

# Starvation Freedom in Multi-Version Transactional Memory Systems

Anonymous user 1, Anonymous user 2, and Anonymous user 3

Anonymous affiliation

{Anonymous\_user1, Anonymous\_user2, Anonymous\_user3}@gmail.com

## 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. For STMs to be efficient, they must guarantee some progress properties. This work explores the notion of one of the progress property, i.e., *starvation-freedom*, in STMs. An STM system is said to be starvation-free if every thread invoking a transaction gets the opportunity to take a step (due to the presence of a fair scheduler) such that the transaction eventually commits.

A few *starvation-free* algorithms have been proposed in the literature in context of single-version STMs. These algorithms are priority based i.e. if two transactions are in 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. So, we propose multi-version *starvation-free* STM system which addresses this issue.

Multi-version STMs maintain multiple-versions for each transactional object. By storing multiple versions, these systems can achieve greater concurrency. In this paper, we propose 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 one to infinity. If *K* is infinite, 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 one, it becomes the same as a single-version *starvation-free* STM system. We prove the correctness and *starvation-freedom* property of the *KSFTM* algorithm.

To the best of our knowledge, this is the first multi-version STM system that satisfies *starvation-freedom*. We implement *KSFTM* and compare its performance with single-version *starvation-free* STM system (*SV-SFTM*) which works on the priority principle. Our experiments show that *KSFTM* achieves more than two-fold speedup over *SV-SFTM* for different workloads. *KSFTM* shows significant performance gain (ranging from 1.3 to 3.3 times) over existing *non-starvation-free* state-of-the-art STMs (ESTM, NOrec, and MVTO) on various workloads.

## 1 Introduction

Software Transactional Memory systems (*STMs*) [13, 25] 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. STMs often use an optimistic approach for concurrent execution of *transactions* (a piece of code invoked by a thread). In optimistic execution, each transaction reads from the shared memory, but all write updates are performed on local memory. On completion, the STM system *validates* the reads and writes of the transaction. If any inconsistency is found, the transaction is *aborted*, and its local writes are discarded. Otherwise, the transaction is *committed*, and its local updates are transferred to the shared memory. A transaction that has begun but has not yet *committed/aborted* is referred to as *live*.

A typical STM is a library which exports the following methods: *tbegin* which begins a transaction, *stm-read* which reads a *transactional object* or *t-object*, *stm-write* which writes to a *t-object*, *tryC* which tries to commit the transaction.

A typical code using STMs is as shown in Algorithm 1. It shows the overview of a concurrent *insert* method which inserts an element  $e$  into a linked-list  $LL$ . It consists of a loop where the thread creates a transaction. This transaction executes the code to insert an element  $e$  in a linked-list  $LL$  using *stm-read* and *stm-write* operations. At the end of the transaction, the thread calls *tryC*. At this point, the STM checks if the given transaction can be *committed* while satisfying the required safety properties (e.g., serializability [22], opacity [10]). If yes, then the transaction is *committed*. Otherwise, it is *aborted*. If the given transaction is *aborted* then the thread retries that transaction again.

---

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

---

```

1: retry = 0;
2: while (true) do
3:   id = tbegin (retry);
4:   ...
5:   ...
6:   v = stm-read(id, x); /* reads the value of x as v */
7:   ...
8:   ...
9:   stm-write(id, x, v'); /* writes a value v' to x */
10:  ...
11:  ...
12:  ret = tryC(id); /* tryC can return commit or abort */
13:  if (ret == commit) then
14:    break;
15:  else
16:    retry++;
17:  end if
18: end while

```

---

**Correctness:** By committing/aborting the transactions, the STM system ensures atomicity and consistency of transactions. Thus, an important requirement of STMs is to precisely identify the criterion as to when a transaction should be *aborted/committed*, referred to as *correctness-criterion*. Several *correctness-criteria* have been proposed for STMs such as opacity [10], virtual world consistency [17], local opacity [19, 20]. All these *correctness-criteria* 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, such as serializability, strict-serializability [22], the *correctness-criteria* for STMs ensure that even *aborted* transactions read correct values. This ensures that programmers do not see any undesirable side-effects due to the reads of *aborted* transactions such as infinite-loops, crashes, etc. in the application due to concurrent executions. This additional requirement on *aborted* transactions is a fundamental requirement of STMs which differentiates STMs from databases as observed by Guerraoui & Kapalka [10, 11].

To ensure correctness such as opacity, most STMs execute optimistically. With this approach, a transaction only reads values written by other *committed* transactions. **To achieve this, all writes are written to local memory first. They are added to the shared memory only when the transaction commits.** This (combined with required validation) can, in turn, ensure that the reads of the transaction are consistent as required by the *correctness-criterion*. Thus in this paper, we focus only on optimistic execution with the *correctness-criterion* being *local opacity* [20] (explained in Section 2).

**Progress Conditions:** Another requirement of the STM system is to ensure that transactions achieve *progress*. A well known blocking progress condition associated with concurrent programming is *starvation-freedom* [15, chap 2], [14]. In the context of STMs, *starvation-freedom* ensures that every *aborted* transaction that is retried infinitely often eventually commits. It can be defined as: an STM system is said to be *starvation-free* if a thread invoking a transaction  $T_i$  gets the opportunity to retry  $T_i$  on every abort (due to the presence of a fair underlying scheduler), then  $T_i$  will eventually commit.

With a *starvation-free* STM, the thread invoking insert in Algorithm 1 will eventually be able to *commit/abort*. Otherwise, every transaction invoked by the thread could potentially abort and the method will never complete.

*Wait-freedom* is another interesting progress condition for STMs in which every transaction commits regardless of the nature of concurrent transactions and the underlying scheduler [12, 14]. But it was shown by Guerraoui and Kapalka [3, 11] that it is not possible to achieve *wait-freedom* in dynamic STMs in which data sets of transactions are not known in advance. So in this paper, we explore the weaker progress condition of *starvation-freedom* for transactional memories. We also assume that the data sets of the transactions are *not* known in advance.

**Existing work on starvation-free STMs:** *Starvation-freedom* in STMs has been explored by a few researchers in literature such as Gramoli et al. [8], Waliullah and Stenstrom [27], Spear et al. [26]. Most of these systems work

by assigning priorities to transactions. In case of a conflict between two transactions, the transaction with lower priority is *aborted*. They ensure that every *aborted* transaction, on the sufficient number of retries, will eventually have the highest priority and hence will commit. We denote such an algorithm as *single-version starvation-free STM* or *SV-SFTM*.

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

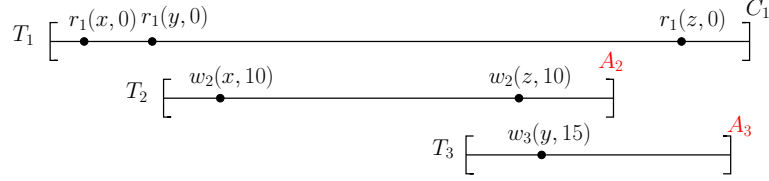


Figure 1: Drawback of single-version *starvation-free* algorithms

Figure 1 illustrates this problem. Consider the execution:  $r_1(x, 0) r_1(y, 0) w_2(x, 10) w_2(z, 10) w_3(y, 15) r_1(z, 0)$ . It has three transactions  $T_1$ ,  $T_2$  and  $T_3$ . Suppose  $T_1$  has the highest priority. After reading  $y$ , suppose  $T_1$  becomes slow. Next  $T_2$  and  $T_3$  want to write to  $x, z$  and  $y$  respectively and commit. But  $T_2$  and  $T_3$ , write operations conflict with  $T_1$ 's reads. Since  $T_1$  has higher priority and has not committed yet,  $T_2$  and  $T_3$  have to *abort*. If these transactions are retried and again conflict with  $T_1$  (if it is still live), they will have to *abort* again. Thus, any transaction with the priority lower than  $T_1$  and conflicts with it has to abort. It is as if  $T_1$  has locked the t-objects  $x, y$  and does not allow any other transaction, writing to these t-objects to commit.

**Solution Direction - Multi-version *starvation-free* STM:** To address this limitation, in this paper, we focus on developing *starv*

*ation-free* algorithms STMs using multiple versions. Many STMs have been proposed which uses the idea of multiple versions [18, 21, 7, 23]. Multi-version STMs, by virtue of multiple versions, can ensure that more methods succeed [18]. Hence, multi-version STMs (MVSTMs) can achieve greater concurrency.

Suppose the execution shown in Figure 1 uses multiple versions for each t-object. Then both  $T_2$  and  $T_3$  create a new version corresponding to each t-object  $x, z$  and  $y$  and return commit while not causing  $T_1$  to abort as well.  $T_1$  reads the initial value of  $z$ , and returns commit. So, by maintaining multiple versions all the transactions  $T_1$ ,  $T_2$ , and  $T_3$  can commit with equivalent serial history as  $T_1 T_2 T_3$  or  $T_1 T_3 T_2$ . Thus multiple versions can help with *starvation-freedom* without sacrificing on concurrency. This motivated us to develop a multi-version *starvation-free* STM system.

### Contributions of the paper:

- We propose a multi-version *starvation-free* STM system as *K-version starvation-free STM* or *KSFTM* for a given parameter  $K$ . Here  $K$  is the number of versions of any t-object and can range from 1 to  $\infty$ . To the best of our knowledge, this is the first *starvation-free* MVSTM. We develop *KSFTM* algorithm in a step-wise manner as follows:
  - First, in SubSection 3.3, we propose a simple priority-based  $K$ -version STM algorithm *Priority-based K-version MVTO* or *PKTO*. This algorithm guarantees the safety properties of strict-serializability and local opacity. However, it is not *starvation-free*.
  - We analyze *PKTO* to identify the characteristics that will help us to achieve preventing a transaction from getting *aborted* forever. This analysis leads us to the development of *starvation-free K-version TO* or *SFKTO* (SubSection 3.5), a multi-version *starvation-free* STM obtained by revising *PKTO*. But *SFKTO* does not satisfy correctness, i.e., strict-serializability, and local opacity.
  - Finally, we extend *SFKTO* to develop *KSFTM* (SubSection 3.6) that preserves the *starvation-freedom*, strict-serializability and local opacity.

- Our experiments show that *KSFTM* achieves more than two-fold speedup over *SV-SFTM* (explained above). *KSFTM* achieves a performance gain by a factor ranging from 1.3 to 3.3 times over existing *non-starvation-free* state-of-the-art STMs (ESTM[6], NOrec[5], and MVTO[18], etc.) on various workloads.

## 2 System Model and Preliminaries

Following [11, 20], we assume a system of  $n$  processes/threads,  $p_1, \dots, p_n$  that access a collection of *transactional objects* (or *t-objects*) via atomic *transactions*. Each transaction has a unique identifier. Within a transaction, processes can perform *transactional operations or methods*: *tbegin* that begins a transaction, *stm-write*( $x, v$ ) operation that updates a t-object  $x$  with value  $v$  in its local memory, the *stm-read*( $x$ ) operation tries to read  $x$ , *tryC*() that tries to commit the transaction and returns *commit* 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 as arguments by *stm-write* operations are unique.

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

For a transaction  $T_k$ , we denote all the t-objects accessed by its read operations as  $rset_k$  and t-objects accessed by its write operations as  $wset_k$ . We denote all the operations of a transaction  $T_k$  as  $T_k.evts$  or  $evts_k$ .

**History:** A *history* is a sequence of *events*, i.e., a sequence of invocations and responses of transactional operations. The collection of events is denoted as  $H.evts$ . For simplicity, we only consider *sequential* histories here: the invocation of each transactional operation is immediately followed by a matching response. Therefore, we treat each transactional operation as one atomic event, and let  $<_H$  denote the total order on the transactional operations incurred by  $H$ . With this assumption, the only relevant events of a transaction  $T_k$  is of the types:  $r_k(x, v)$ ,  $r_k(x, \mathcal{A})$ ,  $w_k(x, v)$ ,  $tryC_k(\mathcal{C})$  (or  $c_k$  for short),  $tryC_k(\mathcal{A})$ ,  $tryA_k(\mathcal{A})$  (or  $a_k$  for short). We identify a history  $H$  as tuple  $\langle H.evts, <_H \rangle$ .

Let  $H|T$  denote the history consisting of events of  $T$  in  $H$ , and  $H|p_i$  denote the history consisting of events of  $p_i$  in  $H$ . We only consider *well-formed* histories here, i.e., no transaction of a process begins before the previous transaction invocation has completed (either *commits* or *aborts*). 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  $H$  is denoted by  $H.committed$  (resp.,  $H.aborted$ ). The set of *incomplete* or *live* transactions in  $H$  is denoted by  $H.incomp = H.live = (H.txns - H.committed - H.aborted)$ .

