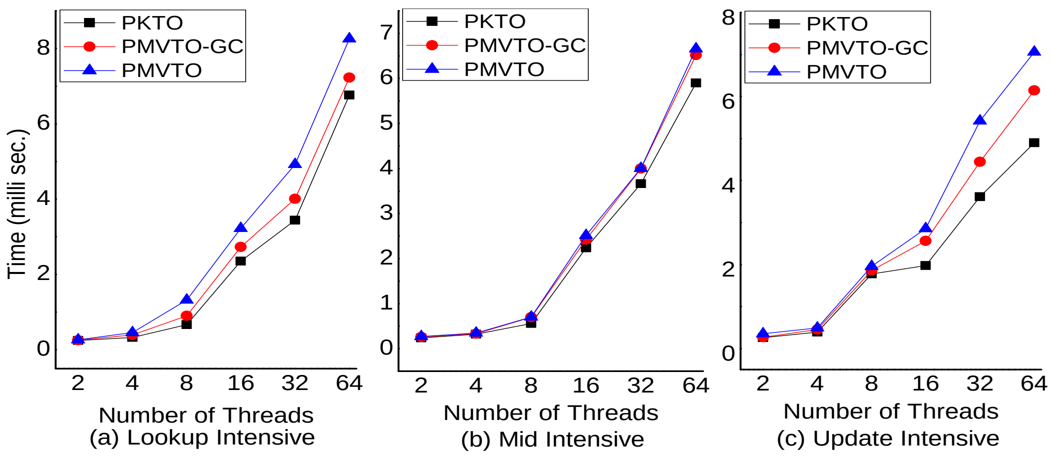


Figure 3.9: Time comparison among variants of *PKTO*

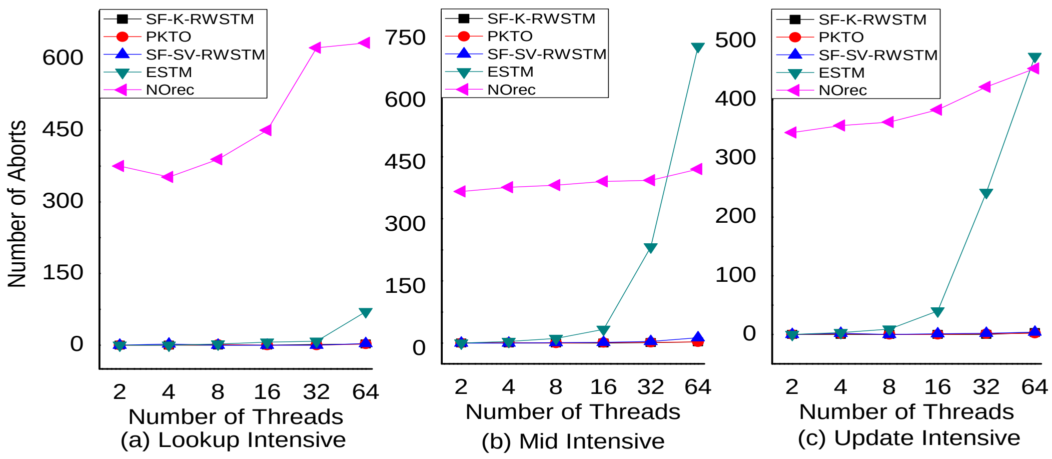


Figure 3.10: Abort Count on workload $W1$, $W2$, and $W3$

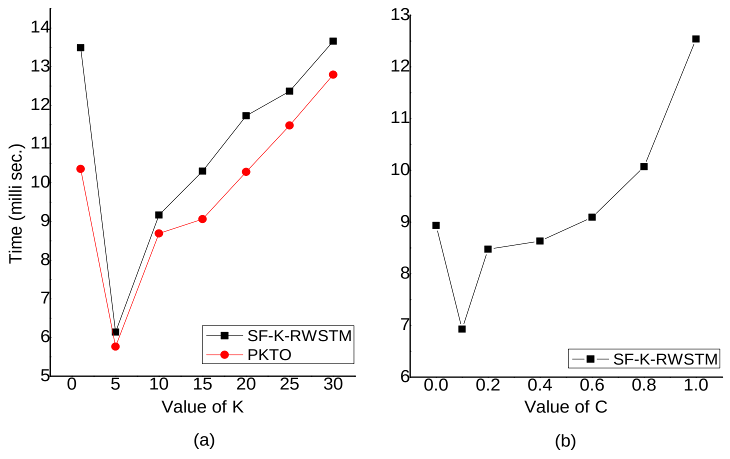


Figure 3.11: Best value of K and optimal value of C for *SF-K-RWSTM*

Figure 3.8 represents three variants of *SF-K-RWSTM* (*SF-UV-RWSTM*, *SF-UV-RWSTM-GC*, and *SF-K-RWSTM*) and Figure 3.9 shows the three variants of *PKTO* (*PMVTO*, *PMVTO-GC*, and *PKTO*) on all the workloads $W1$ $W2$ and $W3$. *SF-K-RWSTM* outperforms *SF-UV-RWSTM* and *SF-UV-RWSTM-GC* by a factor of 2.1 and 1.5. Similarly, *PKTO* outperforms *PMVTO* and *PMVTO-GC* by a factor of 2 and 1.35. These results show that maintaining finite versions corresponding to each t-object performs better than maintaining infinite versions and garbage collection on infinite versions corresponding to each t-object.

Comparison on the basis of Abort count: Figure 3.10 demonstrates the abort count comparisons of *SF-K-RWSTM* with *PKTO*, ESTM [25], NOrec [24], MVTO [10], and *SF-SV-RWSTM* across all workloads ($W1$, $W2$, and $W3$). The number of aborts in ESTM and NOrec are high as compared to all other STM algorithms while all other algorithms (*SF-K-RWSTM*, *PKTO*, MVTO, *SF-SV-RWSTM*) have marginally small differences among them.

Best value of K and optimal value of constant C: To identify the best value of K for *SF-K-RWSTM*, we ran our experiment, varying value of K and keeping the number of threads as 64 on workload $W1$ and obtained the optimal value of K in *SF-K-RWSTM* is 5 as shown in Figure 3.11.(a) for counter application. Similarly, we calculate the best value of K as 5 for *PKTO* on the same parameters (the value of K is application dependent). C, is a constant that is used to calculate wts of a transaction. i.e., $wts_i = cts_i + C * (cts_i - its_i)$; where, C is any constant greater than 0. We run the experiments across workload $W1$ with 64 threads. We obtained the best value of C as 0.1 for counter application as illustrated in Figure 3.11 (b).

3.7 Summary

In this chapter of the thesis, we proposed *SF-K-RWSTM* which ensures starvation-freedom while maintaining K-versions for each t-objects. It uses two insights to ensure starvation-freedom in the context of MV-RWSTMs: (1) using *its* to ensure that older transactions are given a higher priority, and (2) using *wts* to ensure that conflicting transactions do not commit too quickly before the older transaction could commit. We proved that *SF-K-RWSTM* satisfies strict-serializability [5] and local opacity [8, 34]. To the best of our knowledge, this is the first work to explore *starvation-freedom* with MV-RWSTMs.

Our experiments show that *SF-K-RWSTM* performs better than single-version RWSTMs (ESTM, Norec STM) under high contention and also single-version *starvation-free* RWSTM *SF-SV-RWSTM* developed based on the principle of priority. On the other hand, its performance is comparable or slightly worse than multi-version RWSTM, *PKTO* (around 2%). This is the cost of the overhead required to achieve *starvation-freedom* which we believe is a marginal price.

In this document, we have not considered a transactional solution based on two-phase locking (2PL) and its multi-version variants [2]. With the carefully designed 2PL solution, one can ensure that none of the transactions abort [2]. But this will require advance knowledge of the code of the transactions which may not always be available with the STM library. Without such knowledge, it is possible that a 2PL solution can deadlock and cause further aborts which will, raise the issue of *starvation-freedom* again.

Chapter 4

Exploring Starvation-Freedom in Single-Version and Multi-Version OSTMs

4.1 Introduction

To utilize the multi-core processors properly concurrent programming is needed. The main challenge is to design a correct and efficient concurrent program. *Software Transactional Memory systems (STMs)* [3, 4] provide ease of multithreading to the programmer without worrying about concurrency issues as deadlock, livelock, priority inversion, etc. Most of the STMs work on read-write operations known as *Read-Write STMs (or RWSTMs)*.

Some *Software Transactional Memory Systems (STMs)* work at higher-level operations [9, 14, 15] instead of low-level operations (read and write) and ensure greater concurrency than MV-RWSTMs and SV-RWSTMs. They include more semantically rich operations such as push/pop on stack objects, enqueue/dequeue on queue objects and insert/lookup/delete on sets, trees or hash table objects depending upon the underlying data structure used to implement higher-level systems. Such STMs are known as *Single-Version Object-based STMs (SV-OSTMs or OSTMs)*. Some conflicts of RWSTMs do not matter at SV-OSTMs which reduce the number of aborts and improve the concurrency using SV-OSTMs. Figure 1.2 of SubSection 1.2.2 illustrates the advantages of SV-OSTMs over RWSTMs with an interesting example.

A typical SV-OSTM system exports the following methods: (1) *STM_begin()*: begins a transaction T_i with unique timestamp i same as RWSTMs. (2) *STM_lookup$_i$*(k) (or $l_i(k)$): T_i lookups key k from shared memory and returns the value. (3) *STM_insert$_i$*(k, v) (or $i_i(k, v)$): T_i inserts a key k with value v into its local memory. (4) *STM_delete$_i$*(k)(or $d_i(k)$): T_i deletes key k. (5) *STM_tryC$_i$() or (tryC$_i$())*: the actual effect of *STM_insert()* and *STM_delete()* will be visible to the shared memory after successful validation and T_i returns commit otherwise (6) *STM_tryA$_i$()*: T_i returns abort.

The transactions of SV-OSTMs can return $commit$ or $abort$. Aborted SV-OSTMs transaction retry. But in the current setting of SV-OSTMs, transactions may starve. So, we propose a novel *Starvation-Freedom in SV-OSTM as SF-SV-OSTM* which assigns the priority to transaction on abort. Whenever a conflicting transaction T_i aborts in SF-SV-OSTM, it retries with transaction T_j which has higher priority than T_i. To ensure the starvation-freedom, this procedure repeats until T_i gets the highest priority and eventually returns commit. SF-SV-OSTM ensures starvation-freedom and satisfies the correctness criteria *conflict-opacity* [9].

Motivation to Propose Starvation-Freedom in Multi-Version OSTM System: If the highest priority transaction becomes slow (for some reason) in SF-SV-OSTM then it may cause several other transactions to abort and bring down the progress of the system. Database, RWSTMs [10–13] and OSTMs [16] say that maintaining multiple versions corresponding to each key reduces the number of aborts and improves throughput.

So, in this chapter of the thesis, we propose the novel and efficient *Starvation-Free Multi-Version OSTM (SF-MV-OSTM)* which maintains multiple versions corresponding to each key. Figure 1.7 of SubSection 1.4.2 demonstrates the benefits of using SF-MV-OSTM over SF-SV-OSTM with an interesting example. It shows that SF-MV-OSTM system improves the concurrency than SF-SV-OSTM system while reducing the number of aborts and ensures the starvation-freedom.

SF-MV-OSTM system works for unbounded versions with *Garbage Collection (GC)* as SF-MV-OSTM-GC which deletes the unwanted versions from version list of keys and for bounded/finite versions as SF-K-OSTM which stores finite say latest K number of versions corresponding to each key k. So, whenever any thread creates $(K+1)^{th}$ version of key, it replaces the oldest version of it. The most challenging task is achieving starvation-freedom in bounded version OSTM because say, a highest priority transaction rely on the oldest version that has been replaced. So, in this case highest priority transaction has to return abort and hence make it harder to achieve starvation-freedom. Hence, this chapter of the thesis bridges the gap by developing starvation-free OSTM while maintaining bounded number of versions.

We proposed SF-SV-OSTM and SF-MV-OSTM for hash table and linked-list data structure but it is generic for other data structures as well. SF-K-OSTM is best among all proposed Starvation-Free OSTMs (SF-SV-OSTM, SF-MV-OSTM, and SF-MV-OSTM-GC) for both hash table and linked-list data structures. Our experimental analysis demonstrates that proposed hash table based SF-K-OSTM (HT-SF-K-OSTM) achieved an average speedup of 3.9x, 32.18x, 22.67x, 10.8x and 17.1x on *max-time* (maximum time for a transaction to commit) over state-of-the-art STMs, HT-K-OSTM [16], HT-SV-OSTM [9], ESTM [25], RWSTM [2, Chap. 4], and HT-MVTO [10] respectively on various workloads (W1 (Lookup Intensive - 5% insert, 5% delete, and 90% lookup), W2 (Mid Intensive - 25% insert, 25% delete, and 50% lookup), and W3 (Update Intensive - 45% insert, 45% delete, and 10% lookup)). It also illustrates that proposed list based SF-K-OSTM (list-SF-K-OSTM) performs 2.4x, 10.6x, 7.37x, 36.7x, 9.05x, 14.47x, and 1.43x average speedup on *max-time* than state-of-the-art STMs, list-K-OSTM [16], list-SV-OSTM [9], Trans-list [27], Boosting-list [14], NOrec-list [24], list-MVTO [10], and list-SF-K-RWSTM [23] (proposed by us in Chapter 3) respectively on various workloads W1, W2, and W3.

Roadmap: Section 4.2 describes the graph characterization of conflict-opacity [9] which helps to prove the correctness of proposed algorithms. Initially, we propose Starvation-Freedom in Single-Version OSTM as SF-SV-OSTM for *hash table* and *linked-list* data structure describe in SubSection 4.3.2 but it is generic for other data structures as well. Section 4.4 and Section 4.5 shows the liveness and safety proof of SF-SV-OSTM. To achieve the greater concurrency further, we propose Starvation-Freedom for Multi-Version OSTM as SF-K-OSTM in Section 4.6 which maintains K number of versions corresponding to each key and satisfies the correctness criteria as *local opacity* [34]. We propose SF-K-OSTM for *hash table* and *linked-list* data structure describe in SubSection 4.6.1 but it is generic for other data structures as well. Section 4.7 describes the graph characterization of local opacity. Section 4.8 and Section 4.9 shows the liveness and safety proof of SF-K-OSTM. Section 4.10 shows that SF-K-OSTM is best among all proposed Starvation-Free OSTMs (SF-SV-OSTM, SF-MV-OSTM, and SF-MV-OSTM-GC) and state-of-the-art STMs for both hash table and linked-list data structures. Section 4.11 gives a summary of this chapter.

4.2 Graph Characterization of Co-opacity

This section describes the graph characterization of the history H which helps to prove the correctness of STMs. We follow the graph characterization by Guerraoui and Kapalka [35] and modify it for sequential histories with high-level methods.

A graph for conflict-opacity (co-opacity) is represented as $CG(H, \ll) = (V, E)$ which consists of V vertices and E edges. Here, each committed transaction is consider as a vertex and edges are as follows:

- *Conflict (or Conf) edge*: The conflict edges between two transactions depends on the conflicts between them. Two transactions T_i and T_j of the sequential history are said to be in conflict, if one of the following holds:
  - **tryC-tryC** conflict: Two transactions T_i & T_j are in tryC-tryC conflict (1) If T_i and T_j are committed; (2) Both T_i & T_j update the same key k of the hash table, ht, i.e., $(\langle ht, k\rangle \in updtSet(T_i)) \wedge (\langle ht, k\rangle \in updtSet(T_j))$, here $updtSet(T_i)$ is set of keys in which T_i performs update methods (or $upd\_methods$); (3) and STM_tryC() of T_i has completed before STM_tryC() of T_j, i.e., $STM\_tryC_i() \prec_H^{max\text{-}r} STM\_tryC_j()$.
  - **tryC-rv** conflict: Two transactions T_i & T_j are in tryC-rv conflict (1) If T_i has updated the key k of hash table, ht and committed; (2) After that T_j invokes a rv_method rvm_j on the key same k of hash table ht and returns the value updated by T_i, i.e., $STM\_tryC_i() \prec_H^{max\text{-}r} rvm_j$.

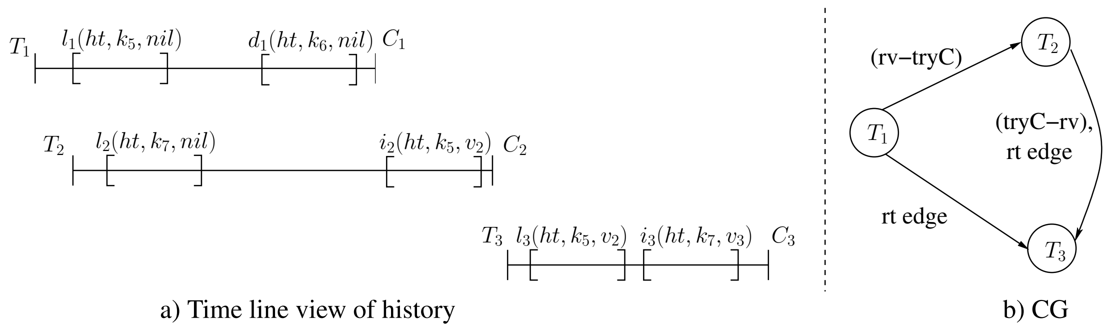


Figure 4.1: Illustration of Graph Characterization of Co-opacity

- **rv-tryC** conflict: Two transactions T_i & T_j are in rv-tryC conflict (1) T_i invokes a rv_method rvm_i on the key k of hash table ht and returns the value updated by T_k, i.e., $STM\_tryC_k() \prec_H^{max\text{-}r}$ rvm_i; (2) After that T_j update the same key k of the hash table, ht, i.e., $(\langle ht, k\rangle \in updtSet(T_j))$ and T_j returns commit, i.e., $rvm_i \prec_H^{max\text{-}r} STM\_tryC_j()$.

If any of the above defined conflicts occur then conflict edge goes from T_i to T_j. As described in Chapter 2, *STM_lookup()*, and *STM_delete()* return the value from underlying data structure so, we called these methods as *return value methods (or $rv\_methods$)*. Whereas, *STM_insert()*, and *STM_delete()* are updating the underlying data structure after successful *STM_tryC()* so, we called these as *update* methods (or $upd\_methods$). So, the conflicts are defined between the methods that accesses 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$_j$())* are the possible conflicting methods.

- *real-time (or real-time) edge*: If transaction T_i returns commit before the beginning of other transaction T_j then real-time edge goes from T_i to T_j. Formally, ($STM\_tryC_i() \prec_H$ *STM_begin$_j$()*) $\implies T_i \rightarrow T_j$.

For better understanding, we consider a history H: $l_1(ht, k_5, nil), l_2(ht, k_7, nil), d_1(ht, k_6, nil), C_1, i_2(ht, k_5, v_2), C_2, l_3(ht, k_5, v_2), i_3(ht, k_7, v_3), C_3$ and show the time line view of it in Figure 4.1.(a). We construct $CG(H, \ll) = (V, E)$ shown in Figure 4.1.(b). There exist a (rv-tryC) edge between T_1 to T_2 because T_2 updates the key k_5 with value v_2 after T_1 does a lookup on it. T_3 begins after the commit of T_1 and T_2 so, real-time edges are going from T_1 to T_3 and T_2 to T_3. Here, T_3 does a lookup on key k_5 after it is updated by T_2 and returns the value v_2. So, (tryC-rv) edge is going from T_2 to T_3. Hence, H constructs an acyclic graph (CG) with equivalent serial schedule $T_1T_2T_3$.

Lemma 49 *For any legal t-sequential history S the conflict graph $CG(S, \ll_S,)$ is acyclic.*

Proof. The t-sequential history S consists of multiple transactions, we order all the them into real-time order on the basis of their increasing order of timestamp (TS). For example, consider two transaction T_i and T_j with $TS(T_i)$ is less than $TS(T_j)$ then T_i will occur before T_j in S. Formally, $TS(T_i) < TS(T_j) \Leftrightarrow T_i <_S T_j$. To prove the order between transactions, we analyze all the edges of $CG(S, \ll_S,)$ one by one:

- real-time edges: It follows that any transaction begins after the commit of the previous transaction only. Hence, all the real-time edges go from a lower TS transaction T_i to higher TS transaction T_j and follow timestamp order.
- Conf edges: If any transaction T_j lookups key k from T_i in S then T_i has to be committed before invoking of lookup of T_j. Similarly, other conflicting edges are following TS order as $TS(T_i) < TS(T_j) \Leftrightarrow T_i <_S T_j$. Thus, all the Conf edges go from a lower TS transaction to higher TS transaction.

Hence, all the edges of $CG(S, \ll_S,)$ are following increasing of the TS of the transactions, i.e. all the edges goes from lower TS transaction to higher TS transaction in S. Conflict graph $CG(S, \ll_S,)$ is acyclic.

Theorem 50 *A history H is co-opaque iff $CG(H, \ll_H)$ is acyclic.*

Proof. (if part): First, we consider $CG(H, \ll_H)$ is acyclic and we need to prove that history H is co-opaque. Since $CG(H, \ll_H)$ is acyclic, we apply topological sort on $CG(H, \ll_H)$ and generate a t-sequential history S such that S is equivalent to $\overline{H}$. $CG(H, \ll_H)$ maintains real-time edges as well and S has been generated from it. So, S also respect real-time order real-time as H. Formally, $\prec_H^{RT} \subseteq \prec_S^{RT}$.

Since $CG(H, \ll_H)$ maintains all the conflicting (or Conf) edges as well defined above. S has been generated by applying topological sort on $CG(H, \ll_H)$. So, S respects all the conflicting edges present in H. Formally, $\prec_H^{Conf} \subseteq \prec_S^{Conf}$.

It can be seen in $CG(H, \ll_H)$ that rv_methods() on any key k by transaction T_i returns the value written on k by previous closest committed transaction T_j. H maintains all the rv_methods() in conflicting (or Conf) edges of $CG(H, \ll_H)$. Since S has been generated by applying topological sort on $CG(H, \ll_H)$. So, S returns all the value of the rv_methods() from previous closest committed transactions. Hence, S is $legal$.

S satisfies all the properties of *co-opacity* and equivalent to H because S has been generated from the topological sort on $CG(H, \ll_H)$. Hence, history H is co-opaque.

(Only if part): Now, we consider H is co-opaque and we have to prove that $CG(H, \ll_H)$ is acyclic. Since H is co-opaque there exists an equivalent legal t-sequential history S to $\overline{H}$ which maintains real-time (real-time) and conflict ($Conf$) order of H. From the Lemma 49, we can say that conflict graph $CG(S, \ll_S,)$ is acyclic. As we know, $CG(H, \ll_H)$ is the subgraph of $CG(S, \ll_S,)$. Hence, $CG(H, \ll_H)$ is acyclic.

4.3 The Proposed SF-SV-OSTM Algorithm

In this section, we propose Starvation-Free Single-Version OSTM (SF-SV-OSTM) algorithm. SubSection 4.3.1 describes the definition of starvation-freedom followed by our assumption about the scheduler that helps us to achieve starvation-freedom in SF-SV-OSTM. SubSection 4.3.2 explains the design and data structure of SF-SV-OSTM. SubSection 4.3.3 shows the working of SF-SV-OSTM algorithm which includes the detail description of SF-SV-OSTM methods and challenges to make it starvation-free.

4.3.1 Description of Starvation-Freedom

Definition 5 ***Starvation-Freedom:*** *An STM system is said to be starvation-free if a thread invoking a non-parasitic transaction T_i gets the opportunity to retry T_i on every abort, due to the presence of a fair scheduler, then T_i will eventually commit.*

Herlihy & Shavit [18] defined the fair scheduler which ensures that none of the thread will crash or delayed forever. Hence, any thread Th_i acquires the lock on the shared data-items while executing transaction T_i will eventually release the locks. So, a thread will never block other threads to progress. Please refer SubSection 3.3.1 for the detailed description of fair scheduler. To satisfy the starvation-freedom for SF-SV-OSTM and SF-K-OSTM, we assumed bounded termination for the fair scheduler.

Assumption 3 ***Bounded-Termination:*** *For any transaction T_i, invoked by a thread Th_i, the fair system scheduler ensures, in the absence of deadlocks, Th_i is given sufficient time on a CPU (and memory, etc) such that T_i terminates ($\mathscr{C}$ or $\mathscr{A}$) in bounded time.*

In the proposed algorithms, we have considered *L* as the maximum time-bound of a transaction T_i within this either T_i will return commit or abort in the absence of deadlock. Approach for achieving the *deadlock-freedom* is motivated from the literature in which threads executing transactions acquire the locks in increasing order of the keys and releases the locks in bounded time either by committing or aborting the transaction. We consider an assumption about the transactions of the system as follows.

Assumption 4 *We assume, if other concurrent conflicting transactions do not exist in the system then every transaction will commit. i.e. (a) If a transaction T_i is executing in the system with the absence of other conflicting transactions then T_i will not self-abort. (b) Transactions of the system are non-parasitic as explained in Section 3.1.*

If transactions self-abort or parasitic then ensuring starvation-freedom is impossible.

4.3.2 Design and Data Structure of SF-SV-OSTM Algorithm

In this subsection, we illustrate the design and underlying data structure of proposed SF-SV-OSTM algorithm to maintain the shared data-items (or keys).

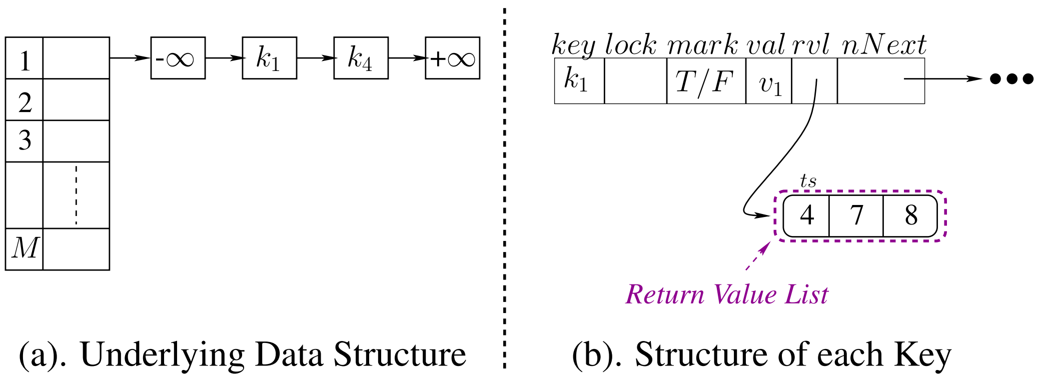

1
2
3
M
-∞
k_1
k_4
+∞
key lock mark val rvl nNext
k_1
T/F
v_1
ts
4
7
8
Return Value List
(a). Underlying Data Structure
(b). Structure of each Key

Figure 4.2: Design and Data Structure of SF-SV-OSTM

To achieve the *Starvation-Freedom* in *Single-Version Object-based STM (SF-SV-OSTM)*, we use chaining hash table (or ht) as an underlying data structure where the size of the hash table is *M* buckets and we propose HT-SF-SV-OSTM as shown in Figure 4.2.(a). Hash table with bucket size one becomes the linked-list data structure for SF-SV-OSTM represented as list-SF-SV-OSTM. The representation of SF-SV-OSTM is similar to SV-OSTM [9]. Each bucket stores multiple nodes in the form of linked-list between the two sentinel nodes *Head*(-∞) and *Tail*(+∞). Figure 4.2.(b) illustrates the structure of each node as ⟨*key, lock, mark, val, rvl, nNext*⟩. Where, *key* is the unique value from the range of [1 to $\mathscr{K}$] stored in the increasing order between the two sentinel nodes similar to linked-list based concurrent set implementation [42,43]. The *lock* field is acquired by the transaction before updating (inserting or deleting) on the node. *mark* is the boolean field which says a node is deleted or not. If *mark* sets to true then node is logically deleted else present in the hash table. Here, the deletion is in a lazy manner similar to concurrent linked-list structure [42]. If value (*val*) is *nil* then node is created by the *STM_delete*() otherwise *STM_insert*() creates a node with not *nil* value. To satisfy the correctness criteria as *co-opacity*, *STM_delete*() also maintains the node corresponding to each key with *mark* field as $true$. We motivate this with an interesting example below. *rvl* stands for *return value list* which maintains the information about lookup transaction that has lookups from a particular node. It maintains the timestamp (*ts*) of rv_methods (*STM_lookup*() or *STM_delete*()) transaction in it. The field *nNext* points to next available node in the linked-list. From now onwards, we will use the term key and node interchangeably.

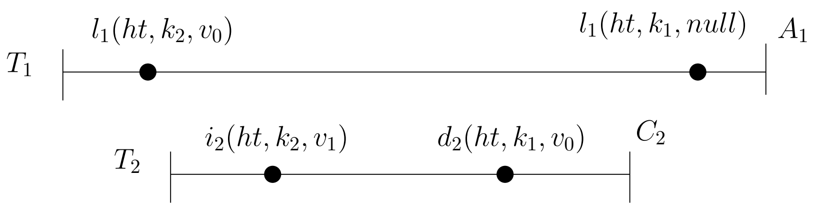


Figure 4.3: History H is not co-opaque

Marked Node: Now, we explain why we need to maintain deleted nodes through Figure 4.3

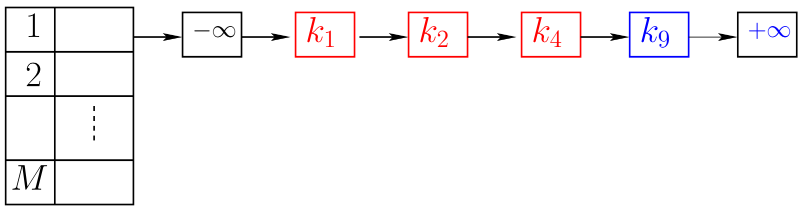


Figure 4.5: Searching k_9 over *lazy-list*

and 4.4. History H shown in Figure 4.3 is not co-opaque [9] 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 $l_1(ht, k_1, null)$ needs to be aborted. And $l_1(ht, k_1, null)$ can only be aborted if SF-SV-OSTM scheduler knows that a conflicting operation $d_2(ht, k_1, v_0)$ 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 [2] and aborts $l_1(ht, k_1, null)$ to ensure co-opacity.

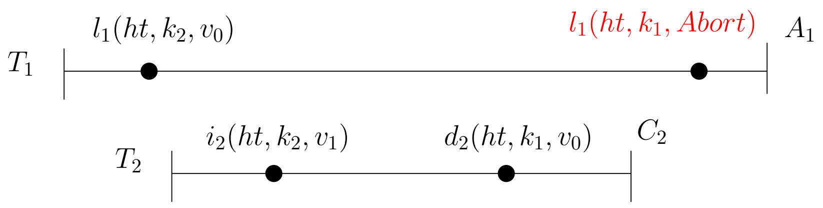


Figure 4.4: Co-opacity History H1

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 $l_1(ht, k_1)$, we get that the history is co-opaque with T_1 and T_2 being the equivalent serial history as shown in Figure 4.4. The deleted keys (nodes with marked field set) can be reused if another transaction comes and inserts the same key back.

