# Starvation Freedom in Multi-Version Transactional Memory Systems

Ved Prakash Chaudhary [1], Sandeep Kulkarni[2], Sweta Kumari[1], Sathya Peri [1]

[1]{cs14mtech11019}@iith.ac.in, [2]sandeep@cse.msu.edu,
[1]{cs15resch01004,sathya_p}@iith.ac.in
[1]Department of Computer Science & Engineering, IIT Hyderabad
[2]Department of Computer Science, Michigan State University



**Abstract**

Software Transactional Memory systems (STMs) have garnered significant interest as an elegant alternative for addressing synchronization and concurrency issues with multi-threaded programming in multi-core systems. In order for STMs to be efficient, they must guarantee some progress properties. This work explores the notion of starvation-freedom in Software Transactional Memory Systems (STMs). A STM systems is said to be starvation-free if every thread invoking a transaction gets opportunity to take a step (due to the presence of a fair scheduler) then the transaction will eventually commit.

A few starvation-free algorithms have been proposed in the literature in the context of single-version STM Systems. These algorithm work on the basis of priority. If two transactions conflict, then the transaction with lower priority will abort. A transaction running for a long time will eventually have the highest priority and hence commit. But the drawback with this approach is that if a set of high-priority transactions become slow then they can cause several other transactions to abort. In that case, this approach becomes similar to pessimistic lock-based approach.

Multi-version STMs maintain multiple-versions for each transactional object or t-object. By storing multiple versions, these systems can achieve greater concurrency. In this paper, we propose a multi-version starvation free STM, KSFTM, which as the name suggests achieves starvation freedom while storing K versions of each t-object. Here K is an input parameter fixed by the application programmer depending on the requirement. Our algorithm is dynamic which can support different values of K ranging from 1 to infinity . If K is infinity then there is no limit on the number of versions. But a separate garbage-collection mechanism is required to collect unwanted versions. On the other hand when K is 1, it becomes same as a single-version STM system.

We prove the correctness and starvation-freedom property of the proposed KSFTM algorithm. To the best of our knowledge this is the first multi-version STM system that is correct and satisfies starvation-freedom as well.

## 1 Introduction

In the past few years *Big Data Analytics* has become a very popular paradigm for solving problems of very diverse fields from engineering to education. It is clear that to solve challenges of big data analytics, huge processing power will be required. Multi-core systems which have become prevalent can address the processing needs of Data Analytics.

Programming multi-core systems is usually performed using multi-threading. But, multi-threading and hence multi-core programming typically involves synchronization and communication which can be very expensive. The cost of synchronization can sometime be high that it can negate the programming power of multi-core systems and thus result in degrading multi-core to single-core systems.

Software Transactional Memory systems (*STMs*) [12, 21] have garnered significant interest as an elegant alternative for addressing synchronization and concurrency issues in multi-core systems. STMs are a convenient programming interface for a programmer to access shared memory without worrying about consistency issues [12, 21]. STM systems uses optimistic approach in which multiple transactions can execute concurrently. On completion, each transaction has to validate and if any inconsistency is found then it is *aborted*. Otherwise it is allowed to *commit*. A transaction that has begun but has not yet been validated is referred to as *live*. A typical TM system is a library which exports the methods: begin which begins a transaction, *read* which reads a *transaction-object* (data-item) or *tobj*, *write* which writes to a tobj, *tryC* which tries to commit.

An important requirement of STM systems is to precisely identify the criterion as to when a transaction should be aborted/committed referred to as *correctness-criterion*. Several correctness-criterion have been proposed for STMs such as opacity [9], virtual world consistency [14], local opacity [16], TMS [1, 5] etc. All these correctness-criterion require that all the transactions including aborted to appear to execute sequentially in an order that agrees with the order of non-overlapping transactions. Unlike the correctness criterion for traditional databases serializability [19], these correctness-criterion ensure that even aborted transactions read consistent values. This is one of the fundamental requirements of STM systems first observed in [9] which differentiates STMs from Databases.

Another important requirement of STM system is to ensure that transactions make *progress* i.e. they do not abort unnecessarily. It would be ideal to abort a transaction only when it does not violate correctness requirement (such as opacity). However it was observed in [2] that many STM systems developed so far spuriously abort transactions even when not required.

Wait-freedom is one of the interesting progress condition for STMs in which every transaction commits regardless of the nature of concurrent processes [11]. But it was shown by Guerraoui and Kapalka [10] that it is not possible to achieve wait-freedom in dynamic TMs in which data sets of transactions are not known in advance. So in this paper, we explore a weaker progress condition *starvation-freedom* [13, chap 2], to ensure that every transaction that is attempted infinitely often eventually succeeds. Intuitively, it is defined as follows in the context of TM systems: Suppose a transaction $T_i$ on getting aborted by the TM system is re-executed. Then, the STM system is said to be starvation-free if it can ensure that $T_i$ will eventually commit if $T_i$ is retried every time it aborts (and $T_i$ does not invoke tryA). It can be seen that in order to ensure starvation-freedom, the STM system must store some state information for each aborted transaction.

Algo 1 illustrates starvation-freedom. It shows the overview of *insert* method which inserts an element $e$ into a linked-list $LL$. Insert method is implemented using transactions to ensure correctness in presence of concurrent threads operating on common data-items. The method has an infinite while loop Line 1 to Line 15. In this while loop, a new transaction is created to read and write onto the shared memory. This corresponds to creating and inserting a new node into the shared memory. If the transaction succeeds then the control breaks out of the loop. Otherwise, this process continues until a transaction is eventually able to succeed. Thus, it can be seen that insert method can execute forever if transactions created by it never successfully commits. To ensure that insert method eventually completes, the STM system must guarantee starvation-freedom of transactions.

Gramoli et.al has proposed fair FairCM contention manager that satisfies starvation-freedom for many-core systems. They have used cumulative time to achive it [7]. In our paper, we explore ideas to achieve starvation-freedom for STMs. We first present *Single-Version Starvation Free STM* or *SV-SFTM* , in which system maintains single version for each tobj. We believe that SV-SFTM is less expensive [Section 3] than TM$^2$C [7] because we need not to calculate cumulative time for each successful transaction.

FairCM gurantees Starvation-freedom [7] but they explained only intution but not formally proved it. To the best of our knowledge, our work is the first that formally proves the Starvation freedom of transactional memory systems.

SV-SFTM is based on Forward-Oriented Optimistic Concurrency Control Protocol (FOCC), a commonly used optimistic algorithm in databases [22, Chap 4]. As per this algorithm, when two transactions $T_i, T_j$ conflict, one of them is aborted. The transaction to be aborted, say $T_j$, is one which has lower priority in terms of how long it has executed. When a transaction $T_i$ begins, it is allotted an *initial-timestamp* or *G_its*. If $T_i$ gets aborted, then it restarts again with a new identity, say $T_p$, but retains the original $G\_its$. In case of conflict of $T_p$ with $T_j$, the conflict is resolved based on $G\_its$ of $T_p$ (which is same as $T_i$) and $T_j$. The transaction with higher $G\_its$ is aborted. The details of this algorithm are described in SubSection3.1.

It was observed that more read operations succeed by keeping multiple versions of each object, i.e.multi-version STMs can ensure that more read operations to return successfully [15, 18].

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

**while** $(true)$ **do**
    $id$ = tbegin ();
    ...
    ...
    v = $read(id, x)$;
    ...
    ...
    $write(id, x, v')$;
    ...
    ...
    $ret$ = $tryC(id)$;
    **if** $(ret == success)$ **then**
        break;
    **end if**
**end while**

Thus, multi-version STMs (MVSTMs) can achieve greater concurrency and progress. Many STM systems have been proposed using the idea of multiple versions [15, 18, 6, 4, 20]. All these MVSTMs do not place a limit on the number of versions created. They have separate thread routines that perform *garbage-collection* on old and unwanted versions periodically. In fact, it was shown in [15], greater the number of versions, lesser the number of aborts. So, we propose K-version Multi-Version STM system that maintains $K$ versions, KSTM, which is the extention of $MVTO$ [15].It is a precursor to KSFTM as KSTM does not guarantee starvation-freedom, but provides an insight into how to achieve starvation-freedom with multi-version STMs.

KSTM maintains $K$ versions where $K$ can range from between $1 - \infty$. When $K$ is 1 then this algorithm boils down to a single-version STM system. If $K$ is $\infty$ then it is similar to existing MVSTMs which do not maintain a upper bound on the number of version. We show KSTM satisfies opacity.

It can be seen that SFTM does not take advantage of multiple versions. As a result, SFTM can still cause abort of many transactions (although it ensures that every transaction commits if it is re-executed sufficient number of times). Consider the case that a transaction $T_i$ with has the lowest $G\_its$. Hence, it cannot be aborted as per SFTM. But if it is slow (for some reason), then it can cause several other conflicting transactions to abort. Hence, the progress of the entire system can be brought down. We can alleviate this situation by using multiple versions.

Hence, we develop a Multi-Version Starvation Free STM System, *KSFTM* that guarantees starvation-freedom of transactions.

To study the efficiency of STMs developed, we will consider a useful metric *commit-throughput* defined as the time taken by a transaction to commit which includes the re-execution time caused by aborts. Naturally, this metric depends on the applications with which the STM system is tested. We plan to measure the performance commit-throughput of SFTM, KSTM and KSFTM using various benchmarks. The advantage of KSTM is that one can tune the value of $K$ to obtain the best commit-throughput for a given application. We want to understand which variant of STM can provide greater commit-throughput: FOCC,SFTM, KSTM, KSFTM. For the latter two, we have to experiment with a suitably chosen value for $K$.

*Overview of our Contributions and Roadmap.* We describe our system model in Section 2. Section 3, Section 4 and Section **??** illustrates Motivation for Starvation Freedom in Multi-Version Systems, Working of KSFTM and Proof outline of Safety & Liveness of KSFTM respectively. We conclude in Section 7. Finally in appendix, we describe proofs in details.

# 2 System Model and Preliminaries

Following [10, 8], we assume a system of $n$ processes, $p_1, \ldots, p_n$ that access a collection of *transactional objects* (or *tobjs*) via atomic *transactions*. Each transaction has a unique identifier. Within a transaction, a processes can execute *transactional methods/operations* or *methods*: tbegin operation that beings the transaction and returns an unique transaction identifier to the application; *stm-write*$(x, v)$ operation that tries to update a t-object $x$ with value $v$; *stm-read*$(x)$ operation tries to read $x$; *tryC*$()$ that tries to commit the transaction and returns $\mathcal{C}$ if it succeeds;

and *tryA*() that aborts the transaction and returns $\mathcal{A}$. For the sake of presentation simplicity, we assume that the values taken by arguments by *stm-write* operations are unique.

Operations *stm-write*, *stm-read* and *tryC*() may return $\mathcal{A}$, 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 returns $\mathcal{A}$ or $\mathcal{C}$. We denote any operation that returns $\mathcal{A}$ or $\mathcal{C}$ as *terminal operations* or as *term-ops*. Hence, operations $tryC$ and $tryA$ are terminal operations. A transaction does not invoke any further operations after terminal operations.

In this document, we use the terms operations and methods interchangeably. We denote all the operations of a transaction as $stm\text{-}methods$. For a transaction $T_k$, we denote all the tobjs accessed by its stm-read operations as $rset(T_k)$ or $rset_k$ and tobjs accessed by its stm-write operations as $wset(T_k)$ or $wset_k$.

*Events and Executions.* Suppose a transaction $T_i$ invokes a stm-method. During the course of the execution of the method, $T_i$ executes several atomic *events* one after another. These events are (1) read, write on shared/local memory objects. Note that these read and write are different from stm-read and stm-write methods; (2) method invocations or *inv* event & responses or *rsp* event on stm-methods. We assume that all events are the atomic and will be executed in a single clock cycle without any interruption. We denote the *execution* of a STM system as a totally ordered collection of events. We formally denote an execution $E$ as the tuple $\langle evts, <_E \rangle$, where $E.evts$ denotes the set of all events of $E$ and $<_E$ is the total order among these events.

*Histories.* A *history* consists only of stm-method inv and rsp events of an execution. In other words, a history views the methods as black boxes without going inside the internals. Similar to an execution, a history $H$ can be formally denoted as $\langle evts, <_H \rangle$ where $evts$ are of type inv & rsp and $<_H$ defines a total order among these events. We now define a few notations on histories which can be extended to the corresponding executions. For a history $H$, we denote the corresponding execution as $H.exec$. Similarly for an execution $E$, we denote the corresponding history as $E.hist$.

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., (1) each $H|T$ consists of a read-only prefix (consisting of read operations only), followed by a write-only part (consisting of write operations only), possibly *completed* with a $tryC$ or $tryA$ operation[a], and (2) each $H|p_i$ is *t-sequential*: no transaction begins before the last transaction completes (commits or a aborts). We also assume that every history has an initial committed transaction $T_0$ that initializes all the t-objects with value 0.

The set of transactions that appear in $H$ is denoted by $H.txns$. The set of committed (resp., aborted) transactions in $H$ consists of the transactions that are committed (resp. aborted) after the last event in $H$ and is formally denoted by all the $H.committed$ (resp., $H.aborted$). The set of *incomplete* or *live* transactions in $H$ is denoted by $H.incomp(= H.live) = \{H.txns - H.committed - H.aborted\}$. It can be seen that a transaction $T_i$ is live in $H$ if $T_i$ does not execute a terminal operation till the last event of $H$.

We define a history $H1$ to be a *prefix* of $H2$ if $(H1.evts \subseteq H2.evts) \wedge (<_{H1} \subseteq <_{H2})$. In this case, we denote $H1 \sqsubseteq H2$. We say that $H1$ is a *strict* prefix of $H2$ if $H1 \neq H2$. We denote $H.prefixes$ be the set of all prefixes of $H$. Analogously, we say that $H2$ is an *extension* of $H1$ if $H1$ is a prefix of $H2$. $H2$ is a strict extension of $H1$ if $H2 \neq H1$. It can be seen that any transaction $T_i$ that is terminated in $H2$ is live in a history $H1$ that is a prefix of $H2$.