For a history  $H$ , we construct the *completion* of  $H$ , denoted as  $\bar{H}$ , by inserting  $tryA_k(\mathcal{A})$  immediately after the last event of every transaction  $T_k \in H.live$ . But for  $tryC_i$  of transaction  $T_i$ , if it released the lock on first t-object successfully that means updates made by  $T_i$  is consistent so,  $T_i$  will immediately return commit.

**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 say that a history is *t-sequential* if all the transactions are ordered by this real-time order. Note that from our earlier assumption all the transactions of a single process are ordered by real-time.

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

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

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

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

**Opacity and Strict Serializability:** We say that two histories  $H$  and  $H'$  are *equivalent* if they have the same set of events. Now a history  $H$  is said to be *opaque* [10, 11] if it is valid and there exists a t-sequential legal history  $S$  such that (1)  $S$  is equivalent to  $\overline{H}$  and (2)  $S$  respects  $\prec_H^{RT}$ , i.e.,  $\prec_H^{RT} \subset \prec_S^{RT}$ . By requiring  $S$  being equivalent to  $\overline{H}$ , opacity treats all the incomplete transactions as aborted. We call  $S$  an (opaque) *serialization* of  $H$ .

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

Serializability is commonly used criterion in databases. But it is not suitable for STMs as it does not consider the correctness of *aborted* transactions as shown by Guerraoui & Kapalka [10]. Opacity, on the other hand, considers the correctness of *aborted* transactions as well. Similarly, local opacity (described below) is another correctness-criterion for STMs but is not as restrictive as opacity.

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

A history  $H$  is said to be *locally-opaque* [19, 20] if all the sub-histories in  $H.subhistSet$  are opaque. It must be seen that in the construction of sub-history of an aborted transaction  $T_i$ , the *subhist* will contain operations from only one aborted transaction which is  $T_i$  itself and no other live/aborted transactions. Similarly, the sub-history of *committed* transaction  $T_l$  has no operations of aborted and live transactions. Thus in local opacity, no aborted or live transaction can cause another transaction to abort. It was shown that local opacity [19, 20] allows greater concurrency than opacity. Any history that is opaque is also locally-opaque but not necessarily the vice-versa. On the other hand, a history that is locally-opaque is also strict-serializable, but the vice-versa need not be true.

**Graph Characterization of Local Opacity:** To prove correctness of STM systems, it is useful to consider graph characterization of histories. In this section, we describe the graph characterization developed by Kumar et al [18] for proving opacity which is based on characterization by Bernstein and Goodman [2]. We extend this characterization for LO.

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

Consider the history  $H2 : r_1(x, 0)r_2(x, 0)r_1(y, 0)r_3(z, 0)w_1(x, 5)w_3(y, 15)w_2(y, 10)w_1(z, 10)c_1c_2r_4(x, 5)r_4(y, 10)w_3(z, 15)c_3r_4(z, 10)$ . Using the notation that a committed transaction  $T_i$  writing to  $x$  creates a version  $x_i$ , a possible version order for  $H2 \ll_{H2}$  is:  $\langle x_0 \ll x_1 \rangle, \langle y_0 \ll y_2 \ll y_3 \rangle, \langle z_0 \ll z_1 \ll z_3 \rangle$ .

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

1. *real-time*(real-time) edges: If  $T_i$  commits before  $T_j$  starts in  $H$ , then there is an edge from  $v_i$  to  $v_j$ . This set of edges are referred to as  $rt(H)$ .
2. *rf*(reads-from) edges: If  $T_j$  reads  $x$  from  $T_i$  in  $H$ , then there is an edge from  $v_i$  to  $v_j$ . Note that in order for this to happen,  $T_i$  must have committed before  $T_j$  and  $c_i <_H r_j(x)$ . This set of edges are referred to as  $rf(H)$ .
3. *mv*(multiversion) edges: The mv edges capture the multiversion relations and is based on the version order. Consider a successful read operation  $r_k(x, v)$  and the write operation  $w_j(x, v)$  belonging to transaction  $T_j$  such that  $r_k(x, v)$  reads  $x$  from  $w_j(x, v)$  (it must be noted  $T_j$  is a committed transaction and  $c_j <_H r_k$ ). Consider a committed transaction  $T_i$  which writes to  $x$ ,  $w_i(x, u)$  where  $u \neq v$ . Thus the versions created  $x_i, x_j$  are related by  $\ll$ . Then, if  $x_i \ll x_j$  we add an edge from  $v_i$  to  $v_j$ . Otherwise  $(x_j \ll x_i)$ , we add an edge from  $v_k$  to  $v_i$ . This set of edges are referred to as  $mv(H, \ll)$ .

Using the construction, the  $OPG(H2, \ll_{H2})$  for history  $H2$  and  $\ll_{H2}$  is shown in Figure 11. The edges are annotated. The only mv edge from  $T4$  to  $T3$  is because of t-objects  $y, z$ .  $T4$  reads value 5 for  $z$  from  $T1$  whereas  $T3$  also writes 15 to  $z$  and commits before  $r_4(z)$ .

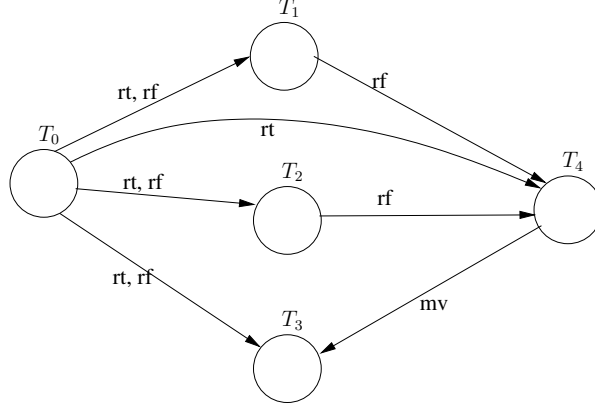


Figure 2:  $OPG(H2, \ll_{H2})$

Kumar et al [18] showed that if a version order  $\ll$  exists for a history  $H$  such that  $OPG(H, \ll_H)$  is acyclic, then  $H$  is opaque. This is captured in the following result.

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

This result can be easily extended to prove LO as follows

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

**Proof.** To prove this theorem, we have to show that each sub-history  $sh$  in  $H.subhistSet$  is valid. Then the rest follows from Result 10. Now consider a sub-history  $sh$ . Consider any read operation  $r_i(x, v)$  of a transaction  $T_i$ . It is clear that  $T_i$  must have read a version of  $x$  created by a previously committed transaction. From the construction of  $sh$ , we get that all the transaction that committed before  $r_i$  are also in  $sh$ . Hence  $sh$  is also valid.

Now, proving  $sh$  to be opaque iff there exists a version order  $\ll_{sh}$  such that  $OPG(sh, \ll_{sh})$  is acyclic follows from Result 10.

### 3 The Working of KSFTM Algorithm

In this section, we propose *K-version starvation-free STM* or *KSFTM* for a given parameter  $K$ . Here  $K$  is the number of versions of each t-object and can range from 1 to  $\infty$ . When  $K$  is 1, it boils down to single-version *starvation-free* STM. If  $K$  is  $\infty$ , then *KSFTM* uses unbounded versions and needs a separate garbage collection mechanism to delete old versions like other MVSTMs proposed in the literature [18, 21]. We denote *KSFTM* using unbounded versions as *UVSFTM* and *UVSFTM* with garbage collection as *UVSFTM-GC*.

Next, we describe some *starvation-freedom* preliminaries in SubSection 3.1 to explain the working of *KSFTM* algorithm. To explain the intuition behind the *KSFTM* algorithm, we start with the modification of MVTO [2, 18] algorithm in SubSection 3.3. We then make a sequence of modifications to it to arrive at *KSFTM* algorithm.

#### 3.1 Starvation-Freedom Preliminaries

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

**Definition 3 Starvation-Freedom:** *A STM system is said to be starvation-free if a thread invoking a transaction  $T_i$  gets the opportunity to retry  $T_i$  on every abort, due to the presence of a fair scheduler, then  $T_i$  will eventually commit.*

As explained by Herlihy & Shavit [14], a fair scheduler implies that no thread is forever delayed or crashed. Hence with a fair scheduler, we get that if a thread acquires locks then it will eventually release the locks. Thus a thread cannot block out other threads from progressing.

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

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

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

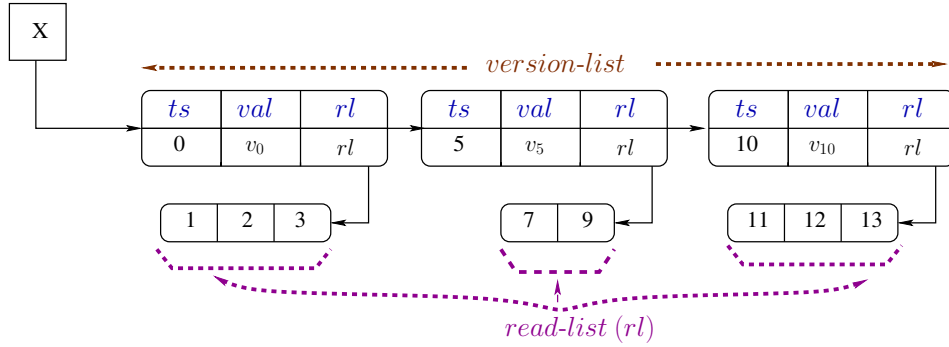


Figure 3: Data Structures for Maintaining Versions

We also assume that in the absence of other concurrent conflicting transactions, every transaction will commit. In other words, if a transaction is executing in a system where other concurrent conflicting transactions are not present then it will not self-abort. If transactions can self-abort then providing *starvation-freedom* is impossible.

**Common Data Structures and STM Methods:** Here we describe the common data structures used by all the algorithms proposed in this section. For each t-object, the algorithms maintain multiple versions in *version-list* (or *vlist*) using list. Similar to versions in MVTO [18], each version of a t-object is a tuple denoted as *vTuple* and consists of three fields: (1) timestamp, (or *ts*) of the transaction that created this version which normally is the CTS; (2) the value (or *val*) of the version; (3) a list, called read-list (or *rl*), consisting of transactions ids (can be CTS as well) that read from this version. The read-list of a version is initially empty. Figure 3 illustrates this structure. For a t-object  $x$ , we use the notation  $x[t]$  to access the version with timestamp  $t$ . Depending on the algorithm considered, the fields change of this structure.

The algorithms have access to a global atomic counter,  $G\_tCntr$  used for generating timestamps in the various transactional methods. We assume that the STM system exports the following methods for a transaction  $T_i$ : (1)  $tbegin(t)$  where  $t$  is provided by the invoking thread,  $Th_x$ . From our earlier assumption, it is the CTS of the first incarnation. In case  $Th_x$  is invoking this transaction for the first time, then  $t$  is *null*. This method returns a unique timestamp to  $Th_x$  which is the CTS/id of the transaction. (2)  $stm-read_i(x)$  tries to read t-object  $x$ . It returns either value  $v$  or  $\mathcal{A}$ . (3)  $stm-write_i(x, v)$  operation that updates a t-object  $x$  with value  $v$  locally. It returns *ok*. (4)  $tryC_i()$  tries to commit the transaction and returns  $\mathcal{C}$  if it succeeds. Otherwise, it returns  $\mathcal{A}$ .

**Correctness Criteria:** For ease of exposition, we initially consider strict-serializability as *correctness-criterion* to illustrate the correctness of the algorithms. But strict-serializability does not consider the correctness of *aborted* transactions and as a result not a suitable *correctness-criterion* for STMs. Finally, we show that the proposed STM algorithm *KSFTM* satisfies local opacity, a *correctness-criterion* for STMs (described in Section 2). We denote the set of histories generated by an STM algorithm, say  $A$ , as  $gen(A)$ .

### 3.2 Motivation for Starvation Freedom in Multi-Version Systems

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

### 3.2.1 Illustration of SV-SFTM

Forward-oriented optimistic concurrency control protocol (FOCC), is a commonly used optimistic algorithm in databases [28, 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 t-objects 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 *SV-SFTM* we use  $\tau_{ss}$  which are monotonically in increasing order. We implement the  $\tau_{ss}$  using atomic counters. Each transaction  $t_i$  has two time-stamps: (i) *current time-stamp or CTS*: this is a unique  $\tau_{ss}$  allotted 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, *SV-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. We show that *SV-SFTM* satisfies opacity and starvation-free.

**Theorem 4** *Any history generated by SV-SFTM is opaque.*

**Theorem 5** *SV-SFTM ensure starvation-freedom.*

We prove the correctness by showing that the conflict graph [28, Chap 3], [19] of any history generated by *SV-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 4 shows the a sample execution of *SV-SFTM*. It compares the execution of FOCC with *SV-SFTM*. The execution on the left corresponds to FOCC, while the execution one the right is of *SV-SFTM* for the same input. It can be seen that each transaction has two  $\tau_{ss}$  in *SV-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 *SV-SFTM*,  $T_{5,3}$  which is re-execution of  $T_{3,3}$  has the same ITS 3. Hence, when  $T_{4,4}$  validates in *SV-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.