But maintaining the deleted node along with the live (not deleted) node will increase the traversal time to search a particular node in the list. Consider Figure 4.5, where red color depicts the deleted node $\langle k_1, k_2, k_4 \rangle$ and blue color depicts the live node $\langle k_9 \rangle$. When any method of SF-SV-OSTM searches the key k_9 then it has to traverse the deleted nodes $\langle k_1, k_2, k_4 \rangle$ as well before reach to k_9 that increases the traversal time.

This motivated us to modify the lazy-list structure of a node to form a skip list based on red and blue links. We called it as a *red-blue lazy-list* or *rblazy-list*. This idea has been explored by Peri et al. in SV-OSTMs [9]. *rblazy-list* maintains two-pointer corresponding to each node such as red link (RL) and blue link (BL). Where BL points to the live node and RL points to live node as well as deleted node. Let us consider the same example as discussed above with this modification, key k_9 is directly searched from the head of the list with the help of BL as shown

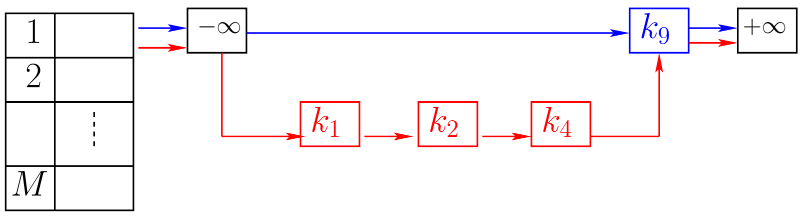


Figure 4.6: Searching k_9 over *rblazy-list*

in Figure 4.6. In this case, traversal time is efficient because any method of SF-SV-OSTM need not traverse the deleted nodes. To maintain the RL and BL in each node we modify the structure of *lazy-list* as ⟨*key, lock, mark, vl, RL, BL*⟩ and called it as *rblazy-list*.

4.3.3 The Working of SF-SV-OSTM Algorithm

SF-SV-OSTM system invokes *STM_begin*(), *STM_lookup*(), *STM_delete*(), *STM_insert*(), and STM_tryC() methods. *STM_lookup*() and *STM_delete*() works as rv_method() which lookup the value of key k from shared memory and returns it. Whereas *STM_insert*() and *STM_delete*() work as upd_method() that modifies the value of k in shared memory. We propose optimistic SF-SV-OSTM, so, upd_method() first update the value of k in its local log $txLog$ and the actual effect of upd_method() will be visible after successful STM_tryC(). This subsection explains the working of each method as follows:

***STM_begin*():** We show the high-level view of *STM_begin*() in Algorithm 19. When a thread Th_i invokes transaction T_i for the first time (or first incarnation) then *STM_begin*() assigns a unique timestamp known as *current timestamp* (cts) as shown in Line 5. It is incremented atomically with the help of atomic global counter ($gcounter$). If T_i gets aborted then thread Th_i executes it again with new incarnation of T_i, say T_j with the new cts until T_i commits but retains its initial cts as *initial timestamp* (its) at Line 8. Th_i uses its to inform the STM system that whether T_i is a new invocation or an incarnation. If T_i is the first incarnation then its and cts are same as cts_i so, Th_i maintains ⟨its_i, cts_i⟩. If T_i gets aborted and retries with T_j then Th_i maintains ⟨its_i, cts_j⟩. By assigning priority to the lowest its transaction (i.e. transaction have been in the system for longer time) in Single-Version OSTM, Starvation-Freedom can easily achieved.

STM_begin() initializes the *transaction local log* ($txLog_i$) for each transaction T_i to store the information in it. Whenever a transaction starts it atomically sets its *status* to be *live* as a global variable at Line 13. Transaction *status* can be $\langle live, commit, false\rangle$. After successful execution of STM_tryC(), T_i sets its *status* to be *commit*. If *status* of the transaction is *false* then it returns *abort*.

***STM_lookup*() and *STM_delete*() as rv_methods():** *rv_method(ht, k, val)* returns the value (*val*) corresponding to the key k from the shared memory as hash table (*ht*). We show the high-level overview of the rv_method() in Algorithm 20. First, it identifies the key k in the

Algorithm 19 *STM_begin(its)*: This method is invoke by a thread to start a new transaction T_i. It pass a parameter its which is the initial timestamp of the first incarnation of T_i. If this is the first incarnation then its is nil.

```
1:  procedure STM_begin($its$)
2:      Creating a local log $txLog_i$ for each transaction.
3:      if ($its == nil$) then
4:          /* Atomically get the value from the global counter and set it to its, and, cts.*/
5:          $its_i = cts_i = gcounter.get\&Inc()$;
6:      else
7:          /*Set the $its_i$ to first incarnation of $T_i$ $its$*/
8:          $its_i = its$;
9:          /*Atomically get the value from the global counter for $cts_i$*/
10:         $cts_i = gcounter.get\&Inc()$.
11:     end if
12:     /*Initially, set the $status_i$ of $T_i$ as $live$*/
13:     $state_i = live$;
14:     return $\langle cts_i, its_i \rangle$
15: end procedure
```

transaction local log as $txLog_i$ for transaction T_i. If k exists then it updates the $txLog_i$ and returns the val at Line 18.

If k does not exist in the $txLog_i$ then rv_method() checks the *status* of T_i before identifying the location in shared memory at Line 21. If *status* of T_i (or i) is *false* then T_i has to *abort* which says that T_i is not having the lowest its among other concurrent conflicting transactions. So to propose starvation-free SV-OSTM other conflicting transactions sets its *status* field as *false* and force transaction T_i to *abort*.

If *status* of T_i is not *false* and k is not exist in the $txLog_i$ then it identifies the location optimistically (without acquiring the locks similar to the *lazy-list* [42]) in the shared memory at Line 23. SF-SV-OSTM maintains the shared memory in the form of hash table with M buckets as shown in SubSection 4.3.2, where each bucket stores the keys in the form of *rblazy-list*. Each node contains two pointer $\langle RL, BL \rangle$. So, it identifies the two *predecessors (pred)* and two *current (curr)* with respect to each node. First, it identifies the pred and curr for key k in BL as $\langle preds[0], currs[1] \rangle$. After that it identifies the pred and curr for key k in RL as $\langle preds[1], currs[0] \rangle$. If $\langle preds[1], currs[0] \rangle$ are not marked then $\langle preds[0] = preds[1], currs[1] = currs[0] \rangle$. SF-SV-OSTM maintains the keys are in increasing order. So the order among the nodes are $\langle preds[0].key \leq preds[1].key < k \leq currs[0].key \leq currs[1].key \rangle$.

rv_method() acquires the lock in predefined order on all the identified preds and currs for key k to avoid the deadlock at Line 24 and do the *rv_Validation()* at Line 25. If $\langle preds[0] \vee currs[1] \rangle$ is marked or preds are not pointing to identified curs as $\langle (preds[0].BL \neq currs[1]) \vee (preds[1].RL \neq currs[0]) \rangle$ shown in Algorithm 21 then it releases the locks on all the preds and curRs and identify the new preds and currs for key k in shared memory.

Algorithm 20 *rv_method(ht, k, val):* It can either be *STM_delete*$_i(ht, k, val)$ or *STM_lookup*$_i(ht, k, val)$ on key k of transaction T_i.

```
16: procedure $rv\_method_i(ht, k, val)$
17:     if ($k \in txLog_i$) then
18:         Update the local log and return $val$.
19:     else
20:         /*Atomically check the status of its own transaction $T_i$ (or $i$)*/
21:         if (i.status == false) then return $\langle abort_i \rangle$.
22:         end if
23:         Identify the preds[] and currs[] for $k$ in bucket $M_k$ of rblazy-list using BL
            and RL.
24:         Acquire locks on preds[] & currs[] in increasing order.
25:         if (!rv_Validation(preds[], currs[])) then
26:             Release the locks and goto Line 23.
27:         end if
28:         if ($k \notin M_k.rblazy\text{-}list$) then
29:             Create a new node $n$ with key $k$ as: ⟨key=k, lock=false, mark=true, rvl=i,
                RL=$\phi$, BL=$\phi$⟩./*$n$ is marked*/
30:             Insert $n$ into $M_k$.rblazy-list s.t. it is accessible only via RLs.
31:             Release locks; update the $txLog_i$ with $k$.
32:             return ⟨val⟩. /*val as $null$*/
33:         else
34:             Add $i$ into the $rvl$ of currs[].
35:             Release the locks; update the $txLog_i$ with $k$ and value.
36:             return $\langle val \rangle$.
37:         end if
38:     end if
39: end procedure
```

If key k is not exist in the *rblazy-list* of corresponding bucket M_k at Line 28 then it creates a new node n with key k as ⟨ *key=k, lock=false, mark=true, rvl=i, RL*=ϕ, *BL*=ϕ⟩ at Line 29. T_i adds its cts_i in the *rvl*. Finally, it insert the node n into $M_k.rblazy\text{-}list$ such that it is accessible via RL only at Line 30. rv_method() releases the locks and update the $txLog_i$ with key k and value as $null$ (Line 31). Eventually, it returns the *val* as $null$ at Line 32.

If key k is exist in the M_k.*rblazy-list* then it adds the cts_i of T_i in the rvl of *currs[]* at Line 34. Finally, it releases the lock and update the $txLog_i$ with key k and value as val at Line 35. Eventually, it returns the *val* at Line 36.

***STM_insert*() and *STM_delete*() as upd_methods():** The actual effect of *STM_insert*() and *STM_delete*() comes after successful STM_tryC(). We shows the high level view of STM_tryC() in Algorithm 22. First, STM_tryC() checks the *status* of the transaction T_i at Line 47. If *status* of T_i is *false* then T_i has to *abort* same as explained above in rv_method().

Algorithm 21 *rv_Validation(preds[], currs[])*: It is mainly used for *rv_method()* validation.

```
40: procedure $rv\_Validation($preds[], currs[]$)$
41:     if $((preds[0].mark)||(currs[1].mark)||(preds[0].BL) \neq currs[1]||(preds[1].RL) \neq currs[0])$ then return $\langle false \rangle$.
42:     else return $\langle true \rangle$.
43:     end if
44: end procedure
```

If *status* is not false then STM_tryC() sort the keys (exist in $txLog_i$ of T_i) of upd_methods() in increasing order. It takes one by one methods (m_{ij}) from the $txLog_i$ and identifies the location of the key k in M_k.*rblazy-list* as explained above in rv_method(). After identifying the preds and currs for k it acquire the locks in predefined order to avoid the deadlock at Line 54 and calls *tryC_Validation()* to validate the methods of T_i.

tryC_Validation() identifies whether the methods of invoking transaction T_i are insert/update a node corresponding to the keys while ensuring the *starvation-freedom*. First, it do the *rv_Validation()* at Line 79 as explained in rv_method(). If *rv_Validation()* is successful and key k is exist in the $M_k.rblazy\text{-}list$ then it maintains the All Return Value List (allRVL) from *currs[]* of key k at Line 82. Acquire the locks on *status* of all the transactions present in allRVL list including T_i it self in predefined order to avoid the deadlock at Line 85. First, it checks the *status* of its own transaction T_i at Line 87. If *status* of T_i is *false* then T_i has to *abort* same reason as explained in rv_method().

If *status* of T_i is not *false* then it compares the its_i of its own transaction T_i with the its_p of other transactions T_p present in the allRVL at Line 90. Along with this it checks the *status* of p. If above conditions $\langle (its_i < its_p) \&\& (p.status == live)) \rangle$ succeed then it includes T_p in the Abort Return Value List (abortRVL) at Line 91 to abort it later otherwise abort T_i itself at Line 92.

At Line 96, STM_tryC() aborts all other conflicting transactions which are present in the abortRVL while modifying the *status* field to be *false* to achieve *starvation-freedom*.

All the steps of the *tryC_Validation()* is successful then the actual effect of the *STM_insert*() and *STM_delete*() will be visible to the shared memory. At Line 61, STM_tryC() checks for *poValidation()*. When two subsequent methods $\langle m_{ij}, m_{ik} \rangle$ of the same transaction T_i identify the overlapping location of preds and currs in *rblazy-list*. Then *poValidation()* updates the current method m_{ik} preds and currs with the help of previous method m_{ij} preds and currs.

If m_{ij} is *STM_insert*() and key k is not exist in the $M_k.rblazy\text{-}list$ then it creates the new node n with key k as ⟨*key=k, lock=false, mark=false, rvl=*ϕ*, nNext=*ϕ⟩ at Line 63. Finally, it insert the node n into $M_k.rblazy\text{-}list$ such that it is accessible via RL as well as BL at Line 64. If m_{ij} is *STM_insert*() and key k is exist in the $M_k.rblazy\text{-}list$ then it updates the value and *rvl* to ϕ for node corresponding to the key k.

Algorithm 22 *STM_tryC (T_i)*: Validate the upd_methods of the transaction and return *commit*.

```
45: procedure $STM\_tryC(T_i)$
46:     /*Atomically check the status of its own transaction $T_i$ (or $i$)*/
47:     if ($i.status == false$) then return $\langle abort_i \rangle$.
48:     end if
49:     /*Sort the $keys$ of $txLog_i$ in increasing order.*/
50:     /*Method ($m$) will be either STM_insert() or STM_delete()*/
51:     for all ($m_{ij} \in txLog_i$) do
52:         if (($m_{ij}$==STM_insert())||($m_{ij}$==STM_delete())) then
53:             Identify the preds[] & currs[] for k in bucket $M_k$ of rblazy-list using BL
                & RL.
54:             Acquire the locks on preds[] & currs[] in increasing order.
55:             if (!$tryC\_Validation()$) then
56:                 return $\langle abort_i \rangle$.
57:             end if
58:         end if
59:     end for
60:     for all ($m_{ij} \in txLog_i$) do
61:         poValidation() modifies the preds[] & currs[] of current method which
            would have been updated by previous method of the same transaction.
62:         if (($m_{ij}$==STM_insert())&&(k$\notin M_k$.rblazy-list)) then
63:             Create new node $n$ with $k$ as: $\langle$key=k, lock=false, mark=false, rvl=$\phi$,
                RL=$\phi$, BL=$\phi\rangle$.
64:             Insert node $n$ into $M_k$.rblazy-list such that it is accessible via RL as well
                as BL.
65:             /*lock sets true*/
66:         else if ($m_{ij}$ == STM_insert()) then
67:             /*Sets rvl as $\phi$ and update the value*/.
68:             Node (currs[]) is accessible via RL and BL.
69:         end if
70:         if ($m_{ij}$ == STM_delete()) then
71:             /*Sets rvl as $\phi$ and mark as true*/.
72:             Node (currs[]) is accessible via RL only.
73:         end if
74:         Update the preds[] & currs[] of $m_{ij}$ in $txLog_i$.
75:     end for
76:     Release the locks; return $\langle commit_i \rangle$.
77: end procedure
```

Algorithm 23 *tryC_Validation():* It is only use from STM_tryC() validation.

```
78: procedure tryC_Validation()
79:     if (!rv_Validation()) then Release the locks and retry.
80:     end if
81:     if (k ∈ $M_k.rblazy\text{-}list$) then
82:         Maintain the list of currs[].rvl as allRVL for all k of $T_i$.
83:         /*p is the tsimestamp of transaction $T_p$*/
84:         if ($p$ ∈ allRVL) then /*Includes $i$ in allRVL*/
85:             Lock status of each $p$ in pre-defined order.
86:         end if
87:         if (i.status == false) then return $\langle false\rangle$.
88:         end if
89:         for all ($T_p$ ∈ allRVL) do
90:             if (($its_i$<$its_p$)&&(p.status==live)) then
91:                 Maintain abort list as abortRVL & includes p in it.
92:             else return $\langle false\rangle$. /*abort $i$ itself*/
93:             end if
94:         end for
95:         for all ($p$ ∈ abortRVL) do
96:             Set the status of $p$ to be $false$.
97:         end for
98:     end if
99:     return $\langle true\rangle$.
100: end procedure
```

If m_{ij} is *STM_delete*() and key k is exist in the $M_k.rblazy\text{-}list$ then it sets the *rvl* as ϕ and *mark* field as $true$ for node corresponding to the key k at Line 72. At last it updates the preds and currs of each m_{ij} into its $txLog_i$ to help the upcoming methods of the same transactions in *poValidation()* at Line 74. Finally, it releases the locks on all the keys in predefined order and returns *commit* at Line 76.

4.4 Liveness Proof of SF-SV-OSTM Algorithm

Proof Notations: Following the notion derived for SF-SV-OSTM algorithm, we assume that all the histories accepted by SF-SV-OSTM algorithm as *gen(SF-SV-OSTM)*. This section considers only histories that are generated by SF-SV-OSTM unless explicitly stated otherwise. For simplicity, we consider the sequential histories for our discussion and we can get the sequential history using the linearization points (or LPs) as *first unlocking point of each successful* *method.*

Let us consider a transaction T_i from the history H as *gen(SF-SV-OSTM)*. Each transaction T_i maintains $\langle its_i, cts_i \rangle$. The value of cts is assigned atomically with the help of atomic global counter $gcounter$. So, we use $gcounter$ to approximate the system time.

Apart from these cts_i and its_i transaction T_i maintains $lock$ and $status$. T_i acquires the $lock$ on the keys before accessing it. $status$ can be $\langle live, false, commit \rangle$. The value of $lock$ and $status$ field change as the execution proceeds. For the sake of understanding, we represent the timestamps of a transaction T_i corresponding to history H as $H.its_i$ and $H.cts_i$.

To satisfy the starvation-freedom for SF-SV-OSTM, We assumed bounded termination for the fair scheduler as described Assumption 3 in SubSection 4.3.1. In the proposed algorithms, we have considered *L* as the maximum time-bound of a transaction T_i within this either T_i will return commit or abort in the absence of deadlock. We consider an assumption about the transactions of the system as described Assumption 4 in SubSection 4.3.1 which will help to achieve and prove about the starvation-freedom of SF-SV-OSTM.

Theorem 51 *SF-SV-OSTM ensures starvation-freedom in presence of a fair scheduler that satisfies Assumption 3 (bounded-termination) and in the absence of parasitic transactions that satisfies Assumption 4.*

Proof. Consider any history H generated by SF-SV-OSTM algorithm with transaction T_i. Initially, thread Th_i calls *STM_begin*() for T_i which maintains ⟨*its, cts*⟩ and set the $status$ as $live$. If T_i is the first incarnation then its *Initial Timestamp (its)* and *Current Timestamp (cts)* are same as i. We represent the $\langle its, cts \rangle$ as $\langle its_i, cts_i \rangle$ for transaction T_i. If T_i is aborted then thread Th_i executes it again with new incarnation of T_i until T_i commits. Let the new incarnation of T_i say T_j then thread Th_i maintain its $\langle its, cts \rangle$ as $\langle its_i, cts_j \rangle$. Th_i stores the first incarnation its_i of T_i to set the reincarnation its_j of T_j is same as its_i. The value of cts is incremented atomically with the help of atomic global counter $gcounter$. So, we use $gcounter$ to approximate the system time.

Through Assumption 3, we can say that T_i will terminate ($\mathscr{C}$ or $\mathscr{A}$) in bounded time. If T_i returns abort then T_i will retry again with new incarnation of T_i, say T_j while satisfying the Assumption 4. The incarnation of T_i transaction as $\langle its_i, cts_j \rangle$. So, it can be seen that, T_i will get the lowest its in the system and achieve the highest priority. Eventually, T_i returns commit. Similarly, all the transactions of the H generated by SF-SV-OSTM will eventually commit. Hence, SF-SV-OSTM ensures starvation-freedom.

4.5 Safety Proof of SF-SV-OSTM Algorithm

This section shows the correctness of proposed SF-SV-OSTM with the help of graph characterization of co-opacity described in Section 4.2. As discussed in Chapter 2, SF-SV-OSTM executes high-level methods through transactions on history H which internally invoke multiple read-write (or lower-level) operations including invocation and response known as *events* (or $evts$). So, high-level methods are interval instead of dots (atomic). Methods of same transaction T_i are always real-time ordered, i.e., none of the methods of T_i overlaps each other. But due to the concurrent execution of history H with methods are interval, two methods from different transactions may overlap. Thus, we order the overlapping methods of transactions based on their *linearization point (LP)*. We consider *first unlocking point* of each successful method as the *LP* of the respective method.

In the concurrent execution of a history H, we make high-level methods of a transaction as atomic based on their *linearization points (LPs)* as defined above. But, as we know from Chapter 2, a transaction internally invokes multiple high-level methods and transactions are overlapping to each other in concurrent history H. So, with the help of *graph characterization of co-opacity*, SF-SV-OSTM ensures the atomicity of the transaction. It proves the correctness of SF-SV-OSTM using following theorems:

Theorem 52 *A legal SF-SV-OSTM history H is co-opaque iff $CG(H, \ll_H)$ is acyclic.*

Proof. **(if part):** First, we consider H is *legal* and $CG(H, \ll_H)$ is acyclic then we need to prove that history H is co-opaque. Since $CG(H, \ll_H)$ is acyclic, we apply topological sort on $CG(H, \ll_H)$ and obtained a t-sequential history S which is equivalent to $\overline{H}$. S also respect real-time edges (or real-time) and Conf edges as $\overline{H}$. Formally, S respects $\prec_H^{RT} = \prec_{\overline{H}}^{RT}$ and $\prec_H^{Conf} = \prec_{\overline{H}}^{Conf}$.

Since Conflict relation between two methods of SF-SV-OSTM in S are also present in $\overline{H}$. Formally, $\prec_{\overline{H}}^{Conf} \subseteq \prec_S^{Conf}$. Given that H is *legal* which implies that $\overline{H}$ is also *legal*. So, we can say that S is *legal*. Collectively, H satisfies all the necessary conditions of *co-opacity*. Hence, history H is co-opaque.

(Only if part): Now, we consider H is co-opaque and legal then we have to prove that $CG(H, \ll_H)$ is acyclic. Since H is co-opaque there exists an equivalent legal t-sequential history S to $\overline{H}$ which maintains real-time (real-time) and conflict ($Conf$) order of H, i.e, S respects $\prec_H^{RT}$ and $\prec_H^{Conf}$ (from the definition of co-opacity [9]). So, we can observe from the conflict graph construction that $CG(H, \ll_H) = CG(H, \ll_{\overline{H}})$ and both are the subgraph of $CG(S, \ll_S)$. Since S is a t-sequential history, so $CG(S, \ll_S)$ is acyclic. As we know, any subgraph of any acyclic graph is also acyclic and $CG(H, \ll_H)$ is the subgraph of $CG(S, \ll_S)$. Hence, $CG(H, \ll_H)$ is acyclic.

Theorem 53 *Any legal history H generated by SF-SV-OSTM satisfies co-opacity.*

Proof. In order to prove this, we construct the co-opacity graph $CG(H, \ll)$ generated by SF-SV-OSTM algorithm and prove that $CG(H, \ll)$ graph is acyclic. After that with the help of

Theorem 52, we can say that generated $CG(H, \ll)$ graph is acyclic so any legal history H generated by SF-SV-OSTM is co-opaque.

To prove the $CG(H, \ll)$ generated by SF-SV-OSTM algorithm is acyclic. We construct $CG(H, \ll) = (V, E)$ which consists of V vertices and E edges. Here, each committed transaction is consider as a vertex and edges are as follows:

- *real-time (or real-time) edge*: If transaction T_i returns commit before the beginning of other transaction T_j then real-time edge goes from T_i to T_j in SF-SV-OSTM. Formally, $(STM\_tryC_i() \prec_H$ *STM_begin*$_j()) \implies TS(T_i) < TS(T_j) \implies T_i \rightarrow T_j$.

- *conflict (or Conf) edge*: The conflict edges between two transactions depends on the conflicts between them. Two transactions T_i and T_j of the sequential history are said to be in conflict, if both of them access same key k and at least one transaction performs *update* method. As described in Chapter 2, *STM_lookup()*, and *STM_delete()* return the value from underlying data structure so, we called these methods as *return value methods (or* $rv\_methods$*)*. Whereas, *STM_insert()*, and *STM_delete()* are updating the underlying data structure after successful *STM_tryC()* so, we called these as *update* methods (or $upd\_methods$). The conflicts are defined between the methods that accesses 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*$_j$*())* are the possible conflicting methods in SF-SV-OSTM. On conflict between two transaction T_i and T_j where $TS(T_i) < TS(T_j)$, the conflict edge going from T_i to T_j. In SF-SV-OSTM, if higher timestamp (TS) transaction T_j has already been committed then lower TS transaction T_i returns abort and retry with higher TS in the incarnation of T_i and returns commit. So, conflicts edges in SF-SV-OSTM follows increasing of their TS. Formally, $TS(T_i) < TS(T_j) \implies T_i \rightarrow T_j$.

So, all the edges of $CG(H, \ll)$ generated by SF-SV-OSTM algorithm follows the increasing order of TS of the transactions. Thus, $CG(H, \ll)$ graph is acyclic. Hence, with the help of Theorem 52, any legal history H generated by SF-SV-OSTM satisfies co-opacity.

4.6 The Proposed SF-K-OSTM Algorithm

In this section, we propose *Starvation-Free K-version OSTM (SF-K-OSTM)* algorithm which maintains K number of versions corresponding to each key. The value of K can vary from 1 to ∞. When K is equal to 1 then SF-K-OSTM boils down to *Starvation-Free Single-Version OSTM (SF-SV-OSTM)* proposed in Section 4.3. When K is ∞ then SF-K-OSTM maintains unbounded versions corresponding to each key known as *Starvation-Free Multi-Version OSTM (SF-MV-OSTM)* algorithm. To delete the unused version from the version list of SF-MV-OSTM, it calls a separate Garbage Collection (GC) method [10] and proposes SF-MV-OSTM-

GC. In this chapter of the thesis, we propose SF-K-OSTM and all the variants of it *(SF-MV-OSTM, SF-MV-OSTM-GC)* for two data structures *hash table* and *linked-list* but it is generic for other data structures as well.

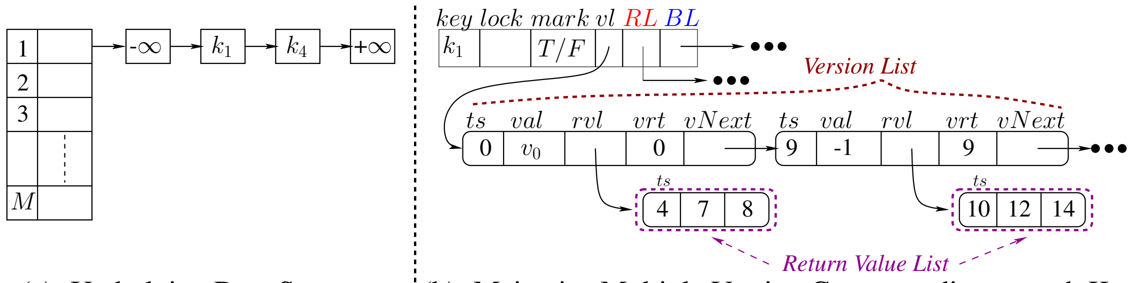
SubSection 4.6.1 explains the design and data structure of SF-K-OSTM. SubSection 4.6.2 shows the working of SF-K-OSTM algorithm. To achieve starvation-freedom in SF-K-OSTM, we followed the definition of *starvation-freedom* along with our assumptions about the scheduler explained in SubSection 4.3.1.

4.6.1 Design and Data Structure of SF-K-OSTM Algorithm

In this subsection, we illustrated the design and underlying data structure of SF-K-OSTM algorithm to maintain the shared data-items (or keys).

To achieve the *Starvation-Freedom* in *K-version Object-based STM (SF-K-OSTM)*, we use chaining hash table (or ht) as an underlying data structure where the size of the hash table is *M* buckets as shown in Figure 4.7.(a) and we propose HT-SF-K-OSTM. Hash table with bucket size one becomes the linked-list data structure for SF-K-OSTM represented as list-SF-K-OSTM. The representation of SF-K-OSTM is similar to MV-OSTM [16]. Each bucket stores multiple nodes in the form of linked-list between the two sentinel nodes *Head*(-∞) and *Tail*(+∞). Figure 4.7.(b) illustrates the structure of each node as $\langle$*key, lock, mark, vl, RL, BL*$\rangle$. Where, *key* is the unique value from the range of [1 to $\mathscr{K}$] stored in the increasing order between the two sentinel nodes similar to linked-list based concurrent set implementation [42, 43]. The *lock* field is acquired by the transaction before updating (inserting or deleting) on the node. *mark* is the boolean field which says a node is deleted or not. If *mark* sets to true then node is logically deleted else present in the hash table. Here, the deletion is in a lazy manner similar to concurrent linked-list structure [42]. The field *vl* stands for version list. SF-K-OSTM maintains the finite say latest K-versions corresponding to each key to achieve the greater concurrency. Whenever $(K + 1)^{th}$ version created for the key then it overwrites the oldest version corresponding to that key. If K is equal to 1, i.e., version list contains only one version corresponding to each key which boils down to proposed *Starvation-Free Single-Version OSTM (SF-SV-OSTM)* explained in Section 4.3. 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 *red-blue lazy list* or *rblazy-list*.

The structure of the *vl* is $\langle$*ts, val, rvl, vrt, vNext*$\rangle$ as shown in Figure 4.7.(b). *ts* is the unique timestamp assigned by the *STM_begin*(). If value (*val*) is *nil* then version is created by the *STM_delete*() otherwise *STM_insert*() creates a version with not *nil* value. To satisfy the correctness criteria as *local opacity*, *STM_delete*() also maintains the version corresponding to each key with *mark* field as $true$. It allows the concurrent transactions to lookup from the older



(a). Underlying Data Structure (b). Maintains Multiple Version Corresponding to each Key
Figure 4.7: Design and Data Structure of SF-K-OSTM

version of the marked node and returns the value as not *nil*. SF-MV-OSTM algorithm does not immediately physically remove deleted keys from the hash table to ensures the correctness criteria as opacity explained in SubSection 4.3.2. *rvl* stands for *return value list* which maintains the information about lookup transaction that has lookups from a particular version. It maintains the timestamp (*ts*) of rv_methods (*STM_lookup*() or *STM_delete*()) transaction in it. *vrt* stands for *version real time* which helps to maintain the *real-time order* among the transactions. *vNext* points to next available version in the version list.

4.6.2 The Working of SF-K-OSTM Algorithm

In this subsection, we describe the working of SF-K-OSTM algorithm which includes the detail description of SF-K-OSTM methods and challenges to make it starvation-free. This description can easily be extended to SF-MV-OSTM and SF-MV-OSTM-GC as well.