A history is said to be *sequential* if the invocation of each transactional operation is immediately followed by a matching response. Thus in sequential histories, 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, in sequential histories the only relevant events of a transaction $T_k$ are of the types: $r_k(x, v)$, $r_k(x, \mathcal{A})$, $w_k(x, v)$, $w_k(x, v, \mathcal{A})$, $tryC_k(\mathcal{C})$ (or $c_k$ for short), $tryC_k(\mathcal{A})$, $tryA_k(\mathcal{A})$ (or $a_k$ for short).

For a history $H$, we construct the *completion* of $H$, denoted $\overline{H}$, by inserting $tryA_k(\mathcal{A})$ immediately after the last event of every transaction $T_k \in H.live$.

*Transaction orders.* For two transactions $T_k, T_m \in H.txns$, we say that $T_k$ *precedes* $T_m$ in the *real-time order* of $H$, denote $T_k \prec_H^{RT} T_m$, if $T_k$ is complete in $H$ and the last event of $T_k$ precedes the first event of $T_m$ in $H$. If neither $T_k \prec_H^{RT} T_m$ nor $T_m \prec_H^{RT} T_k$, then $T_k$ and $T_m$ *overlap* in $H$.

We define a history $H$ to be *serial* [19] or *t-sequential* if it has no overlapping transactions. In other words, in a serial history, all the transactions are ordered by real-time.

*Sub-histories.* A *sub-history*, $SH$ of a history $H$ denoted as the tuple $\langle SH.evts, <_{SH} \rangle$ and is defined as: (1)

[a]It was shown in [17] that this restriction brings no loss of generality.

$<_{SH} \subseteq <_H$; (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 $H.txns$.Then $H.subhist(R)$ denotes the sub-history of $H$ that is formed from the operations in $R$.

*Valid and legal histories.* Consider a sequential history $H$. A successful read $r_k(x, v)$ (i.e., $v \neq A$) in history $H$ (i.e., $v \neq A$), is said to be *valid* if some there is a transaction $T_j$ that wrote $v$ to $x$ and committed before $r_k(x, v)$. Formally, $\langle r_k(x, v)$ is valid $\Leftrightarrow \exists T_j : (c_j <_H r_k(x, v)) \wedge (w_j(x, v) \in T_j.evts) \wedge (v \neq A)\rangle$. 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*$(T_i)$ ($T_i$ 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 $\Leftrightarrow$ $(v \neq A) \wedge (H.lastWrite(r_k(x, v)) = c_i) \wedge (w_i(x, v) \in T_i.stm\text{-}methods)\rangle$. 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.

*Strict Serializability and Opacity.* We say that two histories $H$ and $H'$ are *equivalent* if they have the same set of events. Now a sequential history $H$ is said to be *opaque* [9, 10] it is valid and there exists a serial legal history $S$ such that (1) $S$ is equivalent to $\overline{H}$ and (2) $S$ respects $\prec_H^{RT}$, i.e $\prec_H^{RT} \subset \prec_S^{RT}$.

Unlike this definition, the original definition of opacitywas not restricted to sequential histories. 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 the same lines, a valid history $H$ is said to be *strictly serializable* if $H.subhist(H.committed)$ is opaque. Thus, 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 subsequence of a serialization of $H$ that only contains transactions in $H.committed$.

*History(H').* For each aborted transaction $T_i$ consider all previously committed transactions including $T_i$ while immediately putting commit after last successful operation of $T_i$ and for last committed transaction $T_l$ consider all the previously committed transactions including $T_l$.

*Local opacity*: A history H is said to be local opaque if all the above *History(H')* are opaque.

For the sake of clarity, consider a history $H_5$ with multiple reads and writes on different t-objects: $w_1(x, 1)C_1$ $r_2(x, 1)w_3(x, 3)w_3(y_3)C_3r_4(y, 3)w_4(k, 4)C_4r_5(k, 4)r_5(z, 0)w_2(z, 2)C_2A_5$.

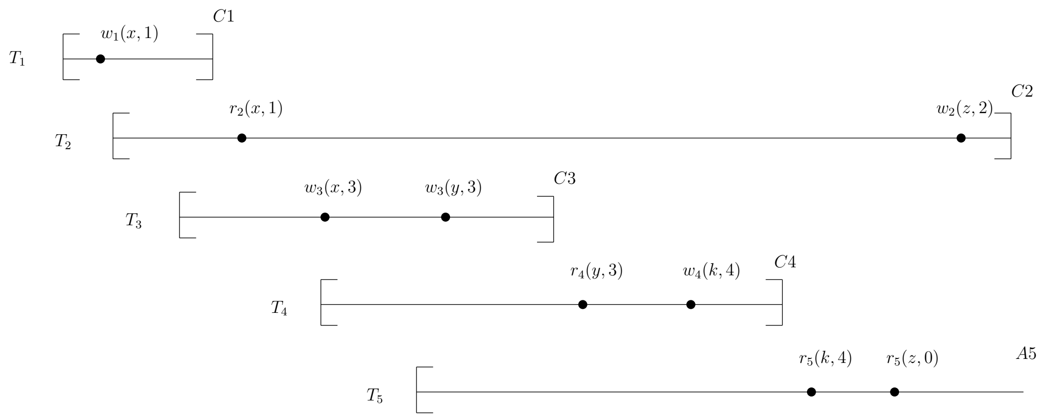



Figure 1: A locally opaque, but not opaque history $H5$

# 3 Motivation for Starvation Freedom in Multi-Version Systems

In this section, first we describe the starvation freedom solution used for single version i.e. SFTM algorithm and then the drawback of it.

## 3.1 Illustration of SFTM

Forward-oriented optimistic concurrency control protocol (FOCC), is a commonly used optimistic algorithm in databases [22, Chap 4]. In fact, several STM Systems are also based on this idea. In a typical STM system (also in database optimistic concurrency control algorithms), a transaction execution is divided can be two phases - a *read/local-write phase* and *try-Commit phase* (also referred to as validation phase in databases). The various algorithms differ in how the try-Commit phase executes. Let the write-set or wset and read-set or rset of a $t_i$ denotes the set of tobjs written & read by $t_i$. In FOCC a transaction $t_i$ in its try-Commit phase is validated against all live transactions that are in their read/local-write phase as follows: $\langle wset(t_i) \cap (\forall t_j : rset^n(t_j)) = \Phi \rangle$. This implies that the wset of $t_i$ can not have any conflict with the current rset of any transaction $t_j$ in its read/local-write phase. Here $rset^n(t_j)$ implies the rset of $t_j$ till the point of validation of $t_i$. If there is a conflict, then either $t_i$ or $t_j$ (all transactions conflicting with $t_i$) is aborted. A commonly used approach in databases is to abort $t_i$, the validating transaction.

In SFTM we use $tss$ which are monotonically in increasing order. We implement the `tss` using atomic counters. Each transaction $t_i$ has two time-stamps: (i) *current time-stamp or CTS*: this is a unique `ts` alloted to $t_i$ when it begins; (ii) *initial time-stamp or ITS*: this is same as CTS when a transaction $t_i$ starts for the first time. When $t_i$ aborts and re-starts later, it gets a new CTS. But it retains its original CTS as ITS. The value of ITS is retained across aborts. For achieving starvation freedom, SFTM uses ITS with a modification to FOCC as follows: a transaction $t_i$ in try-Commit phase is validated against all other conflicting transactions, say $t_j$ which are in their read/local-write phase. The ITS of $t_i$ is compared with the ITS of any such transaction $t_j$. If ITS of $t_i$ is smaller than ITS of all such $t_j$, then all such $t_j$ are aborted while $t_i$ is committed. Otherwise, $t_i$ is aborted. Due to lack of space, we have showed an example illustrates the working of SFTM in Section **??**. We show that SFTM satisfies opacity and starvation-free.

**Theorem 1** *Any history generated by SFTM is opaque.*

**Theorem 2** *SFTM ensure starvation-freedom.*

We prove the correctness by showing that the conflict graph [22, Chap 3], [16] of any history generated by SFTM is acyclic. We show starvation-freedom by showing that for each transaction $t_i$ there eventually exists a global state in which it has the smallest ITS.

Figure 2 shows the a sample execution of SFTM. It compares the execution of FOCC with SFTM. The execution on the left corresponds to FOCC, while the execution one the right is of SFTM for the same input. It can be seen that each transaction has two `tss` in SFTM. They correspond to CTS, ITS respectively. Thus, transaction $T_{1,1}$ implies that CTS and ITS are 1. In this execution, transaction $T_3$ executes the read operation $r_3(z)$ and is aborted due to conflict with $T_2$. The same happens with $T_{3,3}$. Transaction $T_5$ is re-execution of $T_3$. With FOCC $T_5$ again aborts due to conflict with $T_4$. In case of SFTM, $T_{5,3}$ which is re-execution of $T_{3,3}$ has the same ITS 3. Hence, when $T_{4,4}$ validates in SFTM, it aborts as $T_{5,3}$ has lower ITS. Later $T_{5,3}$ commits.

It can be seen that ITSs prioritizes the transactions under conflict and the transaction with lower ITS is given higher priority.

## 3.2 drawback of SFTM

Figure 3 is representing history H: $r_1(x,0)r_1(y,0)w_2(x,10)w_3(y,15)a_2a_3c_1$ It has three transactions $T_1$, $T_2$ and $T_3$. $T_1$ is having lowest time stamp and after reading it became slow. $T_2$ and $T_3$ wants to write to $x$ and $y$ respectively but when it came into validation phase, due to $r_1(x)$, $r_1(y)$ and not committed yet, $T_2$ and $T_3$ gets aborted. However, when we are using multiple version $T_2$ and $T_3$ both can commit and $T_1$ can also read from $T_0$. The equivalent serial history is $T_1T_2T_3$.

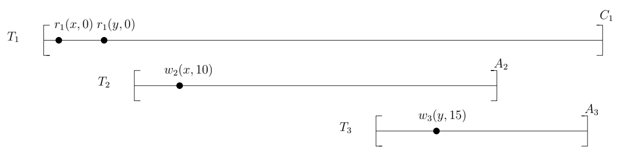



Figure 3: Pictorial representation of execution under SFTM

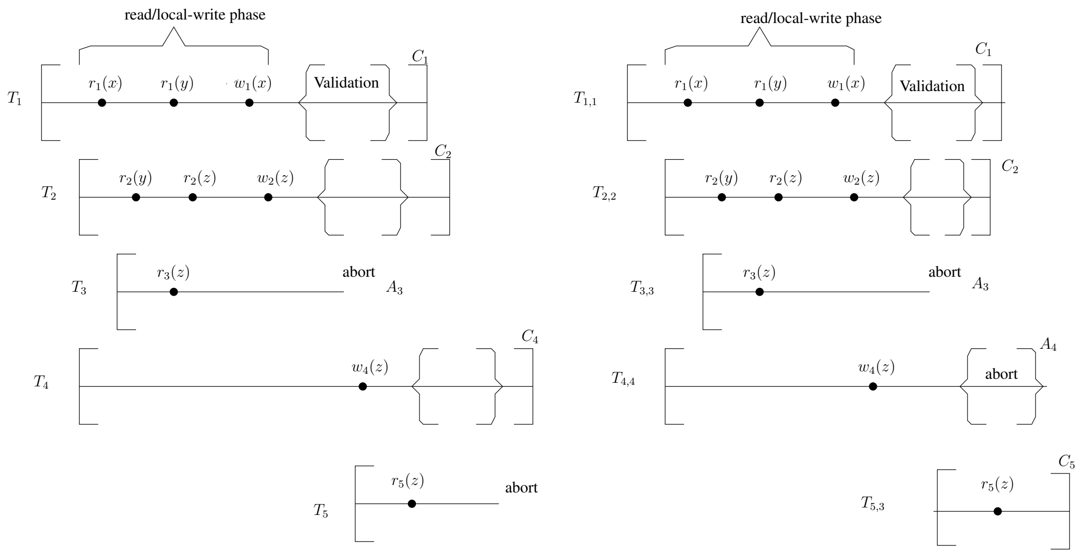



Figure 2: Sample execution of SFTM

# 4 Working of KSFTM

This section starts with the brief introduction of MVTO algorithm [15] and proceed to the main idea of KSTM. After that it describes, how KSTM does not satisfy starvation freedom. Then illustrates the main idea of KSFTM and ends with the pcode of it.

## 4.1 Main idea of KSTM

KSTM algorithm is based on *MVTO* algorithm for STMs [15] which again is similar to the MVTO algorithm proposed for databases [3]. The proposed MVTO algorithm does not maintain any limit on the number of versions. As a result it has to execute a separate garbage-collection procedure.

KSTM algorithm as the name suggests maintains $k$-versions for each tobj and uses `ts`s (like SFTM). Each tobj maintains all its versions as a linked-list. Each version of a tobj has three fields (1) `ts` which is the CTS of the transaction that wrote to it; (2) the value of the version; (3) a list, called read-list, consisting of transactions CTSs that read from this version.

1. $read(x)$: Transaction $t_i$ reads from a version of $x$ with `ts` $j$ such that $j$ is the largest `ts` less than $i$ (among the versions $x$), i.e. there exists no version $k$ such that $j < k < i$ is true. If no such version exists then $t_i$ is aborted.

2. $write(x, v)$: $t_i$ stores this write to value $x$ locally in its wset.

3. $tryC$: This operation consists of multiple steps:

   (a) $t_i$ validates each tobj $x$ in its wset as follows:

      i. $t_i$ finds a version of $x$ with `ts` $j$ such that $j$ is the largest `ts` less than $i$ (like in read).

      ii. Then, among all the transactions that have read from $j$ if there is any transaction $t_k$ such that $j < i < k$ and $t_k$ has already committed then $t_i$ is aborted. Otherwise, if $t_k$ is still live then $t_k$ is aborted. Transaction $t_i$ then proceeds to validate the next tobj in its wset.

      iii. If there exists no version of $x$ with `ts` less than $i$ then $t_i$ is aborted

   (b) After performing the tests of Step 3(a)i, Step 3(a)ii, Step 3(a)iii over each tobjs $x$ in $t_i$'s wset, if $t_i$ has not yet been aborted, then for each $x$: among all the versions of $x$ currently present, the oldest version is over-written with $i$ and $i$'s value. Transaction $t_i$ is then committed.

Further details of KSTM algorithm can found in appendix.

**Theorem 3** *Any history generated by KSTM is opaque.*

We prove the correctness of the algorithm by showing that the equivalent serial history, all the transactions are ordered by their `tss`. But KSTM does not satisfy starvation-freedom which is illustrated in an example.