### 3.2.2 drawback of SV-SFTM

Figure 5 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  $T_1T_2T_3$ .

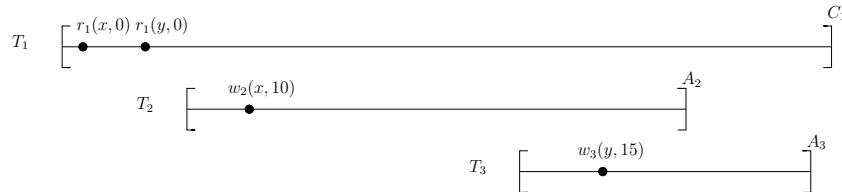


Figure 5: Pictorial representation of execution under SFTM



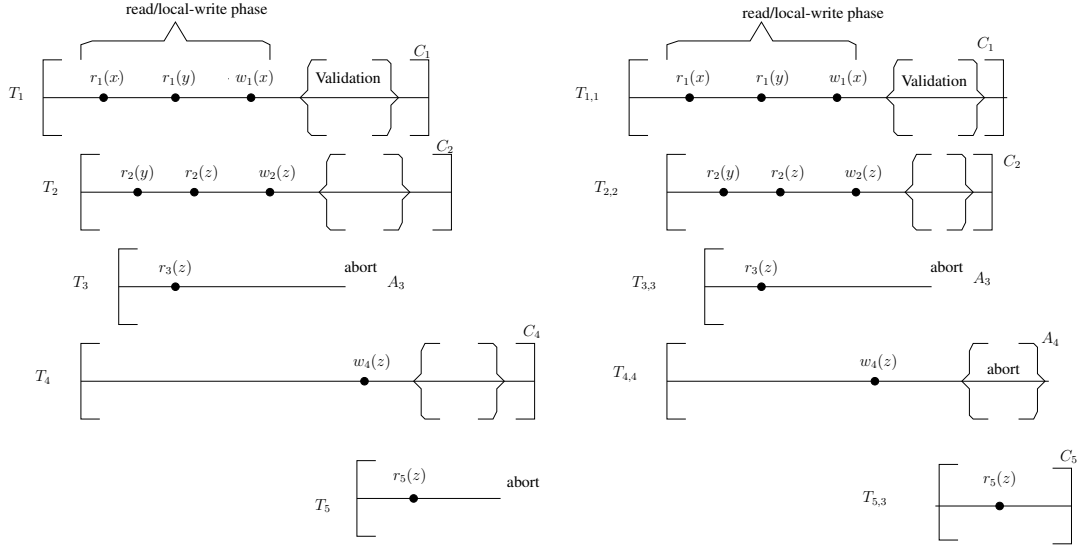


Figure 4: Sample execution of *SV-SFTM*

### 3.2.3 Data Structures and Pseudocode of *SV-SFTM*

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 t-object to which transaction  $T_i$  writes the value  $val$ . Similarly, we refer to a tuple in  $T_i$ 's write-set by  $wset_i[x]$ .

In addition to these local structures, the following shared global structures are maintained that are shared across transactions (and hence, threads). 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 ( $T$ ). If it becomes false ( $F$ ) then  $T_i$  has to be aborted.
- $G\_state_i$ : This is a variable which states the current value of  $T_i$ . It has three states: live, commit or abort.

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

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

---

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

---

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

---

```

1:  $i = \text{unique-id}$ ; /* An unique id to identify this transaction. It could be same as  $G\_cts$ . */
2: if (its == nil) then
3:    $G\_its_i = G\_cts_i = G\_tCntr.get\&Inc()$ ;
4:   /*  $G\_tCntr.get\&Inc()$  returns the current value of  $G\_tCntr$  and atomically increments it by 1. */
5: else
6:    $G\_its_i = its$ ;
7:    $G\_cts_i = G\_tCntr.get\&Inc()$ ;
8: end if
9:  $rset_i = wset_i = \text{null}$ ;
10:  $G\_state_i = \text{live}$ ;
11:  $G\_valid_i = T$ ;
12: return  $\langle i, G\_cts_i \rangle$ 

```

---



---

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

---

```

1: if ( $x \in wset_i$ ) then /* Check if  $x$  is in  $wset_i$  */
2:   return  $wset_i[x].val$ ;
3: else if ( $x \in rset_i$ ) then /* Check if  $x$  is in  $rset_i$  */
4:   return  $rset_i[x].val$ ;
5: else /*  $x$  is not in  $rset_i$  and  $wset_i$  */
6:   lock  $x$ ;
7:   lock  $G\_lock_i$ ;
8:   if ( $G\_valid_i == F$ ) then
9:     return abort(i);
10:  end if
11:   $val = x.val$ ;
12:  add  $T_i$  to  $x.rl$ ;
13:  unlock  $G\_lock_i$ ;
14:  unlock  $x$ ;
15:  return  $val$ ;
16: end if

```

---

---

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

---

```

1: Append the  $d\_tuple\langle x, val \rangle$  to  $wset_i$ ; /* If same dataitem then overwrite the tuple */
2: return  $ok$ ;

```

---



---

**Algorithm 6** STM  $findLLTS(TSet)$ : Find the lowest  $its$  value among all the live trasactions in  $TSet$ .

---

```

1:  $min\_its = \infty$ 
2: for all ( $T_j \in TSet$ ) do
3:   if ( $(G\_its_j < min\_its) \ \&\& \ (G\_state_j == live)$ ) then
4:      $min\_its = G\_its_j$ ;
5:   end if
6: end for
7: return  $min\_its$ ;

```

---



---

**Algorithm 7** STM  $tryC()$ : Returns  $C$  on commit else return Abort  $\mathcal{A}$

---

```

1: lock  $G\_lock_i$ 
2: if ( $G\_valid_i == F$ ) then return  $abort(i)$ ;
3: end if
4:  $TSet = null$  /*  $TSet$  storing transaction Ids */
5: for all ( $x \in wset_i$ ) do
6:   lock  $x$  in pre-defined order;
7:   for all ( $T_j \in x.rl$ ) do
8:      $TSet = TSet \cup \{T_j\}$ 
9:   end for
10: end for /*  $x \in wset_i$  */
11:  $TSet = TSet \cup \{T_i\}$  /* Add current transaction  $T_i$  into  $TSet$  */
12: for all ( $T_k \in TSet$ ) do
13:   lock  $G\_lock_k$  in pre-defined order; /* Note: Since  $T_i$  is also in  $TSet$ ,  $G\_lock_i$  is also locked */
14: end for
15: if ( $G\_valid_i == F$ ) then return  $abort(i)$ ;
16: else
17:   if ( $G\_its_i == findLLTS(TSet)$ ) then /* Check if  $T_i$  has lowest  $its$  among all live transactions in  $TSet$  */
18:     for all ( $T_j \in TSet$ ) do /* ( $T_i \neq T_j$ ) */
19:        $G\_valid_j = F$ 
20:       unlock  $G\_lock_j$ ;
21:     end for
22:   else
23:     return  $abort(i)$ ;
24:   end if
25: end if
26: for all ( $x \in wset_i$ ) do
27:   replace the old value in  $x.val$  with  $newValue$ ;
28:    $x.rl = null$ ;
29: end for
30:  $G\_state_i = commit$ ;
31: unlock all variables locked by  $T_i$ ;
32: return  $C$ ;

```

---

---

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

---

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

**Simplifying Assumptions:** We next describe the main idea behind the starvation-free STM algorithm *KSFTM* through a sequence of algorithms. For ease of exposition, we make two simplifying assumptions (1) We assume that in the absence of other concurrent conflicting transactions, every transaction will commit. In other words, if a transaction is executed in a system by itself, it will not self-abort. (2) We initially consider strict-serializability as correctness-criterion to illustrate the correctness of the algorithms. But strict-serializability does not consider the correctness of aborted transactions and as a result not a suitable correctness-criterion for STMs. Finally, we show that the proposed STM algorithm *KSFTM* satisfies local opacity, a correctness-criterion for STMs.

We denote the set of histories generated by an STM algorithm, say  $A$ , as  $gen(A)$ .

### 3.3 Priority-based MVTO Algorithm

In this subsection, we describe a modification to the multi-version timestamp ordering (MVTO) algorithm [2, 18] to ensure that it provides preference to transactions that have low ITS, i.e., transactions that have been in the system for a longer time. We denote the basic algorithm which maintains unbounded versions as *Priority-based MVTO* or *PMVTO* (akin to the original MVTO). We denote the variant of *PMVTO* that maintains  $K$  versions as *PKTO* and the unbounded versions variant with garbage collection as *PMVTO-GC*. In this sub-section, we specifically describe *PKTO*. But most of these properties apply to *PMVTO* and *PMVTO-GC* as well.

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

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

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

$tryC$  : This operation consists of three steps. In Step 1, it checks whether  $T_i$  can be *committed*. In Step 2, it performs the necessary tasks to mark  $T_i$  as a *committed* transaction and in Step 3,  $T_i$  return commits.

1. Before  $T_i$  can commit, it needs to verify that any version it creates does not violate consistency. Suppose  $T_i$  creates a new version of  $x$  with timestamp  $i$ . Let  $j$  be the largest timestamp smaller than  $i$  for which version of  $x$  exists. Let this version be  $x[j]$ . Now,  $T_i$  needs to make sure that any transaction that has read  $x[j]$  is not affected by the new version created by  $T_i$ . There are two possibilities of concern:
  - (a) Let  $T_k$  be some transaction that has read  $x[j]$  and  $k > i$  ( $k = \text{CTS of } T_k$ ). In this scenario, the value read by  $T_k$  would be incorrect (w.r.t strict-serializability) if  $T_i$  is allowed to create a new version. In this case, we say that the transactions  $T_i$  and  $T_k$  are in *conflict*. So, we do the following:
    - (i) if  $T_k$  has already *committed* then  $T_i$  is *aborted*;
    - (ii) if  $T_k$  is live and  $its_k$  is less than  $its_i$ . Then again  $T_i$  is *aborted*;
    - (iii) If  $T_k$  is still live with  $its_i$  less than  $its_k$  then  $T_k$  is *aborted*.
  - (b) The previous version  $x[j]$  does not exist. This happens when the previous version  $x[j]$  has been overwritten. In this case,  $T_i$  is *aborted* since *PKTO* does not know if  $T_i$  conflicts with any other transaction  $T_k$  that has read the previous version.
2. After Step 1, we have verified that it is ok for  $T_i$  to commit. Now, we have to create a version of each t-object  $x$  in the  $wset$  of  $T_i$ . This is achieved as follows:
  - (a)  $T_i$  creates a  $vTuple \langle i, wset_i.x.v, null \rangle$ . In this tuple,  $i$  (CTS of  $T_i$ ) is the timestamp of the new version;  $wset_i.x.v$  is the value of  $x$  in  $T_i$ 's  $wset$ , and the read-list of the  $vTuple$  is *null*.

- (b) Suppose the total number of versions of  $x$  is  $K$ . Then among all the versions of  $x$ ,  $T_i$  replaces the version with the smallest timestamp with  $vTuple \langle i, wset_i.x.v, null \rangle$ . Otherwise, the  $vTuple$  is added to  $x$ 's  $vlist$ .

3. Transaction  $T_i$  is then *committed*.

The algorithm described here is only the main idea. The actual implementation will use locks to ensure that each of these methods are linearizable [16]. It can be seen that *PKTO* gives preference to the transaction having lower ITS in Step 1a. Transactions having lower ITS have been in the system for a longer time. Hence, *PKTO* gives preference to them.