SF-K-OSTM invokes following methods as: *STM_begin*(), *STM_lookup*(), *STM_delete*(), *STM_insert*(), and STM_tryC(). *STM_lookup*() and *STM_delete*() work as rv_methods() which lookup the value of key k from shared memory and return it. Whereas *STM_insert*() and *STM_delete*() work as upd_methods() that modify the value of k in shared memory. We propose optimistic SF-K-OSTM, so, upd_methods() first update the value of k in transaction local log $txLog$ and the actual effect of upd_methods() will be visible after successful STM_tryC(). Now, we explain the functionality of each method as follows:

***STM_begin*():** When a thread Th_i invokes transaction T_i for the first time (or first incarnation), *STM_begin*() assigns a unique timestamp known as *current timestamp* (cts) using atomic global counter $(gcounter)$ at Line 105. If T_i gets aborted then thread Th_i executes it again with new incarnation of T_i, say T_j with the new cts until T_i commits but retains its initial cts as *initial timestamp* (its) at Line 108. Th_i uses its to inform the SF-K-OSTM system that whether T_i is a new invocation or an incarnation. If T_i is the first incarnation then its and cts are same as cts_i so, Th_i maintains $\langle its_i, cts_i \rangle$. If T_i gets aborted and retries with T_j then Th_i maintains $\langle its_i, cts_j \rangle$.

By assigning priority to the lowest its transaction (i.e. transaction have been in the system for longer time) in *Single-Version OSTM*, *Starvation-Freedom* can easily achieved as explained in Section 4.3. But achieving *Starvation-Freedom* in finite *K-versions OSTM (SF-K-OSTM)* is challenging. Though the transaction T_i has lowest its but T_i may return abort because of finite versions T_i did not find a correct version to lookup from or overwrite a version. Table 4.1 shows the key insight to achieve the starvation-freedom in finite K-versions OSTM. Here, we considered two transaction T_{10} and T_{20} with cts 10 and 20 that performs *STM_lookup()* (or l) and *STM_insert()* (or i) on same key k. We assume that a version of k exists with cts 5, so, *STM_lookup()* of T_{10} and T_{20} find a previous version to lookup and never return abort. Due to the optimistic execution in SF-K-OSTM, effect of *STM_insert()* comes after successful *STM_tryC()*, so *STM_lookup()* of a transaction comes before effect of its *STM_insert()*. Hence, total six permutations are possible as defined in Table 4.1. We can observe from the Table 4.1 that in some cases T_{10} returns abort. But if T_{20} gets the lowest its then T_{20} never returns abort. This ensures that a transaction with lowest its and highest cts will never return abort. But achieving highest cts along with lowest its is bit difficult because new transactions are keep on coming with higher cts using $gcounter$. So, to achieve the highest cts, we introduce a new timestamp as *working timestamp (wts)* which is significantly larger than cts.

S. No.	Execution Sequence	Possible actions by Transactions
1.	$l_{10}(k), i_{10}(k), l_{20}(k), i_{20}(k)$	$T_{20}(k)$ lookups the version inserted by T_{10}. No conflict.
2.	$l_{10}(k), l_{20}(k), i_{10}(k), i_{20}(k)$	Conflict detected at $i_{10}(k)$. Either abort T_{10} or T_{20}.
3.	$l_{10}(k), l_{20}(k), i_{20}(k), i_{10}(k)$	Conflict detected at $i_{10}(k)$. Hence, abort T_{10}.
4.	$l_{20}(k), l_{10}(k), i_{20}(k), i_{10}(k)$	Conflict detected at $i_{10}(k)$. Hence, abort T_{10}.
5.	$l_{20}(k), l_{10}(k), i_{10}(k), i_{20}(k)$	Conflict detected at $i_{10}(k)$. Either abort T_{10} or T_{20}.
6.	$l_{20}(k), i_{20}(k), l_{10}(k), i_{10}(k)$	Conflict detected at $i_{10}(k)$. Hence, abort T_{10}.

Table 4.1: Possible Permutations of Methods

STM_begin() maintains the wts for transaction T_i as wts_i, which is potentially higher timestamp as compare to cts_i. So, we derived,

$$wts_i = cts_i + C * (cts_i - its_i); \tag{4.1}$$

where C is any constant value greater than 0. When T_i is issued for the first time then wts_i, cts_i, and its_i are same at Line 105. If T_i gets aborted again and again then drift between the cts_i and wts_i will increases. The advantage for maintaining wts_i is if any transaction keeps getting aborted then its wts_i will be high and its_i will be low. Eventually, T_i will get chance to commit in finite number of steps to achieve starvation-freedom. For simplicity, we use timestamp (*ts*) i of T_i as wts_i, i.e., $\langle wts_i = i \rangle$ for SF-K-OSTM.

Observation 54 *Any transaction T_i with lowest its_i and highest wts_i will never abort in SF-K-OSTM system.*

Sometimes, the value of wts is significantly larger than cts. So, wts is unable to maintain *real-time order* between the transactions which violates the correctness of SF-K-OSTM.

Violation of Real-Time Order by wts: As described above, cts respects the real-time order among the transactions but SF-K-OSTM uses wts which may not respect real-time order. Sometimes, the value of wts is significantly larger than cts which leads to violate the real-time order among the transactions. Figure 4.8 illustrates it with history H: $l_1(ht, k_1, v_0)l_2(ht, k_2, v_0)$ $i_1(ht, k_1, v_{10})C_1 i_2(ht, k_1, v_{20})C_2 l_3(ht, k_1, v_{10}) i_3(ht, k_3, v_{25})C_3$ consists of three transactions T_1, T_2, T_3 with cts as 100, 110, 130 and wts as 100, 150, 130 respectively. T_1 and T_2 has been committed before the beginning of T_3, so T_1 and T_2 are in real-time order with T_3. Formally, $T_1 \prec_H^{RT} T_3$ and $T_2 \prec_H^{RT} T_3$. But, T_2 has higher wts than T_3. Now, T_3 lookups key k_1 from T_1 and returns the value as v_{10} because T_1 is the available largest wts (100) smaller than T_3 wts (130). The only possible equivalent serial order S to history H is $T_1T_3T_2$ which is legal as well. But S violates real-time order because T_3 is serialized before T_2 in S but T_2 has been committed before the beginning of T_3 in H. It can easily be seen that, such history H can be accepted by the algorithm when it uses only wts instead of cts. But this should not happen because its violating the real-time order which says it does not satisfy the correctness criteria as *local opacity*.

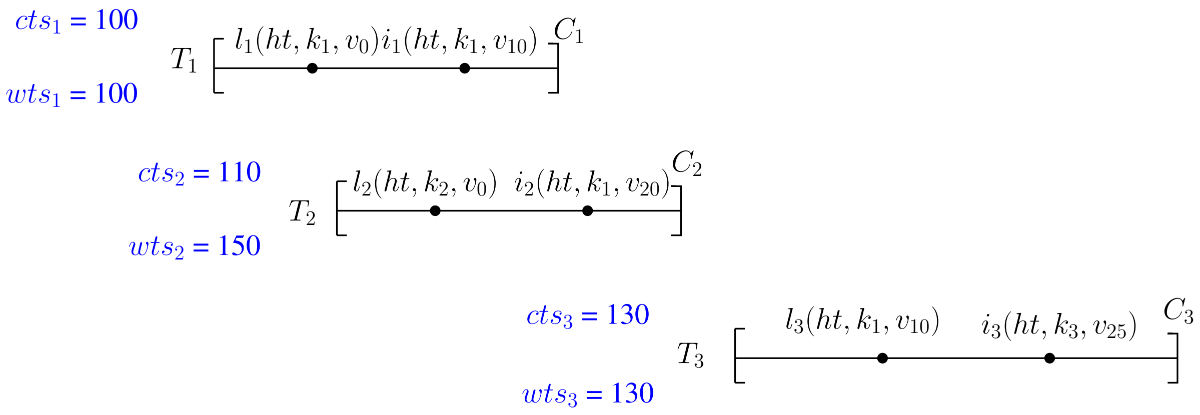
A simple solution to this issue is by delaying the committing transaction say T_i with wts_i until the real-time catches up to the wts_i. Delaying such T_i will ensure the correctness criteria as *local opacity* while making the wts of the transaction same as real-time. But, this is highly unacceptable. It seems like transaction T_i acquires the locks on all the keys it wants to update and wait. It will show the adverse effect and reduces the performance of SF-K-OSTM system.

Regaining the Real-Time Order using Timestamp Ranges along with wts: We require that all the transactions of history H generated by SF-K-OSTM are serialized based on their wts while respecting the real-time order among them. Another efficient solution is to allow the transaction T_i with wts_i to catch up with the actual time if T_i does not violates the real-time order. So, to respect the real-time order among the transactions SF-K-OSTM uses the time constraints. SF-K-OSTM uses the idea of timestamp ranges given by Riegel et al. [39] along with $\langle its_i, cts_i, wts_i \rangle$ for transaction T_i in *STM_begin*(). It maintains the *transaction lower timestamp limit ($tltl_i$)* and *transaction upper timestamp limit ($tutl_i$)* for T_i. Initially, $\langle its_i, cts_i, wts_i, tltl_i \rangle$ are the same for T_i. $tutl_i$ would be set as a largest possible value denoted as $+\infty$ for T_i. After successful execution of *rv_methods()* or STM_tryC() of T_i, $tltl_i$ gets incremented and $tutl_i$ gets decremented[1] to respect the real-time order among the transactions.

For better understanding consider Figure 4.9, which shows the regaining the *real-time order* using timestamp ranges (*tltl* and *tutl*) along with *wts* on history H. Initially, T_1 begins with cts_1 = wts_1 = $tltl_1$ = 100, $tutl_1$ = ∞ and T_1 returns commit. We assume at the time of commit of T_1, $gcounter$ is 120. So, $tutl_1$ reduces to 120. After that T_2 commits and suppose $tutl_2$ reduces to 121 (so, the current value $gcounter$ is 121). T_1 and T_2 both access the key k_1 and T_2

[1]Practically ∞ cannot be decremented for $tutl_i$ so we assign the highest possible value to $tutl_i$ which gets decremented.

is updating k_1. So, T_1 and T_2 are conflicting. Hence, $tltl_2$ is incremented to a value greater than $tutl_1$, say 121. Now, when T_3 begins, it assigns cts_3 = wts_3 = $tltl_3$ = 130, and $tutl_3$ = ∞. At the time of $l_3(ht, k_1)$, as T_3 lookups the version of k_1 from T_1, so, T_3 reduces its $tutl_3$ less than $tltl_2$ (currently, $tltl_2$ is 121). Hence, $tutl_3$ becomes say 120. But, $tltl_3$ is already 130. So, $tltl_3$ has crossed the limit of $tutl_3$ which is causing T_3 to abort. Intuitively, this implies that wts_3 and real-time order are out of synchrony and can not be reconciled. Hence, by using the timestamp ranges H executes correctly by SF-K-OSTM algorithm with equivalent serial schedule T_1T_2.



cts_1 = 100
wts_1 = 100
T_1
$l_1(ht, k_1, v_0)i_1(ht, k_1, v_{10})$
C_1
cts_2 = 110
wts_2 = 150
T_2
$l_2(ht, k_2, v_0)$ $i_2(ht, k_1, v_{20})$
C_2
cts_3 = 130
wts_3 = 130
T_3
$l_3(ht, k_1, v_{10})$ $i_3(ht, k_3, v_{25})$
C_3

Figure 4.8: Violating the *real-time order* by *wts*

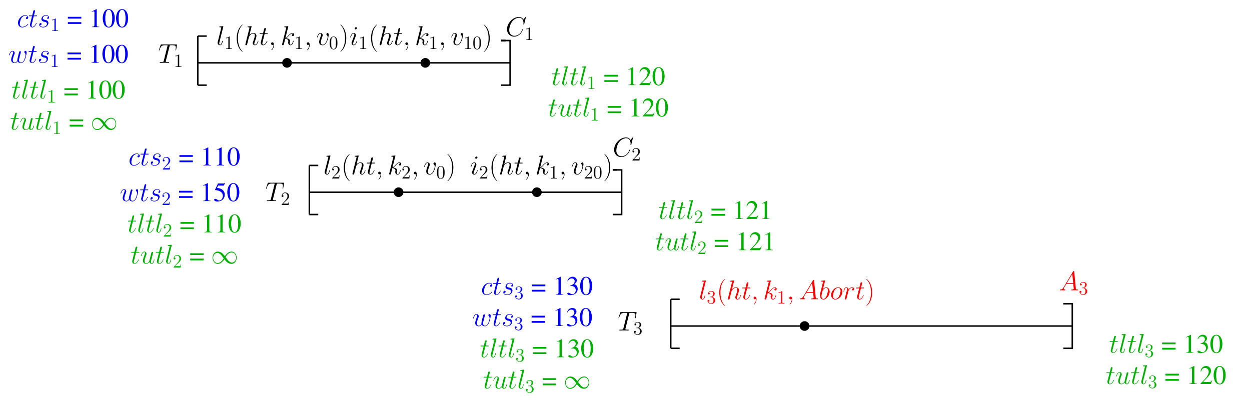


Figure 4.9: Regaining the *real-time order* using Timestamp Ranges along with *wts*

Algorithm 24 *STM_begin(its)*: This method is invoke by a thread Th_i to start a new transaction T_i. It pass a parameter its which is the initial timestamp of the first incarnation of T_i. If T_i is the first incarnation then its is nil.

```
101: procedure STM_begin(its)
102:     Create a local log $txLog_i$ for each transaction.
103:     if ($its == nil$) then
104:         /* Atomically get the value from the global counter and set it to its, cts, and
             wts.*/
```

```
105:         $its_i$ = $cts_i$ = $wts_i$ = gcounter.get&Inc();
106:     else
107:         /*Set the $its_i$ to first incarnation of $T_i$ $its$*/
108:         $its_i$ = $its$;
109:         /*Atomically get the value from the global counter for $cts_i$*/
110:         $cts_i$ = gcounter.get&Inc().
111:         /*Set the $wts$ value with the help of $cts_i$ and $its_i$*/
112:         $wts_i$ = $cts_i$+C*($cts_i$-$its_i$).
113:     end if
114:     /*Set the $tltl_i$ as $cts_i$*/
115:     $tltl$ = $cts_i$.
116:     /*Set the $tutl_i$ as possible large value*/
117:     $tutl_i$ = $\infty$.
118:     /*Initially, set the $status_i$ of $T_i$ as $live$*/
119:     $status_i$ = $live$;
120:     return $\langle cts_i, wts_i \rangle$
121: end procedure
```

STM_begin() initializes the *transaction local log* ($txLog_i$) for each transaction T_i to store the information in it. Whenever a transaction starts it atomically sets its *status* to be *live* as a global variable at Line 119. Transaction *status* can be $\langle live, commit, false \rangle$. After successful execution of STM_tryC(), T_i sets its *status* to be *commit*. If *status* of the transaction is *false* then it returns *abort*.

***STM_lookup*() and *STM_delete*() as rv_methods():** *rv_methods(ht, k, val)* return the value (*val*) corresponding to the key *k* from the shared memory as hash table (*ht*). We show the high level overview of the rv_methods() in Algorithm 26. First, it identifies the key k in the transaction local log as $txLog_i$ for transaction T_i. If k exists then it updates the $txLog_i$ and returns the val at Line 129.

If key k does not exist in the $txLog_i$ then before identify the location in share memory rv_methods() check the *status* of T_i at Line 132. If *status* of T_i (or i) is *false* then T_i has to *abort* which says that T_i is not having the lowest its and highest wts among other concurrent conflicting transactions. So, to propose starvation-freedom in SF-K-OSTM other conflicting transactions set the *status* of T_i as *false* and force it to *abort*.

If *status* of T_i is not *false* and key k does not exist in the $txLog_i$ then it identifies the location of key k optimistically (without acquiring the locks similar to the *lazy-list* [42]) in the shared memory at Line 134. SF-K-OSTM maintains the shared memory in the form of hash table with M buckets as shown in SubSection 4.6.1, where each bucket stores the keys in *rblazy-list*. Each node contains two pointer $\langle RL, BL \rangle$. So, it identifies the two *pre-*

decessors (pred) and two *current (curr)* with respect to each node. First, it identifies the pred and curr for key k in BL as $\langle preds[0], currs[1]\rangle$. After that it identifies the pred and curr for key k in RL as $\langle preds[1], currs[0]\rangle$. If $\langle preds[1], currs[0]\rangle$ are not marked then $\langle preds[0] = preds[1], currs[1] = currs[0]\rangle$. SF-K-OSTM maintains the keys are in increasing order. So, the order among the nodes are $\langle preds[0].key \leq preds[1].key < k \leq currs[0].key \leq currs[1].key\rangle$.

rv_methods() acquire the lock in predefined order on all the identified preds and currs for key k to avoid the deadlock at Line 135 and do the *rv_Validation()* as shown in Algorithm 25. If $\langle preds[0] \vee currs[1]\rangle$ is marked or preds are not pointing to identified curls as $\langle(preds[0].BL \neq currs[1]) \vee (preds[1].RL \neq currs[0])\rangle$ then it releases the locks from all the preds and currs and identify the new preds and currs for k in shared memory.

Algorithm 25 *rv_Validation(preds[], currs[])*: It is mainly used for *rv_method()* validation.

```
122: procedure $rv\_Validation(preds[], currs[])$
123:     if $((preds[0].mark)||(currs[1].mark)||((preds[0].BL) \neq currs[1])||((preds[1].RL) \neq currs[0]))$ then return $\langle false\rangle$.
124:     else return $\langle true\rangle$.
125:     end if
126: end procedure
```

If key k does not exist in the *rblazy-list* of corresponding bucket M_k at Line 139 then it creates a new node n with key k as $\langle$ *key=k, lock=false, mark=true, vl=ver,* RL=ϕ, BL=$\phi\rangle$ at Line 140 and creates a version (*ver*) for transaction T_0 as $\langle$*ts=0, val=nil, rvl=i, vrt=0, vNext=*$\phi\rangle$ at Line 141. Transaction T_i creates the version of T_0, so, other concurrent conflicting transaction (say T_p) with lower timestamp than T_i, i.e., $\langle p < i\rangle$ can lookup from T_0 version. Thus, T_i save T_p to abort while creating a T_0 version and ensures greater concurrency. After that T_i adds its wts_i in the *rvl* of T_0 and sets the *vrt* 0 as timestamp of T_0 version. Finally, it insert the node n into $M_k.rblazy\text{-}list$ such that it is accessible via RL only at Line 142. rv_method() releases the locks and update the $txLog_i$ with key k and value as nil (Line 143). Eventually, it returns the *val* as nil at Line 144.

If key k exists in the $M_k.rblazy\text{-}list$ then it identifies the current version ver_j with ts = j such that j is the *largest timestamp smaller (lts)* than i at Line 146 and there exists no other version with timestamp p by T_p on same key k such that $\langle j < p < i\rangle$. If ver_j is nil at Line 147 then SF-K-OSTM returns $abort$ for transaction T_i because it does not found version to lookup otherwise it identifies the next version with the help of $ver_j.vNext$. If next version ($ver_j.vNext$ as ver_k) exist then T_i maintains the $tutl_i$ with minimum of $\langle tutl_i \vee (ver_k.vrt-1)\rangle$ at Line 151 and $tltl_i$ with maximum of $\langle tltl_i \vee (ver_j.vrt+1)\rangle$ at Line 154 to respect the *real-time order* among the transactions. If $tltl_i$ is greater than $tutl_i$ at Line 156 then transaction T_i returns *abort* (fail to maintains real-time order) otherwise it adds the ts of T_i (wts_i) in the rvl

Algorithm 26 *rv_methods(ht, k, val)*: It can either be *STM_delete*$_i(ht, k, val)$ or *STM_lookup*$_i(ht, k, val)$ on key k by transaction T_i.

```
127: procedure $rv\_methods_i(ht, k, val)$
128:     if ($k \in txLog_i$) then
129:         Update the local log of $T_i$ and return $val$.
130:     else
131:         /*Atomically check the *status* of its own transaction $T_i$ (or $i$).*/
132:         if (*i.status == false*) then return $\langle abort_i \rangle$.
133:         end if
134:         Identify the *preds[]* and *currs[]* for key $k$ in bucket $M_k$ of *rblazy-list* using
            BL and RL.
135:         Acquire locks on *preds[]* & *currs[]* in increasing order of keys to avoid the
            deadlock.
136:         if (*!rv_Validation(preds[], currs[])*) then
137:             Release the locks and goto Line 134.
138:         end if
139:         if ($k \notin M_k.rblazy\text{-}list$) then
140:             Create a new node $n$ with key $k$ as: $\langle$*key=k, lock=false, mark=true,
                vl=ver, RL=$\phi$, BL=$\phi$*$\rangle$./*$n$ is marked*/
141:             Create version $ver$ as:$\langle$*ts=0, val=nil, rvl=i, vrt=0, vNext=$\phi$*$\rangle$.
142:             Insert $n$ into $M_k$*.rblazy-list* s.t. it is accessible only via RLs. /**lock* sets
                *true**/
143:             Release locks; update the $txLog_i$ with $k$.
144:             return $\langle$*val*$\rangle$. /**val* as $nil$*/
145:         end if
146:         Identify the version $ver_j$ with $ts = j$ such that $j$ is the *largest timestamp smaller
            (lts)* than $i$.
147:         if ($ver_j == nil$) then /*Finite Versions*/
148:             return $\langle abort_i \rangle$
149:         else if ($ver_j.vNext$ != $nil$) then
150:             /*$tutl_i$ should be less then *vrt* of next version $ver_j$*/
151:             Calculate $tutl_i$ = min($tutl_i$, $ver_j.vNext\ .vrt - 1$).
152:         end if
153:         /*$tltl_i$ should be greater then $vrt$ of $ver_j$*/
154:         Calculate $tltl_i$ = max($tltl_i$, $ver_j.vrt + 1$).
155:         /*If limit has crossed each other then abort $T_i$*/
156:         if ($tltl_i > tutl_i$) then return $\langle abort_i \rangle$.
157:         end if
158:         Add $i$ into the $rvl$ of $ver_j$.
159:         Release the locks; update the $txLog_i$ with $k$ and value.
160:     end if
161:     return $\langle ver_j.val \rangle$.
162: end procedure
```

of ver_j at Line 158. Finally, it releases the lock and update the $txLog_i$ with key k and value as current version value ($ver_j.val$) at Line 159. Eventually, it returns the value as $ver_j.val$ at Line 161.

***STM_insert*() and *STM_delete*() as upd_methods():** The actual effect of *STM_insert*() and *STM_delete*() come after successful STM_tryC(). They create the version corresponding to the key in shared memory. We show the high-level view of STM_tryC() in Algorithm 28. First, STM_tryC() checks the *status* of the transaction T_i at Line 217. If *status* of T_i is *false* then T_i returns *abort* with similar reasoning explained above in rv_method().

If *status* is not false then STM_tryC() sort the keys (exist in $txLog_i$ of T_i) of upd_methods() in increasing order. It takes the method (m_{ij}) from $txLog_i$ one by one and identifies the location of the key k in M_k*.rblazy-list* as explained above in rv_method(). After identifying the preds and currs for k it acquire the locks in predefined order to avoid the deadlock at Line 224 and calls *tryC_Validation()* to validate the methods of T_i.

***tryC_Validation()*:** It identifies whether the methods of invoking transaction T_i are able to create or delete a version corresponding to the keys while ensuring the *starvation-freedom* and maintaining the *real-time order* among the transactions.

First, it do the *rv_Validation()* at Line 164 as explained in rv_method(). If *rv_Validation()* is successful and key k exists in the $M_k.rblazy\text{-}list$ then it identifies the current version ver_j with $ts = j$ such that j is the *largest timestamp smaller (lts)* than i at Line 167. If ver_j is $null$ then SF-K-OSTM returns $abort$ for transaction T_i at Line 169 because it does not find the version to replace otherwise after identifying the current version ver_j it maintains the Current Version List (currVL), Next Version List (nextVL), All Return Value List (allRVL), Large Return Value List (largeRVL), Small Return Value List (smallRVL) from ver_j of key k at Line 171. currVL and nextVL maintain the previous closest version and next immediate version of all the keys accessed in STM_tryC(). allRVL keeps the currVL.rvl whereas largeRVL and smallRVL stores all the wts of currVL.rvl such that ($wts_{currVL.rvl} > wts_i$) and ($wts_{currVL.rvl}$ $< wts_i$) respectively. Acquire the locks on *status* of all the transactions present in allRVL list including T_i it self in predefined order to avoid the deadlock at Line 174. First, it checks the *status* of its own transaction T_i at Line 176. If *status* of T_i is *false* then T_i has to *abort* the same reason as explained in rv_method().

If the *status* of T_i is not *false* then it compares the its_i of its own transaction T_i with the its_p of other transactions T_p present in the largeRVL at Line 179. Along with this it checks the *status* of p. If above conditions $\langle(its_i < its_p)\&\&(p.status == live)\rangle$ succeed then it includes T_p in the Abort Return Value List (abortRVL) at Line 180 to abort it later otherwise abort T_i itself at Line 181.

Algorithm 27 *tryC_Validation()*: It is use for STM_tryC() validation.

```
163: procedure tryC_Validation()
164:     if (!rv_Validation()) then Release the locks and retry.
165:     end if
166:     if (k ∈ $M_k.rblazy\text{-}list$) then
167:         Identify the version $ver_j$ with $ts = j$ such that $j$ is the largest timestamp
         smaller (lts) than $i$ and there exists no other version with timestamp $p$ by $T_p$ on
         key $k$ such that ⟨$j < p < i$⟩.
168:         if ($ver_j$ == $null$) then /*Finite Versions*/
169:             return ⟨$abort_i$⟩
170:         end if
171:         Maintain the list of $ver_j$, $ver_j.vNext$, $ver_j.rvl$, ($ver_j.rvl > i$), and
         ($ver_j.rvl < i$) as prevVL, nextVL, allRVL, largeRVL and smallRVL respectively
         for all key k of $T_i$.
172:         /*$p$ is the timestamp of transaction $T_p$*/
173:         if ($p$ ∈ allRVL) then /*Includes $i$ as well in allRVL*/
174:             Lock status of each $p$ in pre-defined order.
175:         end if
176:         if (i.status == false) then return ⟨$false$⟩.
177:         end if
178:         for all ($p$ ∈ largeRVL) do
179:             if (($its_i$<$its_p$)&&(p.status==live)) then
180:                 Maintain abort list as abortRVL & includes $p$ in it.
181:             else return ⟨$false$⟩. /*abort $i$ itself*/
182:             end if
183:         end for
184:         for all ($ver$ ∈ nextVL) do
185:             Calculate $tutl_i$ = min($tutl_i$, $ver.vNext.vrt - 1$).
186:         end for
187:         for all ($ver$ ∈ currVL) do
188:             Calculate $tltl_i$ = max($tltl_i$, $ver.vrt + 1$).
189:         end for
190:         /*Store current value of global counter as commit time and increment it.*/
191:         $comTime$ = gcounter.add&get(incrVal);
192:         Calculate $tutl_i$ = min($tutl_i$, $comTime$);
193:         if ($tltl_i > tutl_i$) then /*abort $i$ itself*/
194:             return ⟨$false$⟩.
195:         end if
196:         for all ($p$ ∈ smallRVL) do
197:             if ($tltl_p > tutl_i$) then
198:                 if (($its_i$<$its_p$)&&(p.status==live)) then
199:                     Includes $p$ in abortRVL list.
200:                 else return ⟨$false$⟩. /*abort $i$ itself*/
201:                 end if
202:             end if
203:         end for
```

```
204:         $tltl_i$ = $tutl_i$. /*After this point $i$ can't abort*/
205:         **for all** ($p \in$ smallRVL) **do**
206:             /*Only for *live* transactions*/
207:             Calculate the $tutl_p$ = min($tutl_p$, $tltl_i - 1$).
208:         **end for**
209:         **for all** ($p \in$ abortRVL) **do**
210:             Set the *status* of $p$ to be $false$.
211:         **end for**
212:     **end if**
213:     return $\langle true \rangle$.
214: **end procedure**
```

After that STM_tryC() maintains the $tltl_i$ and $tutl_i$ of transaction T_i at Line 188 and Line 185. The requirement of $tltl_i$ and $tutl_i$ is explained above in the rv_method(). If limit of $tltl_i$ crossed with $tutl_i$ then T_i have to abort at Line 194. If $tltl_p$ greater than $tutl_i$ at Line 197 then it checks the its_i and its_p. If $\langle (its_i < its_p) \&\& (p.status == live)) \rangle$ then add the transaction T_p in the abortRVL for all the smallRVL transactions at Line 199 otherwise, *abort* T_i itself at Line 200.

At Line 204, $tltl_i$ would be equal to $tutl_i$ and after this step transaction T_i will never *abort*. T_i helps the other transaction T_p to update the $tutl_p$ which exists in the smallRVL and still $live$ then it sets the $tutl_p$ to minimum of $\langle tutl_p \vee (tltl_i - 1) \rangle$ to maintain the real-time order among the transaction at Line 207. At Line 210, STM_tryC() aborts all other conflicting transactions which are present in the abortRVL while modifying the *status* field to be *false* to achieve *starvation-freedom*.

If all the steps of the *tryC_Validation()* is successful then the actual effect of the *STM_insert*() and *STM_delete*() will be visible to the shared memory. At Line 231, STM_tryC() checks for *poValidation()*. When two subsequent methods $\langle m_{ij}, m_{ik} \rangle$ of the same transaction T_i identify the overlapping location of preds and currs in *rblazy-list*. Then *poValidation()* updates the current method m_{ik} preds and currs with the help of previous method m_{ij} preds and currs.

If m_{ij} is *STM_insert*() and key k is not exist in the $M_k.rblazy\text{-}list$ then it creates the new node n with key k as ⟨*key=k, lock=false, mark=false, vl=ver,* $RL = \phi, BL = \phi$⟩ at Line 233. Later, it creates a version (ver) for transaction T_0 and T_i as ⟨ *ts=0, val=nil, rvl=i, vrt=0, vNext=i* ⟩ and ⟨*ts=i, val=v, rvl=*ϕ*, vrt=i, vNext=*ϕ⟩ at Line 234. The T_0 version created by transaction T_i to helps other concurrent conflicting transactions (with lower timestamp than T_i) to lookup from T_0 version. Finally, it insert the node n into $M_k.rblazy\text{-}list$ such that it is accessible via RL as well as BL at Line 235. If m_{ij} is *STM_insert*() and key k is exist in the $M_k.rblazy\text{-}list$ then it creates the new version ver_i as ⟨*ts=i, val=v, rvl=*ϕ*, vrt=i, vNext=*ϕ⟩ corresponding to key k. If the limit of the version reach to K then SF-K-OSTM replaces the oldest version with $(K+1)^{th}$ version which is accessible via RL as well as BL at Line 238.