**KSTM illustration:** We now illustrate the working of the algorithm with an example. Figure 4 shows an execution where $K = 3$ and the currently considered versions for a tobj $x$ are $5, 15$ & $25$. Consider version $15$. Its value is 8 and its read-list consists of transactions with `ts`s $17, 22$. The C next to id 22 indicates that $t_{22}$ is already committed. Transactions $t_{17}$ is still live. In this setting suppose transaction $t_{23}$ intends to commit and create a new version. In this case, $15 < 23 < 24$ and $t_{24}$ is still live. Hence, $t_{24}$ is aborted and a new version with `ts` 23 is allowed to be created. Since 5 is the oldest version, the newly created version 23 overwrites 5. Next, consider the case that transaction $t_{26}$ intends to commit and create a new version. Since $t_{29}$ is already committed, $t_{26}$ is not allowed to create a new version.

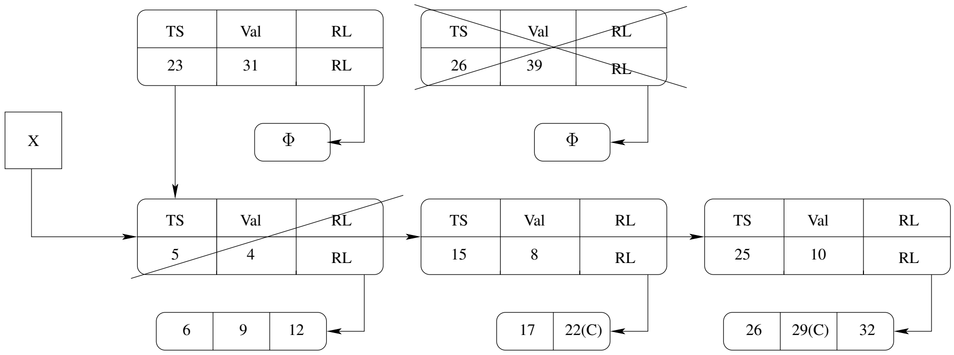



Figure 4: Sample execution of KSTM

In this example suppose $t_{26}$ has the lowest ITS and let $t_{29}$ have a higher ITS. But $t_{26}$ still has to abort due to commit of $t_{29}$. This shows the drawback of KSTM w.r.t starvation-freedom.

Thus, although $t_{26}$ has lowest ITS, it has to abort due to $t_{29}$ which has higher CTS. Suppose there was no transaction with higher CTS than $t_{26}$. Then, it can be seen that $t_{26}$ can not abort since it has lowest ITS and highest CTS.

Thus, the key observation here is that a transaction with lowest ITS and highest CTS can not abort. So, we used this property to build KSFTM.

## 4.2 drawback of KSTM

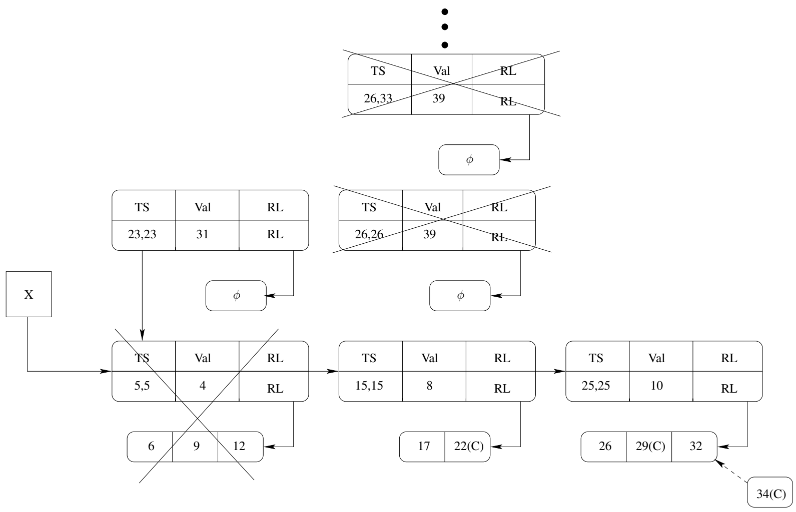



Figure 5: Pictorial representation of execution under KSTM

Figure 5 represents the execution under KSTM algorithm, in which transaction $T_{26}$ is starving. First time $T_{26}$ is getting aborted due to higher timestamp transaction $T_{29}$ has been committed in the readlist of $T_{25}$. After that $T_{26}$ retries with same $G\_its$ 26 but new $G\_cts$ 33. Lets assume the scenario in which before commit of $T_{26}$, transaction $T_{34}$ has been committed in the readlist of $T_{25}$ so, $T_{26}$ returns abort again. If such scenario occurs again and again then $T_{26}$ will starve. So, we proposed one more algorithm as KSFTM that ensures starvation-free STM. We describe a timestamp based algorithm for multi-version STM systems, K-version Starvation Free STM (KSFTM) algorithm that is locally opaque. As the name suggests the algorithm is starvation-free. We formally prove that our algorithm satisfies local opacity [16] using the graph characterization and starvation-freedom.

## 4.3 Outline of KSFTM Algorithm

We assume that in the absence of synchronization conflicts, every transaction will commit. In other words, if a transaction is executed in a system by itself, it will not self-abort. One way to satisfy starvation freedom in such a system is to order transactions based on their arrival time and ensure that we first execute the oldest transaction by itself, then the next oldest and so on. While this approach would provide starvation freedom, it lacks concurrency. Based on this, we require that

> If transaction $T_i$ does not conflict with Transaction $T_j$ (either due to accessing common variables or due to XXX) then (1) $T_i$ is not aborted due to actions of $T_j$, and (2) $T_i$ is not delayed due to transaction $T_j$.

Our goal in KSFTM was to provide priority for transactions that begin early. However, since conflicts between transactions is not known and we cannot abort transactions that have committed already, we need to modify this approach. To illustrate how we can modify KSTM to obtain starvation freedom, consider an example where we have two transactions, say $T_{50}$ and $T_{60}$ with $WTS$ value to be 50 and 60 respectively. Furthermore, assume that these transactions read and write variable $x$. Also, assume that the latest version is available at time $40$. We can view the transacitons in terms of two statements where the transcation first reads the value of $x$. These statements are marked as $r_{50}$ and $r_{60}$ respectively. Likewise, transactions $w_{50}$ and $w_{60}$ denote the corresponding write/tryCommit statement. Given that the reading must occur before writing/committing, there are six possible permutations of these statements. We identify these statements and the action that should be taken for that permutation:

1. $r_{50}, w_{50}, r_{60}, w_{60}$ & $T_{50}$ reads the version at time 40, $T_{60}$ 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 $w_{50}$.We must abort $T_{50}$.
4. $r_{60}, r_{50}, w_{60}, w_{50}$ & Conflict detected at $w_{60}$, We must 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. $r_{60}, w_{60}, r_{50}, w_{50}$ & $T_{50}$ cannot create the version and, hence, must be aborted.

Observe that in Scenario 1, there is no conflict.

In Scenario 2, we cannot allow $w_{50}$, as this would create a version with timestamp $50$ and $T_{60}$ should have read this value instated of the value at time $40$. Since both $T_{50}$ and $T_{60}$ are both live, we can abort any one of them.

In Scenario 3 and 4, when transaction $T_{60}$ commits, we are unaware of the intent by $T_{50}$ to write to $x$. Hence, we can allow $T_{60}$ to commit. However, when transaction $T_{50}$ tries to write and commit later, we must abort it. Allowing version 50 to be created would be inconsistent the values read by $T_{60}$.

In Scenario 5, when transaction $T_{50}$ tries to commit, we detect a conflict. Hence, we must abort either $T_{50}$ or $T_{60}$.

FInally, in Scenario 6, allowing $T_{50}$ to write and commit $x$ is not permitted as it would be inconsistent with values read by $T_{60}$.

This observation is the key to our approach to provide starvation freedom. In particular, if a transaction aborts and is restarted, we want it to choose a higher $WTS$ value. However, we want the transaction to choose this value independently, i.e., without coordinating with other transactions. This will be especially useful if we cannot identify all transactions in the system (e.g., in a distributed system).

We identify the basic structure of the algorithm. Each transaction $T_i$ is associated with three timestamps:
1)*An initial timestamp* $ITS_i$: when $T_i$ starts for the first time, it gets an $ITS_i$. When $T_i$ aborts and re-starts later, it retains same ITS.
2)*Current timestamp* $CTS_i$: This is a unique timestamp alloted to $T_i$ when it begins. It is same as ITS when $T_i$ starts for the first time. When $T_i$ aborts and re-startslater, it gets a new CTS.
3)*Working timestamp* $WTS_i$: Anytime, this transaction begins (either initially or after an abort), it selects a timestamp $WTS_i$. When $T_i$ starts for the first time, $WTS_i$, $CTS_i$ and $ITS_i$ are same. Goal of $T_i$ is to read the shared variables at time $WTS_i$ as well as create new versions at the same time. In other words, goal of $T_i$ is to essentially appear as it took 0 time and it executed at time $WTS_i$. To prevent other transactions from reading values of uncommitted transaction, each transaction performs all writes to local storage. Only when a transaction enters the tryCommit phase, it can potentially create new versions.
$WTS_i = CTS_i + C * (CTS_i - ITS_i)$; Where, $C$ is any constant greater or equal to than 1.
Our algorithm relies on two properties: First, when a new tranasction is initiated $WTS$ and $CTS$ are same. Our second requirement is that $WTS$ is strictly increasing. This implies that if a transaction is aborted several times then the difference between $WTS$ and $CTS$ increases. For sake of simplicity, we present correctness of the algorithm.

We proved that if a transaction has the highest $WTS$ value and the lowest $ITS$ value and this property remains stable then that transaction is guaranteed to commit. In order to utilize this theorem, we need to guarantee that (1) transaction with lowest $ITS$ value will eventually have the highest $WTS$ value and (2) no transaction with higher $WTS$ value will enter the system as long as this transaction is live.

For real time order transactions $T_i$ and $T_j$, only $WTS$ is not sufficient because it's not always in increasing order with respect to other real time transactions. So, we have introduced $G\_tltl_i$ & $G\_tutl_i$ to ensures real time order. Figure 6 represents a history $H : r_1(z,0)r_3(y,0)w_1(z,1)c_1r_2(x,0)w_2(x,2)c_2r_3(z,1)c_3$ with $WTS_1$, $WTS_2$ and $WTS_3$ as 80, 70 and 100 respectively. According to WTS order the serial schedule will be $T_2T_1T_3$. But it's violating the the real time order between $T_1$ and $T_2$. So, we have used $G\_tltl$ & $G\_tutl$ to get the correct serial schedule $T_1T_3T_2$.

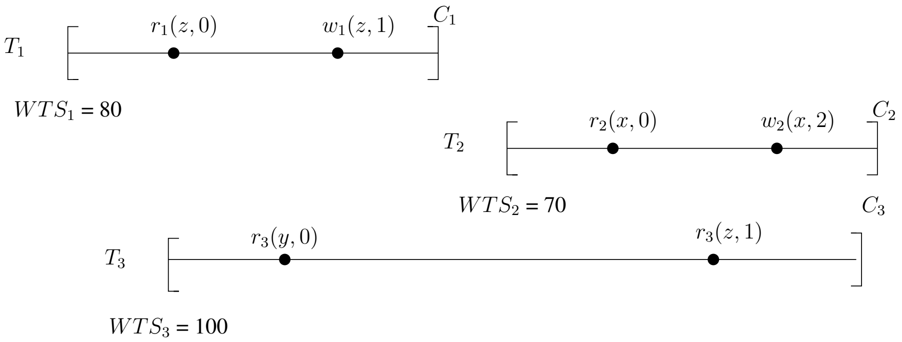



Figure 6: Need of $G\_tltl$ and $G\_tutl$

1. $read(i,x)$: A transaction $T_i$ on invoking read method for t-object x, It will search for the largest available version but less than itself.
   (a) If there exist a transaction $T_j$ such that it successfully created a version of x with ($G\_wts_j < G\_wts_i$) and j is the largest available timestamp $\leq$ i then increase $G\_tltl_i$.
      i. If there exist a transaction $T_k$ such that it successfully created a version of x with ($G\_wts_i < G\_wts_k$) and k is the smallest available timestamp $\geq$ i then $G\_tutl_i$ gets decremented.
      ii. If $G\_tltl_i$ is less than $G\_tutl_i$ $read(i,x)$ then returns the value written by $T_j$.
   (b) Otherwise, $read(i,x)$ returns abort.
2. $write_i(x, val)$: A Transaction $T_i$ writes into local memory.

3. $tryC()$: On invoke of $tryC()$ method by a transaction $T_i$ for each t-object x, in its Wset:

   (a) If there exist a transaction $T_j$ such that it successfully created a version of x with ($G\_wts_j < G\_wts_i$) and j is the largest available timestamp $\leq$ i then find the readlist of j and increment $G\_tltl_i$.

      i. If $T_k$ is in the readlist of j with ($G\_wts_i < G\_wts_k$) and ($T_k$ is committed or $G\_its_i > G\_its_k$) then $T_i$ returns abort. Otherwise, $T_k$ returns abort.

      ii. If $T_k$ is in the readlist of j with ($G\_wts_i > G\_wts_k$), ($G\_tltl_k \geq G\_tutl_i$) and ($T_k$ is committed or $G\_its_i > G\_its_k$) then $T_i$ returns abort. Otherwise, $T_k$ returns abort and $G\_tltl_i$ = $G\_tutl_i$.

   (b) If there exist a transaction $T_{j'}$ such that it successfully created a version of x with $G\_wts_i < G\_wts_{j'}$ and j' is the smallest available timestamp $\geq$ i then $G\_tutl_i$ gets decremented. If $G\_tltl_i$ is greater than $G\_tutl_i$ then $T_i$ returns abort.

   (c) otherwise, $T_i$ creates a new version and returns commit.