### 3.4 Pseudocode of *PKTO*

---

**Algorithm 9** STM *init()*: Invoked at the start of the STM system. Initializes all the t-objects used by the STM System

---

```

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

---



---

**Algorithm 10** 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  $\langle id, G\_cts \rangle$

---

```

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

---

---

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

---

```

1: if ( $x \in rset_i$ ) then /* Check if the t-object  $x$  is in  $rset_i$  */
2:   return  $rset_i[x].val$ ;
3: else if ( $x \in wset_i$ ) then /* Check if the t-object  $x$  is in  $wset_i$  */
4:   return  $wset_i[x].val$ ;
5: else/* t-object  $x$  is not in  $rset_i$  and  $wset_i$  */
6:   lock  $x$ ; lock  $G\_lock_i$ ;
7:   if ( $G\_valid_i == F$ ) then return abort(i);
8:   end if
9:   /* findLTS: From  $x.vl$ , returns the largest  $ts$  value less than  $G\_cts_i$ . If no such version exists, it returns
    $nil$  */
10:   $curVer = findLTS(G\_cts_i, x)$ ;
11:  if ( $curVer == nil$ ) then return abort(i); /* Proceed only if  $curVer$  is not nil */
12:  end if
13:   $val = x[curVer].v$ ; add  $\langle x, val \rangle$  to  $rset_i$ ;
14:  add  $T_i$  to  $x[curVer].rl$ ;
15:  unlock  $G\_lock_i$ ; unlock  $x$ ;
16:  return  $val$ ;
17: end if

```

---



---

**Algorithm 12** STM  $write_i(x, val)$ : A Transaction  $T_i$  writes into local memory

---

```

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

```

---

---

**Algorithm 13** STM *tryC()*: Returns *ok* on commit else return Abort

---

```
1: /* The following check is an optimization which needs to be performed again later */
2: lock  $G\_lock_i$ ;
3: if ( $G\_valid_i == F$ ) then
4:   return abort(i);
5: end if
6: unlock  $G\_lock_i$ ;
7:  $largeRL = allRL = nil$ ; /* Initialize larger read list (largeRL), all read list (allRL) to nil */
8: for all  $x \in wset_i$  do
9:   lock  $x$  in pre-defined order;
10:  /* findLTS: returns the version with the largest  $ts$  value less than  $G\_cts_i$ . If no such version exists, it
    returns nil. */
11:   $prevVer = findLTS(G\_cts_i, x)$ ; /* prevVer: largest version smaller than  $G\_cts_i$  */
12:  if ( $prevVer == nil$ ) then /* There exists no version with  $ts$  value less than  $G\_cts_i$  */
13:    lock  $G\_lock_i$ ; return abort(i);
14:  end if
15:  /* getLar: obtain the list of reading transactions of  $x[prevVer].rl$  whose  $G\_cts$  is greater than  $G\_cts_i$  */
16:   $largeRL = largeRL \cup getLar(G\_cts_i, x[prevVer].rl)$ ;
17: end for /*  $x \in wset_i$  */
18:  $relLL = largeRL \cup T_i$ ; /* Initialize relevant Lock List (relLL) */
19: for all ( $T_k \in relLL$ ) do
20:   lock  $G\_lock_k$  in pre-defined order; /* Note: Since  $T_i$  is also in  $relLL$ ,  $G\_lock_i$  is also locked */
21: end for
22: /* Verify if  $G\_valid_i$  is false */
23: if ( $G\_valid_i == F$ ) then
24:   return abort(i);
25: end if
26:  $abortRL = nil$  /* Initialize abort read list (abortRL) */
27: /* Among the transactions in  $T_k$  in  $largeRL$ , either  $T_k$  or  $T_i$  has to be aborted */
28: for all ( $T_k \in largeRL$ ) do
29:   if ( $isAborted(T_k)$ ) then /* Transaction  $T_k$  can be ignored since it is already aborted or about to be aborted
    */
30:     continue;
31:   end if
32:   if ( $G\_its_i < G\_its_k$ )  $\wedge$  ( $G\_state_k == live$ ) then
33:     /* Transaction  $T_k$  has lower priority and is not yet committed. So it needs to be aborted */
34:      $abortRL = abortRL \cup T_k$ ; /* Store  $T_k$  in abortRL */
35:   else /* Transaction  $T_i$  has to be aborted */
36:     return abort(i);
37:   end if
38: end for
```

---

---

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

---

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

---

---

```

39: /* Store the current value of the global counter as commit time and increment it */
40:  $comTime = G\_tCntr.get\&Inc()$ ;
41: for all  $T_k \in abortRL$  do /* Abort all the transactions in abortRL */
42:    $G\_valid_k = F$ ;
43: end for
44: /* Having completed all the checks,  $T_i$  can be committed */
45: for all  $(x \in wset_i)$  do
46:    $newTuple = \langle G\_cts_i, wset_i[x].val, nil \rangle$ ; /* Create new v_tuple:  $G\_cts$ ,  $val$ ,  $rl$  for  $x$  */
47:   if  $(|x.vl| > k)$  then
48:     replace the oldest tuple in  $x.vl$  with  $newTuple$ ; /*  $x.vl$  is ordered by timestamp */
49:   else
50:     add a  $newTuple$  to  $x.vl$  in sorted order;
51:   end if
52: end for /*  $x \in wset_i$  */
53:  $G\_state_i = commit$ ;
54: unlock all variables;
55: return  $\mathcal{C}$ ;