If m_{ij} is *STM_delete*() and key k is exist in the $M_k.rblazy\text{-}list$ then it creates the new version

Algorithm 28 *STM_tryC* (T_i): Validate the upd_methods() of T_i and returns *commit*.

```
215: procedure $STM\_tryC(T_i)$
216:     /*Atomically check the status of its own transaction $T_i$ (or $i$)*/
217:     if (i.status == false) then return $\langle abort_i \rangle$.
218:     end if
219:     /*Sort the $keys$ of $txLog_i$ in increasing order.*/
220:     /*Method $(m)$ will be either STM_insert() or STM_delete()*/
221:     for all $(m_{ij} \in txLog_i)$ do
222:         if($m_{ij}$==STM_insert()||$m_{ij}$==STM_delete())then
223:             Identify the preds[] & currs[] for key k in bucket $M_k$ of rblazy-list using BL
                 & RL.
224:             Acquire the locks on preds[] & currs[] in increasing order of keys to
                 avoid deadlock.
225:             if (! $tryC\_Validation()$) then
226:                 return $\langle abort_i \rangle$.
227:             end if
228:         end if
229:     end for
230:     for all $(m_{ij} \in txLog_i)$ do
231:         poValidation() modifies the preds[] & currs[] of current method which
             would have been updated by previous method of the same transaction.
232:         if (($m_{ij}$==STM_insert())&&(k$\notin M_k$.rblazy-list)) then
233:             Create new node $n$ with $k$ as: $\langle$key=k, lock=false, mark= false, vl=ver,
                 RL=$\phi$, BL=$\phi\rangle$.
234:             Create first version $ver$ for $T_0$ and next for $i$: $\langle$ts=i, val=v, rvl=$\phi$, vrt=i,
                 vNext=$\phi\rangle$.
235:             Insert node $n$ into $M_k$.rblazy-list such that it is accessible via RL as
                 well as BL.
236:             /*lock sets true*/
237:         else if ($m_{ij}$ == STM_insert()) then
238:             Add $ver$: $\langle$ts=i, val=v, rvl=$\phi$, vrt=i, vNext=$\phi\rangle$ into $M_k$.rblazy-list &
                 accessible via RL, BL. /*mark=false*/
239:         end if
240:         if ($m_{ij}$ == STM_delete()) then
241:             Add ver:$\langle$ts=i, val=nil, rvl=$\phi$, vrt=i, vNext=$\phi\rangle$ into $M_k$.rblazy-list &
                 accessible via RL only. /*mark=true*/
242:         end if
243:         Update preds[] & currs[] of $m_{ij}$ in $txLog_i$.
244:     end for
245:     Release the locks; return $\langle commit_i \rangle$.
246: end procedure
```

ver_i as $\langle$*ts=i, val=nil, rvl=*ϕ*, vrt=i, vNext=*$\phi\rangle$ which is accessible via RL only at Line 241. At last it updates the preds and currs of each m_{ij} into its $txLog_i$ to help the upcoming methods of the same transactions in *poValidation()* at Line 243. Finally, it releases the locks on all the keys in predefined order and returns *commit* at Line 245.

4.7 Graph Characterization of Local Opacity

This section describes the graph characterization of local opacity for the history H which maintains multiple versions corresponding to each key. Graph Characterization helps to prove the correctness of STMs for a given version order. Lets assume a history H with given version order $\ll$. Following the graph characterization by Chaudhary et al. [23] and modified it for sequential histories with high-level methods while maintaining multiple versions corresponding to each key and extend it for local opacity which helps to prove the correctness of SF-K-OSTM.

Similar to Section 4.2, SF-K-OSTM executes a concurrent history H which consists of multiple transactions. Each transaction calls high-level methods which internally invokes multiple read-write (or lower-level) operations including method invocation and response known as *events* (or $evts$) as discussed in Chapter 2. So, we need to ensure the atomicity on both the levels. First, we ensure method level atomicity using the *linearization point (LP)* of respective method. After that we ensure the atomicity of transaction on the basis of graph characterization of local-opacity.

High-level methods are interval instead of dots (atomic). In order to make it atomic, we order the high-level method on the basis of their *linearization point (LP)*. We consider *first unlocking point* of each successful method as the *LP* of the respective method. Now, we need to ensure the atomicity at transactional level, a transaction internally invokes multiple high-level methods and transactions are overlapping to each other in concurrent history H. So, with the help of *graph characterization* of *local-opacity*, SF-K-OSTM ensures the atomicity of the transaction defined below.

We construct a opacity graph represented as $H.lockOpGraph\ll = (V, E)$ which consists of V vertices and E edges. Here, each committed transaction T_i is consider as a vertex and edges are as follows:

- *real-time (or real-time) edge*: This is same as real-time edge defined in $CG(H, \ll_H)$. If transaction T_i returns commit before the beginning of other transaction T_j then real-time edge goes from T_i to T_j. Formally, ($STM\_tryC_i()$ $\prec_H$ *STM_begin*$_j$()) $\implies$ $T_i \rightarrow T_j$.

- *return value from (or rvf) edge*: There exist a *rvf* edge between two transaction T_i and T_j such that (1) If T_i is the latest transaction that has updated ($upd\_method()$) the key k of hash table, ht and committed; (2) After that T_j invokes a rv_method rvm_j on the same key k of hash table ht and returns the value updated by T_i, i.e., $upd\_method_i()$ $\prec_H^{max\_r}$

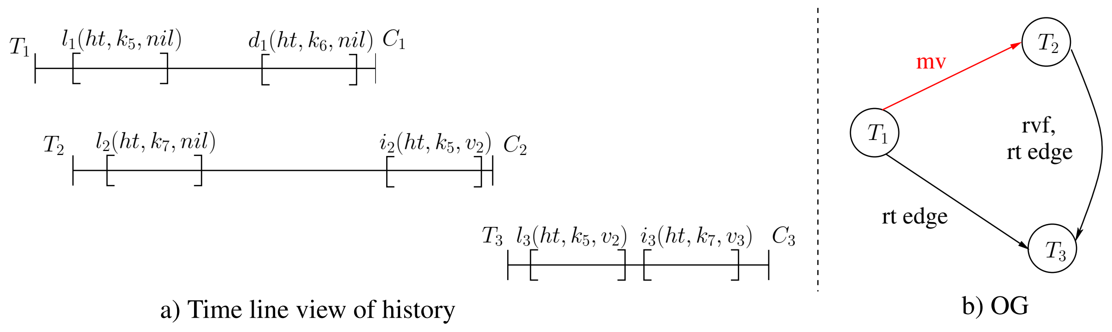


Figure 4.10: Illustration of Graph Characterization of Opacity

rvm$_j$. As defined in Chapter 2, upd_method() can either be *STM_insert()* or *STM_delete()*. If the $upd\_method_i()$ is *STM_insert()* method on key k then rv_method() returns the value updated by T_i, i.e., $i_i(k, v) <_H c_i <_H$ *rvm*$_j(k, v)$. If the $upd\_method_i()$ is *STM_delete()* method on key k then rv_method returns $null$, i.e., $d_i(k, null) <_H c_i <_H$ *rvm*$_j(k, null)$.

- *multi-version (or mv) edge*: It depends on the version order between two transactions T_i and T_j. For the sake of understanding, consider a triplet of three transactions T_i, T_j and T_k with successful methods on key k as *up*$_i(k, u)$, *rvm*$_j(k, u)$, *up*$_k(k, v)$, where $u \neq v$ and up_i stands for upd_method$_i$() of T_i. It can observe that a *return value from edge* is going from T_i to T_j because of *rvm*$_j(k, u)$. If the version order is $k_i \ll k_k$ then the multi-version edge is going from T_j to T_k. Otherwise, multi-version edge is from T_k to T_i because of version order ($k_k \ll k_i$).

For better understanding, we consider a history H: $l_1(ht, k_5, nil), l_2(ht, k_7, nil), d_1(ht, k_6, nil), C_1, i_2(ht, k_5, v_2), C_2, l_3(ht, k_5, v_2), i_3(ht, k_7, v_3), C_3$ and show the time line view of it in Figure 4.10.(a). We construct $H.lockOpGraph\ll = (V, E)$ shown in Figure 4.10.(b). There exist a mv edge between T_1 and T_2 because T_1 lookups the value of key k_5 from T_0 and after that T_2 creates a version on k_5 with value v_2. T_3 begins after the commit of T_1 and T_2 so, real-time edges are going from T_1 to T_3 and T_2 to T_3. Here, T_3 lookups key k_5 from the version created by T_2 and returns the value v_2. So, rvf edge is going from T_2 to T_3. Hence, H constructs an acyclic graph (OG) with equivalent serial schedule $T_1T_2T_3$.

For a given history H and version order $\ll$, we consider a complete graph $\overline{H}$ instead of H and construct the graph $\overline{H}.lockOpGraph\ll$. It can be seen that $\overline{H}$ has more $real - time$ edges than H, i.e., $\prec_H^{RT} \subseteq \prec_{\overline{H}}^{RT}$. But, for the graph construction, we consider only $real - time$ edges of H with the assumption $real - time(H) = real - time(\overline{H})$ that satisfies the following property:

Property 55 *For a given history H and version order $\ll$, the opacity graphs $H.lockOpGraph\ll$ and $\overline{H}.lockOpGraph\ll$ are same.*

Definition 6 *We define a version order $\ll_S$ for t-sequential history S such that if two committed transactions T_i and T_j has created versions on key k as k_i and k_j respectively with version order $k_i \ll_S k_j$ then T_i committed before T_j in S. Formally, $\langle k_i \ll_S k_j \Leftrightarrow T_i <_S T_j \rangle$.*

This definition along with below defined lemmas and theorems will help us to prove the correctness of our graph characterization.

Lemma 56 *The opacity graph for legal t-sequential history S as $S, \ll_S .lockOpGraph$ is acyclic.*

Proof. The proof of this lemma is similar as Lemma 49 of Section 4.2. We order all the transactions of S into real-time order on the basis of their increasing order of timestamp (TS). For example, consider two transaction T_i and T_j with $TS(T_i)$ is less than $TS(T_j)$ then T_i will occur before T_j in S. Formally, $TS(T_i) < TS(T_j) \Leftrightarrow T_i <_S T_j$. We consider all the types edges of $S, \ll_S .lockOpGraph$ and analyze it one by one as follows to show the acyclicity of it:

- real-time edge: It follow that any transaction begin after commit of previous transaction only. Hence, all the real-time edges go from a lower TS transaction T_i to higher TS transaction T_j and follow timestamp order.
- rvf edge: Any transaction T_j lookups key k from T_i in S then T_i has to be committed before invoking of lookup of T_j. So, $TS(T_i) < TS(T_j)$. Hence, all the rvf edges goes from a lower TS transaction to a higher TS transaction.
- mv edge: Consider a triplet of three transactions T_i, T_j and T_k with successful methods on key k as $up_i(k, u)$, $rvm_j(k, u)$, $up_k(k, v)$, where $u \neq v$. Here, $rvm_j(k, u)$ method is returning the latest value written by T_i on key k with value u using $up_i(k, u)$. So, there exist a rvf edge with $TS(T_i) < TS(T_j)$. There are two cases for the version order of k as follows: (1) If the version order is $T_k \ll_S T_i$ which implies that $TS(T_k) < TS(T_i)$ then multi-version edge goes from T_k to T_i which also follows the increasing order of TS. (2) If the version order is $T_i \ll_S T_k$ which implies that $TS(T_i) < TS(T_k)$. Since S is a legal t-sequential history, so, $TS(T_j) < TS(T_k)$ and then multi-version edge goes from T_j to T_k which again follows the increasing order of TS. So, mv edges also follow the increasing order of TS order.

Therefore, all the types of edges follow the increasing order of transaction's TS as defined above. All the edges of S goes from lower TS transaction to higher TS transactions. This implies that the opacity graph generated by legal t-sequential history S as $S, \ll_S .lockOpGraph$ is acyclic.

Lemma 57 *Consider a history H with given version order $\ll_H$. Another history H' is equivalent to H then the mv edges $mv(H, \ll_H)$ induced by $\ll_H$ in H and H' will be same.*

Proof. Since history H and H' are equivalent, so, version order of $\ll_H$ will be same as version order of $\ll_{H'}$. We can observe that mv edges depend on version order $\ll$ and the methods of the history. It is independent from the order of the methods in H. So, being equivalence H' also contains the same version order $\ll_H$ and the methods as in H. Thus, multi-version mv edges are same in H and H'.

Theorem 58 *A valid history H is opaque with a version order $\ll_H$ iff $H.lockOpGraph{\ll_H}$ is acyclic.*

Proof. (if part): First, we consider $H.lockOpGraph{\ll_H}$ is acyclic and we need to prove that history H is opaque. Since $H.lockOpGraph{\ll_H}$ is acyclic, we apply topological sort on $H.lockOpGraph{\ll_H}$ and generate a t-sequential history S such that S is equivalent to $\overline{H}$. $H.lockOpGraph{\ll_H}$ maintains real-time edges as well and S has been generated from it. So, S also respect real-time order (or real-time) as H. Formally, $\prec_H^{RT} \subseteq \prec_S^{RT}$.

$H.lockOpGraph{\ll_H}$ maintains the return value from (or rvf) edges for rv_method() on any key k by transaction T_i returns the value written on k by previously committed transaction T_j. So, S is *valid*. Now, we need to prove that S is *legal*. We prove it by contradiction, so we assume that S is not *legal*. That means, a rv_method() $rvm_j(k, u)$ lookups on key k from a committed transaction T_i which wrote the value of k as u. But between these two transaction T_i and T_j, an another committed transaction T_k exist in S which wrote to k with value v and ($u \neq v$). Formally, $T_i \prec_S^{RT} T_k \prec_S^{RT} T_j$. Consider a given version order in H as $\ll_H$, if the version order is $T_k \ll_S T_i$ then the multi-version edge goes from T_k to T_i. Consider the other version order is $T_i \ll_S T_k$ then the multi-version edge goes from T_j to T_k. So, in both the cases T_k is not coming between T_i and T_j. So, our assumption is wrong. Hence, S is *legal*.

S satisfies all the properties of *opacity* and equivalent to H because S has been generated from the topological sort on $H.lockOpGraph{\ll_H}$. Hence, history H is opaque.

(Only if part): Now, we consider H is opaque and have to prove that $H.lockOpGraph{\ll_H}$ is acyclic. Since H is opaque there exists an equivalent legal t-sequential history S to $\overline{H}$ which maintains real-time (real-time) and conflict ($Conf$) order of H. From the Lemma 56, we can say that opacity graph $S, \ll_S .lockOpGraph$ is acyclic. As we know, $H.lockOpGraph{\ll_H}$ is the subgraph of $S, \ll_S .lockOpGraph$. Hence, $H.lockOpGraph{\ll_H}$ is acyclic.
The above defined lemmas and theorems of *opacity* can be extended the proof of *local-opacity*.

Theorem 59 *A valid history H is locally-opaque iff all the sub-histories $H.subhistSet$ (defined in Chapter 2) for a history H are opaque, i.e., A valid history H is locally-opaque iff the opacity graph $sh.lockOpGraph{\ll_{sh}}$ generated for each sub-history sh of $H.subhistSet$ with given version order $\ll_{sh}$ is acyclic. Formally,*
$\langle (H \text{ is locally-opaque}) \Leftrightarrow (\forall sh \in H.subhistSet, \exists \ll_{sh}: sh.lockOpGraph{\ll_{sh}} \text{ is acyclic}) \rangle$.

Proof. In order to prove it, we need to show that each sub-history sh from the sub-histories of H, $H.subhistSet$ is *valid*. After that the remaining proof follows Theorem 58.

To prove sub-history sh is *valid*, consider a sh with any rv_method() *rvm*$_j(k, u)$ of a transaction T_j. We can easily observe that *rvm* method returns the value of key k that has been written by a committed transaction T_i on k with value u. So, it can be seen that sh has all the transactions that has committed before *rvm*$_j(k, u)$. Similarly, all the *rvm* methods of sh return the value from previously committed transactions. Thus, sub-history sh is *valid*.

Now, we need to show each sub-history sh is opaque with a given version order $\ll_{sh}$ iff $sh.lockOpGraph\ll_{sh}$ is acyclic. The proof of this is directly coming Theorem 58 while replacing H with sh. Similarly, all the sub-histories of $H.subhistSet$ are *valid* and satisfying Theorem 58. So, all the sub-histories of history H are *opaque*. Hence, a valid history H is locally-opaque.

4.8 Liveness Proof of SF-K-OSTM Algorithm

This section describes the liveness proof of SF-K-OSTM algorithm. The liveness proof of SF-K-OSTM is same as the liveness proof of SF-K-RWSTM algorithm described in Section 3.4. It follows the notion derived for SF-K-OSTM algorithm, we assume that all the histories accepted by SF-K-OSTM algorithm as *gen(SF-K-OSTM)*. For simplicity, we consider the sequential histories for our discussion here as well and we can get the sequential history using the linearization points (or LPs) as *first unlocking point of each successful method.*

Theorem 60 *Consider a history $H1$ with T_i be a transaction in $H1.live$. Then there is an extension of $H1$, $H2$ in which an incarnation of T_i, T_j is committed. Formally,* $\langle H1, T_i : (T_i \in H.live) \implies (\exists T_j, H2 : (H1 \sqsubset H2) \wedge (T_j \in H2.incarSet(T_i)) \wedge (T_j \in H2.committed))\rangle$.

Proof. The proof of this theorem is same as the proof of Theorem 39.
From this theorem, we get the following corollary which states that any history generated by SF-K-OSTM is $starvation\text{-}freedom$.

Corollary 61 *SF-K-OSTM algorithm ensures starvation-freedom.*

4.9 Safety Proof of SF-K-OSTM Algorithm

Now, we consider the algorithms defined for each method of *SF-K-OSTM* in SubSection 4.6.2 and prove the correctness of SF-K-OSTM. Consider a history H generated by SF-K-OSTM with two transaction T_i and T_j with status either *live or committed* using $status$ flags as true. Then the edges between T_i and T_j follow the $tltl$ order. SF-K-OSTM algorithm ensures that $tltl$ are keep on increasing order of the transaction timestamp (TS) order using atomic counter $gcounter$ in *STM_begin()*. Though the value of $tltl$ is increasing in the other methods of SF-K-OSTM still its maintaining the increasing order of transactions TS. We assume all the histories generate (or gen) by SF-K-OSTM algorithm as gen(*SF-K-OSTM*).

Lemma 62 *Any history H generated by SF-K-OSTM as $gen(\text{SF-K-OSTM})$ with two transactions T_i and T_j such that status flags of both the transactions are true. If there is an edge from T_i to T_j then $tltl_i$ is less than $tltl_j$. Formally, $T_i \rightarrow T_j \implies tltl_i < tltl_j$.*

Proof. We consider all types of edges in $H.lockOpGraph{\ll_H}$ and analyze it as follows:

- *real-time (or real-time) edge*: Here, the transaction T_i returns commit before the begin of other transaction T_j then real-time edge goes from T_i to T_j. Hence, $tltl_i$ gets the value from $gcounter$ earlier than begin of T_j. So, $T_i \rightarrow T_j \implies tltl_i < tltl_j$.

- *return value from (or rvf) edge*: The transaction T_i has updated ($upd\_method()$) the key k of hash table, ht and committed. After that transaction T_j invokes a rvm_j on the same key k and returns the value updated by T_i. SF-K-OSTM ensures that T_j returns the value of k from the transaction which has lesser TS than T_j i.e., $T_i \rightarrow T_j \implies tltl_i < tltl_j$.

- *multi-version (or mv) edge*: SF-K-OSTM ensures that version order between two transactions T_i and T_j are also following TS order using their $tltl$. Consider a triplet generated by SF-K-OSTM with three transactions T_i, T_j and T_k with successful methods on key k as $up_i(k,u)$, $rvm_j(k,u)$, $up_k(k,v)$, where $u \neq v$. It can observe that a return value from edge is going from T_i to T_j because of $rvm_j(k,u)$, so, $T_i \rightarrow T_j \implies tltl_i < tltl_j$. If the version order is $k_i \ll k_k$ then the multi-version edge is going from T_j to T_k. Hence, the order among the transactions are $(T_i \rightarrow T_j \rightarrow T_k) \implies (tltl_i < tltl_j < tltl_k)$. Otherwise, multi-version edge is from T_k to T_i because of version order ($k_k \ll k_i$). Then the relation is $(T_k \rightarrow T_i \rightarrow T_j) \implies (tltl_k < tltl_i < tltl_j)$.

So, all the edges of $H.lockOpGraph{\ll_H}$ are following increasing order of transactions $tltl$. Hence, $T_i \rightarrow T_j \implies tltl_i < tltl_j$.

Theorem 63 *A valid SF-K-OSTM history H is locally-opaque iff $H.lockOpGraph{\ll_H}$ is acyclic.*

Proof. (if part): We consider H is *valid* and $H.lockOpGraph{\ll_H}$ generated by SF-K-OSTM is acyclic then we need to prove that history H is locally-opaque. Since $H.lockOpGraph{\ll_H}$ is acyclic, we apply topological sort on $H.lockOpGraph{\ll_H}$ and obtained a t-sequential history S which is equivalent to $\overline{H}$. S also respect real-time (or RT), return value from (or RVF) and multi-version (or MV) edges as $\overline{H}$. Formally, S respects $\prec_S^{RT} = \prec_{\overline{H}}^{RT}$, $\prec_S^{RVF} = \prec_{\overline{H}}^{RVF}$ and $\prec_S^{MV} = \prec_{\overline{H}}^{MV}$.

Since rvf and real-time relation between the methods of SF-K-OSTM for a given version in S are also present in $\overline{H}$. Formally, $\prec_{\overline{H}}^{RT} \subseteq \prec_S^{RT}$ and $\prec_{\overline{H}}^{RVF} \subseteq \prec_S^{RVF}$. Given that H is *valid* which implies that $\overline{H}$ is also *valid*. So, we can say that S is *legal* for a given version order. Similarly, we can prove that all the sub-histories $H.subhistSet$ for a history H are opaque. Hence with the help of Theorem 59, collectively, H satisfies all the necessary conditions of *local-opacity*. Hence, history H is locally-opaque.

(Only if part): Now, we consider H is locally-opaque and valid then we have to prove that $H.lockOpGraph{\ll_H}$ is acyclic. We prove it through contradiction, so we assume there exist a cycle in $H.lockOpGraph{\ll_H}$. From Lemma 62, any two transactions T_i and T_j generated by SF-K-OSTM such that both their status flags are true and $T_i \rightarrow T_j \implies tltl_i < tltl_j$. Consider the transitive case with k transactions $T_1, T_2, T_3 ... T_k$ such that status flags of all the k transactions are true. If edges exist like ($T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow .... \rightarrow T_k$) $\implies$ ($tltl_1 < tltl_2 < tltl_3$ $< .... < tltl_k$).
Now, we consider our assumption, there exist a cycle in $H.lockOpGraph{\ll_H}$. So, $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow .... \rightarrow T_k \rightarrow T_1$ that implies $tltl_1 < tltl_2 < tltl_3 < .... < tltl_k < tltl_1$.
Hence, above assumption says that, $tltl_1 < tltl_1$ but this is not possible. So, our assumption there exist a cycle in $H.lockOpGraph{\ll_H}$ is wrong.
Therefore, $H.lockOpGraph{\ll_H}$ produced by SF-K-OSTM is acyclic.

Theorem 64 *Any valid history H generated by SF-K-OSTM satisfies local-opacity.*

Proof. With the help of Lemma 62, we can say that any history H gen(*SF-K-OSTM*) with two transactions T_i and T_j such that status flags of both the transactions are true. If there is an edge from T_i to T_j then $tltl_i$ is less than $tltl_j$. Formally, $T_i \rightarrow T_j \implies tltl_i < tltl_j$. So, we can infer that any valid history H generated by SF-K-OSTM following the edges $T_i \rightarrow T_j$ in increasing order of $tltl_i < tltl_j$ with the help of atomic G_tCntr. Hence, we can conclude that SF-K-OSTM always produce an acyclic $H.lockOpGraph{\ll_H}$ graph.

Now, using Theorem 63, we can infer that if a valid history H generated by SF-K-OSTM always produces an acyclic $H.lockOpGraph{\ll_H}$ graph then H is locally-opaque. Hence, any valid history H generated by SF-K-OSTM satisfies local-opacity.

4.10 Experimental Evaluations

This section represents the experimental analysis of variants of the proposed Starvation-Free Object-based STMs (SF-SV-OSTM, SF-MV-OSTM, SF-MV-OSTM-GC, and SF-K-OSTM)[2] for two data structure *hash table* (HT-SF-SV-OSTM, HT-SF-MV-OSTM, HT-SF-MV-OSTM-GC and HT-SF-K-OSTM) and *linked-list* (list-SF-SV-OSTM, list-SF-MV-OSTM, list-SF-MV-OSTM-GC and list-SF-K-OSTM) implemented in C++. We analyzed that HT-SF-K-OSTM and list-SF-K-OSTM perform best among all the proposed algorithms. So, we compared our HT-SF-K-OSTM with hash table based state-of-the-art STMs HT-K-OSTM [16], HT-SV-OSTM [9], ESTM [25], RWSTM [2, Chap. 4], HT-MVTO [10] and our list-SF-K-OSTM with list based state-of-the-art STMs list-K-OSTM [16], list-SV-OSTM [9], Trans-list [27], Boosting-list [14], NOrec-list [24], list-MVTO [10], list-SF-K-RWSTM [23].

[2]Code is available here: https://github.com/PDCRL/SF-MVOSTM.

Experimental Setup: The system configuration for experiments is 2 socket Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz with 14 cores per socket and 2 hyper-threads per core, a total of 56 threads. A private 32KB L1 cache and 256 KB L2 cache is with each core. It has 32 GB RAM with Ubuntu 16.04.2 LTS running Operating System. Default scheduling algorithm of Linux with all threads have the same base priority is used in our experiments. This satisfies Assumption 3 (bounded-termination) of the scheduler and we ensure the absence of parasitic transactions for our setup to satisfy Assumption 4.

Methodology: We have considered three different types of workloads namely, W1 (Lookup Intensive - 5% insert, 5% delete, and 90% lookup), W2 (Mid Intensive - 25% insert, 25% delete, and 50% lookup), and W3 (Update Intensive - 45% insert, 45% delete, and 10% lookup). To analyze the absolute benefit of starvation-freedom, we used a customized application called as the *Counter Application* (described in SubSection 4.10.1) which provides us the flexibility to create a high contention environment where the probability of transactions undergoing starvation on an average is very high. Our *high contention* environment includes only 30 shared data-items (or keys), number of threads ranging from 50 to 250, each thread spawns upon a transaction, where each transaction performs 10 operations depending upon the workload chosen. To study starvation-freedom of various algorithms, we have used *max-time* which is the maximum time required by a transaction to finally commit from its first incarnation, which also involves time taken by all its aborted incarnations. For accuracy, all the experiments are averaged over 11 runs in which the first run is discarded and considered as a warm-up run.

4.10.1 Pseudocode of Counter Application

To analyze the absolute benefit of starvation-freedom, we use a *Counter Application* which provides us the flexibility to create a high contention environment where the probability of transactions undergoing starvation on an average is very high. In this subsection we described the detailed functionality of *Counter Application* though pseudocode as follows:

Algorithm 29 *main()*: The main function invoked by *Counter Application*.

```
247: /*Each thread $th_i$ log abort counts, average time taken by each transaction to commit and worst case time (maximum time to commit the transaction) in $abortCount_{th_i}$, $timeTaken_{th_i}$ and $worstTime_{th_i}$ respectively;*/
248: for all (numOfThreads) do /*Multiple threads call the helper function*/
249:     helperFun();
250: end for
251: for all (numOfThreads) do
252:     /*Join all the threads*/
253: end for
254: for all (numOfThreads) do
255:     if ($maxWorstTime < worstTime_{th_i}$) then
256:         /*Calculate the Maximum Worst Case Time*/
```

```
257:            $maxWorstTime = worstTime_{th_i}$;
258:        end if
259:        /*Calculate the Total Abort Count*/
260:        $totalAbortCount$ += $abortCount_{th_i}$;
261:        /*Calculate the Average Time Taken*/
262:        $AvgTimeTaken$ /= $TimeTaken_{th_i}$;
263:    end for
```

Algorithm 30 *helperFun()*:Multiple threads invoke this function.

```
264:    Initialize the Transaction Count $txCount_i$ of $T_i$ as 0;
265:    while (numOfTransactions) do /*Execute until number of transactions are non zero*/
266:        $startTime_{th_i}$ = timeRequest(); /*get the start time of thread $th_i$*/
267:        /*Execute the transactions $T_i$ by invoking testSTM functions;*/
268:        $abortCount_{th_i} = testSTM_i()$;
269:        Increment the $txCount_i$ of $T_i$ by one.
270:        $endTime_{th_i}$ = timeRequest(); /*get the end time of thread $th_i$*/
271:        /*Calculate the Total Time Taken by each thread $th_i$*/
272:        $timeTaken_{th_i}$ += ($endTime_{th_i}$ - $startTime_{th_i}$);
273:        /*Calculate the Worst Case Time taken by each thread $th_i$*/
274:        if ($worstTime_{th_i} <$ ($endTime_{th_i}$ - $startTime_{th_i}$)) then
275:            $worstTime_{th_i}$ = ($endTime_{th_i}$ - $startTime_{th_i}$);
276:        end if
277:        Atomically, decrement the numOfTransactions;
278:    end while
279:    /*Calculate the Average Time taken by each thread $th_i$*/
280:    $TimeTaken_{th_i}$ /= $txCount_i$;
```

Algorithm 31 $testSTM_i()$: Main function which executes the methods of the transaction T_i (or i) by thread th_i.

```
281:    while (true) do
282:        if (i.its != nil) then
283:            STM_begin(i.its); /*If $T_i$ is an incarnation*/
284:        else
285:            STM_begin(nil); /*If $T_i$ is first invocation*/
286:        end if
287:        for all (numOfMethods) do
288:            $k_i$ = rand()%totalKeys;/*Select the key randomly*/
289:            $m_i$ = rand()%100;/*Select the method randomly*/
```

```
290:         switch ($m_i$) do
291:             case ($m_i \leq$ STM_lookup()):
292:                 $v \leftarrow$ STM_lookup($k_i$); /*Lookup key $k$ from a shared memory*/
293:                 if ($v$ == $abort$) then
294:                     $txAbortCount_i++$; /*Increment the transaction abort count*/
295:                     goto Line 282;
296:                 end if
297:             case (STM_lookup() $< m_i \leq$ STM_insert()):
298:                 /*Insert key $k_i$ into $T_i$ local memory with value $v$*/
299:                 STM_insert($k_i, v$);
300:             case (STM_insert() $< m_i \leq$ STM_delete()):
301:                 /*Actual deletion happens after successful STM_tryC()*/
302:                 STM_delete($k_i$);
303:             case default:
304:                 /*Neither lookup nor insert/delete on shared memory*/
305:             $v$ = STM_tryC(); /*Validate all the methods of $T_i$ in tryC*/
306:             if ($v$ == abort) then
307:                 $txAbortCount_i++$;
308:                 goto Line 282;
309:             end if
310:     end for
311:     return $\langle txAbortCount_i \rangle$;
312: end while
```

4.10.2 Result Analysis

All our results reflect the same ideology as proposed showcasing the benefits of Starvation-Freedom in Multi-Version OSTMs. We started our experiments with *hash table* data structure of bucket size 5.