### 4.4 Execution under KSFTM

Figure 7 represents the execution under KSFTM algorithm which has three versions (K=3) for x t-object. All versions are connected as linklist in which each version is having three fields: TS as timestamp, Val as value written by the transaction and RL represents readlist i.e. all the reading transaction that has read from this verion. TS consists of 3 fields $G\_its$, $G\_cts$ and $G\_wts$. Whenever any transaction begins first time then all the timestamps will be same i.e.($G\_its = G\_cts = G\_wts$). Every time a transaction gets aborted it gets a new $G\_cts$ but retains same $G\_its$. For each transaction $G\_wts$ is calculated as ($G\_wts$ = 2* $G\_cts$ -$G\_its$).

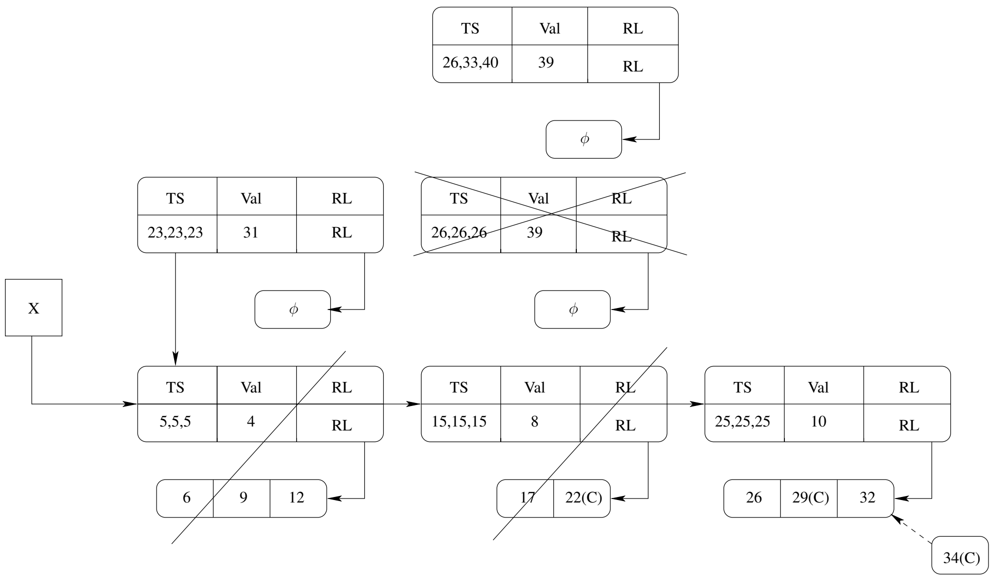



Figure 7: Pictorial representation of execution under KSFTM

Initially, Figure 7 is having three versions of x-tobject with timestamp 5, 15 and 25. Transaction $T_{23}$ creats a verion successfully and overwrites version with timestamp 5. After that transaction $T_{26}$ wants to create a version buts its returning abort because higher timestamp transaction $T_{29}$ has been committed in the readlist of $T_{25}$. So, $T_{26}$ retries with new $G\_cts$ and $G\_wts$ as 33 and 44 respectively and it returns commit.

# 5 K-Version Starvation Free STM

We describe a timestamp based algorithm for multi-version STM systems, K-version Starvation Free STM (KS-FTM) algorithm that is locally opaque. As the name suggests the algorithm is starvation-free. We formally prove that our algorithm satisfies opacity [10, 9] using the graph characterization and starvation-freedom.

## 5.1 Data Structures and Pseudocode

The STM system maintains a set of $n$ *transaction objects* or *tobjs* $\mathcal{T}$ onto which all the reads & writes are performed by the threads. We assume that all the tobjs are ordered as $x_1, x_2, ...x_n$.

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 $v$ 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 tobj 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). We name all the shared variable starting 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 time-stamps:

- $G\_lock_i$: A lock for accessing all the shared variables of $T_i$.
- $G\_its_i$ (initial timestamp): It is a time-stamp assigned to $T_i$ when it was invoked for the first time.
- $G\_cts_i$ (current timestamp): It is a time-stamp 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.
- $G\_wts_i$ (working timestamp): It is the time-stamp that $T_i$ works with. It is either greater than or equal to $T_i$'s G_cts.
- $G\_valid_i$: 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`.
- $G\_tltl$ (transaction lower time limit): It is G_cts of $T_i$ when transaction begins. It increases as the $T_i$ reads further values.
- $G\_tutl$ (transaction upper time limit): This field is a reducing value starting with $\infty$ when the $T_i$ is created. Suppose $T_i$ reads a version of tobj $x$. Then this field reduces as later versions of $x$ are created.

For each tobj $x$ in $\mathcal{T}$, we maintain:

- $x$.`vl` (version list): It is a list consisting of version tuples ($v\_tuple$) of the form $\langle \texttt{ts}, \texttt{rl}, \texttt{vrt} \rangle$. 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 time-stamp): It is the G_tltl value (which is same as G_tutl) of the transaction $T_i$ that created this version at the time of creation of this version.

Figure 8 illustrates the how the version list and read list are managed. For simplicity, we refer to a tuple $\langle j, v, rl, vu \rangle$ in $x$.`vl` as $x[j]$ and the corresponding elements as $x[j].v$ etc.

The STM system consists of the following methods: $init(), tbegin(), read(i, x), write_i(i, x, v), tryC(i)$.

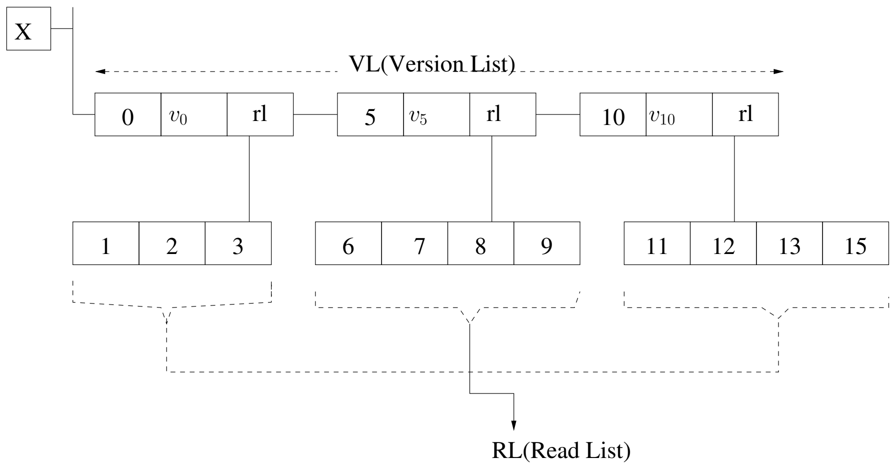



Figure 8: Data Structures for Maintaining Versions

**Algorithm 2** STM $init()$: Invoked at the start of the STM system. Initializes all the tobjs used by the STM System

$G\_tCntr$ = 1;
**for all** $x$ in $\mathcal{T}$ **do** // All the tobjs used by the STM System
  /* $T_0$ is creating the first version of $x$: $\texttt{ts} = 0, val = 0, \texttt{rl} = nil, \texttt{vrt} = 0$ */
  add $\langle 0, 0, nil, 0\rangle$ to $x.\texttt{vl}$;
**end for**;

**Algorithm 3** STM $tbegin(its)$: 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 $\langle id, G\_wts, G\_cts\rangle$

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

**Algorithm 4** STM $read(i, x)$: Invoked by a transaction $T_i$ to read tobj $x$. It returns either the value of $x$ or $\mathcal{A}$

**if** ($x \in rset_i$) **then** // Check if the tobj $x$ is in $rset_i$
  return $rset_i[x].val$;
**else if** ($x \in wset_i$) **then** // Check if the tobj $x$ is in $wset_i$
  return $wset_i[x].val$;
**else**// tobj $x$ is not in $rset_i$ and $wset_i$
  lock $x$; lock $G\_lock_i$;
  **if** ($G\_valid_i == F$) **then** return abort(i);
  **end if**
  /* findLTS: From $x.\texttt{vl}$, returns the largest $\texttt{ts}$ value less than $G\_wts_i$. If no such version exists, it returns $nil$ */
  $curVer = findLTS(G\_wts_i, x)$;
  **if** ($curVer == nil$) **then** return abort(i); // Proceed only if $curVer$ is not nil
  **end if**
  /* findSTL: From $x.\texttt{vl}$, returns the smallest $\texttt{ts}$ value greater than $G\_wts_i$. If no such version exists, it returns $nil$ */
  $nextVer = findSTL(G\_wts_i, x)$;
  **if** ($nextVer \neq nil$) **then**
    // Ensure that $G\_tutl_i$ remains smaller than $nextVer$'s $\texttt{vrt}$
    $G\_tutl_i = min(G\_tutl_i, x[nextVer].\texttt{vrt} - 1)$;
  **end if**
  // $G\_tltl_i$ should be greater than $x[curVer].\texttt{vrt}$
  $G\_tltl_i = max(G\_tltl_i, x[curVer].\texttt{vrt} + 1)$;
  **if** ($G\_tltl_i > G\_tutl_i$) **then** // If the limits have crossed each other, then $T_i$ is aborted
    return abort(i);
  **end if**
  $val = x[curVer].v$; add $\langle x, val \rangle$ to $rset_i$;
  add $T_i$ to $x[curVer].rl$;
  unlock $G\_lock_i$; unlock $x$;
  return $val$;
**end if**

**Algorithm 5** STM $write_i(x, val)$: A Transaction $T_i$ writes into local memory

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

**Algorithm 6** STM $tryC()$: Returns $ok$ on commit else return Abort

// The following check is an optimization which needs to be performed again later
lock $G\_lock_i$;
**if** $(G\_valid_i == F)$ **then** return abort(i);
**end if**
unlock $G\_lock_i$;
// Initialize smaller read list (smallRL), larger read list (largeRL), all read list (allRL) to nil
$smallRL = largeRL = allRL = nil$;
// Initialize previous version list (prevVL), next version list (nextVL) to nil
$prevVL = nextVL = nil$;
**for all** $x \in wset_i$ **do**
    lock $x$ in pre-defined order;
    /* findSTL: returns the version with the largest ts value less than $G\_wts_i$. If no such version exists, it returns $nil$. */
    $prevVer = findSTL(G\_wts_i, x)$; // prevVer: largest version smaller than $G\_wts_i$
    **if** $(prevVer == nil)$ **then** // There exists no version with ts value less than $G\_wts_i$
        lock $G\_lock_i$; return abort(i);
    **end if**
    $prevVL = prevVL \cup prevVer$; // Store the previous version in prevVL
    $allRL = allRL \cup x[prevVer].rl$; // Store the read-list of the previous version
    // **getLar**: obtain the list of reading transactions of $x[prevVer].rl$ whose $G\_wts$ is greater than $G\_wts_i$
    $largeRL = largeRL \cup getLar(G\_wts_i, x[prevVer].rl)$;
    // **getSm**: obtain the list of reading transactions of $x[prevVer].rl$ whose $G\_wts$ is smaller than $G\_wts_i$
    $smallRL = smallRL \cup getSm(G\_wts_i, x[prevVer].rl)$;
    /* findLTS: returns the version with the smallest ts value greater than $G\_wts_i$. If no such version exists, it returns $nil$. */
    $nextVer = findSTL(G\_wts_i, x)$; // prevVer: largest version smaller than $G\_wts_i$
    **if** $(nextVer \neq nil))$ **then**
        $nextVL = nextVL \cup nextVer$; // Store the next version in nextVL
    **end if**
**end for**// $x \in wset_i$
$relLL = allRL \cup T_i$; // Initialize relevant Lock List (relLL)
**for all** $(T_k \in relLL)$ **do**
    lock $G\_lock_k$ in pre-defined order; // Note: Since $T_i$ is also in $relLL$, $G\_lock_i$ is also locked
**end for**
// Verify if $G\_valid_i$ is false
**if** $(G\_valid_i == F)$ **then** return abort(i);
**end if**
$abortRL = nil$ // Initialize abort read list (abortRL)
// Among the transactions in $T_k$ in $largeRL$, either $T_k$ or $T_i$ has to be aborted
**for all** $(T_k \in largeRL)$ **do**
    **if** $(isAborted(T_k))$ **then**
        // Transaction $T_k$ can be ignored since it is already aborted or about to be aborted
        continue;
    **end if**
    **if** $(G\_its_i < G\_its_k) \wedge (G\_state_k == \texttt{live})$ **then**
        // Transaction $T_k$ has lower priority and is not yet committed. So it needs to be aborted
        $abortRL = abortRL \cup T_k$; // Store $T_k$ in abortRL
    **else**// Transaction $T_i$ has to be aborted
        return abort(i);
    **end if**
**end for**
// Ensure that $G\_tltl_i$ is greater than vrt of the versions in $prevVL$
**for all** $(ver \in prevVL)$ **do**
    $x$ = tobj of $ver$;
    $G\_tltl_i = max(G\_tltl_i, x[ver].\texttt{vrt} + 1)$;
**end for**