```

---

**Algorithm 15**  $abort(i)$ : 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  $\mathcal{A}$ ;

```

---

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

**Property 6** Any history generated by *PKTO* is strict-serializable.

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

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

This issue is not restricted only to *PKTO*. It can occur in *PMVTO* (and *PMVTO-GC*) due to the point (2) described above.



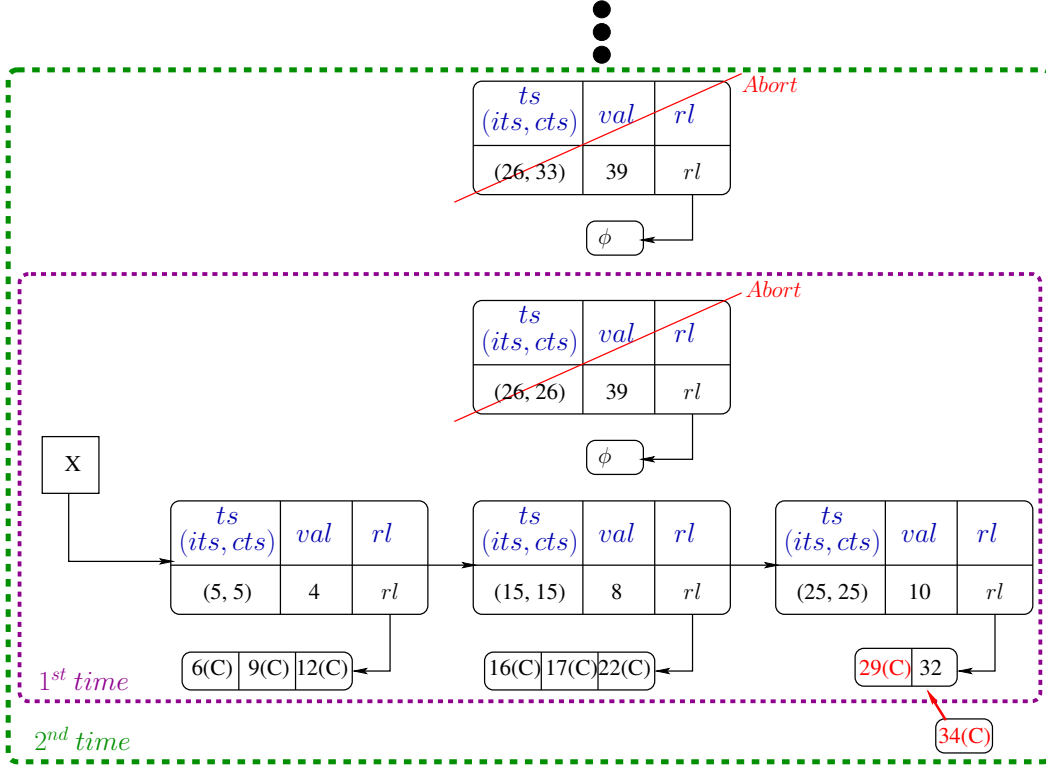


Figure 6: Pictorial representation of execution under *PKTO*

We illustrate this problem in *PKTO* with Figure 6. Here transaction  $T_{26}$ , with ITS 26 is the lowest among all the live transactions, starves due to Step 1a.(i) of the *tryC*. *First time*,  $T_{26}$  gets *aborted* due to higher timestamp transaction  $T_{29}$  in the read-list of  $x[25]$  has *committed*. We have denoted it by a ‘(C)’ next to the version. The *second time*,  $T_{26}$  retries with same ITS 26 but new CTS 33. Now when  $T_{33}$  comes for commit, suppose another transaction  $T_{34}$  in the read-list of  $x[25]$  has already *committed*. So this will cause  $T_{33}$  (another incarnation of  $T_{26}$ ) to abort again. Such scenario can possibly repeat again and again and thus causing no incarnation of  $T_{26}$  to ever commit leading to its starvation.

**Garbage Collection in *UVSFTM-GC* and *PMVTO-GC*:** Having multiple versions to increase the performance and to decrease the number of aborts, leads to creating too many versions which are not of any use and hence occupying space. So, such garbage versions need to be taken care of. Hence we come up with a garbage collection over these unwanted versions. This technique help to conserve memory space and increases the performance in turn as no more unnecessary traversing of garbage versions by transactions is necessary. We have used a global, i.e., across all transactions a list that keeps track of all the live transactions in the system. We call this list as *live-list*. Each transaction at the beginning of its life cycle creates its entry in this *live-list*. Under the optimistic approach of STM, each transaction in the shared memory performs its updates in the *tryC* phase. In this phase, each transaction performs some validations, and if all the validations are successful then the transaction make changes or in simple terms creates versions of the corresponding t-object in the shared memory. While creating a version every transaction, check if it is the least timestamp live transaction present in the system by using *live-list* data structure, if yes then the current transaction deletes all the version of that t-object and create one of its own. Else the transaction does not do any garbage collection or delete any version and look for creating a new version of next t-object in the write set, if at all. Figure 8 shows that both *UVSFTM-GC* and *PMVTO-GC* performs better than *UVSFTM* and *PMVTO* across all workloads.

### 3.5 Modifying *PKTO* to Obtain *SFKTO*: Trading Correctness for *Starvation-Freedom*

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

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

Now, there are six possible permutations of these statements. We identify these permutations and the action that should be taken for that permutation in Table 1. In all these permutations, the read operations of a transaction come before the write operations as the writes to the shared memory occurs only in the *tryC* operation (due to optimistic execution) which is the final operation of a transaction.

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

Table 1: Permutations of operations

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

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

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

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

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

$$wts_i = cts_i + C * (cts_i - its_i); \quad (1)$$

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

Furthermore, in SFKTO algorithm, CTS is replaced with WTS for *stm-read*, *stm-write* and *tryC* operations of *PKTO*. In SFKTO, a transaction  $T_i$  uses  $wts_i$  to read a version in *stm-read*. Similarly,  $T_i$  uses  $wts_i$  in *tryC* to find the appropriate previous version (in Step 1b) and to verify if  $T_i$  has to be aborted (in Step 1a). Along the same lines, once  $T_i$  decides to commit and create new versions of  $x$ , the timestamp of  $x$  will be same as its  $wts_i$  (in Step 3). Thus the timestamp of all the versions in *vlist* will be WTS of the transactions that created them. Now, we have the following property about SFKTO algorithm.

**Property 7** *SFKTO* algorithm ensures *starvation-freedom*.

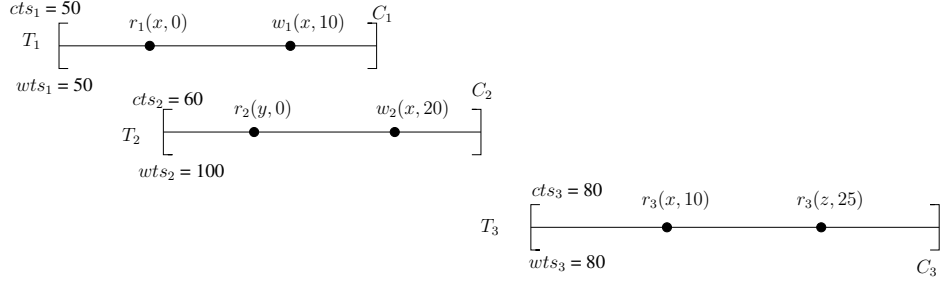


Figure 7: Correctness of SFKTO Algorithm

While the proof of this property is somewhat involved, the key idea is that the transaction with lowest ITS value, say  $T_{low}$ , will eventually have highest WTS value than all the other transactions in the system. Moreover, after a certain duration, any *new* transaction arriving in the system (i.e., whose *ITS* value sufficiently higher than that of  $T_{low}$ ) will have a lower *WTS* value than  $T_{low}$ . This will ensure that  $T_{low}$  will not be aborted. In fact, this property can be shown to be true of SFMVTO as well.

**The drawback of SFKTO:** Although SFKTO satisfies starvation-freedom, it, unfortunately, does not satisfy strict-serializability. Specifically, it violates the real-time requirement. *PKTO* uses CTS for its working while SFKTO uses WTS. It can be seen that CTS is close to the real-time execution of transactions whereas WTS of a transaction  $T_i$  is artificially inflated based on its ITS and might be much larger than its CTS. We illustrate this with an example. Consider the history  $H1$  as shown in Figure 7:  $r_1(x, 0)r_2(y, 0)w_1(x, 10)$

$C_1w_2(x, 20)C_2r_3(x, 10)r_3(z, 25)C_3$  with CTS as 50, 60 and 80 and WTS as 50, 100 and 80 for  $T_1, T_2, T_3$  respectively. Here  $T_1, T_2$  are ordered before  $T_3$  in real-time with  $T_1 \prec_{H1}^{RT} T_3$  and  $T_2 \prec_{H1}^{RT} T_3$  although  $T_2$  has a higher WTS than  $T_3$ .

Here, as per SFKTO algorithm,  $T_3$  reads  $x$  from  $T_1$  since  $T_1$  has the largest WTS (50) smaller than  $T_3$ 's WTS (80). It can be verified that it is possible for SFKTO to generate such a history. But this history is not strict-serializable. The only possible serial order equivalent to  $H1$  and legal is  $T_1T_3T_2$ . But this violates real-time order as  $T_3$  is serialized before  $T_2$  but in  $H1$ ,  $T_2$  completes before  $T_3$  has begun. Since  $H1$  is not strict-serializable, it is not locally-opaque as well. Naturally, this drawback extends to SFMVTO as well.

### 3.6 Design of *KSFTM*: Regaining Correctness while Preserving *Starvation-Freedom*

In this section, we discuss how principles of *PKTO* and SFKTO can be combined to obtain *KSFTM* that provides both correctness (strict-serializability and locally-opaque) as well as *starvation-freedom*. To achieve this, we first understand why the initial algorithm, *PKTO* satisfies strict-serializability. This is because CTS was used to create the ordering among committed transactions. CTS is closely associated with real-time. In contrast, SFKTO uses WTS which may not correspond to the real-time, as WTS may be significantly larger than CTS as shown by  $H1$  in Figure 7.

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

We can allow the transaction  $T_i$  to commit before its  $wts_i$  has caught up with the actual time if it does not violate the real-time ordering. Thus, to ensure that the notion of real-time order is respected by transactions in the course of their execution in SFKTO, we add extra time constraints. We use the idea of timestamp ranges. This notion of timestamp ranges was first used by Riegel et al. [24] in the context of multi-version STMs. Several other researchers have used this idea since then such as Guerraoui et al. [9], Crain et al. [4], Aydonat & Abdelrahman [1].

Thus, in addition to ITS, CTS and WTS, each transaction  $T_i$  maintains a timestamp range: *Transaction Lower Timestamp Limit* or  $tltl_i$ , and *Transaction Upper Timestamp Limit* or  $tutl_i$ . When a transaction  $T_i$  begins,  $tltl_i$  is assigned  $cts_i$  and  $tutl_i$  is assigned a largest possible value which we denote as infinity. When  $T_i$  executes a method  $m$  in which it reads a version of a t-object  $x$  or creates a new version of  $x$  in  $tryC$ ,  $tltl_i$  is incremented while  $tutl_i$  gets decremented <sup>1</sup>.

<sup>1</sup>Technically  $\infty$ , which is assigned to  $tutl_i$ , cannot be decremented. But here as mentioned earlier, we use  $\infty$  to denote the largest possible value that can be represented in a system.

We require to serialize all the transactions based on their WTS while maintaining their real-time order. On executing  $m$ ,  $T_i$  is ordered w.r.t to other transactions that have created a version of  $x$  based on increasing order of WTS. For all transactions  $T_j$  which also have created a version of  $x$  and whose  $wts_j$  is less than  $wts_i$ ,  $tltl_i$  is incremented such that  $tutl_j$  is less than  $tltl_i$ . Note that all such  $T_j$  are serialized before  $T_i$ . Similarly, for any transaction  $T_k$  which has created a version of  $x$  and whose  $wts_k$  is greater than  $wts_i$ ,  $tutl_i$  is decremented such that it becomes less than  $tltl_k$ . Again, note that all such  $T_k$  is serialized after  $T_i$ .

Note that in the above discussion,  $T_i$  need not have created a version of  $x$ . It could also have read the version of  $x$  created by  $T_j$ . After the increments of  $tltl_i$  and the decrements of  $tutl_i$ , if  $tltl_i$  turns out to be greater than  $tutl_i$  then  $T_i$  is aborted. Intuitively, this implies that  $T_i$ 's WTS and real-time orders are out of *sync* and cannot be reconciled.

Finally, when a transaction  $T_i$  commits: (1)  $T_i$  records its commit time (or  $comTime_i$ ) by getting the current value of  $G\_tCntr$  and incrementing it by  $incrVal$  which is any value greater than or equal to 1. Then  $tutl_i$  is set to  $comTime_i$  if it is not already less than it. Now suppose  $T_i$  occurs in real-time before some other transaction,  $T_k$  but does not have any conflict with it. This step ensures that  $tutl_i$  remains less than  $tltl_k$  (which is initialized with  $cts_k$ ); (2) Ensure that  $tltl_i$  is still less than  $tutl_i$ . Otherwise,  $T_i$  is aborted.

We illustrate this technique with the history  $H1$  shown in Figure 7. When  $T_1$  starts its  $cts_1 = 50, tltl_1 = 50, tutl_1 = \infty$ . Now when  $T_1$  commits, suppose  $G\_tCntr$  is 70. Hence,  $tutl_1$  reduces to 70. Next, when  $T_2$  commits, suppose  $tutl_2$  reduces to 75 (the current value of  $G\_tCntr$ ). As  $T_1, T_2$  have accessed a common t-object  $x$  in a conflicting manner,  $tltl_2$  is incremented to a value greater than  $tutl_1$ , say 71. Next, when  $T_3$  begins,  $tltl_3$  is assigned  $cts_3$  which is 80 and  $tutl_3$  is initialized to  $\infty$ . When  $T_3$  reads 10 from  $T_1$ , which is  $r_3(x, 10)$ ,  $tutl_3$  is reduced to a value less than  $tltl_2 (= 71)$ , say 70. But  $tltl_3$  is already at 80. Hence, the limits of  $T_3$  have crossed and thus causing  $T_3$  to abort. The resulting history consisting of only committed transactions  $T_1 T_2$  is strict-serializable.

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

Now, we describe the main idea behind *tbegin*, *stm-read*, *stm-write* and *tryC* operations of a transaction  $T_i$  which is an extension of *PKTO*. Note that as per our notation  $i$  represents the CTS of  $T_i$ .

*tbegin(t)*: A unique timestamp  $ts$  is allocated to  $T_i$  which is its CTS ( $i$  from our assumption) which is generated by atomically incrementing the global counter  $G\_tCntr$ . If the input  $t$  is null then  $cts_i = its_i = ts$  as this is the first incarnation of this transaction. Otherwise, the non-null value of  $t$  is assigned to  $its_i$ . Then, WTS is computed by Eq.(1). Finally,  $tltl$  and  $tutl$  are initialized:  $tltl_i = cts_i, tutl_i = \infty$ .

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

1. The version  $x[j]$  is created by a transaction with  $wts_j$  which is less than  $wts_i$ . Hence,  $tltl_i = \max(tltl_i, x[j].vrt + 1)$ .
2. Let  $p$  be the timestamp of smallest version larger than  $i$ . Then  $tutl_i = \min(tutl_i, x[p].vrt - 1)$ .
3. After these steps, abort  $T_i$  if  $tltl$  and  $tutl$  have crossed, i.e.,  $tltl_i > tutl_i$ .

*stm-write(x, v)*:  $T_i$  stores this write to value  $x$  locally in its  $wset_i$ .

*tryC*: This operation consists of multiple steps:

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

- (a)  $T_i$  finds a version of  $x$  with timestamp  $j$  such that  $j$  is the largest timestamp less than  $wts_i$  (like in *stm-read*). If there exists no version of  $x$  with a timestamp less than  $wts_i$  then  $T_i$  is aborted. This is similar to Step 1b of the *tryC* of *PKTO* algorithm.
- (b) Among all the transactions that have previously read from  $j$  suppose there is a transaction  $T_k$  such that  $j < wts_i < wts_k$ . Then (i) if  $T_k$  has already committed then  $T_i$  is aborted; (ii) Suppose  $T_k$  is live, and  $its_k$  is less than  $its_i$ . Then again  $T_i$  is aborted; (iii) If  $T_k$  is still live with  $its_i$  less than  $its_k$  then  $T_k$  is aborted.

This step is similar to Step 1a of the *tryC* of *PKTO* algorithm.

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

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

3. Transaction  $T_i$  is then committed.

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

### 3.7 Data Structures and Pseudocode of *KSFTM*

The STM system consists of the following methods: *init()*, *tbegin()*, *read(i, x)*, *write<sub>i</sub>(i, x, v)* and *tryC(i)*. We assume that all the t-objects are ordered as  $x_1, x_2, \dots, x_n$  and belong to the set  $\mathcal{T}$ . We describe the data-structures used by the algorithm.

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

- *G\_tCntr* (counter): This a numerical valued counter that is incremented when a transaction begins and terminates.

For each transaction  $T_i$  we maintain the following shared time-stamps:

- *G\_lock<sub>i</sub>*: A lock for accessing all the shared variables of  $T_i$ .
- *G\_its<sub>i</sub>* (initial timestamp): It is a time-stamp assigned to  $T_i$  when it was invoked for the first time without any aborts. The current value of *G\_tCntr* is atomically assigned to it and then incremented. If  $T_i$  is aborted and restarts later then the application assigns it the same *G\_its*.
- *G\_cts<sub>i</sub>* (current timestamp): It is a time-stamp when  $T_i$  is invoked again at a later time after an abort. Like *G\_its*, the current value of *G\_tCntr* is atomically assigned to it and then incremented. When  $T_i$  is created for the first time, then its *G\_cts* is same as its *G\_its*.
- *G\_wts<sub>i</sub>* (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*. It is computed as follows:  $G_wts_i = G_cts_i + C * (G_cts_i - G_its_i)$ .

- $G\_valid_i$ : This is a boolean variable which is initially true. If it becomes false then  $T_i$  has to be aborted.
- $G\_state_i$ : This is a variable which states the current value of  $T_i$ . It has three states: *live*, *committed* or *aborted*.
- $G\_ttl_i, G\_tutl_i$  (transaction lower & upper time limits): These are the time-limits described in the previous section used to keep the transaction WTS and real-time orders in sync.  $G\_ttl_i$  is  $G\_cts$  of  $T_i$  when transaction begins and is a non-decreasing value. It continues to increase (or remains same) as  $T_i$  reads t-objects and later terminates.  $G\_tutl_i$  on the other hand is a non-increasing value starting with  $\infty$  when the  $T_i$  is created. It reduces (or remains same) as  $T_i$  reads t-objects and later terminates. If  $T_i$  commits then both  $G\_ttl_i$  &  $G\_tutl_i$  are made equal.

Two transactions having the same ITS are said to be incarnations. No two transaction can have the same CTS. For simplicity, we assume that no two transactions have the same WTS as well. In case, two transactions have the same WTS, one can use the tuple  $\langle WTS, CTS \rangle$  instead of WTS. But we ignore such cases. For each t-object  $x$  in  $\mathcal{T}$ , we maintain:

- $x.vl$  (version list): It is a list consisting of version tuples or *vTuple* of the form  $\langle ts, val, rl, vrt \rangle$ . The details of the tuple are explained below.
- $ts$  (timestamp): Here  $ts$  is the  $G\_wts_i$  of a committed transaction  $T_i$  that has created this version.
- $val$ : The value of this version.
- $rl$  (readList):  $rl$  is the read list consists of all the transactions that have read this version. Each entry in this list is of the form  $\langle rts \rangle$  where  $rts$  is the  $G\_wts_j$  of a transaction  $T_j$  that read this version.
- $vrt$  (version real-time timestamp): It is the  $G\_tutl$  value (which is same as  $G\_ttl$ ) of the transaction  $T_i$  that created this version at the time of commit of  $T_i$ .

---

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

---

```

1:  $G\_tCntr = 1$ ; /* Global Transaction Counter */
2: for all  $x$  in  $\mathcal{T}$  do /* All the t-objects used by the STM System */
3:   /*  $T_0$  is creating the first version of  $x$ :  $ts = 0, val = 0, rl = nil, vrt = 0$  */
4:   add  $\langle 0, 0, nil, 0 \rangle$  to  $x.vl$ ;
5: 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  $\langle id, G\_wts, G\_cts \rangle$

---

```

1:  $i = \text{unique-id}$ ; /* An unique id to identify this transaction. It could be same as  $G\_cts$  */
2: /* Initialize transaction specific local & global variables */
3: if ( $its == nil$ ) then
4:    $G\_its_i = G\_wts_i = G\_cts_i = G\_tCntr.get\&Inc()$ ; /*  $G\_tCntr.get\&Inc()$  returns the current value of  $G\_tCntr$  and atomically increments it */
5: else
6:    $G\_its_i = its$ ;
7:    $G\_cts_i = G\_tCntr.get\&Inc()$ ;
8:    $G\_wts_i = G\_cts_i + C * (G\_cts_i - G\_its_i)$ ; /*  $C$  is any constant greater or equal to than 1 */
9: end if
10:  $G\_ttl_i = G\_cts_i$ ;  $G\_tutl_i = comTime_i = \infty$ ;
11:  $G\_state_i = live$ ;  $G\_valid_i = T$ ;
12:  $rset_i = wset_i = nil$ ;
13: return  $\langle i, G\_wts_i, G\_cts_i \rangle$ 
```