First, we compared *max-time* taken by all our proposed HT-SF-SV-OSTM and variations of HT-SF-K-OSTM (HT-SF-MV-OSTM and HT-SF-MV-OSTM-GC) while varying the number of threads from 50 to 250. We did these experiments on *high contention environment* which includes only 30 keys, number of threads ranging from 50 to 250, each thread spawns upon a transaction, where each transaction performs 10 operations (insert, delete and lookup) depending upon the workload chosen. We analyze through Figure 4.11 that the finite version HT-SF-K-OSTM performs best among all the proposed algorithms on all the three types of workloads (W1, W2, and W3) with value of K and C as 5 and 0.1 respectively. Here, K is the number of versions in the version list and C is the variable used to derive the wts. Similarly, we consider

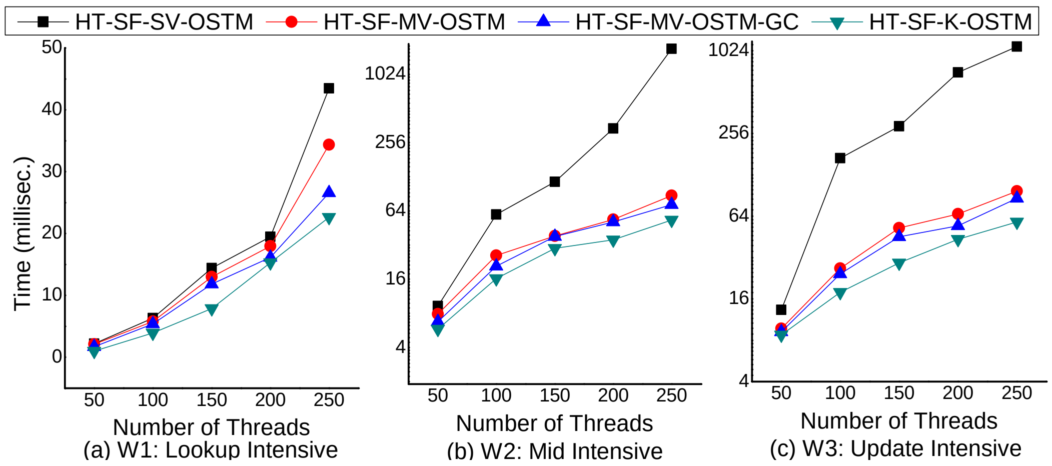


Figure 4.11: Performance analysis among SF-SV-OSTM and variants of SF-K-OSTM on hash table

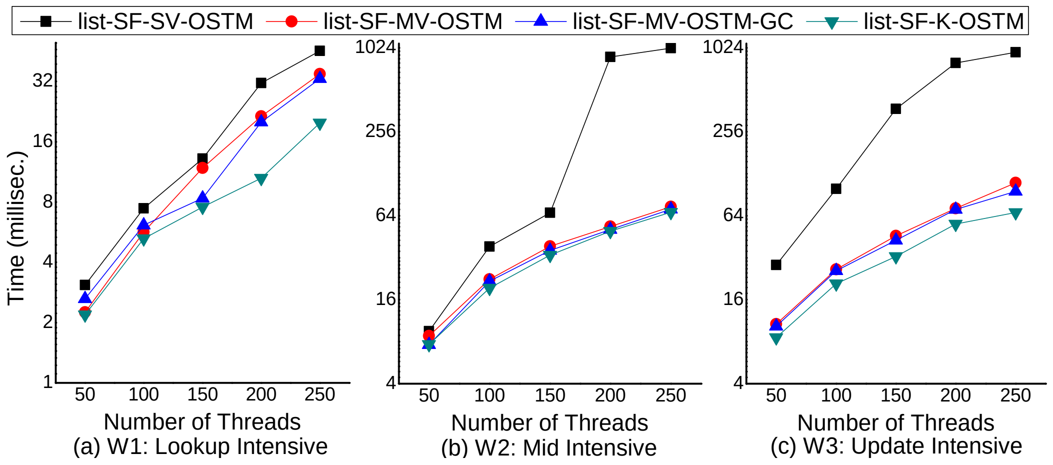


Figure 4.12: Performance analysis among SF-SV-OSTM and variants of SF-K-OSTM on list

another data structure as linked-list on *high contention environment* and compared *max-time* taken by all our proposed list-SF-SV-OSTM, list-SF-MV-OSTM, list-SF-MV-OSTM-GC, and list-SF-K-OSTM algorithms. Figure 4.12 represents that list-SF-K-OSTM performs best for all the workloads (W1, W2, and W3) with value of K and C as 5 and 0.1 respectively.

We compared *max-time* for a transaction to commit by proposed HT-SF-K-OSTM with hash table based state-of-the-art STMs. HT-SF-K-OSTM achieved an average speedup of 3.9x, 32.18x, 22.67x, 10.8x and 17.1x over HT-K-OSTM [16], HT-SV-OSTM [9], ESTM [25], RW-STM [2, Chap. 4], and HT-MVTO [10] respectively as shown in Figure 4.13.

We further considered another data structure *linked-list* and compared *max-time* for a transaction to commit by proposed list-SF-K-OSTM with list based state-of-the-arts STMs. list-SF-K-OSTM achieved an average speedup of 2.4x, 10.6x, 7.37x, 36.7x, 9.05x, 14.47x, and 1.43x over list-K-OSTM [16], list-SV-OSTM [9], Trans-list [27], Boosting-list [14], NOrec-list [24], list-MVTO [10] and list-SF-K-RWSTM [23] respectively as shown in Figure 4.14. We consider number of versions in the version list K as 5 and value of C as 0.1.

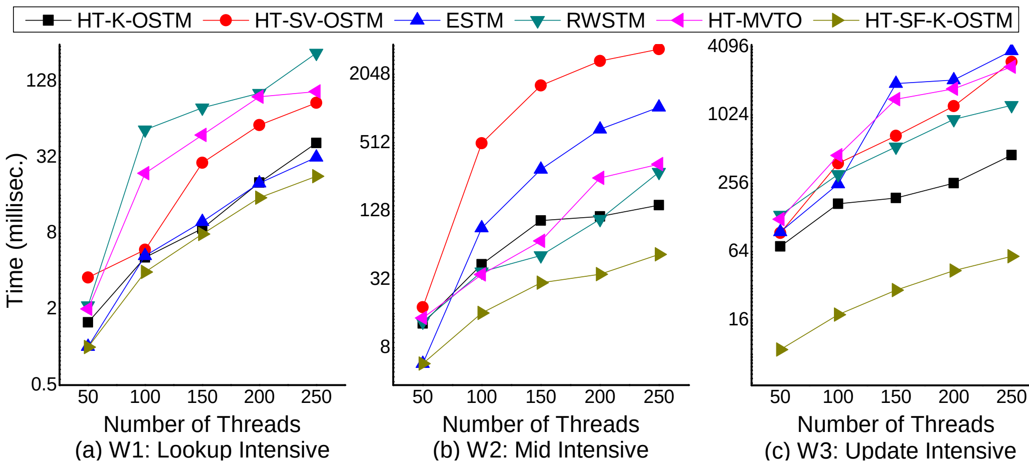


Figure 4.13: Performance analysis of SF-K-OSTM and State-of-the-art STMs on hash table

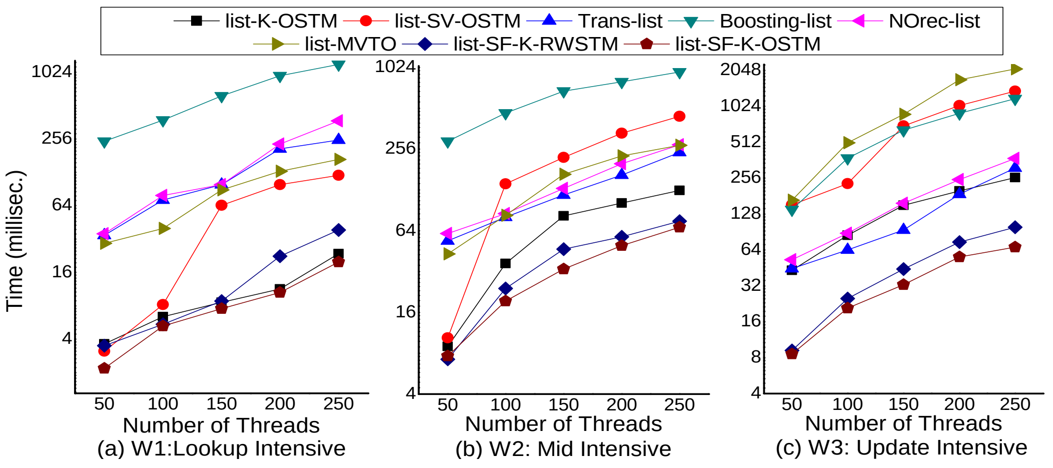


Figure 4.14: Performance analysis of SF-K-OSTM and State-of-the-art STMs on list

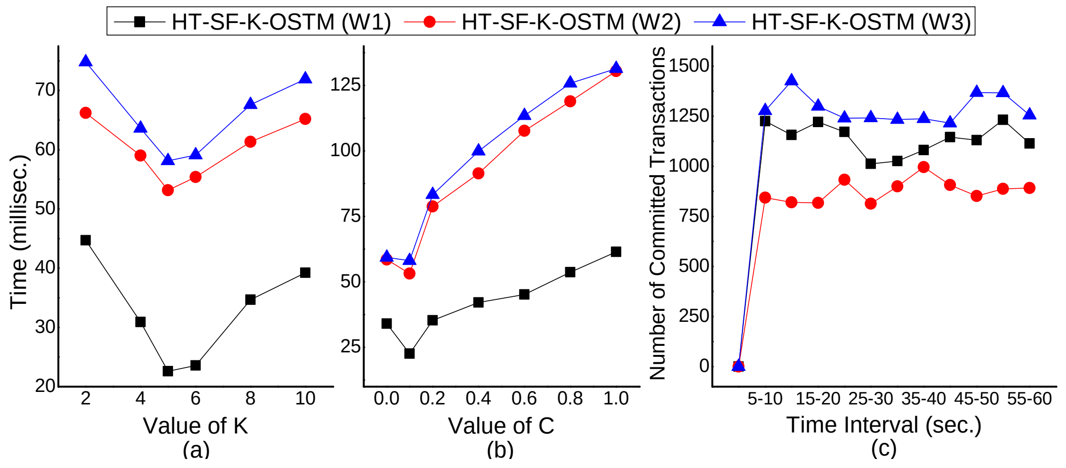


Figure 4.15: Optimal value of K and C along with Stability for hash table

Best value of K, C, and Stability in SF-K-OSTM: We identified the best value of K for both HT-SF-K-OSTM and list-SF-K-OSTM algorithms. The best value of K depends on the application. Figure 4.15.(a). demonstrates the best value of K as 5 for HT-SF-K-OSTM on counter application. We achieve this while varying value of K on *high contention environment* with 64 threads on all the workloads $W1, W2, W3$. Figure 4.15.(b). illustrates the best value of C as 0.1 for HT-SF-K-OSTM on all the workloads $W1, W2, W3$. Figure 4.15.(c). represents the $stability$ of HT-SF-K-OSTM algorithm overtime for the counter application. For this experiment, we fixed 32 threads, 1000 shared data-items (or keys), the value of K as 5, and C as 0.1. Each thread invokes transactions until its time-bound of 60 seconds expires. We calculate the number of transactions committed in the incremental interval of 5 seconds. Figure 4.15.(c). shows that over time HT-SF-K-OSTM is stable which helps to hold the claim that the performance of HT-SF-K-OSTM will continue in the same manner if time is increased to higher orders. Similarly, we perform the same experiments for the linked-list data structure as well. Figure 4.16.(a). and Figure 4.16.(b). demonstrate the best value of K as 5 and C as 0.1 for list-SF-K-OSTM on all the workloads $W1, W2, W3$. Similarly, Figure 4.16.(c). illustrates the $stability$ of list-SF-K-OSTM and shows that it is stable over time on all the workloads $W1, W2, W3$.

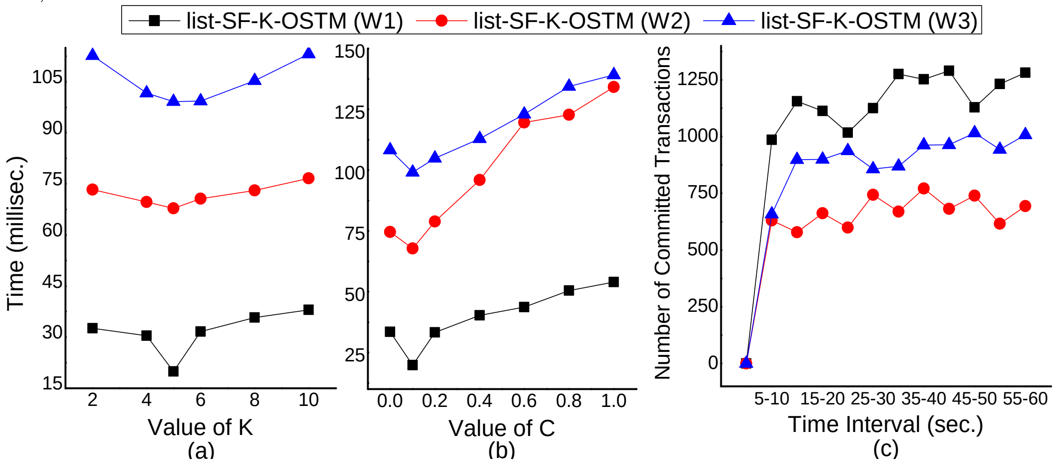


Figure 4.16: Optimal value of K and C along with Stability for list

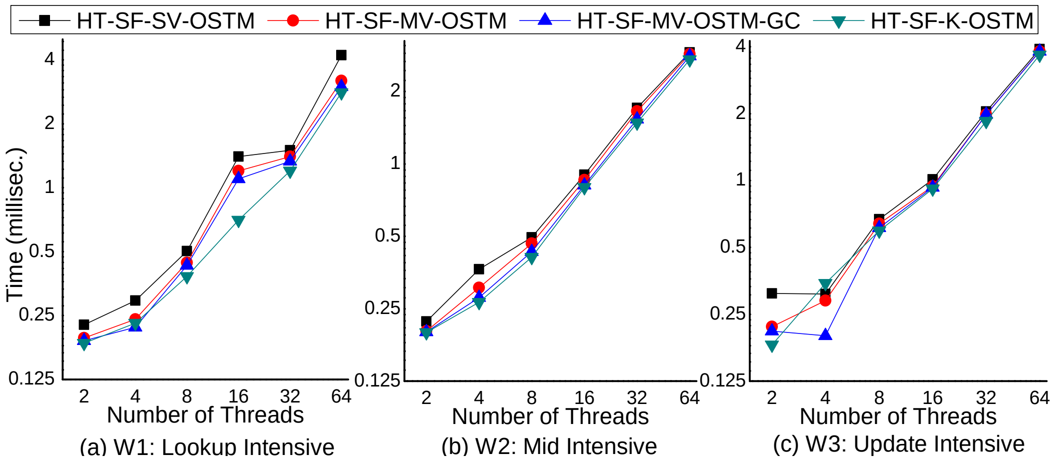


Figure 4.17: Time comparison among SF-SV-OSTM and variants of SF-K-OSTM on hash table

We have done some experiments on *low contention environment* which involves 1000 keys, threads varying from 2 to 64 in power of 2, each thread spawns one transaction and each transaction executes 10 operations (insert, delete and lookup) depending upon the workload chosen. We observed that HT-SF-K-OSTM performs best out of all the proposed algorithms on $W1$ and $W2$ workload as shown in Figure 4.17. For a lesser number of threads on $W3$, HT-SF-K-OSTM was taking a bit more time than other proposed algorithms as shown in Figure 4.17.(c). This may be because of the finite version, finding and replacing the oldest version is taking time. After that, we consider HT-SF-K-OSTM and compared against state-of-the-art STMs. Figure 4.18 shows that our proposed algorithm performs better than all other state-of-art-STMs algorithms but slightly lesser than the non starvation-free HT-K-OSTM. But to provide the guarantee of starvation-freedom this slight slag of time is worth paying. For the better clarification of speedups, please refer to Table 4.2.

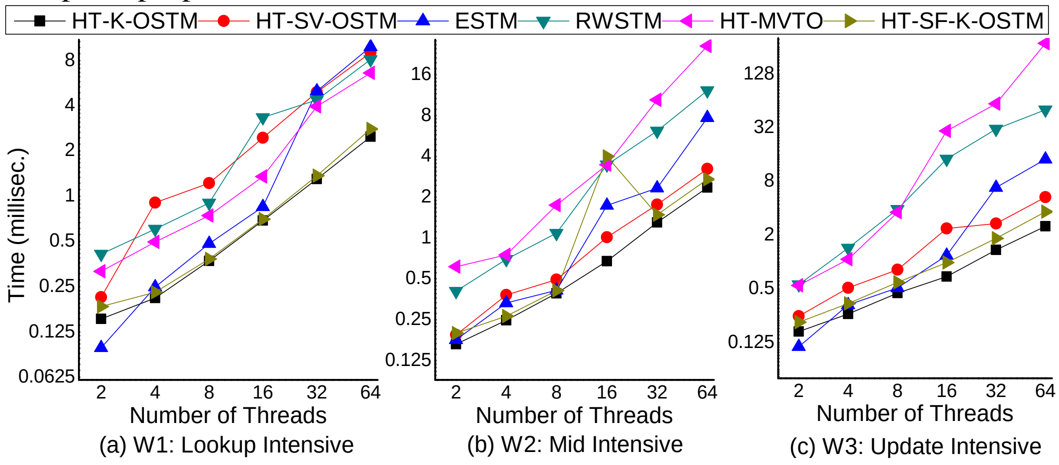


Figure 4.18: Time comparison of SF-K-OSTM and State-of-the-art STMs on hash table

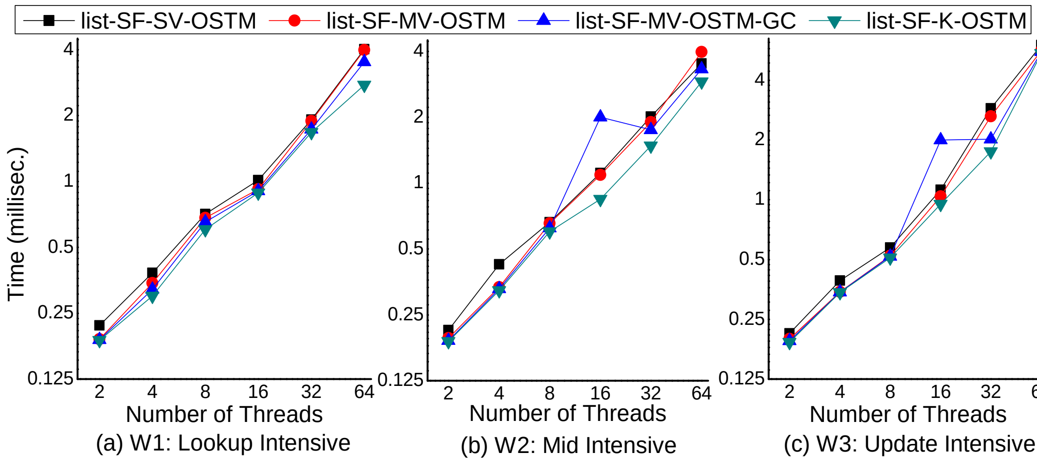


Figure 4.19: Time comparison among SF-SV-OSTM and variants of SF-K-OSTM on list

Similarly, Figure 4.19 represents the analysis of *low contention environment* for list data structure where list-SF-K-OSTM performs best out of all the proposed algorithms. Figure 4.20 demonstrates the comparison of proposed list-SF-K-OSTM with list based state-of-the-art STMs and shows the significant performance gain in terms of a speedup as presented in Table 4.3. For low contention environment, starvation-freedom is appearing as an overhead so, both HT-

SF-K-OSTM and list-SF-K-OSTM achieve a bit less speedup than HT-K-OSTM and list-K-OSTM. But for high contention environment, starvation-free algorithms are always better so, both HT-SF-K-OSTM and list-SF-K-OSTM achieve better speedup than HT-K-OSTM and list-K-OSTM.

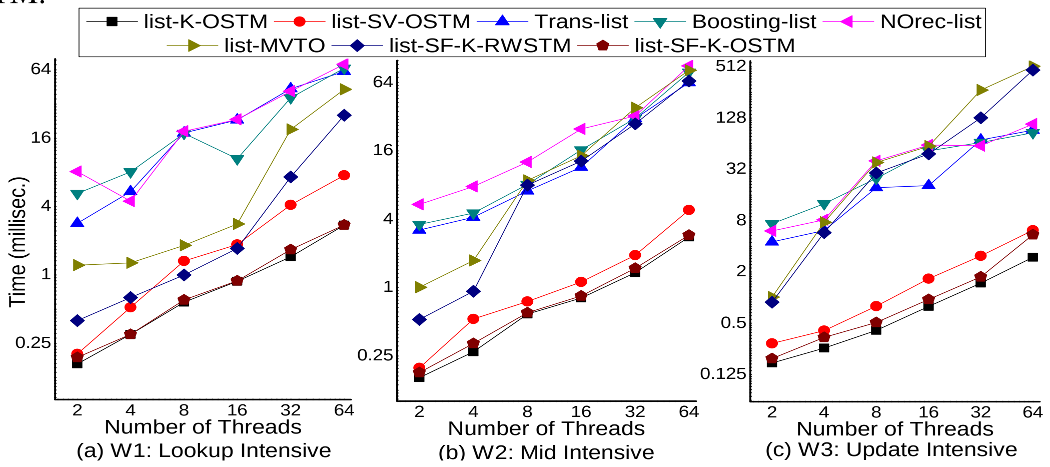


Figure 4.20: Time comparison of SF-K-OSTM and State-of-the-art STMs on list

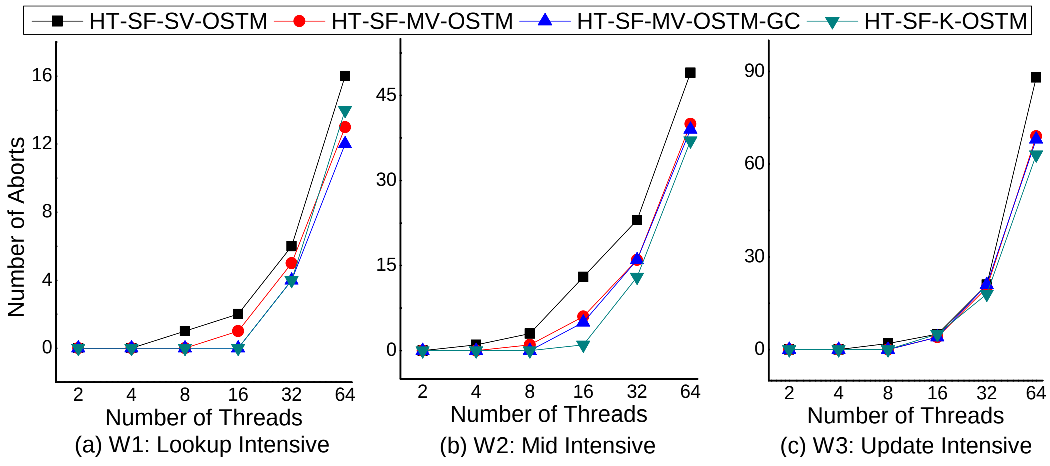


Figure 4.21: Abort Count of SF-SV-OSTM and variants of SF-K-OSTM on hash table

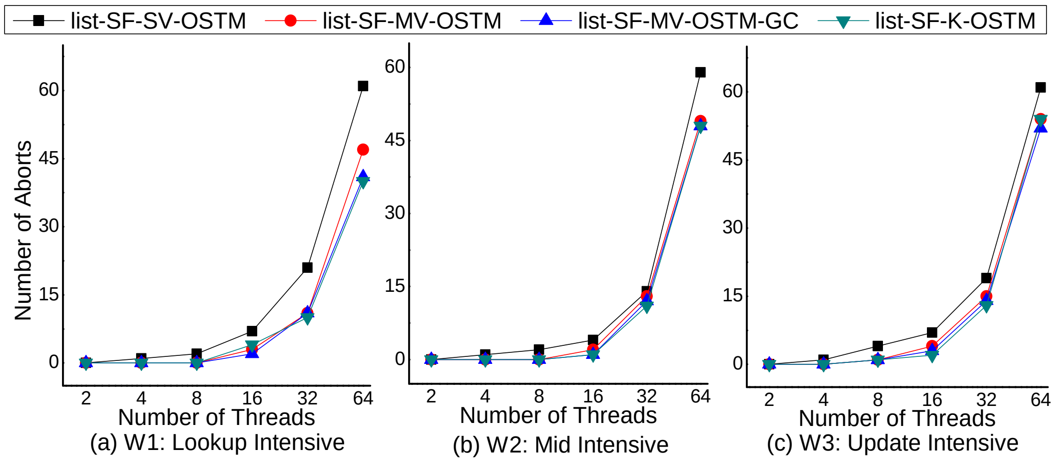


Figure 4.22: Abort Count of SF-SV-OSTM and variants of SF-K-OSTM on list

Abort Count: We analyzed the number of aborts on low contention environment as defined above. Figure 4.21 and Figure 4.22 show the number of aborts comparison among all the proposed variants in all three workloads W1, W2 and W3 for both data structures hash table and linked-list. The results show that HT-SF-K-OSTM and list-SF-K-OSTM have relatively less number of aborts than other proposed algorithms. Similarly, Figure 4.23 and Figure 4.24 shows the number of aborts comparison among proposed HT-SF-K-OSTM with hash table based state-of-the-art STMs and proposed list-SF-K-OSTM with list based state-of-the-art STMs in all three workloads W1, W2, and W3. The result shows that the least number of aborts are happening with HT-SF-K-OSTM and list-SF-K-OSTM.

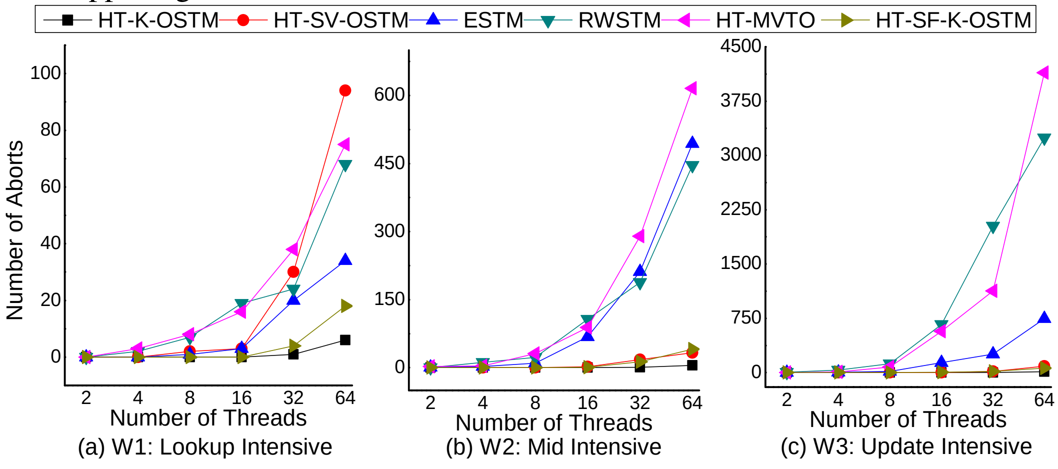


Figure 4.23: Abort Count of SF-K-OSTM and State-of-the-art STMs on hash table

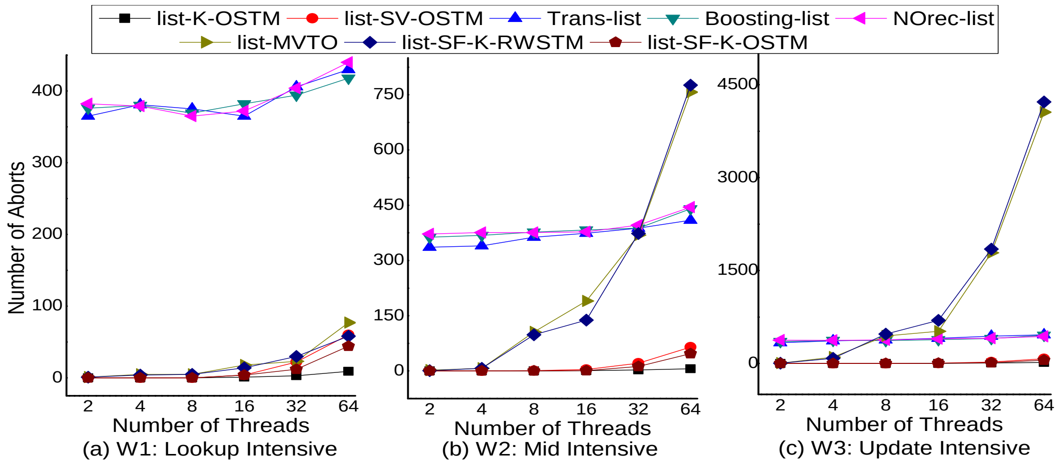


Figure 4.24: Abort Count of SF-K-OSTM and State-of-the-art STMs on list

Garbage Collection: To delete the unwanted versions, we use garbage collection mechanism in SF-MV-OSTM and proposed SF-MV-OSTM-GC. The results show that SF-MV-OSTM-GC performs better than SF-MV-OSTM. Garbage collection method deletes the unwanted version corresponding to the key. In garbage collection we use a *livelist*, this livelist contains all the transaction that are live, which means every transaction on the start of its first incarnation logs its time in livelist and when commit/abort remove its entry from livelist in sorted order of transactions on the basis of wts. Garbage collection is achieved by deleting the version which is not latest, whose timestamp is smaller than the wts of smallest live transaction. Figure 4.25

represents the ***Memory Consumption*** by SF-MV-OSTM-GC and SF-K-OSTM algorithms for *high contention (or HC)* and *low contention (or LC)* environment on workload $W3$ for the linked-list data structure. Here, each algorithm creates a version corresponding to the key after successful *STM_tryC()*.