**Algorithm 7** STM $tryC()$: Continued

// Ensure that $\texttt{vutl}_i$ is less than $\texttt{vrt}$ of versions in $nextVL$<br>
**for all** $(ver \in nextVL)$ **do**<br>
    $x$ = tobj of $ver$;<br>
    $G\_tutl_i = min(G\_tutl_i, x[ver].\texttt{vrt} - 1)$;<br>
**end for**<br>
// Store the current value of the global counter as commit time and increment it<br>
$comTime = G\_tCntr.add\&Get(incrVal)$; // $incrVal$ can be constant $\geq 2$<br>
$G\_tutl_i = min(G\_tutl_i, comTime)$; // Ensure that $G\_tutl_i$ is less than or equal to $comTime$<br>
// Abort $T_i$ if its limits have crossed<br>
**if** $(G\_tltl_i > G\_tutl_i)$ **then** return abort(i);<br>
**end if**<br>
**for all** $(T_k \in smallRL)$ **do** // Iterate through $smallRL$ to see if $T_k$ or $T_i$ has to aborted<br>
    **if** $(isAborted(T_k))$ **then**<br>
        // Transaction $T_k$ can be ignored since it is already aborted or about to be aborted<br>
        continue;<br>
    **end if**<br>
    **if** $(G\_tltl_k \geq G\_tutl_i)$ **then** // Ensure that the limits do not cross for both $T_i$ & $T_k$<br>
        **if** $(G\_state_k == live)$ **then** // Check if $T_k$ is live<br>
            **if** $(G\_its_i < G\_its_k)$ **then**<br>
                // Transaction $T_k$ has lower priority and is not yet committed. So it needs to be aborted<br>
                $abortRL = abortRL \cup T_k$; // Store $T_k$ in abortRL<br>
            **else**// Transaction $T_i$ has to be aborted<br>
                return abort(i);<br>
            **end if**// $(G\_its_i < G\_its_k)$<br>
        **else**// ($T_k$ is committed. Hence, $T_i$ has to be aborted)<br>
            return abort(i);<br>
        **end if**// $(G\_state_k == live)$<br>
    **end if**// $(G\_tltl_k \geq G\_tutl_i)$<br>
**end for**$(T_k \in smallRL)$<br>
// After this point $T_i$ can't abort.<br>
$G\_tltl_i = G\_tutl_i$;<br>
**for all** $T_k \in abortRL$ **do** // Abort all the transactions in abortRL since $T_i$ can't abort<br>
    $G\_valid_k = F$;<br>
**end for**<br>
// Having completed all the checks, $T_i$ can be committed<br>
**for all** $(x \in wset_i)$ **do**<br>
    /* Create new v_tuple: $G\_wts, val, \texttt{rl}, \texttt{vrt}$ for $x$ */<br>
    $newTuple = \langle G\_wts_i, wset_i[x].val, nil, G\_tltl_i \rangle$; // $\texttt{vl} = G\_tltl_i$<br>
    **if** $(|x.vl| > k)$ **then**<br>
        replace the oldest tuple in $x.\texttt{vl}$ with $newTuple$; // $x.\texttt{vl}$ is ordered by $\texttt{ts}$<br>
    **else**<br>
        add a $newTuple$ to $x.vl$ in sorted order;<br>
    **end if**<br>
**end for**// $x \in wset_i$<br>
$G\_state_i$ = $\texttt{commit}$;<br>
unlock all variables;<br>
return $\mathcal{C}$;

**Algorithm 8** $isAborted(T_k)$: Verifies if $T_i$ is already aborted or its G_valid flag is set to false implying that $T_i$ will be aborted soon

```
1: if (G_valid_k == F) ∨ (G_state_k == abort) ∨ (T_k ∈ abortRL) then
2:     return T;
3: else
4:     return F;
5: end if
```

**Algorithm 9** $abort(i)$: Invoked by various STM methods to abort transaction $T_i$. It returns $\mathcal{A}$

```
1: G_valid_i = F; G_state_i = abort;
2: unlock all variables locked by T_i;
3: return A;
```

## 5.2 Proof of Liveness

**Proof Notations:** Let $gen(KSFTM)$ consist of all the histories accepted by KSFTM algorithm. In the follow sub-section, we only consider histories that are generated by KSFTM 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 KSFTM. Once it executes tbegin method, its ITS, CTS, WTS values 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 1 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 $H.incarCt(T_i)$

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

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

**Observation 4** *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,* $\langle H, T_i : (T_i \in H.txns) \wedge (H.incarCt(T_i)) \implies (T_i \in H.terminated)\rangle$.

**Observation 5** *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,* $\langle H1, H2, T_i, T_j : (H1 \sqsubseteq H2) \wedge (H1.incarCt(T_i)) \wedge (T_j \in H2.txns) \wedge (T_i \in H2.incarSet(T_j)) \implies (H2.incarCt(T_j))\rangle$.

**Lemma 6** *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,* $\langle H1, H2, T_i, T_j : (H1 \sqsubset H2) \wedge (H1.incarCt(T_i)) \wedge (T_j \in H2.txns) \wedge (T_j \notin H1.txns) \implies (H1.its_i \neq H2.its_j)\rangle$.

**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 ITSs 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 7** *Consider a history $H1$ generated by gen(KSFTM). Then there is a history $H2$ in gen(KSFTM) such that $H2$ is a strict extension of $H1$. Formally,* $\langle \forall H1 : (H1 \in gen(ksftm)) \implies (\exists H2 : (H2 \in gen(ksftm)) \wedge (H1 \sqsubset H2)\rangle$.

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

**Observation 8** *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 9** *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,* $\langle H1, T_i : (T_i \in H1.aborted) \implies (\exists T_j, H2 : (H1 \sqsubseteq H2) \wedge (T_j \in H2.live) \wedge (H2.its_i = H2.its_j) \wedge (H2.cts_i < H2.cts_j))\rangle$.

**Observation 10** *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,* $\langle H1, T_i : (T_i \in H1.committed) \implies (\nexists T_j, H2 : (H1 \sqsubseteq H2) \wedge (T_j \in H2.live) \wedge (H2.its_i = H2.its_j))\rangle$.

**Lemma 11** *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,* $\langle H1, H2, T_i, T_j : (H1 \sqsubset H2) \wedge (T_i \in H1.txns) \wedge (T_j \in H2.txns) \wedge (T_i \in H2.incarSet(T_j)) \wedge (H1.wts_i < H2.wts_j) \implies (H1.cts_i < H2.cts_j)\rangle$

**Proof.** Here we are given that

$$H1.wts_i < H2.wts_j \tag{1}$$

The definition of WTS of $T_i$ is: $H1.wts_i = H1.cts_i + C * (H1.cts_i - H1.its_i$. Substituting for $c$ to be 1, we get that $H1.wts_i = 2 * H1.cts_i - H1.its_i$. Combining this Eqn(1), we get that

$$2 * H1.cts_i - H1.its_i < 2 * H2.cts_j - H2.its_j \xrightarrow[H1.its_i = H2.its_j]{T_i \in H2.incarSet(T_j)} H1.cts_i < H2.cts_j. \quad \square$$

**Lemma 12** *Consider a live transaction $T_i$ in a history $H1$ with its $wts_i$ less than a constant $\alpha$. Then there is a strict extension of $H1$, $H2$ in which an incarnation of $T_i$, $T_j$ is live with WTS greater than $\alpha$. Formally,* $\langle H1, T_i : (T_i \in H1.live) \wedge (H1.wts_i < \alpha) \implies (\exists T_j, H2 : (H1 \sqsubseteq H2) \wedge (T_i \in H2.incarSet(T_j)) \wedge ((T_j \in H2.committed) \vee ((T_j \in H2.live) \wedge (H2.wts_j > \alpha))))\rangle$.

**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 $H2$.

On the other hand if $T_i$ is aborted, as seen in Observation 9 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 $\alpha$. Hence, we have that either an incarnation of $T_i$ will get committed or will eventually have WTS greater than or equal to $\alpha$. □

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

**Lemma 13** *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 tbegin and tryC. It can be seen that CTS of $T_i$ gets assigned in the tbegin method. So if the last method of $H$ is the tbegin 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 tbegin 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 14** *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,* $\langle T_i, H1, H2 : (H1 \sqsubset H2) \wedge (T_i \notin H1.txns) \wedge (T_i \in H2.txns) \implies (H2.cts_i > H1.sys\text{-}time)\rangle$.

Now, we have lemma about the methods of KSFTM completing in finite time.

**Lemma 15** *If all the locks are fair and the underlying system scheduler is fair then all the methods of KSFTM 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 tobjs 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 KSFTM will not deadlock.

It can also be seen that none of the methods have any unbounded while loops. All the loops in tryC method iterate through all the tobjs 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 16** *Every transaction either commits or aborts in finite time.*

**Proof.** This theorem comes directly from the Lemma 15. Since every method of KSFTM 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 17** *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 18** *Consider a history $H1$ with two transactions $T_i, T_j$. Let $T_i$ be in $H1.live$. Then for $T_j$, we have that* $\langle H, T_i, T_j : (\{T_i, T_j\} \subseteq H.txns) \wedge (T_i \in H.live) \wedge (H.wts_j \geq H.wts_i) \Longrightarrow (H.its_i + 2L \geq H.its_j)\rangle$.

**Proof.** Since $T_i$ is live in $H1$, from Corollary 17, 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 WTSs of both the transactions.
$wts_j - wts_i = (cts_j + C * (cts_j - its_j)) - (cts_i + C * (cts_i - its_i))$
$= (C+1)(cts_j - cts_i) - C(its_j - its_i)$
$\leq -(C+1)L - C(its_i - its_j) \qquad [\because cts_j \leq cts_i + L]$
$= C(its_j - its_i) - (C+1)L$
$= its_j - its_i - 2L \qquad [\because C = 1]$

Thus, we have that: $\langle (its_j - its_i - 2L) \geq (wts_j - wts_i) \rangle$. 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 KSFTM 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 1** *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,*

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

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

**Lemma 19** *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, $\langle H1, H2, T_i : (H1 \sqsubseteq H2) \land (T_i \in H1.live) \land (T_i \in H2.live) \land (H1.itsEnabled(T_i)) \implies (H2.itsEnabled(T_i)) \rangle$.*

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 2.

**Lemma 20** *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, $\langle H, \{T_i, T_j\} \in H.txns : (tryC_i <_H term\text{-}op_j) \implies (comTime_i + incrVal \leq comTime_j) \rangle$.*

**Proof.** When $T_i$ commits, let the value of the global $tCntr$ be $\alpha$. It can be seen that in tbegin method, $comTime_j$ get initialized to $\infty$. The only place where $comTime_j$ gets modified is at Line 61 of tryC. Thus if $T_j$ gets aborted before executing tryC method or before this line of tryC we have that $comTime_j$ remains at $\infty$. Hence in this case we have that $\langle comTime_i + incrVal < comTime_j \rangle$.

If $T_j$ terminates after executing Line 61 of tryC method then $comTime_j$ is assigned a value, say $\beta$. It can be seen that $\beta$ will be greater than $\alpha$ 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 21** *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 tbegin method, $comTime_i$ get initialized to $\infty$. The only place where $comTime_i$ gets modified is at Line 61 of 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 61, $comTime_i$ 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 5 of tbegin method. In that line, $cts_i$ reads $tCntr$ and increments it atomically. Then in Line 61, $comTime_i$ 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 20 of read, Line 53 and Line 85 of 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 20 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,
$$tltl_i = 1 + x[j].\texttt{vrt} \tag{2}$$
It can be seen that $x[j].\texttt{vrt}$ is same as $tltl_j$ when $T_j$ executed Line 92 of tryC. Further, $tltl_j$ in turn is same as $tutl_j$ due to Line 85 of tryC. From Line 62, it can be seen that $tutl_j$ is less than or equal to $comTime_j$ when $T_j$ committed. Thus we have that
$$x[j].\texttt{vrt} = tltl_j = tutl_j \leq comTime_j \tag{3}$$
It is clear that from the above discussion that $T_j$ executed tryC method before $T_i$ terminated (i.e. $tryC_j <_{H1}$ $term\text{-}op_i$). From Eqn(2) and Eqn(3), we get
$tltl_i \leq 1 + comTime_j < 2 + comTime_j \xrightarrow{incrVal \geq 2} tltl_i < incrVal + comTime_j \xrightarrow{Lemma\ 20} tltl_i < comTime_i$

2. Line 53 of tryC method: The reasoning in this case is very similar to the above case.

3. Line 85 of tryC method: In this line, $tltl_i$ is made equal to $tutl_i$. Further, in Line 62, $tutl_i$ 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 22** *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,* $\langle H \in gen(KSFTM), \{T_i\} \in H.txns : (H.tutl_i < H.comTime_i) \implies (\exists T_j \in H.committed : H.wts_j > H.wts_i) \rangle$.

**Proof.** It can be seen that $G\_tutl_i$ initialized in tbegin method to $\infty$. $tutl_i$ is updated in Line 17 of read method, Line 58 & Line 62 of tryC method. If $T_i$ executes Line 17 of read method and/or Line 58 of tryC method then $tutl_i$ gets decremented to some value less than $\infty$, say $\alpha$. Further, it can be seen that in both these lines the value of $tutl_i$ is possibly decremented from $\infty$ 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 $\alpha$. 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 tryC, we have that $x[nextVer].vrt = tltl_j$ (from Line 92). From Lemma 21, 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 20, we get that $comTime_j < comTime_i$. Hence, we have that,
$$\alpha < comTime_i \tag{4}$$

Now let us consider Line 62 executed by $T_i$ which causes $tutl_i$ to change. This line will get executed only after both Line 17 of read method, Line 58 of tryC method. This is because every transaction executes tryC method only after read method. Further within tryC method, Line 62 follows Line 58.

There are two sub-cases depending on the value of $tutl_i$ before the execution of Line 62: (i) If $tutl_i$ was $\infty$ and then get decremented to $comTime_i$ upon executing this line, then we get $comTime_i = tutl_i$. Thus, we can ignore this case. (ii) Suppose the value of $tutl_i$ before executing Line 62 was $\alpha$. Then from Eqn(4) we get that $tutl_i$ remains at $\alpha$. 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 23** *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, $\langle H1, H2 \in gen(KSFTM), \{T_i, T_j\} \subseteq H1, H2.txns : (H1 \sqsubseteq H2) \wedge (T_j \in H1.committed) \wedge (T_i \in H1.live) \implies (H2.tltl_j < H2.comTime_i)\rangle$.*

**Proof.** As observed in the previous proof of Lemma 21, if $T_i$ is live or aborted in $H2$, then its comTime is $\infty$. 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 $\infty$. In this case, it can be seen that $T_j$ committed before $T_i$. Hence, we have that $comTime_j < comTime_i$. From Lemma 21, 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 24** *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.** Here $T_i$ is itsEnabled in $H1$. Since it is live in $H2$, from Lemma 19, we get that $T_i$ is itsEnabled in $H2$ as well. Note that $H2$ could be same as $H1$ as well.

To show this lemma, w.l.o.g we assume that $T_i$ on executing either read or tryC in $H2$ gets aborted resulting in $H3$. 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 tryC method.

1. STM tryC:

   (a) Line 3 : This line invokes abort() method on $T_i$ which releases all the locks and returns $\mathcal{A}$ 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 tryC algorithm. This can occur in Lines: 45, 75 where $T_i$ is added to $T_j$'s abortRL set. Later in Line 87, $T_i$'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 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 15: 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 34 : This case is similar to Case 1a, i.e., Line 3.

   (d) Line 47 : 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 64 : 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 tbegin method to $\infty$. As long as it is $\infty$, these limits cannot cross each other. Later, $tutl_i$ is updated in Line 17 of read method, Line 58 & Line 62 of tryC method. Suppose $tutl_i$ gets decremented to some value $\alpha$ by one of these lines.