---

---

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

---

```

1: if ( $x \in wset_i$ ) then /* Check if the t-object  $x$  is in  $wset_i$  */
2:   return  $wset_i[x].val$ ;
3: else if ( $x \in rset_i$ ) then /* Check if the t-object  $x$  is in  $rset_i$  */
4:   return  $rset_i[x].val$ ;
5: else /* t-object  $x$  is not in  $rset_i$  and  $wset_i$  */
6:   lock  $x$ ; lock  $G\_lock_i$ ;
7:   if ( $G\_valid_i == F$ ) then return abort( $i$ );
8:   end if
9:   /* findLTS: From  $x.vl$ , returns the largest  $ts$  value less than  $G\_wts_i$ . If no such version exists, it returns
    $nil$  */
10:   $curVer = findLTS(G\_wts_i, x)$ ;
11:  if ( $curVer == nil$ ) then return abort( $i$ ); /* Proceed only if  $curVer$  is not  $nil$  */
12:  end if
13:  /* findSTL: From  $x.vl$ , returns the smallest  $ts$  value greater than  $G\_wts_i$ . If no such version exists, it
   returns  $nil$  */
14:   $nextVer = findSTL(G\_wts_i, x)$ ;

```

---

```

15:  if ( $nextVer \neq nil$ ) then
16:    /* Ensure that  $G\_ttl_i$  remains smaller than  $nextVer$ 's  $vrt$  */
17:     $G\_ttl_i = \min(G\_ttl_i, x[nextVer].vrt - 1)$ ;
18:  end if
19:  /*  $G\_ttl_i$  should be greater than  $x[curVer].vrt$  */
20:   $G\_ttl_i = \max(G\_ttl_i, x[curVer].vrt + 1)$ ;
21:  if ( $G\_ttl_i > G\_ttl_i$ ) then /* If the limits have crossed each other, then  $T_i$  is aborted */
22:    return abort( $i$ );
23:  end if
24:   $val = x[curVer].v$ ; add  $\langle x, val \rangle$  to  $rset_i$ ;
25:  add  $T_i$  to  $x[curVer].rl$ ;
26:  unlock  $G\_lock_i$ ; unlock  $x$ ;
27:  return  $val$ ;
28: end if

```

---



---

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

---

```

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

```

---

---

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

---

```

1: /* The following check is an optimization which needs to be performed again later */
2: lock  $G\_lock_i$ ;
3: if ( $G\_valid_i == F$ ) then return abort( $i$ );
4: end if
5: unlock  $G\_lock_i$ ;
6: /* Initialize smaller read list (smallRL), larger read list (largeRL), all read list (allRL) to nil */
7:  $smallRL = largeRL = allRL = nil$ ;
8: /* Initialize previous version list (prevVL), next version list (nextVL) to nil */
9:  $prevVL = nextVL = nil$ ;
10: for all  $x \in wset_i$  do
11:   lock  $x$  in pre-defined order;
12:   /* findLTS: returns the version of  $x$  with the largest  $ts$  less than  $G\_wts_i$ . If no such version exists, it
      returns nil. */
13:    $prevVer = findLTS(G\_wts_i, x)$ ; /* prevVer: largest version smaller than  $G\_wts_i$  */
14:   if ( $prevVer == nil$ ) then /* There exists no version with  $ts$  value less than  $G\_wts_i$  */
15:     lock  $G\_lock_i$ ; return abort( $i$ );
16:   end if
17:    $prevVL = prevVL \cup prevVer$ ; /* prevVL stores the previous version in sorted order */
18:    $allRL = allRL \cup x[prevVer].rl$ ; /* Store the read-list of the previous version */
19:   /* getLar: obtain the list of reading transactions of  $x[prevVer].rl$  whose  $G\_wts$  is greater than  $G\_wts_i$ 
      */
20:    $largeRL = largeRL \cup getLar(G\_wts_i, x[prevVer].rl)$ ;
21:   /* getSm: obtain the list of reading transactions of  $x[prevVer].rl$  whose  $G\_wts$  is smaller than  $G\_wts_i$ 
      */
22:    $smallRL = smallRL \cup getSm(G\_wts_i, x[prevVer].rl)$ ;

```

---



---

**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 ( $G\_valid_k == F$ )  $\vee$  ( $G\_state_k == abort$ )  $\vee$  ( $T_k \in abortRL$ ) 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:  $G\_valid_i = F$ ;  $G\_state_i = abort$ ;
2: unlock all variables locked by  $T_i$ ;
3: return  $\mathcal{A}$ ;

```

---

We get the following nice properties on *KSFTM*. For simplicity, we assumed  $C$  and  $incrVal$  to be 0.1 and 1 respectively in our analysis. But the proof and the analysis holds for any value greater than 0.

**Theorem 8** Any history generated by *KSFTM* is strict-serializable and locally-opaque.

**Theorem 9** *KSFTM* algorithm ensures starvation-freedom.

As explained in the description Property 7, the proof of this property is somewhat involved. As expected, this proof can be extended to *UVSFTM* as well.

**Garbage Collection:** Having described the *starvation-free* algorithm, we now describe how garbage collection can be performed on the unbounded variant, *UVSFTM* to achieve *UVSFTM-GC*. This is achieved by deleting



---

```

23:  /* findSTL: returns the version with the smallest  $ts$  value greater than  $G\_wts_i$ . If no such version exists,
    it returns  $nil$ . */
24:   $nextVer = findSTL(G\_wts_i, x)$ ; /* nextVer: smallest version larger than  $G\_wts_i$  */
25:  if ( $nextVer \neq nil$ ) then
26:     $nextVL = nextVL \cup nextVer$ ; /* nextVL stores the next version in sorted order */
27:  end if
28: end for /*  $x \in wset_i$  */
29:  $relLL = allRL \cup T_i$ ; /* Initialize relevant Lock List (relLL) */
30: for all ( $T_k \in relLL$ ) do
31:   lock  $G\_lock_k$  in pre-defined order; /* Note: Since  $T_i$  is also in  $relLL$ ,  $G\_lock_i$  is also locked */
32: end for
33: /* Verify if  $G\_valid_i$  is false */
34: if ( $G\_valid_i == F$ ) then return abort(i);
35: end if
36:  $abortRL = nil$  /* Initialize abort read list (abortRL) */
37: /* Among the transactions in  $T_k$  in  $largeRL$ , either  $T_k$  or  $T_i$  has to be aborted */
38: for all ( $T_k \in largeRL$ ) do
39:   if ( $isAborted(T_k)$ ) then
40:     /* Transaction  $T_k$  can be ignored since it is already aborted or about to be aborted */
41:     continue;
42:   end if
43:   if ( $G\_its_i < G\_its_k$ )  $\wedge$  ( $G\_state_k == live$ ) then
44:     /* Transaction  $T_k$  has lower priority and is not yet committed. So it needs to be aborted */
45:      $abortRL = abortRL \cup T_k$ ; /* Store  $T_k$  in abortRL */
46:   else /* Transaction  $T_i$  has to be aborted */
47:     return abort(i);
48:   end if
49: end for
50: /* Ensure that  $G\_ttl_i$  is greater than  $vrt$  of the versions in  $prevVL$  */
51: for all ( $ver \in prevVL$ ) do
52:    $x = t\text{-object of } ver$ ;
53:    $G\_ttl_i = \max(G\_ttl_i, x[ver].vrt + 1)$ ;
54: end for
55: /* Ensure that  $vutl_i$  is less than  $vrt$  of versions in  $nextVL$  */
56: for all ( $ver \in nextVL$ ) do
57:    $x = t\text{-object of } ver$ ;
58:    $G\_tutl_i = \min(G\_tutl_i, x[ver].vrt - 1)$ ;
59: end for
60: /* Store the current value of the global counter as commit time and increment it */
61:  $comTime_i = G\_tCntr.add\&Get(incrVal)$ ; /*  $incrVal$  can be any constant  $\geq 1$  */
62:  $G\_tutl_i = \min(G\_tutl_i, comTime_i)$ ; /* Ensure that  $G\_tutl_i$  is less than or equal to  $comTime$  */
63: /* Abort  $T_i$  if its limits have crossed */
64: if ( $G\_ttl_i > G\_tutl_i$ ) then return abort(i);
65: end if

```

---

---