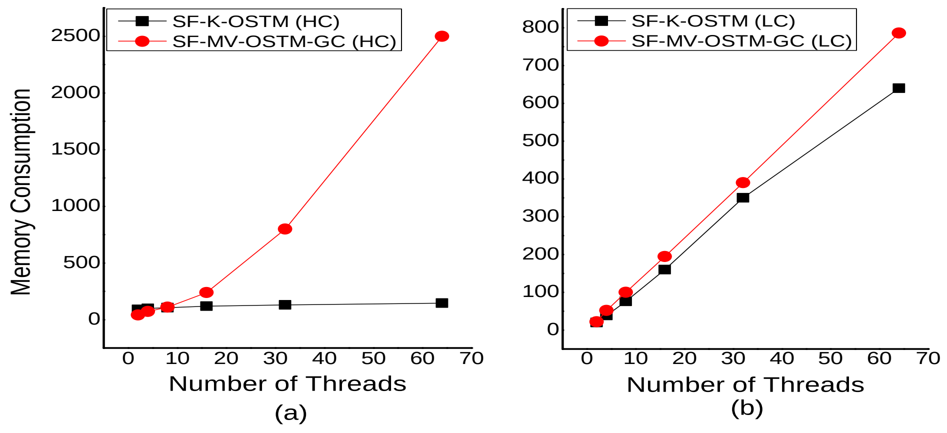


Figure 4.25: Comparison of Memory Consumption in SF-K-OSTM and SF-MV-OSTM-GC

Algorithm	W1	W2	W3
HT-SF-SV-OSTM	1.49	1.13	1.09
HT-SF-MV-OSTM	1.22	1.08	1.04
HT-SF-MV-OSTM-GC	1.13	1.03	1.02
HT-SV-OSTM	3.3	0.77	1.57
ESTM	2.92	1.4	3.03
RWSTM	3.13	2.65	13.36
HT-MVTO	2.37	4.74	49.39
HT-K-OSTM	0.91	0.7	0.8

Table 4.2: Speedup by HT-SF-K-OSTM

Algorithm	W1	W2	W3
list-SF-SV-OSTM	1.29	1.26	1.22
list-SF-MV-OSTM	1.25	1.29	1.12
list-SF-MV-OSTM-GC	1.14	1.5	1.13
list-SV-OSTM	2.4	1.5	1.4
Trans-list	24.15	19.06	23.26
Boosting-list	22.43	22.52	27.2
NOrec-list	26.12	27.33	31.05
list-MVTO	10.8	23.1	19.57
list-SF-K-RWSTM	5.7	18.4	74.20
list-K-OSTM	0.96	0.98	0.8

Table 4.3: Speedup by list-SF-K-OSTM

We calculate the memory consumption based on the *Version Count (or VC)*. If version is created then VC is incremented by 1 and after garbage collection, VC is decremented by 1. Figure 4.25. (a). demonstrates that memory consumption is kept on increasing in SF-MV-OSTM-GC but memory consumption by SF-K-OSTM is constant because of maintaining finite versions corresponding to the less number of keys (high contention). Figure 4.25. (b). shows for *low contention* environment where memory consumption are keeps on increasing in SF-MV-OSTM-GC as well as SF-K-OSTM. But once limit of K-version reach corresponding to all the keys in SF-K-OSTM, memory consumption will be stable. Similar observation can be found for other workloads W_1, W_2 and other data structure hash table as well.

4.11 Summary

We proposed a novel *Starvation-Free K-Version Object-based STM (SF-K-OSTM)* which ensure the *starvation-freedom* while maintaining the latest K-versions corresponding to each key and satisfies the correctness criteria as *local-opacity*. The value of K can vary from 1 to ∞. When K is equal to 1 then SF-K-OSTM boils down to *Single-Version Starvation-Free OSTM (SF-SV-OSTM)*. When K is ∞ then SF-K-OSTM algorithm maintains unbounded versions corresponding to each key known as *Multi-Version Starvation-Free OSTM (SF-MV-OSTM)*. To delete the unused version from the version list, SF-MV-OSTM calls a separate Garbage Collection (GC) method and proposed SF-MV-OSTM-GC. SF-K-OSTM provides greater concurrency and higher throughput using higher-level methods. We implemented all the proposed algorithms for *hash table* and *linked-list* data structure but it is generic for other data structures as well. Results of SF-K-OSTM shows significant performance gain over state-of-the-art STMs.

Chapter 5

Application of Efficient Multi-Version STMs: Blockchain

5.1 Introduction

It is commonly believed that blockchain is a revolutionary technology for doing business over the Internet. Blockchain is a decentralized, distributed database or ledger of records. Cryptocurrencies such as Bitcoin [28] and Ethereum [29] were the first to popularize the blockchain technology. Blockchains ensure that the records are tamper-proof but publicly readable. This distributed database is maintained in a peer-to-peer network where the copy of the entire blockchain is stored at each node of the system. Realizing the effectiveness of blockchains they are being used in several other applications apart from cryptocurrencies. For instance, several governments worldwide are considering to use blockchains for automating and securely storing user records such as land sale documents, vehicle records, insurance records, etc.

5.1.1 Current Blockchain Design

Basically, the blockchain network consists of multiple peers (or nodes) where the peers do not necessarily trust each other. Each node maintains a copy of the distributed ledger. *Clients*, users of the blockchain, send requests or *transactions* to the nodes of the blockchain called as *miners*. The miners collect multiple transactions from the clients and form a *block*. Miners then propose these blocks to be added to the blockchain. They follow a global consensus protocol to agree on which blocks are chosen to be added and in what order. While adding a block to the blockchain, the miner incorporates the hash of the previous block into the current block. This makes it difficult to tamper with the distributed ledger. The resulting structure is in the form of a linked list or a chain of blocks and hence the name blockchain.

The transactions sent by clients to miners are part of a larger code called as *smart contracts* that provide several complex services such as managing the system state, ensuring rules, or credentials checking of the parties involved [30]. Smart contracts are like a ‘class’ in programming languages that encapsulate data and methods which operate on the data. The data represents the state of the smart contract (as well as the blockchain) and the methods (or functions) are the transactions that possibly can change contract state. A transaction invoked by a client is typically such a method or a collection of methods of the smart contracts. Ethereum uses Solidity [31] while Hyperledger [32] supports language such as Java, Golang, Node.js etc.

5.1.2 Bottleneck in Current Blockchain Design

A peer m on receiving sufficient number of smart contract transactions (SCTs) from clients, packages them to a block, say b. Such a peer is called *miner* in Ethereum. Miner m sequentially executes these smart contract transactions one after another to obtain the final state of the blockchain which it stores in the block as well. To maintain the chain structure, m adds the hash of the previous block to the current block b and proposes this new block to be added to the blockchain.

All the nodes in the system execute a global consensus protocol to decide the order of b in the blockchain. As a part of the consensus protocol, the remaining nodes, *validators* validate the contents of the block b. They execute all the smart contract transactions of b one after another sequentially to obtain the final state of the blockchain, assuming that b will be added to the blockchain. If the computed final state is the same as the final state in b then it is accepted by the validators. In this case, the miner m gets an incentive for adding b to the blockchain (in Ethereum and other cryptocurrency-based blockchains). On the other hand, if the computed final state does not match with the final state in the block, then b is rejected, and m does not get any incentive.

It can be seen that the working of Ethereum blockchain described above follows order-execute model [32]: blockchain orders smart contract transactions first, and then re-executes them in the same order on all peers. Several existing blockchains such as Bitcoin, EOS [33] follow this model.

5.1.3 Motivation for Concurrent Execution of Smart Contracts

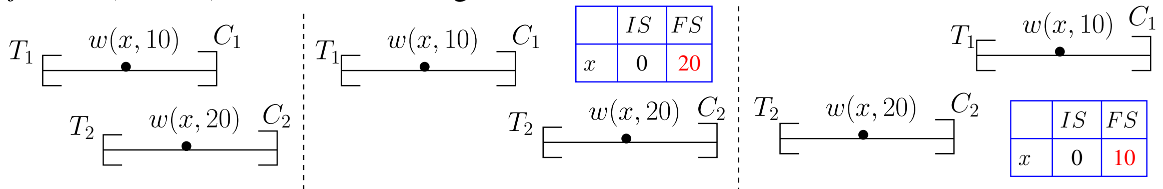
As observed by Dickerson et al. [30], smart contract transactions are executed in two different contexts specifically in Ethereum. First, they are executed by miners while forming a block. A miner selects a sequence of client request transactions, executes the smart contract code of these transactions in sequence, transforming the state of the associated contract in this process. The miner then stores the sequence of transactions, the resulting final state of the contracts in the block along with the hash of the previous block. After creating the block, the miner proposes it to be added to the blockchain through the consensus protocol.

Once a block is added, the other peers in the system, referred to as *validators* in this context, validate the contents of the block. They re-execute the smart contract transactions in the block to verify the block's final states match or not. If final states match, then the block is accepted as valid and the miner who appended this block is rewarded. Otherwise, the block is discarded. Thus the transactions are executed by every peer in the system. In this setting, it turns out that the validation code runs several times more than miner code [30].

This design of smart contract execution is not very efficient as it does not allow any concurrency. Both the miner and the validator execute transactions serially one after another. In today's world of multi-core systems, the serial execution does not utilize all the cores and hence results in lower throughput. It is clear that the concurrent execution of smart contract transactions can improve the overall performance of the blockchain system. Dickerson et al. [30] observed another interesting advantage of concurrent execution in the context of blockchains like Ethereum that support cryptocurrencies. Here once a block gets accepted, the miner receives the incentive. However, all the remaining validators who re-execute the blocks get no such reward. So, a validator given a choice has a greater incentive to pick a block that supports concurrent execution and hence obtain higher throughput.

But the concurrent execution of smart contract transactions is not an easy task. The various transactions requested by the clients could consist of conflicting access to the shared data-objects. Arbitrary execution of these transactions by the miners might result in the data-races leading to the inconsistent final state of the blockchain. Unfortunately, it is not possible to statically identify if two contract transactions are conflicting or not since they are developed in Turing-complete languages. The common solution for correct execution of concurrent transactions is to ensure that the execution is *serializable* [5]. A usual correctness-criterion in databases, serializability ensure that the concurrent execution is equivalent to some serial execution of the same transactions. Thus the miners must ensure that their execution is serializable [30] or one of its variants as described later.

The concurrent execution of the smart contract transactions of a block by the validators although highly desirable can further complicate the situation. Suppose a miner ensures that the concurrent execution of the transactions in a block are serializable. Later a validator executes the same transactions concurrently. But during the concurrent execution, the validator may execute two conflicting transactions in an order different from what was executed by the miner. Thus the serialization order of the miner is different from the validator. Then this can result in the validator obtaining a final state different from what was obtained by the miner. Consequently, the validator may incorrectly reject the block although it is valid. Figure 5.1 illustrates this in the following example. Figure 5.1 (a) consists of two concurrent conflicting transactions T_1 and T_2 working on same shared data-objects x which are part of a block. Figure 5.1 (b) represents the concurrent execution by miner with an equivalent serial schedule as T_1, T_2 and final state (or FS) as 20 from the initial state (or IS) 0. Whereas Figure 5.1 (c), shows the concurrent execution by a validator with an equivalent serial schedule as T_2, T_1, and final state as 10 from IS 0 which is different from the final state proposed by the miner. 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.



(a) Concurrent transactions (b) Equivalent execution by miner (c) Equivalent execution by validator

Figure 5.1: Execution of concurrent transactions by miner and validator

5.1.4 Related Work on Concurrent Execution of Smart Contracts

Nowadays, blockchain is one of the most revolutionary technologies in the world. The first *blockchain* concept has been given by Satoshi Nakamoto in 2009 [28]. He proposed a system as bitcoin [28] which performs electronic transactions without the involvement of the third party. Cryptocurrencies such as Ethereum [44] and several other blockchains run the complex code known as a smart contract. The term smart contract [45] has been introduced by Nick Szabo. *Smart contract* is an interface to reduce the computational transaction cost and provides secure relationships on public networks. Basically, there are two aspects where researchers are exploring: (1) Security aspects [46–48] in which researchers are working to make the blockchain technology more secure. (2) Concurrency aspect [30, 49, 50] in which researchers are working to execute the blockchain efficiently and concurrently. Our proposed work is on concurrent execution of smart contract transactions by miner and validators so, we compared our work with serial miner and validator.

Sergey et al. [50] elaborates a new perspective between smart contracts and concurrent objects. Zhang et al. [49] have developed a multi-threaded miner and validator solution. In their solution, the miner can use any concurrency control mechanism developed in databases while the validator uses multi-version timestamp order (or MVTO) [38] based solution developed for databases. To avoid the FBR error, the miner stores the read-write sets in the block instead of a block graph (or BG). Their solution, unlike STMs, is not optimistic. Hence, they even have to re-execute committed transactions several times to ensure consistency which brings down the performance. Yu et al. [51] developed a pipeline model to verify and create blocks in parallel. Later, Yu et al. [52] proposed a parallel smart contract model which ensures better transactions processing. However, they faced synchronization issues that they resolved using transaction splitting algorithm.

As mentioned above in the motivation SubSection 5.1.3, Dickerson et al. [30] were the first to observe the inefficiency in the execution of smart contract transactions in Ethereum due to lack of concurrency. They pointed out the lack of concurrency both on the part of the miners and validators in Ethereum. Dickerson et al. addressed these issues by developing a multi-threaded miner algorithm. They used locks for synchronization between the threads. Similarly, they developed a multi-threaded solution for the validators.

To avoid the FBR error by the validators, Dickerson et al. proposed the following. As the miner executes the smart contract transactions in a block using multiple threads, it identifies the dependencies between the Smart Contract Transactions (SCTs) of the block and provide a *happens-before* graph in the block. The happens-before graph is a direct acyclic graph over all the transaction of the block. If there is a path from a transaction T_i to T_j then the validator has to execute T_i before T_j. Transactions with no path between them can execute concurrently. The miner uses locks to protect the shared variables accessed by the threads executing the dependent smart contract transactions.

The validator using the happens-before graph in the block executes all the transactions concurrently using the fork-join approach. A validator on receiving a block, to be added to the blockchain, analyzes the happens-before graph in it to identify the dependencies among the smart contract transactions. Any two smart contract transactions not having a path in the happens-before graph can be executed in parallel without being concerned about synchronization between them since they do not have any dependency. The validator identifies all such smart contract transactions and execute them in parallel using *fork-join* approach [30].

This methodology ensures that the final state of the blockchain generated by the miners and the validators are the same for a valid block and hence not rejected by the validators. The presence of tools such as a happens-before graph in the block provides greater enhancement to validators to consider such blocks as it helps them execute quickly by means of parallelization as opposed to a block which does not have any tools for parallelization. This, in turn, entices the miners to provide such tools in the block for concurrent execution by the validators.

5.1.5 Our Solution Approach: Optimization of Blockchain using Efficient STMs

In our solution approach miner executes the smart contract transactions concurrently using efficient and optimistic STMs and generates lock-free graph. Dickerson et al. [30] developed a solution to the problem of concurrent miner and validators using locks and inverse logs. It is well known that locks are pessimistic in nature. So, in this thesis, we explore a novel and efficient framework for concurrent miners using optimistic Software Transactional Memory Systems (STMs).

The requirement of the miner, as explained above, is to concurrently execute the smart contract transactions correctly and output a graph capturing dependencies among the transactions of the block such as happens-before graph. We denote this graph as *Block Graph* (or *BG*). In the proposed solution, the miner uses the services of an optimistic STM system to concurrently execute the smart contract transactions. Since STMs also work with transactions, we differentiate between smart contract transactions and STM transactions. The STM transactions invoked by an STM system is a piece of code that it tries to execute atomically even in presence of other concurrent STM transactions. If the STM system is not able to execute it atomically, then the STM transaction is aborted.

The expectation of a smart contract transaction is that it will be executed serially. Thus, when it is executed in a concurrent setting, it is expected to be executed atomically (or serialized). To differentiate between smart contract transaction from STM transaction, we denote smart contract transaction as *Atomic Unit* or *atomic-unit* and STM transaction as transaction in the rest of the document. Thus the miner uses the STM system to invoke a transaction for each atomic-unit. In case the transaction gets aborted, then the STM repeatedly invokes new transactions for the same atomic-unit until a transaction invocation eventually commits.

A popular correctness guarantee provided by STM systems is *opacity* [7] which is stronger than serializability. Opacity like serializability requires that the concurrent execution including the aborted transactions be equivalent to some serial execution. This ensures that even aborted transaction reads consistent value until the point of abort. As a result, that the application such as a miner using an STM does not encounter any undesirable side-effects such as crash failures, infinite loops, divide by zero etc. STMs provide this guarantee by executing optimistically and support atomic (opaque) reads, writes on *transactional objects* or *t-objects*.

Among the various STMs available, we have chosen two timestamp based STMs in our design: (1) *Basic Timestamp Ordering* or *BTO* STM [2, Chap 4], maintains only one version for each t-object. We called such miner as *BTO Miner*. (2) *Multi-Version Timestamp Ordering* or *MVTO* STM [10], maintains multiple versions corresponding to each t-object which further reduces the number of aborts and improves the throughput. We called such miner as *MVTO Miner*.

The advantage of using timestamp based STM is that in these systems the equivalent serial history is ordered based on the timestamps of the transactions. Thus using the timestamps, the miner can generate the BG of the atomic-units. Dickerson et al. [30], developed the BG in a serial manner. In our approach, the graph is developed by the miner in concurrent and lock-free [18] manner.

The validator process creates multiple threads. Each of these threads parses the BG and re-execute the atomic-units for validation. The BG provided by concurrent miner shows dependency among the atomic-units. Each validator thread, claims a node which does not have any dependency, i.e. a node without any incoming edges by marking it. After that, it executes the corresponding atomic-units deterministically. Since the threads execute only those nodes that do not have any incoming edges, the concurrently executing atomic-units will not have any conflicts. Hence the validator threads need not to worry about synchronization issues. We denote this approach adopted by the validator as a decentralized approach (or Decentralized Validator) as the multiple threads are working on BG concurrently in the absence of master thread. So, we proposed two decentralized validators as *BTO Decentralized* and *MVTO Decentralized Validator*.

The approach adopted by Dickerson et al. [30], works on fork-join in which a master thread allocates different tasks to slave threads. The master thread will identify those atomic-units which do not have any dependencies from the BG and allocates them to different slave threads to work on. So, we proposed two fork-join validators as *BTO Fork-join Validator* and *MVTO Fork-join Validator*. In this paper, we compare the performance of both these approaches with the serial validator.

Our experimental analysis demonstrates that BTO Miner and MVTO miner achieve an average speedup of 3.6x and 3.7x over serial miner respectively. Along with, BTO validator (average of BTO Fork-join Validator and BTO Decentralized Validator) and MVTO validator (average of MVTO Fork-join Validator and MVTO Decentralized Validator) outperform with an average speedup of 40.8x and 47.1x than serial validator respectively.

Roadmap: First, we studied and analyzed the requirements of concurrent miner, concurrent validator and BG in Section 5.2. We introduced a novel way to execute the smart contract transactions by concurrent miner using optimistic STMs in SubSection 5.3.2. Here, we implemented the concurrent miner with the help of BTO and MVTO protocol of STMs but it is generic to any STM protocol. To get rid of FBR error, concurrent miner proposes a lock-free graph library to generate the BG. After that, we proposes concurrent validator in SubSection 5.3.3 which re-executes the smart contract transactions deterministically and efficiently with the help of BG given by concurrent miner. We proved the correctness of BG, concurrent miner and concurrent validator in Section 5.4. Experimental analysis shown in Section 5.5, followed by summary of this chapter in Section 5.6.

5.2 Requirements of Concurrent Miner, Validator and Block Graph

This section describes the requirements of concurrent miner, validator and block graph to ensure correct concurrent execution of the smart contract transactions.

5.2.1 Requirements of the Concurrent Miner

The miner process invokes several threads to concurrently execute the smart contract transactions or atomic-units. With the proposed optimistic execution approach, each miner thread invokes an atomic-unit as a transaction.

The miner should ensure the correct concurrent execution of the smart contract transactions. The incorrect concurrent execution (or consistency issues) may occur when concurrency involved. Any inconsistent read may leads system to divide by zero, infinite loops, crash failure etc. All smart contract transactions take place within a virtual machine [30]. When miner executes the smart contract transactions concurrently on the virtual machine then infinite loop and inconsistent read may occur. So, to ensure the correct concurrent execution, the miner should satisfy the correctness-criterion as opacity [7].

To achieve better efficiency, sometimes we need to adapt the non-virtual machine environment which necessitates with the safeguard of transactions. There as well miner needs to satisfies the correctness-criterion as opacity to ensure the correct concurrent execution of smart contract transactions.

Concurrent miner maintains a BG and provides it to concurrent validators which ensures the dependency order among the conflicting transactions. As we discussed in SubSection 5.1.3, if concurrent miner will not maintain the BG then a valid block may get rejected by the concurrent validator.

5.2.2 Requirements of the Concurrent Validator

The correct concurrent execution by validator should be equivalent to some serial execution. The serial order can be obtained by applying the topological sort on the BG provided by the concurrent miner. BG gives partial order among the transactions while restricting the dependency order same as the concurrent miner. So, validator executes those transactions concurrently which are not having any dependency among them with the help of BG. Validator need not to worry about any concurrency control issues because BG ensures conflicting transactions never execute concurrently.

5.2.3 Requirements of the Block Graph

As explained above, the miner generates a BG to capture the dependencies between the smart contract transactions which is used by the validator to concurrently execute the transactions again later. The validator executes those transactions concurrently which do not have any path (implying dependency) between them. Thus the execution by the validator is given by a topological sort on the BG.

Now it is imperative that the execution history generated by the validator, H_v is 'equivalent' to the history generated by the miner, H_m. The precise equivalence depends on the STM protocol followed by the miners and validators. If the miner uses Multi-version STM such as MVTO then the equivalence between H_v and H_m is *Multi-Version View Equivalent (MVVE)* [2, Chap. 5] explained in Chapter 2. In this case, the graph generated by the miner would be multi-version serialization graph [2, Chap. 5].

On the other hand, if the miner uses single version STM such as BTO then the equivalence between H_v and H_m is view-equivalence (VE) which can be approximated by conflict-equivalence (CE) explained in Chapter 2. Hence, in this case, the graph generated by the miner would be conflict graph [2, Chap. 3].

5.3 Proposed Mechanism

This section presents the methods of lock-free concurrent block graph library followed by concurrent execution of smart contract transactions by miner and validator.

5.3.1 Lock-free Concurrent Block Graph

Data Structure of Lock-free Concurrent Block Graph: We use *adjacency list* to maintain the block graph BG(V, E) as shown in Figure 5.2 (a). Where V is set of vertices (or *vNode*s) which are stored in the vertex list (or *vlist*) in increasing order of timestamp between two sentinel node *vHead* (-∞) and *vTail* (+∞). Each vertex node (or *vNode*) contains $\langle ts = i, AU_{id} = id, inCnt = 0, vNext = nil, eNext = nil\rangle$. Where i is a unique timestamp (or ts) of committed transactions T_i. AU_{id} is the id of atomic-unit which is executed by transaction T_i. To maintain the indegree count of each *vNode* we initialize *inCnt* as 0. *vNext* and *eNext* initializes as nil.

Here, E is a set of edges which maintains all the conflicts of *vNode* in the edge list (or *eList*) as shown in Figure 5.2 (a). *eList* stores *eNode*s (or conflicting transaction nodes say T_j) in increasing order of timestamp (or ts) between two sentinel nodes *eHead* (-∞) and *eTail* (+∞).

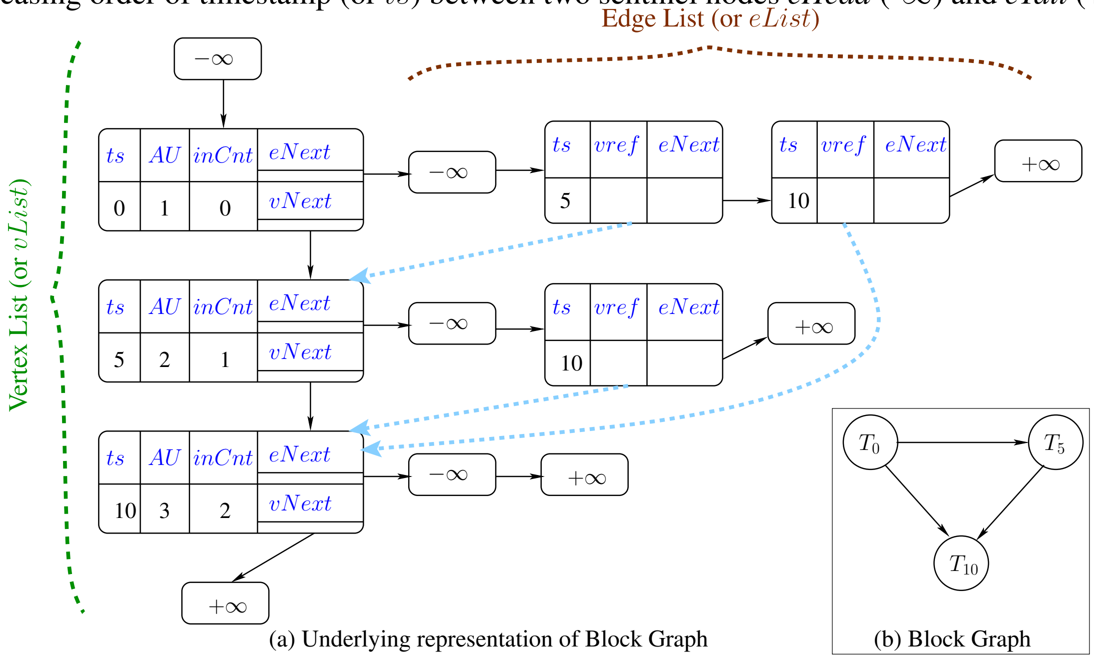


Figure 5.2: Pictorial representation of Block Graph

Edge node (or *eNode*) contains $\langle ts = j, vref, eNext = nil \rangle$. Here, j is a unique timestamp (or

ts) of *committed* transaction T_j which is having conflict with T_i and $ts(T_i)$ is less than $ts(T_j)$. To maintain the acyclicity of the BG, we add a conflict edge from lower timestamp transaction to higher timestamp transaction i.e. conflict edge is from T_i to T_j in the BG. Figure 5.2 (b) illustrates this using three transactions with timestamp 0, 5, and 10, which maintain the acyclicity while adding an edge from lower to higher timestamp. *Vertex node reference (or vref)* keeps the reference of its own vertex which is present in the *vlist*. *eNext* initializes as nil.

Block graph generated by the concurrent miner which helps to execute the validator concurrently and deterministically through graph library methods. Graph library consists of five methods as follows: *addVert(), addEdge(), searchLocal(), searchGlobal()* and *decInCount()*. Among these five methods *addVert()* and *addEdge()* are lock-free [18] methods of graph library.

Algorithm 32 BG(*vNode*, STM): It generates a block graph for all the atomic-unit nodes.

```
procedure BG(vNode, STM)
    /*Get the confList of committed transaction $T_i$ from STM*/
    clist ← STM.getConfList (vNode.$ts_i$);
    /*Transaction $T_i$ conflicts with $T_j$ and $T_j$ existes in conflict list of $T_i$*/
    for all ($ts_j \in$ clist) do
        addVert ($ts_j$);
        addVert (vNode.$ts_i$);
        if ($ts_j <$ vNode.$ts_i$) then
            addEdge ($ts_j$, vNode.$ts_i$);
        else
            addEdge (vNode.$ts_i$, $ts_j$);
        end if
    end for
end procedure
```

Graph Library Methods Accessed by Concurrent Miner: Concurrent miner uses two lock-free methods addVert() and addEdge() of graph library to build a block graph. When concurrent miner wants to add a node in the block graph then first it calls addVert() method. addVert() method identifies the correct location of that node (or *vNode*) in the *vlist* at Line 16. If *vNode* is not part of *vlist* then it creates the node and adds it into *vlist* at Line 19 in lock-free manner with the help of atomic compare and swap operation. Otherwise, *vNode* is already present in *vlist* at Line 24.

After successful addition of *vNode* in the block graph concurrent miner calls addEdge() method to add the conflicting node (or *eNode*) corresponding to *vNode* in the *eList*. First, addEdge() method identifies the correct location of *eNode* in the *eList* of corresponding *vNode* at Line 28. If *eNode* is not part of *eList* then it creates the node and adds it into *eList* of *vNode* at Line 31 in lock-free manner with the help of atomic compare and swap operation. After successful addition of *eNode* in the *eList* of *vNode*, it increment the *inCnt* of *eNode*.$vref$ (to maintain the indegree count) node which is present in the *vlist* at Line 32.

Algorithm 33 *addVert*(ts_i): It finds the appropriate location of vertex graph node (or *vNode*) which is having a ts as i in the *vlist* and add into it.

```
15: procedure addVert($ts_i$)
16:     Identify the ⟨vPred, vCurr⟩ of vNode of $ts_i$ in vlist of $BG$;
17:     if (vCurr.$ts_i \neq$ vNode.$ts_i$) then
18:         Create new Graph Node (or vNode) of $ts_i$ in vlist;
19:         if (vPred.vNext.CAS(vCurr, vNode)) then
20:             return⟨Vertex added⟩; /*vNode is successfully inserted in vlist*/
21:         end if
22:         goto Line 16; /*Start with the vPred to identify the new ⟨vPred, vCurr⟩*/
23:     else
24:         return⟨Vertex already present⟩; /*vNode is already present in vlist*/
25:     end if
26: end procedure
```

Algorithm 34 *addEdge(fromNode, toNode)*: It adds an edge from *fromNode* to *toNode*.

```
27: procedure addEdge(fromNode, toNode)
28:     Identify the ⟨ePred, eCurr⟩ of toNode in eList of the fromNode vertex in $BG$;
29:     if (eCurr.$ts_i \neq$ toNode.$ts_i$) then
30:         Create new Graph Node (or eNode) in eList;
31:         if (ePred.eNext.CAS(eCurr, eNode)) then
32:             Increment the inCnt atomically of eNode.vref in vlist;
33:             return⟨Edge added⟩; /*toNode is successfully inserted in eList*/
34:         end if
35:         goto Line 28; /*Start with the ePred to identify the new ⟨ePred, eCurr⟩*/
36:     else
37:         return⟨Edge already present⟩; /*toNode is already present in eList*/
38:     end if
39: end procedure
```

Graph Library Methods Accessed by Concurrent Validator: Concurrent validator uses searchLocal(), searchGlobal() and decInCount() methods of graph library. First, concurrent validator thread calls searchLocal() method to identify the source node (having indegree (or *inCnt*) 0) in its local *cacheList* (or thread local memory). If any source node exist in the local *cacheList* with *inCnt* 0 then it sets *inCnt* field to be -1 at Line 41 atomically.