   Now there are two cases here: (1) Suppose $tutl_i$ gets decremented to $comTime_i$ due to Line 62 of tryC method. Then from Lemma 21, we have $tltl_i \leq comTime_i = tutl_i$. Thus in this case, $T_i$ will not abort. (2) $tutl_i$ gets decremented to $\alpha$ which is less than $comTime_i$. Then from Lemma 22, 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 77: This case is similar to Case 1a, i.e., Line 3.

(g) Line 80 : In this case, $T_k$ is in $T_i$'s smallRL and is committed in $H1$. And, from this we have that

$$H2.tutl_i \leq H2.tltl_k \tag{5}$$

From the assumption of this case, we have that $T_k$ commits before $T_i$. Thus, from Lemma 23, we get that $comTime_k < comTime_i$. From Lemma 21, we have that $tltl_k < comTime_k$. Thus, we get that $tltl_k < comTime_i$. Combining this with the inequality of this case Eqn(5), we get that $tutl_i < comTime_i$.

Combining this inequality with Lemma 22, 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 7: This case is similar to Case 1a, i.e., Line 3

   (b) Line 22: The reasoning here is similar to Case 1e, i.e., Line 64.

□

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 notion, we 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 KSFTM algorithm and Lemma 18, 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 $T_i$ + $2 * L$, $T_i$'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 have their incarCt 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 2** *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 25** *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 $\phi$ 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 26** *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

$$H1.cts_i \geq H1.its_i + 2 * L \tag{6}$$

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

$$H2.its_k < H2.its_j + 2 * L \tag{7}$$

Since $T_i$ and $T_j$ are incarnations of each other, their ITS are the same. Combining this with Eqn(7), we get that $H2.its_k < H1.its_i + 2 * L$.

$$H2.its_k < H1.its_i + 2 * L \tag{8}$$

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 14 we get that

  $H2.cts_k > H1.sys\text{-}time \xrightarrow[H2.cts_k = H2.its_k]{T_k \text{ starts afresh}} H2.its_k > H1.sys\text{-}time \xrightarrow[H1.sys\text{-}time \geq H1.cts_i]{(T_i \in H1) \wedge Lemma\ 13} H2.its_k > H1.cts_i \xrightarrow{Eqn(6)} H2.its_k > H1.its_i + 2 * L \xrightarrow{H1.its_i = H2.its_j} H2.its_k > H2.its_j + 2 * L$

  But this result contradicts with Eqn(7). Hence, this case is not possible.

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

  $$H1.its_l = H2.its_k \tag{9}$$

  Now combing this result with Eqn(8), we get that $H1.its_l < H1.its_i + 2 * L$. This implies that $T_l$ is in affectSet of $T_i$. 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, $(H1 \sqsubset H2) \wedge (H1.incarCt(T_l)) \wedge (T_k \in H2.txns) \wedge (T_k \notin H1.txns)$. Combining these with Lemma 6, we get that $(H1.its_l \neq H2.its_k)$. But this result contradicts Eqn(9). Hence, this case is also not possible.

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

**Lemma 27** *Consider two histories $H1, H2$ $H2$ 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,* $\langle H1, H2, T_i, T_j, T_k : (H1 \sqsubseteq H2) \wedge (T_i \in H1.txns) \wedge (\{T_j, T_k\} \in H2.txns) \wedge (H1.cts_i > H1.its_i + 2 * L) \wedge (T_i \in H2.incarSet(T_j)) \wedge (T_k \in H2.affectSet(T_j)) \implies (\exists T_l : (T_l \in H2.incarSet(T_k)) \wedge (T_l \in H1.txns))\rangle$

**Proof.**
This proof is similar to the proof of Lemma 26. We are given that

$$H1.cts_i \geq H1.its_i + 2 * L \tag{10}$$

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 after $H1$. Thus, we have that

$H3.its_k > H1.sys\text{-}time \xrightarrow{Lemma\ 13} H3.its_k > H1.cts_i \xrightarrow{Eqn(10)} H3.its_k > H1.its_i + 2 * L \xrightarrow{H1.its_i = H2.its_j} H3.its_k > H2.its_j + 2 * L \xrightarrow[definition]{affectSet} T_k \notin H2.affectSet(T_j)$

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 28** *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, $\langle H1, H2, T_i, T_j : (H1 \sqsubseteq H2) \wedge (H1.cdsEnabled(T_i)) \wedge (T_j \in H2.txns) \wedge (T_i \in H2.incarSet(T_j)) \implies ((H1.affectSet(T_i) = H2.affectSet(T_j))) \rangle$*

**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

$$T_k \in H1.txns \tag{11}$$

Combining Eqn(11) with Observation 8, we get that

$$T_k \in H2.txns \tag{12}$$

From the definition of ITS, we get that

$$H1.its_k = H2.its_k \tag{13}$$

Since $T_i, T_j$ are incarnations we have that .

$$H1.its_i = H2.its_j \tag{14}$$

From the definition of affectSet, we get that,

$H1.its_k < H1.its_i + 2 * L \xrightarrow{Eqn(13)} H2.its_k < H1.its_i + 2 * L \xrightarrow{Eqn(14)} H2.its_k < H2.its_j + 2 * L$

Combining this result with Eqn(12), 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 26, we get that $T_k \in H1.txns$. Now, this case is similar to the above case. It can be seen that Equations 11, 12, 13, 14 hold good in this case as well.

Since $T_k$ is in $H2.affectSet(T_j)$, we get that

$H2.its_k < H2.its_i + 2 * L \xrightarrow{Eqn(13)} H1.its_k < H2.its_j + 2 * L \xrightarrow{Eqn(14)} H1.its_k < H1.its_i + 2 * L$

Combining this result with Eqn(11), 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 29** *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

$$H1.cts_i \geq H2.its_i + 2 * L \tag{15}$$

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

$$\begin{aligned} H2.cts_j &> H1.cts_i \\ &> H1.its_i + 2 * L && \text{[From Eqn(15)]} \\ &> H2.its_j + 2 * L && [its_i = its_j] \end{aligned}$$

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 28, 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., $H1.cds(T_i)$ is true, we get that $H1.incarSet(T_k)$ is true. Combining this with Observation 5, we get that $T_k$ must have its incarCt as true in $H2$ as well, i.e. $H2.incarSet(T_k)$. 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 30** *For any transaction $T_i$ in $H$, we have that $wts_i$ is less than or 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 31** *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

$$H.affWTS(T_i) = max\{H.maxWTS(T_k) | (T_k \in H1.affectSet(T_i))\} \tag{16}$$

$$H.affWTS(T_j) = max\{H.maxWTS(T_l) | (T_l \in H2.affectSet(T_j))\} \tag{17}$$

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 (16) & (17), we get that

$$H1.maxWTS(T_p) \geq H2.maxWTS(T_q) \tag{18}$$

$$H2.maxWTS(T_q) \geq H1.maxWTS(T_p) \tag{19}$$

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 3** *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 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 $T_i$ 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,

$$\begin{split}\langle(H1 \sqsubseteq H2) \land (T_i \in H1.live) \land (T_j \in H2.live) \land (T_i \in H2.incarSet(T_j)) \land (H1.cts_i < H2.cts_j) \\ \land(H1.cdsEnabled(T_i))\rangle\end{split} \tag{20}$$

Combining Eqn(20) with Lemma 29, 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 $T_i$ is live in $H2$ which in turn implies that $T_i$ is in $H1.txns$. Thus changing this in Eqn(20), we get the following

$$\begin{split}\langle(H1 \sqsubseteq H2) \land (T_j \in H2.txns) \land (T_i \in H2.incarSet(T_j)) \land (H1.cts_i < H2.cts_j) \\ \land(H1.cdsEnabled(T_i))\rangle\end{split} \tag{21}$$

Combining Eqn(21) with Lemma 28 we get that

$$H1.affWTS(T_i) = H2.affWTS(T_j) \tag{22}$$

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

$$H1.wts_i < H2.wts_j \tag{23}$$

Since $T_i$ is finEnabled in $H1$, we have that
$H1.wts_i > H1.affWTS(T_i) \xrightarrow{Eqn(23)} H2.wts_j > H1.affWTS(T_i) \xrightarrow{Eqn(22)} H2.wts_j > H2.affWTS(T_j)$

□

Now, we show that a transaction that is finEnabled will eventually commit.

**Lemma 33** *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, $\langle H1, T_i : (T_i \in H1.live) \land (H1.valid_i) \land (H1.finEnabled(T_i)) \implies (\exists H3 : (H1 \sqsubset H3) \land (T_i \in H3.committed))\rangle$.*

**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 25, we get that $T_i$ is itsEnabled in $H1$. Thus from Lemma 24, 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,

$$\langle(\exists H2, T_j : (H1 \sqsubseteq H2 \sqsubset H3) \wedge (T_i \in H2.live) \wedge (T_j \in H2.txns) \wedge (H2.wts_i < H2.wts_j) \\ \wedge (T_j \in H3.committed))\rangle \tag{24}$$

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 32, we get that $T_i$ is finEnabled in $H2$ as well. Now, let us consider $T_j$ in $H2$. From Eqn(24), we get that $(H2.wts_i < H2.wts_j)$. Combining this with the observation that $T_i$ being live in $H2$, Lemma 18 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

$$(H2.affWTS(T_i) \geq H2.maxWTS(T_j)) \tag{25}$$

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

$$(H2.wts_i > H2.affWTS(T_i)) \tag{26}$$

Now combining Equations 25, 26 we get,

$$\begin{aligned} H2.wts_i &> H2.affWTS(T_i) \geq H2.maxWTS(T_j) \\ &> H2.affWTS(T_i) \geq H2.maxWTS(T_j) \geq H2.wts_j \qquad \text{[From Observation 30]} \\ &> H2.wts_j \end{aligned}$$

But this equation contradicts with Eqn(24). 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 ITSs 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 34** *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 14 we get the following

$H3.its_k > H1.sys\text{-}time \xrightarrow{Lemma\ 13} H3.its_k > H1.cts_i \implies H3.its_k > H1.its_i + 2 * L \xrightarrow{H1.its_i = H2.its_j} H3.its_k > H2.its_j + 2 * L \xrightarrow[definitions]{affectSet, depIts} H2.its_k \notin H2.depIts(T_j)$

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 35** *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 itsEnabled 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

$$|H1.depIts(T_i)| = k + 1 \tag{27}$$

Let $\alpha$ be $H1.partAffWTS(T_i)$. Suppose $H1.wts_i < \alpha$. Then from Lemma 12, 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 $\alpha$. 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 24, 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 21 we get that in $H4$,

$$H4.its_m \leq H4.its_l + 2*L \tag{28}$$

Eqn(28) 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 27, we get that there exists an incarnation of $T_m$, $T_n$ in $H1$.

Combining Eqn(28) 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 Eqn(27), we have that $H1.depIts(T_i)$ is $k+1$. From Lemma 34, 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

$$|H4.depIts(T_i)| \leq k \implies |H4.depIts(T_l)| \leq k \tag{29}$$

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

$$H4.cts_l > H4.cts_i \xrightarrow{given} H4.cts_l > H4.its_i + 2 * L \xrightarrow{H4.its_i = H4.its_l} H4.cts_l > H4.its_l + 2 * L \tag{30}$$

We also have that $T_l$ is live in $H4$. Combining this with Equations 29, 30 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 36** *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 $\alpha$. From Lemma 12, 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 $\alpha$.
In the second case, we get that

$$(T_i \in H1.live) \wedge (T_j \in H2.live) \wedge (H.cdsEnabled(T_i)) \wedge (T_j \in H2.incarSet(T_i)) \wedge \\ (H1.wts_i < H2.wts_j) \tag{31}$$

Combining the above result with Lemma 11, we get that $H1.cts_i < H2.cts_j$. Thus the modified equation is

$$(T_i \in H1.live) \wedge (T_j \in H2.live) \wedge (H1.cdsEnabled(T_i)) \wedge (T_j \in H2.incarSet(T_i)) \wedge \\ (H1.cts_i < H2.cts_j) \tag{32}$$

Next combining Eqn(32) with Lemma 28, we get that

$$H1.affectSet(T_i) = H2.affectSet(T_j) \tag{33}$$

Similarly, combining Eqn(32) with Lemma 29 we get that $T_j$ is cdsEnabled in $H2$ as well. Formally,

$$H2.cdsEnabled(T_j) \tag{34}$$

Now combining Eqn(33) with Lemma 31, we get that

$$H1.affWTS(T_i) = H2.affWTS(T_j) \tag{35}$$

From our initial assumption we have that $H1.affWTS(T_i)$ is $\alpha$. From Eqn(35), we get that $H2.affWTS(T_j) = \alpha$. Further, we had earlier also seen that $H2.wts_j$ is greater than $\alpha$. Hence, we have that $H2.wts_j > H2.affWTS(T_j)$. Combining the above result with Eqn(34), $H2.cdsEnabled(T_j)$, we get that $T_j$ is finEnabled, i.e., $H2.finEnabled(T_j)$. □

Next, we show that every live transaction eventually become itsEnabled.

**Lemma 37** *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.** There are two cases: (1) Either incarnation $T_i, T_j$ is committed in $H2$. This trivially proves the lemma; (2) Otherwise, $T_j$ is $itsEnabled$.
Induction on ITS: There are n live transactions in H1 then either $T_i$ or incarnation $T_i, T_j$ is $itsEnabled$ in H2.
**Base case:** Consider only one live transaction $T_i$ in $H1$. So from the definition of $itsEnabled$, either $T_i$ or incarnation $T_i, T_j$ is $itsEnabled$ in H2.

**Induction hypothesis:** The induction statement holds n transactions.

**Inductive step:** Now, we need to prove that The induction statement holds for (n+1) transactions.
In order to prove this, we need to show when the live transaction $T_n$ will commit. From induction hypothesis, $T_n$ is $itsEnabled$ in H2. From Lemma 35,
$H1, T_n : (T_n \in H1.live) \wedge (H1.itsEnabled(T_n)) \implies (\exists H2, T_{n'} : (H1 \sqsubset H2) \wedge (T_{n'} \in H2.incarSet(T_n)) \wedge ((T_{n'} \in H2.committed) \vee (H2.cdsEnabled(T'_n))))$.
Now, from Lemma 36,
$H1, T_n : (T_n \in H1.live) \wedge (H1.cdsEnabled(T_n)) \implies (\exists H2, T_{n'} : (H1 \sqsubset H2) \wedge (T_{n'} \in H2.incarSet(T_n)) \wedge ((T_{n'} \in H2.committed) \vee (H2.finEnabled(T'_n)))$.
From Lemma 33,
$H1, T_n : (T_n \in H1.live) \wedge (H1.valid_n) \wedge (H1.finEnabled(T_n)) \implies (\exists H2 : (H1 \sqsubset H2) \wedge (T_n \in H2.committed))$. Hence, $T_n$ returns commit in H2. Therefore, $T_{n+1}$ is becomes itsEnabled in H2.