```

66: for all ( $T_k \in smallRL$ ) do
67:   if ( $isAborted(T_k)$ ) then
68:     continue;
69:   end if
70:   if ( $G\_tltl_k \geq G\_tutl_i$ ) then /* Ensure that the limits do not cross for both  $T_i$  &  $T_k$  */
71:     if ( $G\_state_k == live$ ) then /* Check if  $T_k$  is live */
72:       if ( $G\_its_i < G\_its_k$ ) then
73:         /* Transaction  $T_k$  has lower priority and is not yet committed. So it needs to be aborted */
74:          $abortRL = abortRL \cup T_k$ ; /* Store  $T_k$  in abortRL */
75:       else /* Transaction  $T_i$  has to be aborted */
76:         return abort(i);
77:       end if /* ( $G\_its_i < G\_its_k$ ) */
78:     else /* ( $T_k$  is committed. Hence,  $T_i$  has to be aborted) */
79:       return abort(i);
80:     end if /* ( $G\_state_k == live$ ) */
81:   end if /* ( $G\_tltl_k \geq G\_tutl_i$ ) */
82: end for ( $T_k \in smallRL$ )
83: /* After this point  $T_i$  can't abort. */
84:  $G\_tltl_i = G\_tutl_i$ ;
85: /* Since  $T_i$  can't abort, we can update  $T_k$ 's  $G\_tutl$  */
86: for all ( $T_k \in smallRL$ ) do
87:   if ( $isAborted(T_k)$ ) then
88:     continue;
89:   end if
90:   /* The following line ensure that  $G\_tltl_k \leq G\_tutl_k < G\_tltl_i$ . Note that this does not cause the limits of
      $T_k$  to cross each other because of the check in Line 70.*/
91:    $G\_tutl_k = \min(G\_tutl_k, G\_tltl_i - 1)$ ;
92: end for
93: for all  $T_k \in abortRL$  do /* Abort all the transactions in abortRL since  $T_i$  can't abort */
94:    $G\_valid_k = F$ ;
95: end for
96: /* Having completed all the checks,  $T_i$  can be committed */
97: for all ( $x \in wset_i$ ) do
98:   /* Create new  $v\_tuple$ :  $ts, val, rl, vrt$  for  $x$  */
99:    $newTuple = \langle G\_wts_i, wset_i[x].val, nil, G\_tltl_i \rangle$ ;
100:  if ( $|x.vl| > k$ ) then
101:    replace the oldest tuple in  $x.vl$  with  $newTuple$ ; /*  $x.vl$  is ordered by  $ts$  */
102:  else
103:    add a  $newTuple$  to  $x.vl$  in sorted order;
104:  end if
105: end for /*  $x \in wset_i$  */
106:  $G\_state_i = commit$ ;
107: unlock all variables;
108: return  $\mathcal{C}$ ;

```

---

non-latest version (i.e., there exists a version with greater  $ts$ ) of each t-object whose timestamp,  $ts$  is less than the CTS of smallest live transaction. It must be noted that *UVSFTM* (*KSFTM*) works with WTS which is greater or equal to CTS for any transaction. Interestingly, the same garbage collection principle can be applied for *PMVTO* to achieve *PMVTO-GC*.

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

## 4 Experimental Evaluation

In this section, we have evaluated the performance of *KSFTM* with state-of-the-art STM systems. We have compared *KSFTM* with *UVSFTM*, *UVSFTM-GC*, *PMVTO*, *PMVTO-GC*, *PKTO*, NOrec STM by Dalessandro et al. [5], Elastic STM (ESTM) by Felber et al. [6], Multi-Version timestamp order STM (MVTO) by Kumar et al. [18] and *SV-SFTM* based on priority principle as described in Section 1 [8, 27, 26]. We have used the implementation of NOrec STM, MVTO, and ESTM published by the authors where all are in C++. We implement the other algorithms - *UVSFTM*, *UVSFTM-GC*, *KSFTM*, *PMVTO*, *PMVTO-GC*, *PKTO*, *SV-SFTM* in C++.

Our experimental goals are to: (G1) identify the best value of  $K$  (i.e., number of versions) in the propose algorithm *KSFTM*; (G2) to evaluate the performance of all our proposed algorithms (*KSFTM*, *UVSFTM*, *UVSFTM-GC*, *PKTO*, *PMVTO*, *PMVTO-GC*); (G3) to evaluate the overhead of *starvation-freedom* by comparing the performance of *KSFTM* and *non-starvation-free PKTO* STM, and (G4) to compare the performance of *KSFTM* with state-of-the-art STMs (NOrec, ESTM, MVTO, and *SV-SFTM*) on different workloads.

**Experimental system:** The experimental system is a 2-socket Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz with 14 cores per socket and 2 hyper-threads (HTs) per core, for a total of 56 threads. Each core has a private 32KB L1 cache and 256 KB L2 cache (which is shared among HTs on that core). All cores on a socket share a 35MB L3 cache. The machine has 32GB of RAM and runs Ubuntu 16.04.2 LTS. All code was compiled with the GNU C++ compiler (G++) 5.4.0 with the build target x86\_64-Linux-gnu and compilation option `-std=c++1x -O3`.

**Methodology:** To test the algorithms, we consider a test-application in which each thread invokes a single transaction. Within a transaction, a thread performs 10 reads/writes operations on randomly chosen t-objects from a set of 1000 t-objects. A thread continues to invoke a transaction until it successfully commits. We have considered three types of workloads: (W1) Li - Lookup intensive(90% read, 10% write), (W2) Mi - Mid intensive(50% read, 50% write) and (W3) Ui - Update intensive(10% read, 90% write). For accurate results, we took an average of ten runs as the final result for each algorithm. In all the plots, the x-axis is the number of threads and the y-axis is the time taken for each thread to invoke a transaction until it commits.

**Results Analysis:** To identify the best value of  $K$  for *KSFTM*, we run our experiment, varying value of  $K$  and keeping the number of threads as 64 on workload  $W1$ . Figure 9 (d) shows that the optimal value of  $K$  in *KSFTM* is 5. Similarly, we calculate the best value of  $K$  as 5 for *PKTO* on the same parameters. The optimal value of  $C$  as 0.1 for *KSFTM* on the above parameters.

Figure 8 (a), (c) and (e) represent the performance analysis among variants of *KSFTM* (*UVSFTM*, *UVSFTM-GC*, and *KSFTM*) and Figure 8 (b), (d) and (f) represent the performance analysis among variants of *PKTO* (*PMVTO*, *PMVTO-GC*, and *PKTO*) on different workloads  $W1$  (Li),  $W2$  (Mi) and  $W3$  (Ui) respectively. *KSFTM* outperforms *UVSFTM* and *UVSFTM-GC* by a factor of 2.1 and 1.5. Similarly, *PKTO* outperforms *PMVTO* and *PMVTO-GC* by a factor of 2 and 1.35. These results show that maintaining (optimal number of) finite versions corresponding to each t-object performs better than maintaining infinite versions. And even implementing garbage collection on infinite versions corresponding to each t-object under performs finite version implementation.

Figure 9 (a), (b) and (c) shows the performance of all the algorithms (proposed as well as state-of-the-art STMs) for workloads  $W1$ ,  $W2$  and  $W3$  respectively. For workload  $W1$ , the graph shows that *KSFTM* outperforms *SV-SFTM*, NOrec, ESTM, and MVTO by a factor of 2.5, 3, 1.7 and 1.5. For workload  $W2$ , *KSFTM* exceeds *SV-SFTM*, NOrec, ESTM, and MVTO by a factor of 1.5, 2, 1.6 and 1.3 respectively. For workload  $W3$ , *KSFTM* again beats *SV-SFTM*, NOrec, ESTM, and MVTO by 1.7, 3.3, 3 and 1.4 at thread count 64. So, *KSFTM* outperforms all the other STM algorithms in low as well as high contention.

Even though *KSFTM* performs better than *SV-SFTM*, NOrec, ESTM and MVTO. *KSFTM*'s performance is comparable to *PKTO* in all workloads. In fact, in all workloads, the performance of *KSFTM* is only 2% less than *PKTO*. But as discussed in SubSection 3.3, that the transactions can possibly starve with *PKTO* while this is not the case with *KSFTM*. We believe that this is the overhead that one has to pay to achieve *starvation-freedom*. On

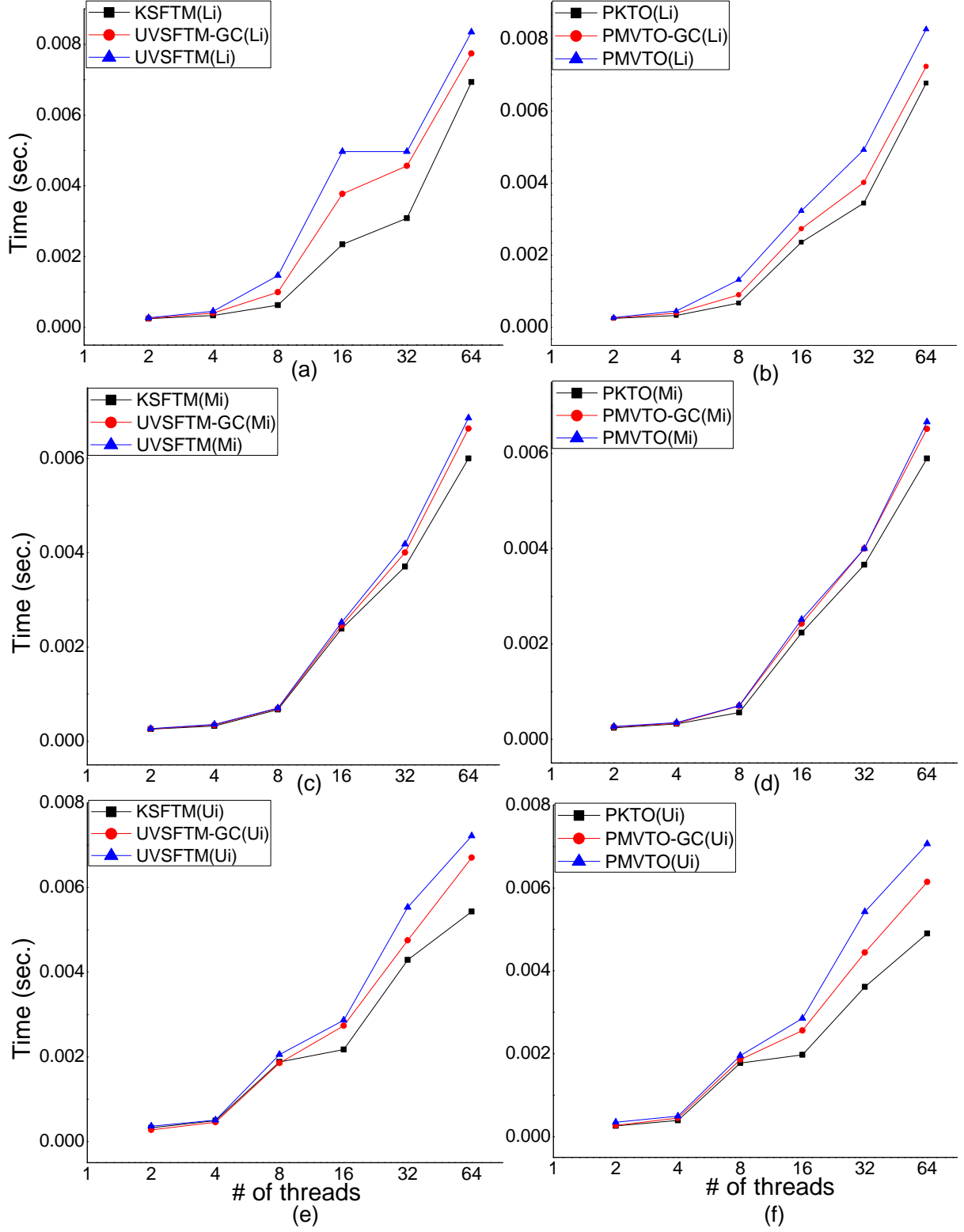


Figure 8: Execution under variants of *KSFTM* and *PKTO*

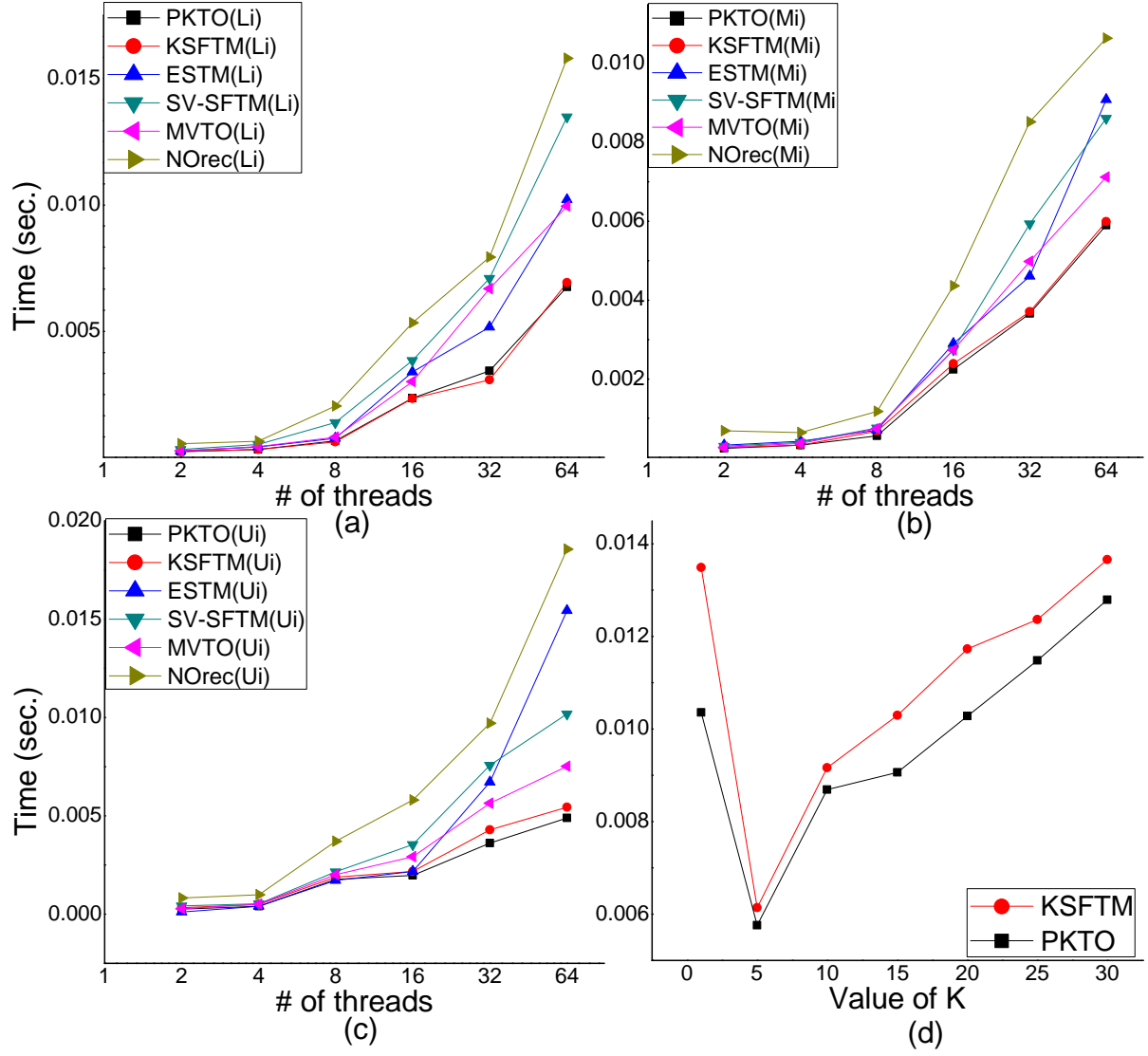


Figure 9: Performance on workload  $W1$ ,  $W2$ ,  $W3$  and Optimal value of  $K$  as 5

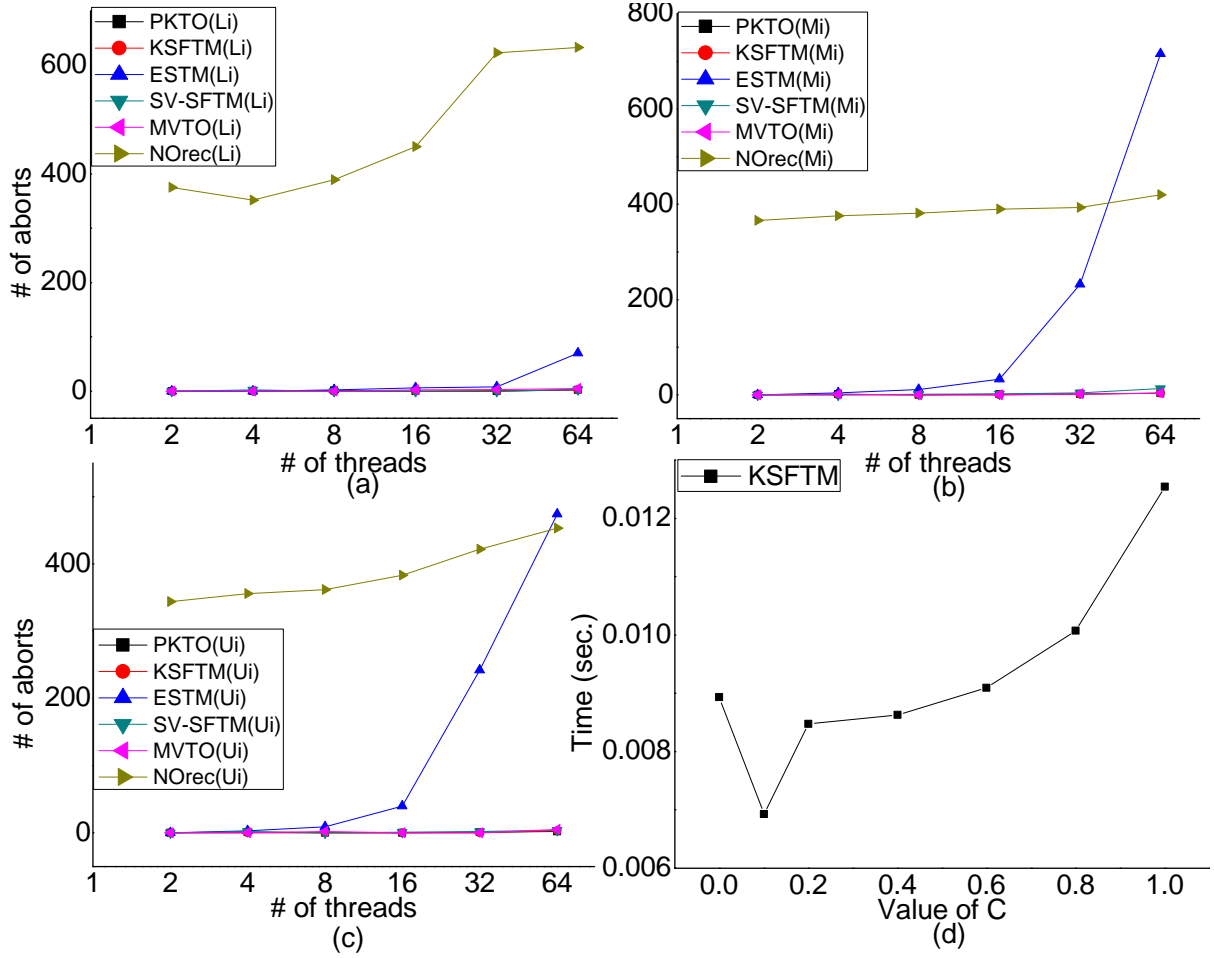