If source node does not exist in the local *cacheList* then concurrent validator thread calls searchGlobal() method to identify the source node in the block graph at Line 52. If any source node exists in the block graph then it will do the same process as done by searchLocal() method.

After that validator thread calls the decInCount() method to decreases the *inCnt* of all the conflicting nodes atomically which are present in the *eList* of corresponding source node at Line 63. While decrementing the *inCnt* of each conflicting nodes in the block graph, it again checks if any conflicting node became a source node then it adds that node into its local *cacheList* to optimize the search time of identifying the next source node at Line 65.

Algorithm 35 *searchLocal*(cacheVer, AU_{id}): First validator thread search into its local *cacheList*.

```
40: procedure searchLocal($cacheVer$, $AU_{id}$)
41:     if (cacheVer.inCnt.CAS(0, -1)) then
42:         nCount ← nCount.$get\&Inc()$;
43:         $AU_{id}$ ← cacheVer.$AU_{id}$;
44:         return⟨cacheVer⟩;
45:     else
46:         return⟨$nil$⟩;
47:     end if
48: end procedure
```

Algorithm 36 *searchGlobal*(BG, AU_{id}): Search the source node in the block graph whose *inCnt* is 0.

```
49: procedure searchGlobal(BG, $AU_{id}$)
50:     vNode ← BG.vHead;
51:     while (vNode.vNext ≠ BG.vTail) do /*Search into the Block Graph*/
52:         if (vNode.inCnt.CAS(0, -1)) then
53:             nCount ← nCount.$get\&Inc()$;
54:             $AU_{id}$ ← vNode.$AU_{id}$;
55:             return⟨vNode⟩;
56:         end if
57:         vNode ← vNode.vNext;
58:     end while
59:     return⟨$nil$⟩;
60: end procedure
```

Algorithm 37 decInCount(remNode): Decrement the *inCnt* of each conflicting node of source node.

```
61: procedure decInCount(remNode)
62:     while (remNode.eNext ≠ remNode.eTail) do
63:         Decrement the inCnt atomically of remNode.vref in the vlist;
64:         if (remNode.vref.inCnt == 0) then
65:             Add remNode.verf node into cacheList of thread local log, thLog;
66:         end if
```

```
            remNode ← remNode.$eNext.verf$;
            return⟨remNode⟩;
        end while
        return⟨$nil$⟩;
    end procedure
```

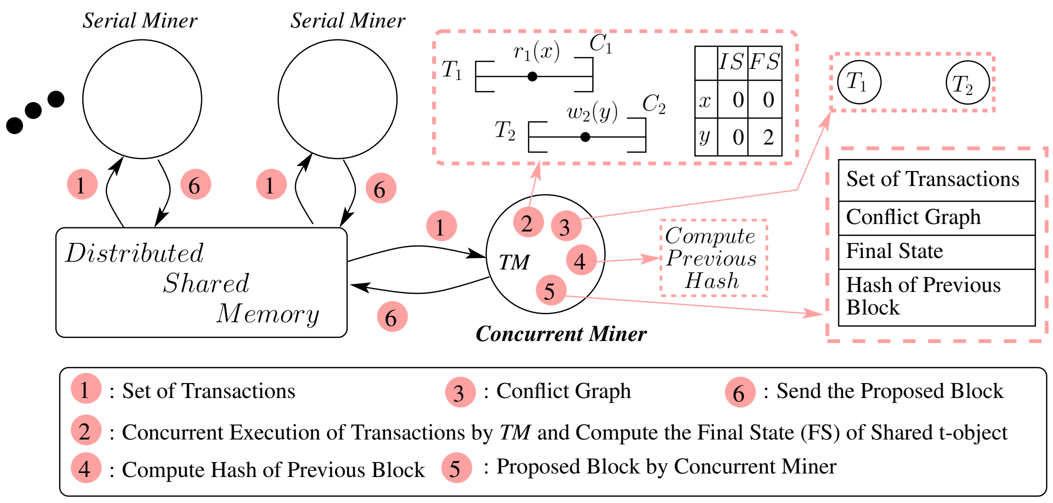
Algorithm 38 *executeCode*(curAU): Execute the current atomic-units.

```
    procedure executeCode($curAU$)
        while (curAU.steps.hasNext()) do /*Assume that curAU is a list of steps*/
            curStep = currAU.steps.next(); /*Get the next step to execute*/
            switch (curStep) do
            case read($x$):
                Read Shared data-object $x$ from a shared memory;
            case write($x, v$):
                Write Shared data-object $x$ in shared memory with value $v$;
            case default:
                /*Neither read from or write to a shared memory Shared data-objects*/;
                execute curStep;
        end while
        return ⟨$void$⟩
    end procedure
```

5.3.2 Concurrent Miner

Smart contracts in blockchain are executed in two different context. First, by miner to propose a new block and after that by multiple validators to verify the block proposed by miner. In this subsection, we describe how miner executes the smart contracts concurrently and proposes the block. [1] *Concurrent miner* gets the set of transactions from the *distributed shared memory* as shown in Figure 5.3. Each transaction associated with the functions (or atomic-units) of smart contracts. To run the smart contracts concurrently we have faced the challenge to identify the conflicting transactions at run-time because smart contract language are Turing-complete. Two transactions are in conflict if they are accessing common shared data-objects and at least one of them perform write operation on it. [2] In *concurrent miner*, conflicts are identified at run-time with the help of efficient framework provided by optimistic software transactional memory system (STMs). STMs access the shared data-objects called as t-objects. Each shared t-object having initial state (or IS) which modified by the atomic-units and change IS to some other valid state. Eventually, it reaches to final state (or FS) at the end of block creation. As shown in Algorithm 39, first, each transaction T_i gets the unique timestamp i from STM_begin() at

Line 91. Then transaction T_i executes the atomic-unit of smart contracts. *Atomic-unit* consists of multiple steps such as $read$ and $write$ on shared t-objects as x. Internally, these $read$ and $write$ steps are handled by the STM_read() and STM_write(), respectively. At Line 95, if current atomic-unit step (or curStep) is $read(x)$ then it calls the STM_read(x). Internally, STM_read() identify the shared t-object x from transactional memory (or TM) and validate it. If validation is successful then it gets the value as v at Line 96 and execute the next step of atomic-unit otherwise re-execute the atomic-unit if v is *abort* at Line 97.



Serial Miner
Serial Miner
Distributed Shared Memory
TM
Concurrent Miner
T_1
$r_1(x)$
C_1
T_2
$w_2(y)$
C_2
IS
FS
x 0 0
y 0 2
Compute Previous Hash
Set of Transactions
Conflict Graph
Final State
Hash of Previous Block
1 : Set of Transactions
3 : Conflict Graph
6 : Send the Proposed Block
2 : Concurrent Execution of Transactions by TM and Compute the Final State (FS) of Shared t-object
4 : Compute Hash of Previous Block
5 : Proposed Block by Concurrent Miner

Figure 5.3: Execution of Concurrent Miner

Algorithm 39 *Concurrent Miner*(*auList[]*, STM): Concurrently m threads are executing atomic-units of smart contract from *auList[]*(or list of atomic-units) with the help of STM.

```
86: procedure Concurrent Miner(auList[], STM)
87:     curAU ← $curInd.get\&Inc(auList[])$;
88:     /*curAU is the current atomic-unit taken from the auList[] */
89:     /*Execute until all the atomic-units successfully completed*/
90:     while (curAU < size_of(auList[])) do
91:         $T_i$ ← STM_begin();/*Create a new transaction $T_i$ with timestamp $i$*/
92:         while (curAU.steps.hasNext()) do /*Assume that curAU is a list of steps*/
93:             curStep = currAU.steps.next(); /*Get the next step to execute*/
94:             switch (curStep) do
95:             case read($x$):
96:                 $v$ ← STM_$read_i(x)$; /*Read t-object $x$ from a shared memory*/
97:                 if ($v$ == $abort$) then
98:                     goto Line 91;
99:                 end if
100:            case write($x, v$):
101:                /*Write t-object $x$ into $T_i$ local memory with value $v$*/
```

```
102:                STM_$write_i(x, v)$;
103:            case default:
104:                /*Neither read from or write to a shared memory t-object*/
105:                execute curStep;
106:        end while
107:        /*Try to commit the current transaction $T_i$ and update the confList[i]*/
108:        $v \leftarrow$ STM_tryC$_i$();
109:        if ($v == abort$) then
110:            goto Line 91;
111:        end if
112:        Create vNode with ⟨i, $AU_{id}$, 0, nil, nil⟩ as a vertex of Block Graph;
113:        BG(vNode, STM);
114:        curAU $\leftarrow$ $curInd.get\&Inc$(auList[]);
115:    end while
116: end procedure
```

If curStep is $write(x, v)$ at Line 100 then it calls the STM_write(x, v). Internally, STM_write() stores the information corresponding to the shared t-object x into transaction local log (or *txlog*) in write-set (or $wset_i$) for transaction T_i. We use an optimistic approach in which effect of the transaction will reflect onto the TM after the successful STM_tryC(). If validation is successful for all the $wset_i$ of transaction T_i in STM_tryC() i.e. all the changes made by the T_i is consistent then it updates the TM otherwise re-execute the atomic-unit if v is $abort$ at Line 109. After successful validation of STM_tryC(), it also maintains the conflicting transaction of T_i into conflict list in TM.

③ Once the transaction commits, it stores the conflicts in the block graph (or BG). To maintain the BG it calls two lock-free method addVert() and addEdge() of graph library. The internal details of addVert() and addEdge() methods are explained in SubSection 5.3.1. ④ Once the transactions successfully executed the atomic-units and completed the construction of BG then *concurrent miner* compute the hash of the previous block. Eventually, ⑤ *concurrent miner* proposes a block which consists of a set of transactions, BG, final state of each shared t-object, hash of the previous block of the blockchain and ⑥ sends it to all other existing nodes in the *distributed shared memory* to validate it as shown in Figure 5.3.

5.3.3 Concurrent Validator

Concurrent validator validates the block proposed by the concurrent miner. It executes the set of transactions concurrently and deterministically with the help of block graph given by the *concurrent miner*. BG consists of dependency among the conflicting transactions that restrict them to execute serially whereas non-conflicting transactions can run concurrently. In *concur-*

rent validator multiple threads are executing the atomic-units of smart contracts concurrently by *executeCode()* method at Line 123 and Line 130 with the help of searchLocal(), and searchGlobal() and decInCount() methods of graph library at Line 122, Line 129 and (Line 125, Line 132) respectively. The functionality of these graph library methods are explained in SubSection 5.3.1.

After the successful execution of all the atomic-units, *concurrent validator* compares its computed final state of each shared data-object with the final states given by the *concurrent miner*. If the final state matches for all the shared data-objects then the block proposed by the *concurrent miner* is valid. Finally, the block is appended to the blockchain and respective *concurrent miner* is rewarded.

Algorithm 40 *Concurrent Validator*(*auList[]*, BG): Concurrently V threads are executing atomic-units of smart contract with the help of BG given by the miner.

```
117: procedure Concurrent Validator(auList[], BG)
118:     /*Execute until all the atomic-units successfully completed*/
119:     while (nCount < size_of(auList[])) do
120:         while (cacheList.hasNext()) do /*First search into thread local cacheList */
121:             cacheVer ← cacheList.next();
122:             cacheVertex ← searchLocal(cacheVer, $AU_{id}$);
123:             executeCode($AU_{id}$); /*Execute the atomic-unit of cacheVertex*/
124:             while (cacheVertex) do
125:                 cacheVertex ← decInCount(cacheVertex);
126:             end while
127:             Remove the current node (or cacheVertex) from local cacheList;
128:         end while
129:         vexNode ← searchGlobal(BG, $AU_{id}$); /*Search into the BG*/
130:         executeCode($AU_{id}$); /*Execute the atomic-unit of vexNode*/
131:         while (verNode) do
132:             verNode ← decInCount(verNode);
133:         end while
134:     end while
135: end procedure
```

5.4 Correctness of Block Graph, Concurrent Miner and Concurrent Validator

This section describes the proof of theorems stated for the correctness of BG, concurrent miner, and validator in Section 5.3. In order to define the correctness of BG, we identify the linearization points (LPs) of each method as follows:

1. *addVert*(*vNode*): (*vPred*.*vNext*.CAS(*vCurr*, *vNode*)) in Line 19 is the LP point of *addVert*() method if *vNode* is not exist in the BG. If *vNode* is exist in the BG then (*vCurr*.ts_i

$\neq$ *vNode*.ts_i) in Line 17 is the LP point.

2. *addEdge(fromNode, toNode)*: (*ePred*.*eNext*.CAS(*eCurr*, *eNode*)) in Line 31 is the LP point of *addEdge*() method if *eNode* is not exist in the BG. If *eNode* is exist in the BG then (*eCurr*.$ts_i \neq$ toNode.ts_i) in Line 29 is the LP point.

3. *searchLocal*(cacheVer, AU_{id}): (cacheVer.*inCnt*.CAS(0, -1)) in Line 41 is the LP point of *searchLocal*() method.

4. *searchGlobal*(BG, AU_{id}): (*vNode*.*inCnt*.CAS(0, -1)) in Line 52 is the LP point of *search-Global*() method.

5. *decInCount*(remNode): Line 63 is the LP point of *decInCount()* method.

Theorem 65 *All the dependencies between the conflicting nodes are captured in the BG.*

Proof. SubSection 5.3.1 represents the construction of BG by concurrent miner using BTO and MVTO protocol. BG considers each committed SCT as a vertex and edges (or dependencies) depends on the used STM protocols such as BTO and MVTO. So, the underlying STM protocol ensures that all the dependencies have been covered correctly among the conflicting nodes of BG. Hence, all the dependencies between the conflicting nodes are captured in the BG.

Theorem 66 *Any history H_m generated by concurrent miner using BTO satisfies co-opacity.*

Proof. Multiple miner threads execute SCTs concurrently using BTO and generate a concurrent history H_m. The underlying BTO protocol ensures the correctness of concurrent execution of H_m. BTO proves that any history generated by it satisfies co-opacity [2, Chap 4]. So, implicitly BTO proves that the history H_m generated by concurrent miner using BTO satisfies co-opacity.

Theorem 67 *Any history H_m generated by concurrent miner using MVTO satisfies opacity.*

Proof. In order to achieve the greater concurrency further, a concurrent miner uses the MVTO protocol which maintains multiple version corresponding to each shared data-item. MVTO ensures the correct concurrent execution of the history H_m with the equivalent serial history S_m. Any history generated by MVTO satisfies opacity [10]. So, implicitly MVTO proves that the history H_m generated by concurrent miner using MVTO satisfies opacity.

Theorem 68 *Any history H_m generated by BTO protocol and history H_v generated by concurrent validator are view equivalent.*

Proof. Concurrent miner execute the SCTs concurrently using BTO protocol to propose a block and generates H_m along with BG. After that concurrent miner broadcasts H_m and BG to concurrent validators to verify the proposed block. Concurrent validator applies the topological sort on BG and obtained an equivalent serial schedule H_v. Since BG constructed from H_m while considering all the conflicts and H_v obtained from the topological sort on BG. So, H_v will be equivalent to H_m. Similarly, H_v will also follow the *read value from* relation to H_m. Hence, H_v is legal. Since H_v and H_m, are equivalent to each other, and H_v is legal. So, H_m and H_v are view equivalent.

Theorem 69 *Any history H_m generated by MVTO protocol and history H_v generated by concurrent validator are multi-version view equivalent.*

Proof. Following the proof of Theorem 68, concurrent miner executes H_m using MVTO and constructs the BG. MVTO maintains multiple-version corresponding to each shared data-item while executing the SCTs by concurrent miner. Later, concurrent validator obtained H_v by applying topological sort on BG given by miner. Since, H_v obtained from topological sort on BG so, H_v will be equivalent to H_m. Similarly, BG maintains the *read value from* relations of H_m. So, from MVTO protocol if T_j returns a value of the method for shared data-item k say $rv_j(k)$ from T_i in H_m then T_i committed before $rv_j(k)$ in H_v. Therefore, H_v is valid. Since H_v and H_m are equivalent to each other and H_v is valid. So, H_m and H_v are multi-version view equivalent.

5.5 Experimental Evaluations

For the experiment, we consider a set of benchmarks generated for Ballot, Simple Auction, and Coin contracts from Solidity documentation [31]. Experiments are performed by varying the number of atomic-units, threads, and shared data-objects. The analysis focuses on two main objectives: (1) Evaluate and analyzes the speedup achieved by concurrent miner over the serial miner. (2) Appraise the speedup achieved by concurrent validator over serial validator on various experiments.

Experimental system: The system configuration for experiments is 2 socket Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz with 14 cores per socket and 2 hyper-threads per core, a total of 56 threads. A private 32KB L1 cache and 256 KB L2 cache is with each core. It has 32 GB RAM with Ubuntu 16.04.2 LTS running Operating System.

Methodology: We have considered three types of workload, (W1) The number of atomic-units varies from 50 to 400, while threads and shared data-objects are fixed to 50 and 40 respectively. (W2) The number of threads varies from 10 to 60 while atomic-units and shared data-objects are fixed to 400 and 40 respectively. (W3) The number of the shared data-objects varies from 10 to 60 while fixing the number of threads and atomic-units to 50 and 400 respectively. For accuracy, all the experiments of concurrent miners and validators are averaged over 11 runs in which the first run is discarded and considered as a warm-up run.

5.5.1 Benchmarks

Smart Contracts: The transactions sent by clients to miners are part of a larger code called as *smart contracts* that provide several complex services such as managing the system state, ensures rules, or credentials checking of the parties involved, etc. [30]. In reality, miner forms a block which consists of a set of transactions from different contracts. So, we consider four benchmarks Ballot, Simple Auction, Coin including Mixed contract which is the combination of above three. In Ethereum blockchain, smart contracts are written in Solidity and runs on the Ethereum Virtual Machine (EVM). The issue with EVM is that it does not support multi-threading and hence give poor throughput. Therefore, to exploit the efficient utilization of multi-core resources and to improve the performance, we convert smart contract from Solidity language into C++ and execute them using multi-threading. The details of the benchmarks are as follows:

1. **Simple Auction:** The functionality of Simple Auction contract is shown in Algorithm 41. Line 136 declares the contract, followed by public state variables as "highestBidder, highestBid, and pendingReturn" which records the state of the contract. A single owner of the contract initiates the auction by executing constructor "SimpleAuction()" which function initialize bidding time as auctionEnd (Line 138). There can be any number of participants to bid. The bidders may get their money back whenever the highest bid is raised. For this, a public state variable declared at Line 142 (pendingReturns) uses solidity built-in complex data type mapping to maps bidder addresses with unsigned integers (withdraw amount respective to bidder). Mapping can be seen as a hash table with key-value pair. This mapping uniquely identifies account addresses of the clients in the Ethereum blockchain. A bidder withdraws the amount of their earlier bid by calling withdraw() method [31].

   At Line 143, a contract function "bid()" is declared, which is called by bidders to bid in the auction. Next, "auctionEnd" variable is checked to identify whether the auction already called off or not. Further, bidders "msg.value" check to identify the highest bid value at Line 147. Smart contract methods can be aborted at any time via throw when the auction is called off, or bid value is smaller than current "highestBid". When execution reaches to Line 151, the "bid()" method recovers the current highest bidder data from mapping through the "highestBidder" address and updates the current bidder pending return amount. Finally, at Line 153 and Line 154, it updates the new highest bidder and highest bid amount. Conflict can occur if at least two bidders are going to request for bidPlusOne() simultaneously.

Algorithm 41 SimpleAuction: It allows every bidder to send their bids throughout the bidding period.

```
procedure SimpleAuction
    address public beneficiary;
    uint public auctionEnd;
    /*current state of the auction*/
    address public highestBidder;
    uint public highestBid;
    mapping(address => uint) pendingReturns;
    function bid() public payable
        if (now $\geq$ auctionEnd) then
            throw;
        end if
        if (msg.value < highestBid) then
            thorw;
        end if
        if (highestBid != 0) then
            pendingReturns[highestBidder] += highestBid;
        end if
        highestBidder = msg.sender;
        highestBid = msg.value;
    end function
end procedure
```

2. **Coin:** It is the simplest form of a cryptocurrency in which accounts are the shared data-objects. All accounts are uniquely identified by Etherum addresses. Only the contract deployer known as minter will be able to generate the coins and initialize the accounts at the beginning. Anyone having an account can send coins to another account with the condition that they have sufficient coins in their account or can check their balance. In the initial state, minter initializes all the accounts with some coins. Conflict can occur if at least two senders are transferring the amount into the same receiver account simultaneously or when one send() and getbalance() have an account in common.

3. **Ballot:** It implements an electronic voting contract in which voters and proposals are the shared data-objects. All the voters and proposals are already registered and have unique Ethereum address. At first, all the voters are given rights by the chairperson (or contract deployer) to participate in the ballot. Voters either cast their vote to the proposal of their choice or delegate vote to another voter whom they can trust using delegate(). A voter is allowed to delegate or vote once throughout the ballot. Conflict can occur if at least two

voters are going to delegate their vote to the same voter or cast a vote to the same proposal simultaneously. Once the ballot period is over, the winner of the ballot is decided based on the maximum vote count.

4. **Mixed:** In this benchmark, we have combined all the above benchmarks in equal proportions. Conflicts can occur when AUs of the same contract executed simultaneously and operate on common shared data-objects.

In all the above contracts, conflicts can very much transpire when *miner* executes them concurrently. So, we use Optimistic STMs to ensure consistency and handle the conflicts.

5.5.2 Result Analysis

This subsection represents the speedup of concurrent miner and validator relative to the serial miner and validator for all the smart contracts (Simple Auction, Coin, Ballot, Mixed) on workload W1, W2, and W3. It shows average speedup of 3.6x and 3.7x by the BTO and MVTO concurrent miner over serial miner respectively. Along with, BTO and MVTO validator outperforms average 40.8x and 47.1x than serial validator respectively[1]. The results from the serial execution of the miner and validator are served as the baseline.

Figure 5.4 demonstrates the speedup of concurrent miner and validator relative to the serial miner and validator for Simple Auction Contract on workload W1, W2, and W3. It can be seen from Figure 5.4 (a) that maximum speedup has been achieved by concurrent miner with smaller number of atomic-units. Figure 5.4 (b) illustrates that speedup of concurrent miner and validator increase while increasing the number of threads up to fix number depending on system configuration. Here, system has 56 logical threads. Therefore, the speedup of concurrent miner and validator at thread 60 is slightly less than the speedup at thread 50. Figure 5.4 (c) observed that with the increase in the number of shared data-objects speedup achieved by concurrent miner and validator also increases compared to the serial miner and validator. The reason behind this speedup is that data conflicts will decrease with the increase in shared data-objects, and a higher number of atomic-units can be executed concurrently.

Similarly, Figure 5.5, Figure 5.6, and Figure 5.6 illustrate the speedup of concurrent miner and validator relative to the serial miner and validator on workload W1, W2, and W3 for Coin, Ballot, and Mixed Contract respectively. The similar observations can be found in all the contracts as explained above for Simple Auction. Table 5.1 represents the speedup of concurrent miner and validator over serial miner and validator on various workloads W1, W2, and W3 for each contract.

Apart from this, Figure 5.4 and Figure 5.5 show the speedup achieved by MVTO Miner is greater than BTO Miner for Simple Auction and Coin contract on all the workloads. A plausible reason can be that MVTO gives good performance for read-intensive workloads [10]. Simple

[1]Code is available here: https://github.com/pdcrl/Blockchain

Auction and Coin contracts are read-intensive as defined in Solidity documentation [31]. On the other hand, Figure 5.6 captured better speedup achieved by BTO Miner as compared to MVTO Miner for all the workloads because Ballot contract is write-intensive as defined in Solidity documentation [31]. Along with this, Figure 5.7 represents that the speedup achieved by MVTO Miner is greater than BTO Miner for Mixed contract on all the workloads. Due to equal proportions of all the above three contracts, the Mixed contract becomes read-intensive. So, the properties of the Mixed contract are same as Simple Auction and Coin contract with similar reasoning.

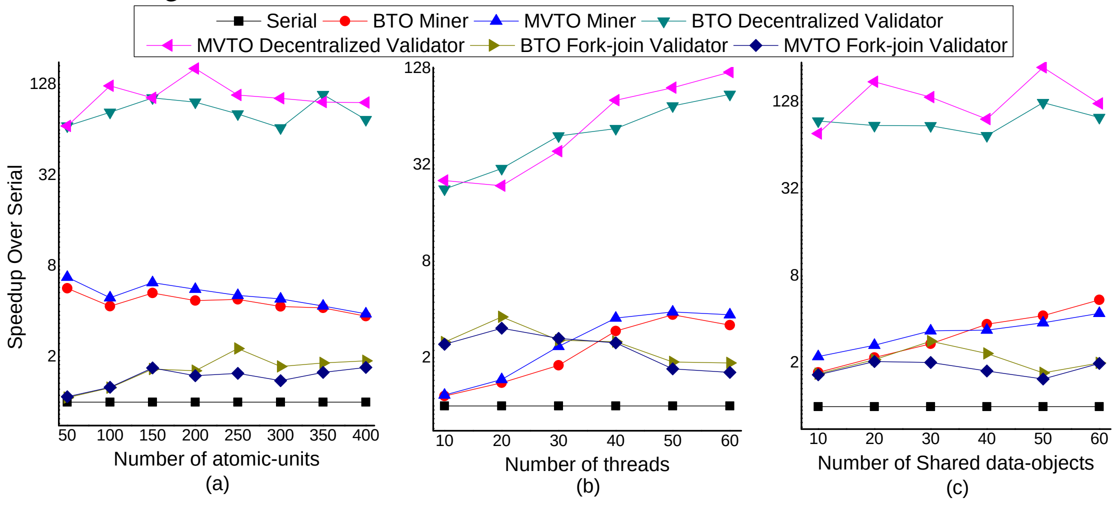


Figure 5.4: Simple Auction Contract

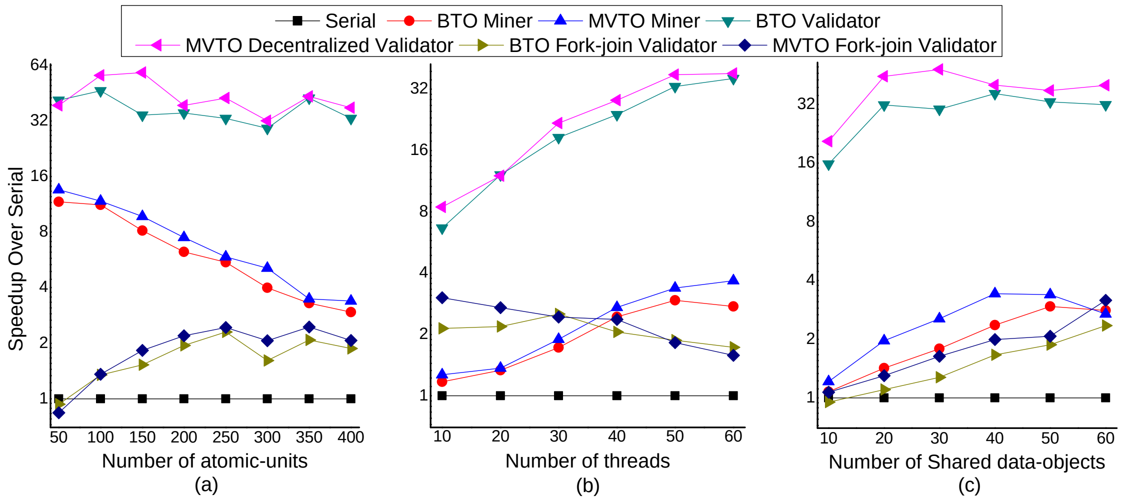


Figure 5.5: Coin Contract

The speedup achieved by the concurrent decentralized validator is very high as compared to serial validator because concurrent decentralized validator executes contracts concurrently and deterministically using BG given by concurrent miner. BG simplifies the parallelization task for the validator as validator need not to determine the conflicts, and directly executes non-conflicting transactions concurrently. It is clear from all the figures (Figure 5.4, Figure 5.5, Figure 5.6, and Figure 5.7) that BTO and MVTO Decentralized Validator is giving far better

Table 5.1: Speedup achieved by concurrent Miner and Validator

	Simple Auction			Coin			Ballot			Mixed		
	W1	**W2**	**W3**	**W1**	**W2**	**W3**	**W1**	**W2**	**W3**	**W1**	**W2**	**W3**
BTO Miner	4.6	2.4	3.4	6.6	2.1	2.1	3.8	3.1	2.7	4.8	1.6	2
MVTO Miner	5.2	2.7	3.3	7.5	2.4	2.5	2.3	1.5	1.8	5.7	1.8	2.4
BTO Decentralized Validator	85.7	53.1	95.23	36.9	21.7	29.8	126.7	152.1	279	90.7	68.6	112.5
MVTO Decentralized Validator	108.5	64.6	139.1	43.5	24.4	38.6	135.8	180.8	282.1	109.5	67.4	109.2
BTO Fork-join Validator	1.7	2.5	2.1	1.7	2.1	1.5	2.1	3.8	3.3	1.5	1.9	2.5
MVTO Fork-join Validator	1.5	2.3	1.8	1.9	2.3	1.9	1.8	3.8	2.2	1.5	2.7	1.9

performance than BTO and MVTO Fork-join Validator. A possible reason can be master thread of BTO and MVTO Fork-join Validator becomes slow to assign the task to slave threads. Even, the speedup achieved by BTO and MVTO Fork-join Validators are less than serial for 50 AUs on Coin contract due to the overhead of allocating the task by master thread.