So, $T_j$ is either committed or itsEnabled in H2.
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 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 $T_i$, $T_j$ 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.** As transaction $T_i$ is live in H1. So from Lemma **??**,
$H1, T_i : (T_i \in H.live) \implies (\exists T_j, H2 : (H1 \sqsubset H2) \wedge (T_j \in H2.incarSet(T_i)) \wedge (T_j \in H2.committed) \vee (H.itsEnabled(T_i)))$.
i.e. Either $T_i$ or incarnation $T_i, T_j$ is either committed or itsEnabled in H2.

Now from Lemma 35, $H1, T_i : (T_i \in H.live) \wedge (H.itsEnabled(T_i)) \implies (\exists H2, T_j : (H1 \sqsubset H2) \wedge (T_j \in H2.incarSet(T_i)) \wedge ((T_j \in H2.committed) \vee (H2.cdsEnabled(T_j))))$.
i.e. Either $T_i$ or incarnation $T_i, T_j$ is either committed or cdsEnabled in H2.
So from Lemma 36,
$H1, T_i : (T_i \in H.live) \wedge (H1.cdsEnabled(T_i)) \implies (\exists H2, T_j : (H1 \sqsubset H2) \wedge (T_j \in H2.incarSet(T_i)) \wedge ((T_j \in H2.committed) \vee (H2.finEnabled(T_j))))$.
i.e. Either $T_i$ or incarnation $T_i, T_j$ is either committed or finEnabled in H2.
Now from Lemma 33,
$H1, T_i : (T_i \in H1.live) \wedge (H1.valid_i) \wedge (H1.finEnabled(T_i)) \implies (\exists H3 : (H1 \sqsubset H3) \wedge (T_i \in H3.committed))$.

Hence, incarnation of $T_i, T_j$ is committed in H2.

# 6 Proof of safety

**Lemma 39** *Consider a history $H$ in $gen(KSFTM)$ with two transactions $T_i$ and $T_j$ such that both their G_valid flags are true. 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 MVSG.

1. Real-time edge: Since, transaction $T_i$ and $T_j$ are in real time order so $comTime_i < G\_cts_j$. As we know from Lemma 21 ($G\_tltl_i \leq comTime_i$). So, ($G\_tltl_i \leq CTS_j$).

We know from STM $tbegin(its)$ 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 92 $tryC(T_i)$, $G\_tltl_i = \texttt{vrt}_i$.
and from Line 20 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 $w_j(x_j)r_k(x_j)w_i(x_i)$ in which there are two possibilities of version order:

   (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 53 of $tryC(T_j)$, $\texttt{vrt}_i < G\_tltl(j)$.
      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 58 of $tryC(T_i)$, $G\_tutl_i < \texttt{vrt}_j$. As we have assumed $G\_valid_i$ is true so definitely it will execute the Line 85 $tryC(T_i)$ 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 **??** of $tryC(T_i)$, either transaction $T_i$ or $T_k$ $G\_valid$ flag is set to be false. If $T_i$ returns abort then this case will not be considered in Lemma 39. Otherwise, as $T_j$ has already been committed and later $T_i$ will execute the Line 92 $tryC(T_i)$, 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 17 of $read(k, x)$, $G\_tutl_k < \texttt{vrt}_i$ and from Line 20 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 92 $tryC(T_i)$, $G\_tltl_i = \texttt{vrt}_i$.
         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 58 of $tryC(T_j)$, $G\_tutl_j < \texttt{vrt}_i$ i.e. $G\_tutl_j < G\_tltl_i$. Here, $T_k$ read from $T_j$. So, From Line 17 of $read(k, x)$, $G\_tutl_k < \texttt{vrt}_i \rightarrow G\_tutl_k < G\_tltl_i$ from Line 20 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 92 $tryC(T_j)$, $G\_tltl_j = \texttt{vrt}_j$.
      Hence, $G\_tltl_j < G\_tltl_k < G\_tutl_k < G\_tltl_i$.
      Therefore, $G\_tltl_j < G\_tltl_k < G\_tltl_i$.

**Lemma 40** *Consider a transaction $T_i$ in history $H$ $gen(KSFTM)$, if all the methods of $T_i$ returns successful then the $G\_valid_i$ will definitely be true.*

**Proof.** There are two method w.r.t. transaction $T_i$ return successful:

1. read(i,x): Here again there are two possibilities for $G\_valid_i$ set to be false:

   (a) $G\_valid_i$ set to be false by other transactions: Due to this $T_i$ will terminate at Line 7 of read(i,x) method so, $T_i$ will never execute Line **??** of read(i,x) method. hence, read(i,x) method of $T_i$ returns successful then the $G\_valid_i$ will definitely be true.

(b) $G\_valid_i$ set to be false by itself: Due to this $T_i$ will terminate at Line **??** and Line 22 of read(i,x) method so, $T_i$ will never execute Line **??** of read(i,x) method. hence, read(i,x) method of $T_i$ returns successful then the $G\_valid_i$ will definitely be true.

2. tryC(): Here again there are two possibilities for $G\_valid_i$ set to be false:

   (a) $G\_valid_i$ set to be false by other transactions: Due to this $T_i$ will terminate at Line 3 and Line 34 of tryC() method so, $T_i$ will never execute Line **??** of tryC() method. hence, tryC() method of $T_i$ returns successful then the $G\_valid_i$ will definitely be true.

   (b) $G\_valid_i$ set to be false by itself: Due to this $T_i$ will terminate at Line 15, Line 47 and Line 80 of tryC() method so, $T_i$ will never execute Line **??** of read(i,x) method. hence, tryC() method of $T_i$ returns successful then the $G\_valid_i$ will definitely be true.

**Theorem 41** *Any history H gen(KSFTM) is local opaque iff for a given version order $\ll$ H, MVSG(H,$\ll$) is acyclic.*

**Proof.** We are proving it by contradiction, so Assuming MVSG(H,$\ll$) has cycle. From Lemma 39, 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, MVSG(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, MVSG(H,$\ll$) produced by KSFTM 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 $tryC_j()$: If they are accessing common t-objects then, let say $read_i(x)$ acquires the lock on x before $tryC_j()$ 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 $tryC_j()$ 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. $tryC_i()$ and $tryC_j()$: If they are accessing common t-objects then, let say $tryC_i()$ acquires the lock on x before $tryC_j()$ 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 $tryC_i()$ acquires the lock on $relLL_i$ before $tryC_j()$ 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. $tbegin_i$ and $tbegin_j$: Both the $tbegin$ are accessing shared counter variable $G\_tCntr$. If $tbegin_i$ executes $G\_tCntr.get\&Inc()$ before $tbegin_j$ then the *M_Order$_H$* is $op_i \rightarrow op_j$.

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

*Linearization:* 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 (OPG):* 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 42** *All the locks in history H ($L\_Order_H$) gen(KSFTM) follows strict partial order. So, operation graph (OPG(H)) is acyclic. If ($op_i \rightarrow op_j$) in OPG, then atleast one of them will definitely true: ($Fpu_i(\alpha) < Lpl\_op_j(\alpha)$) $\cup$ ($access.G\_tCntr_i < access.G\_tCntr_j$) $\cup$ ($Fpu\_op_i(\alpha) < access.G\_tCntr_j$) $\cup$ ($access.G\_tCntr_i < Lpl\_op_j(\alpha)$). Here, $\alpha$ 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 OPG(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, OPG(H, $M\_Order$) produced by KSFTM is acyclic.

**Lemma 43** *Any history H gen(KSFTM) with $\alpha$ linearization such that it respects $M\_Order_H$ then (H, $\alpha$) 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 **??**, it returns the largest `ts` value less than $G\_wts_i$ that has already been committed and return the value successfully from Line **??**. 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, $\alpha$) is valid.

**Lemma 44** *Any history H gen(KSFTM) with $\alpha$ and $\beta$ 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(KSFTM) such that two transactions $T_i$ and $T_j$ are in real time order which respects $M\_Order_H$ i.e. $tryC_i < tbegin_j$. As $\alpha$ and $\beta$ are linearizations of H so, $tryC_i <_{(H,\alpha)} tbegin_j$ and $tryC_i <_{(H,\beta)} tbegin_j$. 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 45** *Any history H gen(KSFTM) with $\alpha$ and $\beta$ linearization such that both respects $M\_Order_H$ i.e. $M\_Order_H \subseteq \alpha$ and $M\_Order_H \subseteq \beta$ then $(H, \alpha)$ is local opaque iff $(H, \beta)$ is local opaque.*

**Proof.** As $\alpha$ and $\beta$ are linearizations of history H gen(KSFTM) so, from Lemma 43 (H, $\alpha$) and (H, $\beta$) are valid histories.

Now assuming (H, $\alpha$) is local opaque so we need to show (H, $\beta$) is also local opaque. Since (H, $\alpha$) 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 ($\overline{H}$, $\alpha$). As we know $\beta$ is a linearization of H so ($\overline{H}$, $\beta$) 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 44, $\prec^{RT}_{(H,\alpha)} = \prec^{RT}_{(H,\beta)}$ that implies $\prec^{RT}_{(H,\beta)} \subseteq \prec^{RT}_{S}$. Hence, $(H, \beta)$ is local opaque.

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

Hence, $(H, \alpha)$ is local opaque iff $(H, \beta)$ is local opaque.

**Lemma 46** *Any history H gen(KSFTM) is deadlock-free.*

**Proof.** In our algorithm, each transaction $T_i$ is following lock order in every method ($read(x, i)$ and $tryc()$) that are locking t-object first then $G\_lock$.

Since transaction $T_i$ is acquiring locks on t-objects in predefined order at Line **??** of $tryC()$ and it is also following predefined locking order of all conflicting $G\_lock$ including itself at Line **??** of $tryC()$.

Hence, history H gen(KSFTM) is deadlock-free.

# 7 Discussion and Conclusion

Software Transactional Memory systems (*STMs*) have garnered significant interest as an elegant alternative for addressing synchronization and concurrency issues in multi-core systems. In order to be efficient, STMs must guarantee some progress properties. In this paper, we explored the notion of starvation-freedom [13, chap 2] for TM systems. Gramoli et.al has proposed starvation-freedom for TM$^2$C systems by implementing FairCM contention manager [7].

We presented a starvation-free STM system, SV-SFTM using single versions. It is based on FOCC, a popular algorithm in databases. SV-SFTM satisfies opacity and ensures starvation-freedom. It assures any transaction with lowest $G\_its$ will definitely commit and abort all conflicting transactions.

It was observed that more read operations succeed by keeping multiple versions of each object [15, 18]. Since SV-SFTM does not consider multiple versions, we observed that it is possible that a slow running old transaction can cause several newer transactions to abort while ensuring starvation-freedom. To address this issue, we proposed KSTM, a MVSTM that maintains fixed number of versions.

But, KSTM does not guarantee starvation-freedom. By understanding the cases where KSTM fails to provide starvation-freedom, So, we develop a Multi-Version Starvation Free STM System, *KSFTM* that guarantees starvation-freedom of transactions. The key observation in working of KSFTM is that a transaction with lowest $G\_its$ and highest $G\_wts$ will definitely commit.

# References


[1] Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. Safety of Live Transactions in Transactional Memory: TMS is Necessary and Sufficient. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, pages 376–390, 2014.

[2] Hagit Attiya and Eshcar Hillel. A Single-Version STM that is Multi-Versioned Permissive. *Theory Comput. Syst.*, 51(4):425–446, 2012.

[3] Philip A. Bernstein and Nathan Goodman. Multiversion Concurrency Control: Theory and Algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, December 1983.

[4] Joao Cachopo and Antonio Rito-Silva. Versioned boxes as the basis for memory transactions. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, oct 2005.

[5] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards Formally Specifying and Verifying Transactional Memory. In *REFINE*, 2009.

[6] Sérgio Miguel Fernandes and Joao Cachopo. Lock-free and Scalable Multi-version Software Transactional Memory. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 179–188, New York, NY, USA, 2011. ACM.

[7] Vincent Gramoli, Rachid Guerraoui, and Vasileios Trigonakis. Tm2c: A software transactional memory for many-cores. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 351–364, New York, NY, USA, 2012. ACM.

[8] Rachid Guerraoui, Thomas Henzinger, and Vasu Singh. Permissiveness in Transactional Memories. In *DISC '08: Proc. 22nd International Symposium on Distributed Computing*, pages 305–319, sep 2008. Springer-Verlag Lecture Notes in Computer Science volume 5218.

[9] Rachid Guerraoui and Michal Kapalka. On the Correctness of Transactional Memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, New York, NY, USA, 2008. ACM.

[10] Rachid Guerraoui and Michal Kapalka. *Principles of Transactional Memory, Synthesis Lectures on Distributed Computing Theory*. Morgan and Claypool, 2010.

[11] Maurice Herlihy. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[12] Maurice Herlihy and J. Eliot B.Moss. Transactional memory: Architectural Support for Lock-Free Data Structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[13] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[14] Damien Imbs, José Ramon de Mendivil, and Michel Raynal. Brief announcement: virtual world consistency: a new condition for STM systems. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 280–281, New York, NY, USA, 2009. ACM.

[15] Priyanka Kumar, Sathya Peri, and K. Vidyasankar. A TimeStamp Based Multi-version STM Algorithm. In *ICDCN*, pages 212–226, 2014.

[16] Petr Kuznetsov and Sathya Peri. Non-interference and Local Correctness in Transactional Memory. In *ICDCN*, pages 197–211, 2014.

[17] Petr Kuznetsov and Srivatsan Ravi. On the cost of concurrency in transactional memory. In *OPODIS*, pages 112–127, 2011.

[18] Li Lu and Michael L. Scott. Generic multiversion STM. In *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, pages 134–148, 2013.