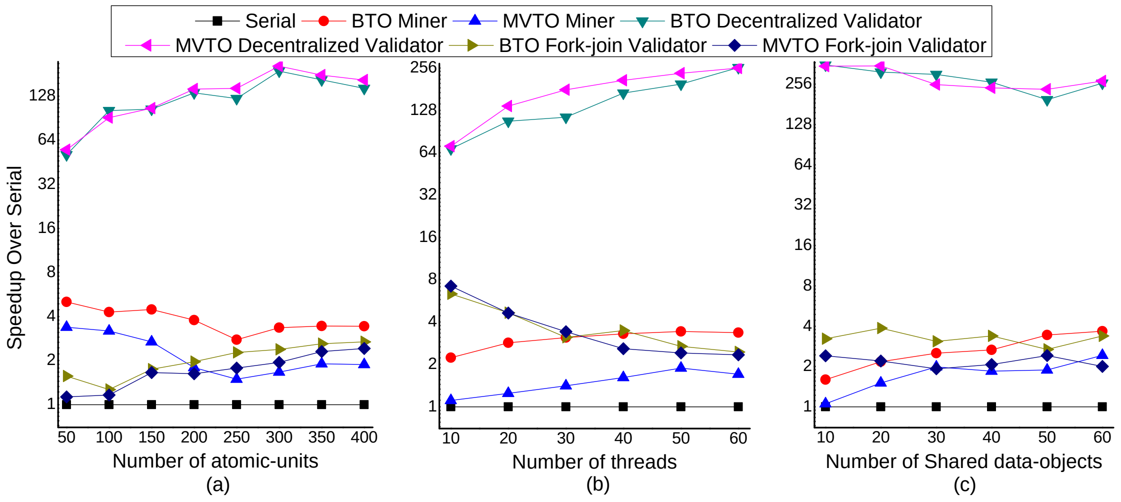


Figure 5.6: Ballot Contract

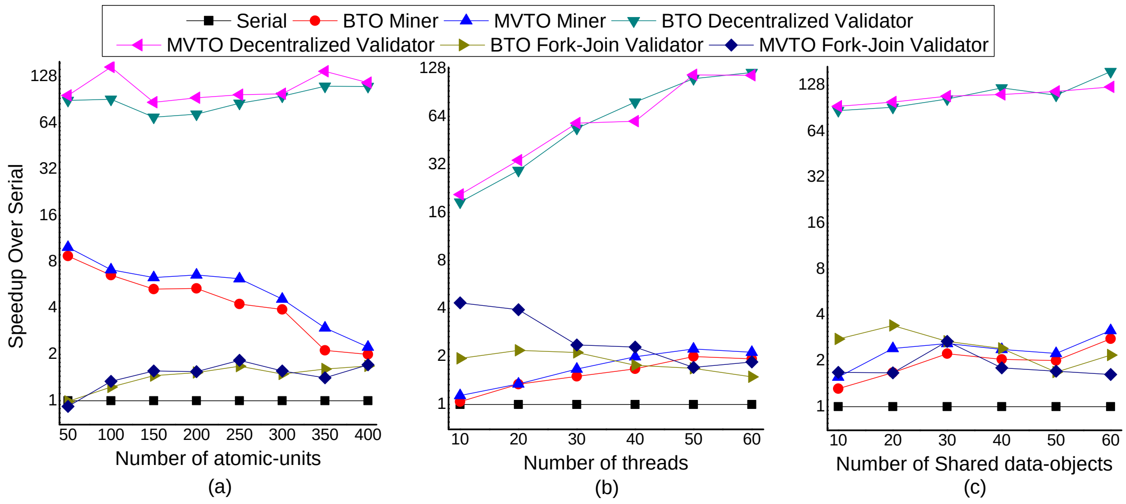


Figure 5.7: Mixed Contract

Average Time taken by each Contract: Figure 5.8, Figure 5.9, Figure 5.10 and Figure 5.11, represents the average time taken by Simple Auction, Coin, Ballot and Mixed contracts benchmark respectively. It can be seen that time taken by serial miner and validator is higher than the proposed concurrent miner and validator. Moreover, the serial validator is taking less time than the serial miner this is because validator will only get the valid transaction in the block given by the miner. From Figure 5.10, this can be observed that for write-intensive workload performance of BTO Miner is better than MVTO Miner. However, for all other workloads which are read-intensive MVTO Miner gives better performance than BTO Miner.

For all the benchmarks, the concurrent validator is taking less time compared to the concurrent miner because concurrent miner did all the required task to find the data conflict and generates the BG to help the validator. In this process, the validator will get a deterministic order of execution in the form of BG and without bothering about data conflicts, validator executes the atomic-units concurrently. However, it can be seen in all the figures, BTO and MVTO Decentralized Validator dominates a BTO and MVTO Fork-join Validator because of overhead associated with a master thread to assign the tasks (atomic-units) to slave threads.

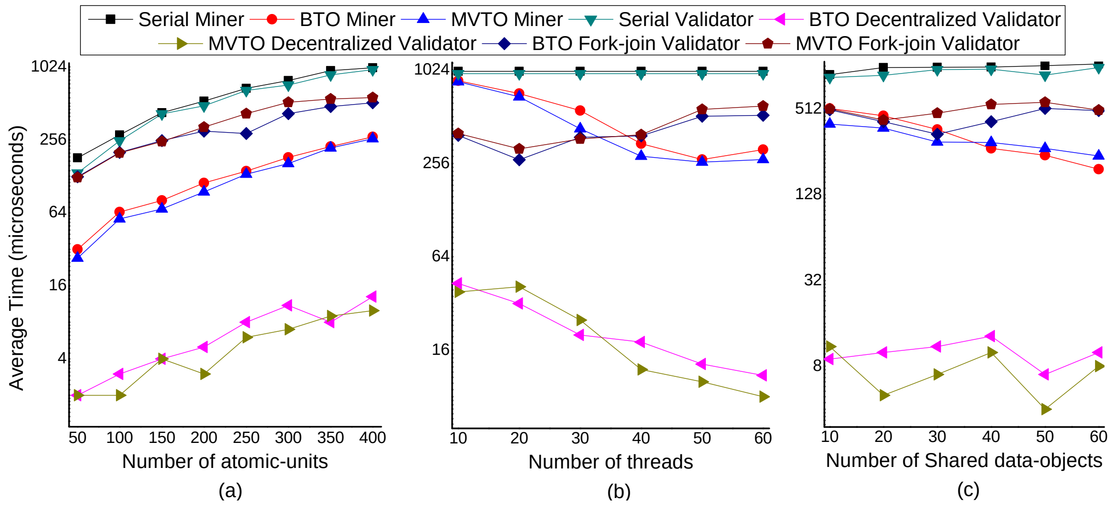


Figure 5.8: Average Time taken by Simple Auction Contract

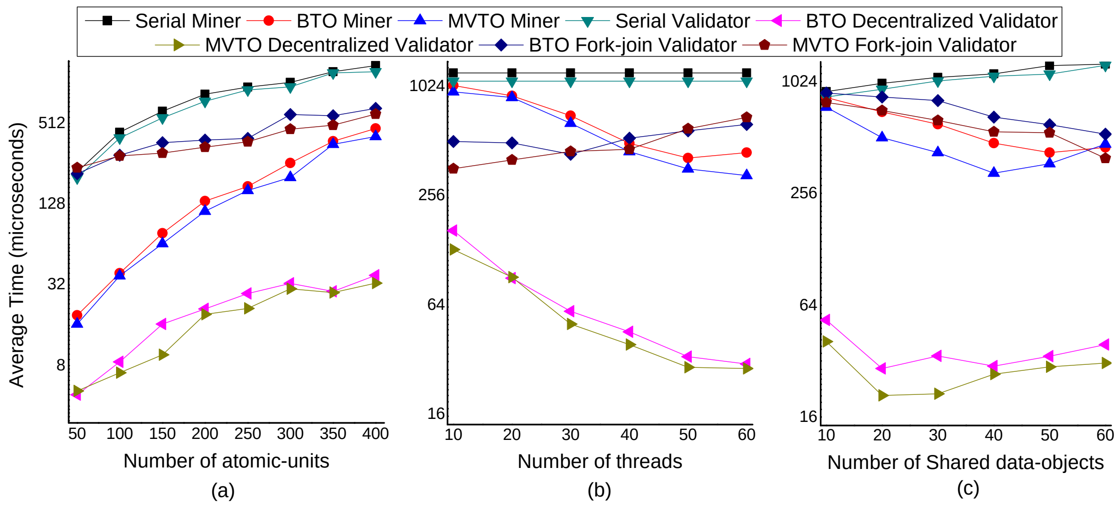


Figure 5.9: Average Time taken by Coin Contract

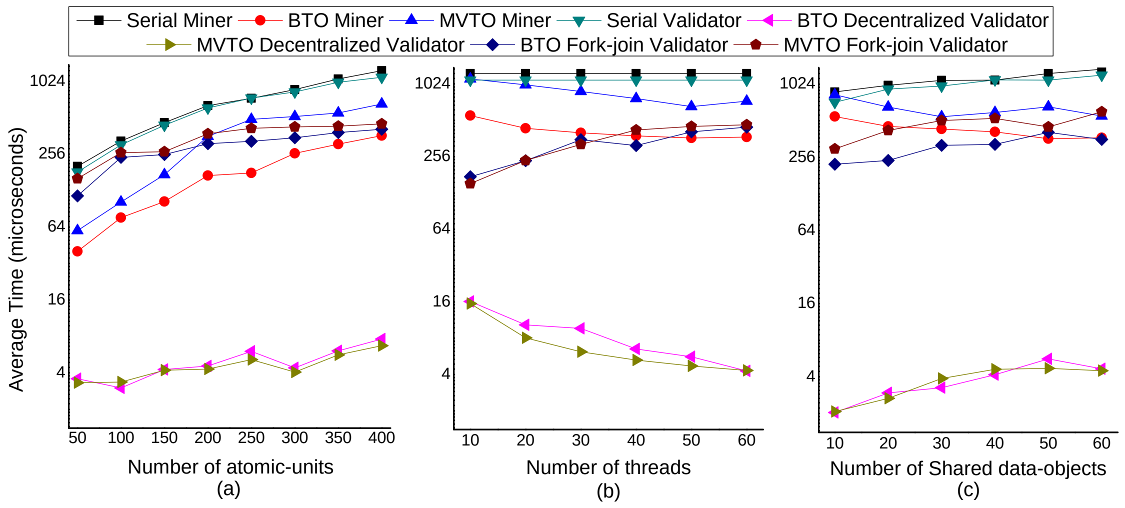


Figure 5.10: Average Time taken by Ballot Contract

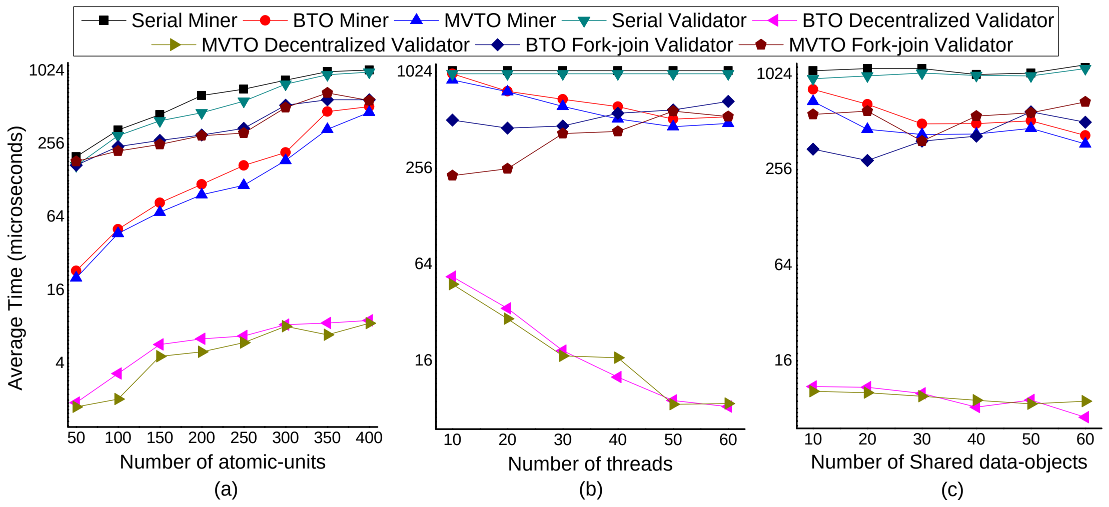


Figure 5.11: Average Time taken by Mixed Contract

5.6 Summary

To exploit the multi-core processors, we have proposed the concurrent execution of smart contract by miners and validators which improves the throughput. Initially, miner executes the smart contracts concurrently using optimistic STM protocol as BTO. To reduce the number of aborts and improves the efficiency further, the concurrent miner uses MVTO protocol which maintains multiple versions corresponding to each data-object. Concurrent miner proposes a block which consists of a set of transactions, BG, hash of the previous block and final state of each shared data-object. Later, the validators re-execute the same smart contract transactions concurrently and deterministically with the help of BG given by miner which capture the conflicting relations among the transactions to verify final state. If the validation is successful then proposed block appended into the blockchain and miner gets incentive otherwise discard the proposed block. Overall, BTO and MVTO miner performs 3.6x and 3.7x speedups over serial miner respectively. Along with, BTO and MVTO validator outperform average 40.8x and 47.1x than serial validator respectively.

Chapter 6

Conclusion and Future Directions

This thesis explores the progress guarantees in *Multi-Version Software Transactional Memory systems (MVSTMs).* We observed that *wait-freedom* is an interesting progress guarantee for STMs in which every transaction commits regardless of the nature of concurrent transactions and the underlying scheduler [18]. But it was shown by Guerraoui and Kapalka [19] that it is not possible to achieve *wait-freedom* in dynamic STMs in which data sets or t-objects of transactions are not known in advance. So in this thesis, we explored the weaker progress condition of *starvation-freedom* for transactional memory systems while assuming that the t-objects of the transactions are *not* known in advance.

6.1 Conclusion of the Thesis

RWSTMs: Initially, we explored starvation-freedom in RWSTMs and proposed a novel and efficient *Starvation-free Multi-Version Read-Write Software Transactional Memory systems (SF-MV-RWSTMs)* in Chapter 3. We introduced the *Garbage Collection (or GC)* mechanism in SF-MV-RWSTM to delete the unwanted versions denoted as *SF-MV-RWSTM-GC*. Although SF-MV-RWSTM-GC provide greater concurrency, they suffer from the cost of garbage collection. So, to avoid this issue, we proposed an efficient and novel starvation-free bounded-version RWSTM system as *Starvation-Free K-version RWSTM* or *SF-K-RWSTM* for a given parameter K in Chapter 3. Here, K is the number of versions of each t-object and can range from 1 to ∞. SF-K-RWSTM ensures the starvation-freedom while satisfying the correctness-criteria as strict-serializability [5] and local opacity [34]. The experimental analysis demonstrates that the proposed *SF-K-RWSTM* algorithm performs best on *max time* (maximum time for a transaction to commit) among its variants (*SF-MV-RWSTM* and *SF-MV-RWSTM-GC*) along with state-of-the-art STMs under long-running transactions with high contention.
OSTMs: Some STMs work at higher-level operations [9, 14, 15] and ensure greater concurrency than MV-RWSTMs and SV-RWSTMs. They include semantically rich operations such as push/pop on stack objects, enqueue/dequeue on queue objects and insert/lookup/delete on

sets, trees or hash table objects depending upon the underlying data structure used to implement higher-level systems. Such STMs are known as *Single-Version Object-based STMs (or SV-OSTMs).*

To obtain higher concurrency while achieving the starvation-freedom initially, we proposed a new algorithm of Starvation-Freedom for Single-Version OSTM as SF-SV-OSTM which satisfied conflict-opacity (or co-opacity) [9] as explained in Chapter 4. To achieve the greater concurrency further while ensuring the starvation-freedom, we maintain multiple versions corresponding to each t-object in starvation-free OSTMs and proposed a novel and efficient *Starvation-Freedom Multi-Version OSTM (or SF-MV-OSTM)* which satisfied local opacity [34]. To utilize the memory efficiently we developed two variants: (1) One in which we use *Garbage Collection (GC)* with SF-MV-OSTM. We denote this as *SF-MV-OSTM-GC*. (2) Another one is bounded with the latest *K-versions* denoted as *SF-K-OSTM*. These are similar to the ones developed for SF-MV-RWSTMs. Our experimental analysis shows that the proposed SF-K-OSTM performs best among its variants (*SF-MV-OSTM* and *SF-MV-OSTM-GC*) along with state-of-the-art single and multi-version RWSTMs, single-version OSTMs. SF-K-OSTM ensures starvation-freedom and satisfies the correctness criteria as local opacity [8].

So in this thesis, we explored the weaker progress condition *starvation-freedom* for single and multi-version RWSTM, single and multi-version OSTM systems while assuming that the t-objects of the transactions are *not* known in advance. Proposed multi-version SF-MV-RWSTMs and SF-MV-OSTMs show better performance than single-version SF-SV-RWSTMs and SF-SV-OSTMs respectively. So, we can conclude that maintaining multiple versions improves the concurrency than single-version while reducing the number of aborts and increases the throughput. This motivated us to use multi-version STMs as an effective tool to improve the performance of smart contract executions in blockchain systems described in Chapter 5.

We introduced a framework to execute the smart contract transactions by concurrent miners using efficient MVSTMs in Chapter 5. We implemented the concurrent miner with the help of single-version BTO [2, Chap 4] and multi-version MVTO [10] STM protocols that satisfy opacity [7]. We proposed concurrent validator that re-executes the smart contract transactions deterministically and efficiently with the help of BG given by concurrent miner to avoid the FBR error. BTO and MVTO miner performs 3.6x and 3.7x average speedups over serial miner respectively. Along with, BTO and MVTO validator outperform an average 40.8x and 47.1x than serial validator respectively.

So, this thesis addressed the bottleneck of blockchain by executing the smart contract transactions concurrently using efficient MVSTM protocols along with exploring the progress guarantees starvation-freedom in single and multi-version RWSTM, single and multi-version OSTM systems.

6.2 Directions for Future Research

An interesting application of the proposed efficient *Multi-Version Software Transactional Memory* is blockchain systems. MVSTMs improve the performance of blockchain while executing the transactions concurrently. A few appealing directions for future research are as follows:

- Identification of Malicious Miner by the Validators
- Concurrent Execution of Smart Contract Transactions (or SCTs) without Block Graph (BG)
- Concurrent Execution of Smart Contract Transactions using Object-based STMs

The detailed description of all the future research is as follows:

6.2.1 Identification of Malicious Miner by the Validators

There is a possibility of malicious activities by a miner with the BG based execution proposed by us in Chapter 5. Later, validators re-execute the smart contract transactions concurrently with the help of BG given by miner to verify the proposed block such as Dickerson et al. [30]. **Effect of Malicious Miners:** Suppose the miner that produces a block is malicious and does not add some edges to the BG. This can result in the blockchain systems entering inconsistent states due to *double spend*. We motivate this with an example. Consider three bank accounts A, B, C maintained on the blockchain with the current balance being \$100 in each of them. Now consider two smart contract transactions T_i, T_j which are conflicting where (a) T_i transfers \$50 from A to B; (b) T_j transfers \$60 from A to C. Considering the initial balance of \$100 in A account, both the transactions cannot be executed.

If a malicious miner, say mm does not add an edge between these two transactions in the BG then both these smart contract transactions can execute concurrently by validators. Then such execution could result in the final state with the balances in the accounts A, B, C as 40, 150, 160 respectively or 50, 150, 160. As we can see, neither of these final states can be obtained from any serial execution and are not correct states. Suppose the miner mm stores 40, 150, 160 for A, B, C in the final state and a validator v on concurrent execution arrives at the same state. Then, v will accept this block which results in its state becoming inconsistent. If the majority of validators similarly accept this block, then the state of the blockchain essentially has become inconsistent. We denote this problem as *edge missing BG* or *EMB*.

To handle EMB, the validator must reject any block which has missing edges in its BG. Execution of such a graph by the validator threads can allow conflicting transactions to execute concurrently, leading to an inconsistent state. To detect such an execution, the validator threads watch and identify transactions performing conflicting access on the same variables while executing concurrently. So, to handle this issue, we plan to develop a *Smart Concurrent Validator* *(or SCV)* which is capable of identifying missing edges in the BG proposed by the (possibly malicious) miner and reject the corresponding block, if required.

6.2.2 Concurrent Execution of SCTs without Block Graph (BG)

There is another future direction related to the size of the BG. A natural question is whether BG is becoming overhead. Currently, the number of smart contract transactions in a block is $\approx$ 100 in Ethereum [44]. So, storing BG inside the block is not consuming much space. However, it can grow over time, so storing the large BG will consume more space. Hence, constructing storage optimal BG is an interesting challenge. Or achieving the concurrent execution of smart contract transactions correctly without any extra BG without compromising with the speedup will be another interesting future direction.

In Ethereum blockchain, smart contracts are written in Solidity and runs on the Ethereum Virtual Machine (EVM). The issue with EVM is that it does not support multi-threading and hence give poor throughput. So, it is not possible to test the proposed approach on Ethereum. To analyze the performance, we did the simulation. So, an interesting research direction can be proposing multi-threaded EVM.

Another interesting aspect for future research is to test our proposed approach to several other blockchains such as Bitcoin [28], EOS [33] that support multi-threading. Analyzing the performance benefits of MVSTMs in the actual blockchain instead of serial execution. Concurrent execution of smart contract transactions in Bitcoin, EOS will be very useful to the research community, industries, and the society to develop efficient decentralized applications.

6.2.3 Concurrent Execution of SCTs using Object-based STMs

As explained in Chapter 4, working on higher-level objects (such as hash tables, lists, etc.), *Single-Version Object-based STMs (SV-OSTMs or OSTMs)* achieve greater concurrency, better throughput as compared to RWSTMs. Further, it was observed that greater concurrency can be obtained using *Multi-Version OSTMs (MV-OSTMs)* by maintaining multiple versions for each shared data-item as opposed to traditional SV-OSTMs. So, another interesting future research direction of the thesis can be developing an efficient framework to execute the smart concurrent transactions concurrently by miners and validators based on object semantics.

6.3 Summary

This thesis comprises of mainly three contributions. First, it explored the weaker progress condition *starvation-freedom* for single and multi-version RWSTM while satisfying the correctness-criteria as *conflict-opacity* [9] and *local opacity* [34]. Second, to achieve the greater concurrency further, it explored the weaker progress condition *starvation-freedom* for single and multi-version OSTM systems while ensuring the correctness-criteria as *conflict-opacity* and *local opacity*. It showed that maintaining multiple versions improves the concurrency than single-version while reducing the number of aborts and increases the throughput. So in the third contribution, we use efficient multi-version STMs as an application to improve the performance of *blockchain*. This thesis addressed the bottleneck of blockchain by executing the smart contract transactions concurrently using an efficient MVSTM system.

Several directions for future research have been proposed in the thesis. It includes identifying the malicious miner by the smart concurrent validator, optimizing the size of the BG developed by a concurrent miner, multi-threaded EVM, and concurrent execution of smart contract transactions using object semantics to achieve better performance.

Bibliography

[1] 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.

[2] G. Weikum and G. Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[3] 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, May 1993. [Online]. Available: http://doi.acm.org/10.1145/173682.165164

[4] 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

[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] R. Guerraoui, T. A. Henzinger, and V. Singh, “Permissiveness in transactional memories,” in *Proceedings of the 22Nd International Symposium on Distributed Computing*, ser. DISC ’08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 305–319. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87779-0_21

[7] 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

[8] P. Kuznetsov and S. Peri, “Non-interference and local correctness in transactional memory,” *Theor. Comput. Sci.*, vol. 688, pp. 103–116, 2017. [Online]. Available: https://doi.org/10.1016/j.tcs.2016.06.021

[9] 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

[10] 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

[11] L. Lu and M. L. Scott, "Generic multiversion STM," in *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, 2013, pp. 134–148. [Online]. Available: https://doi.org/10.1007/978-3-642-41527-2_10

[12] S. M. Fernandes and J. a. Cachopo, "Lock-free and scalable multi-version software transactional memory," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 179–188. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941579

[13] D. Perelman, A. Byshevsky, O. Litmanovich, and I. Keidar, "Smv: Selective multi-versioning stm," in *Distributed Computing*, D. Peleg, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 125–140.

[14] 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: https://doi.org/10.1145/1345206.1345237

[15] 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

[16] 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

[17] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[18] ——, "On the nature of progress," ser. OPODIS 2011.

[19] V. Bushkov, R. Guerraoui, and M. Kapalka, "On the liveness of transactional memory," in *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, ser. PODC '12. New York, NY, USA: ACM, 2012, pp. 9–18. [Online]. Available: http://doi.acm.org/10.1145/2332432.2332435

[20] V. Gramoli, R. Guerraoui, and V. Trigonakis, "Tm2c: A software transactional memory for many-cores," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: ACM, 2012, pp. 351–364. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168872

[21] M. M. Waliullah and P. Stenstrom, "Schemes for avoiding starvation in transactional memory systems," *Concurr. Comput. : Pract. Exper.*, vol. 21, no. 7, pp. 859–873, May 2009. [Online]. Available: http://dx.doi.org/10.1002/cpe.v21:7

[22] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott, "A comprehensive strategy for contention management in software transactional memory," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 141–150. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504199

[23] V. P. Chaudhary, C. Juyal, S. S. Kulkarni, S. Kumari, and S. Peri, "Achieving starvation-freedom in multi-version transactional memory systems," in *Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers*, 2019, pp. 291–310. [Online]. Available: https://doi.org/10.1007/978-3-030-31277-0_20

[24] 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

[25] 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

[26] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing." in *IISWC*, D. Christie, A. Lee, O. Mutlu, and

B. G. Zorn, Eds. IEEE Computer Society, 2008, pp. 35–46. [Online]. Available: http://dblp.uni-trier.de/db/conf/iiswc/iiswc2008.html#MinhCKO08

[27] 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

[28] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[29] "Ethereum," http://github.com/ethereum, [Acessed 26-9-2019].

[30] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, ser. PODC '17. New York, NY, USA: ACM, 2017, pp. 303–312. [Online]. Available: http://doi.acm.org/10.1145/3087801.3087835

[31] "Solidity Documentation," https://solidity.readthedocs.io/, [Accessed 26-9-2019].

[32] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," ser. EuroSys '18. New York, NY, USA: ACM, 2018, pp. 30:1–30:15.

[33] "EOS," https://eos.io/, [Accessed 26-3-2019].

[34] P. Kuznetsov and S. Peri, "Non-interference and local correctness in transactional memory," in *Distributed Computing and Networking - 15th International Conference, ICDCN 2014, Coimbatore, India, January 4-7, 2014. Proceedings*, 2014, pp. 197–211. [Online]. Available: https://doi.org/10.1007/978-3-642-45249-9_13

[35] 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

[36] P. Kuznetsov and S. Ravi, "On the cost of concurrency in transactional memory," in *Principles of Distributed Systems*, A. Fernàndez Anta, G. Lipari, and M. Roy, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 112–127.

[37] 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, Jul. 1990. [Online]. Available: http://doi.acm.org/10.1145/78969.78972

[38] P. A. Bernstein and N. Goodman, "Multiversion concurrency control&mdash;theory and algorithms," *ACM Trans. Database Syst.*, vol. 8, no. 4, pp. 465–483, Dec. 1983. [Online]. Available: http://doi.acm.org/10.1145/319996.319998

[39] T. Riegel, P. Felber, and C. Fetzer, "A lazy snapshot algorithm with eager validation," in *Proceedings of the 20th International Conference on Distributed Computing*, ser. DISC'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 284–298. [Online]. Available: http://dx.doi.org/10.1007/11864219_20

[40] T. Crain, D. Imbs, and M. Raynal, "Read invisibility, virtual world consistency and probabilistic permissiveness are compatible," in *Algorithms and Architectures for Parallel Processing - 11th International Conference, ICA3PP, Melbourne, Australia, October 24-26, 2011, Proceedings, Part I*, 2011, pp. 244–257. [Online]. Available: https://doi.org/10.1007/978-3-642-24650-0_21

[41] V. Bushkov and R. Guerraoui, "Liveness in transactional memory," in *Transactional Memory. Foundations, Algorithms, Tools, and Applications - COST Action Euro-TM IC1001*, 2015, pp. 32–49. [Online]. Available: https://doi.org/10.1007/978-3-319-14720-8_2

[42] 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. [Online]. Available: http://dx.doi.org/10.1142/S0129626407003125

[43] 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

[44] "Ethereum," http://github.com/ethereum, [Acessed 26-3-2019].

[45] N. Szabo, "Formalizing and securing relationships on public networks." *First Monday*, vol. 2, no. 9, 1997. [Online]. Available: http://dblp.uni-trier.de/db/journals/firstmonday/firstmonday2.html#Szabo97

[46] L. Luu, J. Teutsch, R. Kulkarni, and P. Saxena, "Demystifying incentives in the consensus computer," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 706–719.

[47] K. Delmolino, M. Arnett, A. E. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016.*

[48] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16. New York, NY, USA: ACM, 2016, pp. 254–269. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978309

[49] 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.

[50] I. Sergey and A. Hobor, “A concurrent perspective on smart contracts,” in *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers*, 2017, pp. 478–493. [Online]. Available: https://doi.org/10.1007/978-3-319-70278-0_30

[51] L. Yu, W.-T. Tsai, G. Li, Y. Yao, C. Hu, and E. Deng, “Smart-contract execution with concurrent block building,” *2017 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pp. 160–167, 2017.

[52] W. Yu, K. Luo, Y. Ding, G. You, and K. Hu, “A parallel smart contract model,” in *Proceedings of the 2018 International Conference on Machine Learning and Machine Intelligence*, ser. MLMI2018. New York, NY, USA: ACM, 2018, pp. 72–77. [Online]. Available: http://doi.acm.org/10.1145/3278312.3278321

List of Publications[1]

Publications part of the Thesis:

- Conference Papers:
  1. "Achieving Starvation-Freedom in Multi-Version Transactional Memory Systems", Ved Chaudhary, Chirag Juyal, Sandeep Kulkarni, **Sweta Kumari**, and Sathya Peri in 7th International Conference On Networked Systems, 2019, Morocco.
  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. "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.

- Posters:
  4. "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)**.
  5. "Modified FOCC Protocol That Satisfies Opacity and Starvation Freedom" Ved Prakash Chaudhary, Raj Kripal Danday, Hima Varsha Dureddy, Sandeep Kulkarni, **Sweta Kumari**, and Sathya Peri in 30th IEEE International Parallel and Distributed Processing Symposium, 2016, Chicago, Illinois, USA. **(Microsoft Research Travel Grant)**

- Accepted as Work-in-Progress Paper:
  6. "Starvation Freedom in Transactional Memory Systems" Ved Prakash Chaudhary, Raj Kripal Danday, Hima Varsha Dureddy, Sandeep Kulkarni, **Sweta Kumari**, and Sathya Peri in 1st International Workshop on Algorithms & Architectures for Distributed Data Analytics (AADDA) held in conjunction with ICDCN 2017 at Hyderabad, India.

[1]Author sequence follows lexical order of last names.

Publications not part of the Thesis:

- Conference 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 20th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2018), Tokyo, Japan **(Best Student Paper Award)**.
  2. "Achieving Greater Concurrency in Execution of Smart Contracts using Object Semantics", Parwat Singh Anjana, **Sweta Kumari**, Sathya Peri, and Archit Somani. [submitted in OPODIS 2019]
- Journal Paper:
  3. "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]
- Accepted as Work-in-Progress Paper:
  4. "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.
- Poster:
  5. "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)**.