[19] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[20] Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. SMV: Selective Multi-Versioning STM. In *DISC*, pages 125–140, 2011.

[21] Nir Shavit and Dan Touitou. Software Transactional Memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

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

# Appendices

## A PCode of SFTM

**Data Structure:** 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 $v$ 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 tobj 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). We name all the shared variable starting 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 time-stamps:

- $G\_lock_i$: A lock for accessing all the shared variables of $T_i$.
- $G\_its_i$ (initial timestamp): It is a time-stamp assigned to $T_i$ when it was invoked for the first time.
- $G\_cts_i$ (current timestamp): It is a time-stamp 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. 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`.

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

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

**Algorithm 10** STM $init()$: Invoked at the start of the STM system. Initializes all the data items used by the STM System

$G\_tCntr$ = 1;
**for all** data item $x$ used by the STM System **do**
    add $\langle 0, nil\rangle$ to $x.val$;// $T_0$ is initializing $x$
**end for**;

**Algorithm 11** STM $tbegin(its)$: 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 $\langle id, G\_cts\rangle$

$i$ = unique-id; // An unique id to identify this transaction. It could be same as G_cts
**if** ($its == nil$) **then**
    $G\_its_i = G\_cts_i = G\_tCntr.get\&Inc()$;
    // $G\_tCntr.get\&Inc()$ returns the current value of G_tCntr and atomically increments it
**else**
    $G\_its_i = its$;
    $G\_cts_i = G\_tCntr.get\&Inc()$;
**end if**
$rset_i = wset_i = null$;
$G\_state_i$ = `live`;
$G\_valid_i = T$;
return $\langle i, G\_cts_i\rangle$

**Algorithm 12** STM $read(i, x)$: Invoked by a transaction $T_i$ to read $x$. It returns either the value of $x$ or $\mathcal{A}$

**if** ($x \in rset_i$) **then** // Check if $x$ is in $rset_i$
  return $rset_i[x].val$;
**else if** ($x \in wset_i$) **then** // Check if $x$ is in $wset_i$
  return $wset_i[x].val$;
**else**// $x$ is not in $rset_i$ and $wset_i$
  lock $x$; lock $G\_lock_i$;
  **if** ($G\_valid_i == F$) **then**
    return abort(i);
  **end if**
  // Find available value from $x.val$, returns the value
  $curVer = findavilval(G\_cts_i, x)$;
  $val = x[curVer].v$; add $\langle x, val\rangle$ to $rset_i$;
  add $T_i$ to $x[curVer].rl$;
  unlock $G\_lock_i$;
  unlock $x$;
  return $val$;
**end if**

**Algorithm 13** STM $write_i(x, val)$: A Transaction $T_i$ writes into local memory

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

**Algorithm 14** STM $tryC()$: Returns $ok$ on commit else return Abort

// The following check is an optimization which needs to be performed again later
Set<int> TSet $\leftarrow \phi$ // TSet storing transaction Ids
**for all** $x \in wset_i$ **do**
  lock $x$ in pre-defined order;
  **for** <each transaction $t_j$ of [$x$].rl> **do**
    TSet = [$x$].rl
  **end for**
  TSet = TSet $\cup \{t_i\}$
**end for**// $x \in wset_i$
lock $G\_lock_i$;
**if** ($G\_valid_i == F$) **then** return abort(i);
**else**
  Find LTS in TSet // lowest time stamp
  **if** (TS($t_i$) == LTS) **then**
    **for** <each transaction $t_j$ of [$x$].rl> **do**
      $G\_valid_j \leftarrow$ false
      unlock $G\_lock_j$;
    **end for**
  **else**
    return abort(i);
  **end if**
**end if**
// Store the current value of the global counter as commit time and increment it
$comTime = G\_tCntr.get\&Inc()$;

**for all** $x \in wset_i$ **do**
    replace the old value in $x.\texttt{vl}$ with $newValue$;
**end for**
$G\_state_i$ = `commit`;
unlock all variables;
return $\mathcal{C}$;

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

$G\_valid_i = F$; $G\_state_i$ = `abort`;
unlock all variables locked by $T_i$;
return $\mathcal{A}$;

# B Pcode of KSTM

**Algorithm 16** STM $init()$: Invoked at the start of the STM system. Initializes all the tobjs used by the STM System

$G\_tCntr$ = 1;
**for all** $x$ in $\mathcal{T}$ **do** // All the tobjs used by the STM System
    add $\langle 0, 0, nil \rangle$ to $x.\texttt{vl}$; // $T_0$ is initializing $x$
**end for**;

**Algorithm 17** STM $tbegin(its)$: 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 $\langle id, G\_cts \rangle$

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

**Algorithm 18** STM $read(i, x)$: Invoked by a transaction $T_i$ to read tobj $x$. It returns either the value of $x$ or $\mathcal{A}$

**if** ($x \in rset_i$) **then** // Check if the tobj $x$ is in $rset_i$
    return $rset_i[x].val$;
**else if** ($x \in wset_i$) **then** // Check if the tobj $x$ is in $wset_i$
    return $wset_i[x].val$;
**else**// tobj $x$ is not in $rset_i$ and $wset_i$
    lock $x$; lock $G\_lock_i$;
    **if** ($G\_valid_i == F$) **then** return abort(i);
    **end if**
    // findLTS: From $x$.`vl`, returns the largest `ts` value less than $G\_cts_i$. If no such version exists, it returns $nil$
    $curVer = findLTS(G\_cts_i, x)$;
    **if** ($curVer == nil$) **then** return abort(i); // Proceed only if $curVer$ is not nil
    **end if**
    $val = x[curVer].v$; add $\langle x, val \rangle$ to $rset_i$;
    add $T_i$ to $x[curVer].rl$;
    unlock $G\_lock_i$; unlock $x$;
    return $val$;
**end if**

**Algorithm 19** STM $write_i(x, val)$: A Transaction $T_i$ writes into local memory

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

**Algorithm 20** STM $tryC()$: Returns $ok$ on commit else return Abort

// The following check is an optimization which needs to be performed again later
lock $G\_lock_i$;
**if** ($G\_valid_i == F$) **then**
    return abort(i);
**end if**
unlock $G\_lock_i$;
$largeRL = allRL = nil$; // Initialize larger read list (largeRL), all read list (allRL) to nil
**for all** $x \in wset_i$ **do**
    lock $x$ in pre-defined order;
    // findLTS: returns the version with the largest `ts` value less than $G\_cts_i$. If no such version exists, it returns $nil$.
    $prevVer = findLTS(G\_cts_i, x)$; // prevVer: largest version smaller than $G\_cts_i$
    **if** ($prevVer == nil$) **then** // There exists no version with `ts` value less than $G\_cts_i$
        lock $G\_lock_i$; return abort(i);
    **end if**
    // **getLar**: obtain the list of reading transactions of $x[prevVer].rl$ whose $G\_cts$ is greater than $G\_cts_i$
    $largeRL = largeRL \cup getLar(G\_cts_i, x[prevVer].rl)$;
**end for**// $x \in wset_i$
$relLL = largeRL \cup T_i$; // Initialize relevant Lock List (relLL)
**for all** ($T_k \in relLL$) **do**
    lock $G\_lock_k$ in pre-defined order; // Note: Since $T_i$ is also in $relLL$, $G\_lock_i$ is also locked
**end for**
// Verify if $G\_valid_i$ is false

**if** $(G\_valid_i == F)$ **then**
    return abort(i);
**end if**
$abortRL = nil$ // Initialize abort read list (abortRL)
// Among the transactions in $T_k$ in $largeRL$, either $T_k$ or $T_i$ has to be aborted
**for all** $(T_k \in largeRL)$ **do**
    **if** $(isAborted(T_k))$ **then** // Transaction $T_k$ can be ignored since it is already aborted or about to be aborted
        continue;
    **end if**
    **if** $(G\_cts_i < G\_cts_k) \wedge (G\_state_k == \texttt{live})$ **then**
        // Transaction $T_k$ has lower priority and is not yet committed. So it needs to be aborted
        $abortRL = abortRL \cup T_k$; // Store $T_k$ in abortRL
    **else**// Transaction $T_i$ has to be aborted
        return abort(i);
    **end if**
**end for**
// Store the current value of the global counter as commit time and increment it
$comTime = G\_tCntr.get\&Inc()$;
**for all** $T_k \in abortRL$ **do** // Abort all the transactions in abortRL
    $G\_valid_k = F$;
**end for**
// Having completed all the checks, $T_i$ can be committed
**for all** $(x \in wset_i)$ **do**
    $newTuple = \langle G\_cts_i, wset_i[x].val, nil \rangle$; // Create new v_tuple: G_cts, val, `rl` for $x$
    **if** $(|x.vl| > k)$ **then**
        replace the oldest tuple in $x.\texttt{vl}$ with $newTuple$; // $x.\texttt{vl}$ is ordered by time stamp
    **else**
        add a $newTuple$ to $x.vl$ in sorted order;
    **end if**
**end for**// $x \in wset_i$
$G\_state_i$ = `commit`;
unlock all variables;
return $\mathcal{C}$;

**Algorithm 21** $isAborted(T_k)$: Verifies if $T_i$ is already aborted or its G_valid flag is set to false implying that $T_i$ will be aborted soon

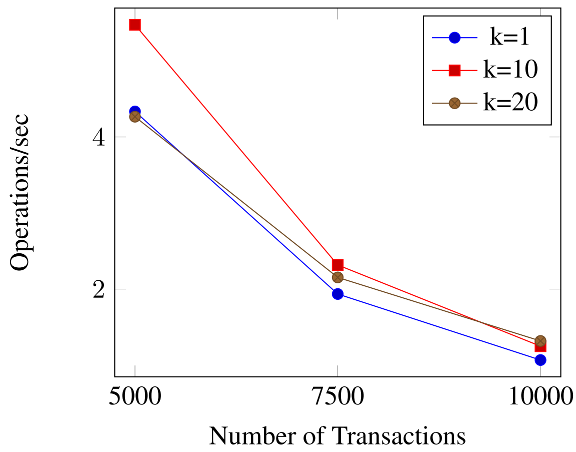
1: **if** $(G\_valid_k == F) \vee (G\_state_k == \texttt{abort}) \vee (T_k \in abortRL)$ **then**
2: return $T$;
3: **else**
4: return $F$;
5: **end if**

**Algorithm 22** $abort(i)$: Invoked by various STM methods to abort transaction $T_i$. It returns $\mathcal{A}$

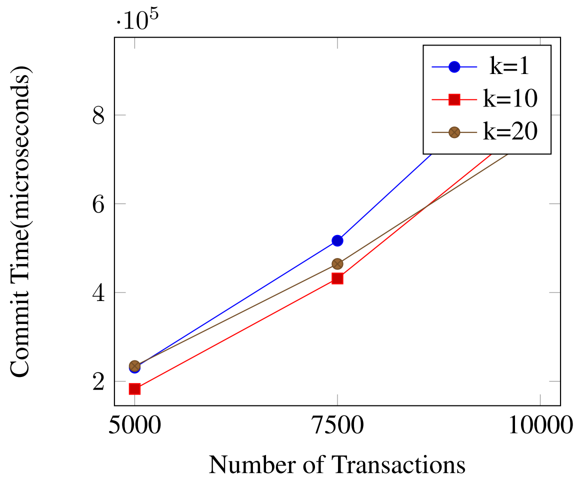
1: $G\_valid_i = F$; $G\_state_i$ = $\texttt{abort}$;
2: unlock all variables locked by $T_i$;
3: return $\mathcal{A}$;

# C Some Preliminary Results

The below graphs have been produced by using a linked list application to compare the performance of KSTM with different values of k. In the application chosen below, there were 90% lookups and remaining were 9:1 ratio of inserts and deletes. Varying number of threads were generated and each thread in turn generated 100 transactions.

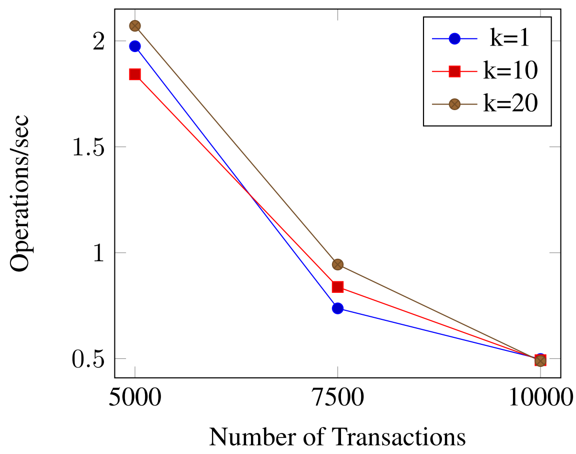


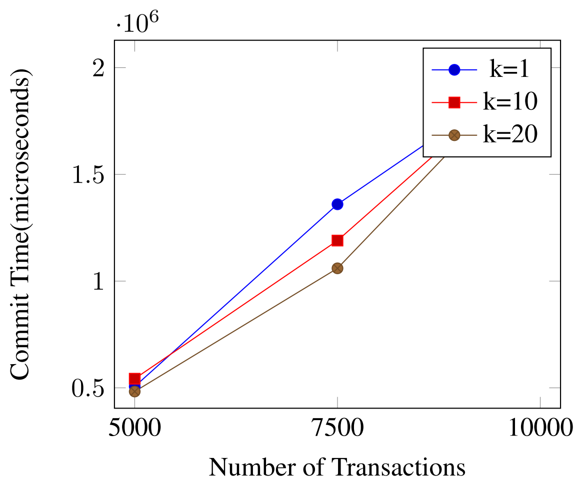

As per the results obtained, multiversion performs better than single version STM. This is because the multiple versions used in KSTM decreases the number of aborts per transaction, thereby effectively increasing the operations/sec performed.



The commit time (time taken per transaction to commit) observed during KSTM (k = 10 here) is the least since is inversely proportional to the operations/sec. As the number of transactions are increasing, they need more versions to read from, to attain higher concurrency leading to lesser abort counts.

In the application chosen below, there were 50% lookups and remaining were 9:1 ratio of inserts and deletes into the linked list. This kind of setup will have more read-write conflicts between the transactions involved when compared to the previous setup.




As per the graph, k = 20 gives the best operations/sec and the least commit time. Hence, having multiple versions(KSTM) performs better than single version STM in this setup too.