Figure 10: Aborts on workload  $W1, W2, W3$  and Optimal value of  $C$  as 0.1

the positive side, the overhead is very small being only around 2% as compared to *PKTO*. On the other hand, the performance of *KSFTM* is much better than single-version starvation-free algorithm *SV-SFTM*.

**Comparison on the basis of Abort count:** Figure 10 (a), (b) and (c) shows the abort count comparisons of *KSFTM* with *PKTO*, *ESTM*, *NOrec*, *MVTO*, and *SV-SFTM* across all workloads ( $W1, W2$ , and  $W3$ ). The number of aborts in *ESTM* and *NOrec* are high as compared to all other STM algorithms while all other algorithms (*KSFTM*, *PKTO*, *MVTO*, *SV-SFTM*) have marginally small differences among them.

**An optimal value of constant  $C$ :**  $C$ , is a constant that is used to calculate  $WTS$  of a transaction. i.e.,  $wts_i = cts_i + C * (cts_i - its_i)$ ; where,  $C$  is any constant greater than 0. We run our experiments across load  $W1$ , for 64 threads and other parameters are same as defined in the methodology of Section 4, we achieve the best value of  $C$  as 0.1. Experimental results are shown in Figure 10 (d).

## 5 Graph Characterization of Local Opacity & *KSFTM* Correctness

To prove correctness of STM systems, it is useful to consider graph characterization of histories. In this section, we describe the graph characterization developed by Kumar et al [18] for proving opacity which is based on characterization by Bernstein and Goodman [2]. We extend this characterization for LO.

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

Consider the history  $H2 : r_1(x, 0)r_2(x, 0)r_1(y, 0)r_3(z, 0)w_1(x, 5)w_3(y, 15)w_2(y, 10)w_1(z, 10)c_1c_2r_4(x, 5)r_4(y, 10)w_3(z, 15)c_3r_4(z, 10)$ . Using the notation that a committed transaction  $T_i$  writing to  $x$  creates

a version  $x_i$ , a possible version order for  $H2 \ll_{H2}$  is:  $\langle x_0 \ll x_1 \rangle, \langle y_0 \ll y_2 \ll y_3 \rangle, \langle z_0 \ll z_1 \ll z_3 \rangle$ .

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

1. *real-time*(real-time) edges: If  $T_i$  commits before  $T_j$  starts in  $H$ , then there is an edge from  $v_i$  to  $v_j$ . This set of edges are referred to as  $rt(H)$ .
2. *rf*(reads-from) edges: If  $T_j$  reads  $x$  from  $T_i$  in  $H$ , then there is an edge from  $v_i$  to  $v_j$ . Note that in order for this to happen,  $T_i$  must have committed before  $T_j$  and  $c_i <_H r_j(x)$ . This set of edges are referred to as  $rf(H)$ .
3. *mv*(multiversion) edges: The mv edges capture the multiversion relations and is based on the version order. Consider a successful read operation  $r_k(x, v)$  and the write operation  $w_j(x, v)$  belonging to transaction  $T_j$  such that  $r_k(x, v)$  reads  $x$  from  $w_j(x, v)$  (it must be noted  $T_j$  is a committed transaction and  $c_j <_H r_k$ ). Consider a committed transaction  $T_i$  which writes to  $x$ ,  $w_i(x, u)$  where  $u \neq v$ . Thus the versions created  $x_i, x_j$  are related by  $\ll$ . Then, if  $x_i \ll x_j$  we add an edge from  $v_i$  to  $v_j$ . Otherwise  $(x_j \ll x_i)$ , we add an edge from  $v_k$  to  $v_i$ . This set of edges are referred to as  $mv(H, \ll)$ .

Using the construction, the  $OPG(H2, \ll_{H2})$  for history  $H2$  and  $\ll_{H2}$  is shown in Figure 11. The edges are annotated. The only mv edge from  $T4$  to  $T3$  is because of t-objects  $y, z$ .  $T4$  reads value 5 for  $z$  from  $T1$  whereas  $T3$  also writes 15 to  $z$  and commits before  $r_4(z)$ .

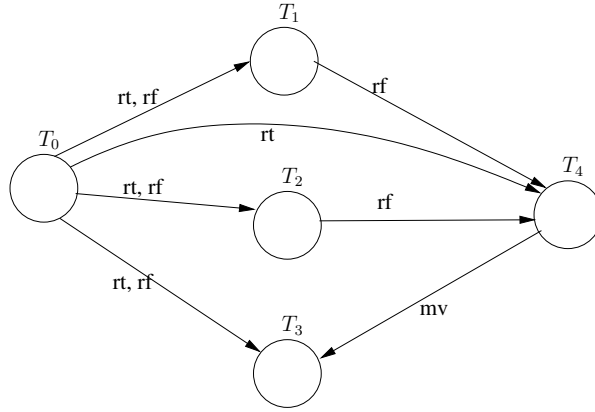


Figure 11:  $OPG(H2, \ll_{H2})$

Kumar et al [18] showed that if a version order  $\ll$  exists for a history  $H$  such that  $OPG(H, \ll_H)$  is acyclic, then  $H$  is opaque. This is captured in the following result.

**Result 10** A valid history  $H$  is opaque iff there exists a version order  $\ll_H$  such that  $OPG(H, \ll_H)$  is acyclic.

This result can be easily extended to prove LO as follows

**Theorem 11** A valid history  $H$  is locally-opaque iff for each sub-history  $sh$  in  $H.subhistSet$  there exists a version order  $\ll_{sh}$  such that  $OPG(sh, \ll_{sh})$  is acyclic. Formally,  $\langle (H \text{ is locally-opaque}) \Leftrightarrow (\forall sh \in H.subhistSet, \exists \ll_{sh}: OPG(sh, \ll_{sh}) \text{ is acyclic}) \rangle$ .

**Proof.** To prove this theorem, we have to show that each sub-history  $sh$  in  $H.subhistSet$  is valid. Then the rest follows from Result 10. Now consider a sub-history  $sh$ . Consider any read operation  $r_i(x, v)$  of a transaction  $T_i$ . It is clear that  $T_i$  must have read a version of  $x$  created by a previously committed transaction. From the construction of  $sh$ , we get that all the transaction that committed before  $r_i$  are also in  $sh$ . Hence  $sh$  is also valid.

Now, proving  $sh$  to be opaque iff there exists a version order  $\ll_{sh}$  such that  $OPG(sh, \ll_{sh})$  is acyclic follows from Result 10.

**Lemma 12** 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  $G\_ttl_i < G\_ttl_j$ .

**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 38 ( $G\_ttl_i \leq comTime_i$ ). So, ( $G\_ttl_i \leq CTS_j$ ).  
We know from STM  $tbegin(its)$  method,  $G\_ttl_j = G\_cts_j$ .  
Eventually,  $G\_ttl_i < G\_ttl_j$ .
2. Read-from edge: Since, transaction  $T_i$  has been committed and  $T_j$  is reading from  $T_i$  so, from Line 99  $tryC(T_i)$ ,  $G\_ttl_i = vrt_i$ .  
and from Line 20 STM  $read(j, x)$ ,  $G\_ttl_j = max(G\_ttl_j, x[curVer].vrt + 1) \Rightarrow (G\_ttl_j > vrt_i) \Rightarrow (G\_ttl_j > G\_ttl_i)$   
Hence,  $G\_ttl_i < G\_ttl_j$ .
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)  $i \ll j \Rightarrow G\_wts_i < G\_wts_j$   
There are two possibilities of commit order:
    - i.  $comTime_i <_H comTime_j$ : Since,  $T_i$  has been committed before  $T_j$  so  $G\_ttl_i = vrt_i$ . From Line 53 of  $tryC(T_j)$ ,  $vrt_i < G\_ttl(j)$ .  
Hence,  $G\_ttl_i < G\_ttl_j$ .
    - ii.  $comTime_j <_H comTime_i$ : Since,  $T_j$  has been committed before  $T_i$  so  $G\_ttl_j = vrt_j$ . From Line 58 of  $tryC(T_i)$ ,  $G\_ttl_i < vrt_j$ . As we have assumed  $G\_valid_i$  is true so definitely it will execute the Line 84  $tryC(T_i)$  i.e.  $G\_ttl_i = G\_ttl_i$ .  
Hence,  $G\_ttl_i < G\_ttl_j$ .
  - (b)  $j \ll i \Rightarrow G\_wts_j < G\_wts_i$   
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 45 to Line 47 of  $tryC(i)$ , either transaction  $T_k$  or  $T_i$ ,  $G\_valid$  flag is set to be false. If  $T_i$  returns abort then this case will not be considered in Lemma 12. Otherwise, as  $T_j$  has already been committed and later  $T_i$  will execute the Line 99  $tryC(T_i)$ , Hence,  $G\_ttl_j < G\_ttl_i$ .
      - B.  $G\_wts_k < G\_wts_i$ : That means  $T_k$  is in smallRL of  $T_i$ . From Line 17 of  $read(k, x)$ ,  $G\_ttl_k < vrt_i$  and from Line 20 of  $read(k, x)$ ,  $G\_ttl_k > vrt_j$ . Here,  $T_j$  has already been committed so,  $G\_ttl_j = vrt_j$ . As we have assumed  $G\_valid_i$  is true so definitely it will execute the Line 99  $tryC(T_i)$ ,  $G\_ttl_i = vrt_i$ .  
So,  $G\_ttl_k < G\_ttl_i$  and  $G\_ttl_k > G\_ttl_j$ . While considering  $G\_valid_k$  flag is true  $\rightarrow G\_ttl_k < G\_ttl_k$ .  
Hence,  $G\_ttl_j < G\_ttl_k < G\_ttl_i$ .  
Therefore,  $G\_ttl_j < G\_ttl_k < G\_ttl_i$ .
    - ii.  $comTime_i <_H comTime_j$ : Since,  $T_i$  has been committed before  $T_j$  so,  $G\_ttl_i = vrt_i$ . From Line 58 of  $tryC(T_j)$ ,  $G\_ttl_j < vrt_i$  i.e.  $G\_ttl_j < G\_ttl_i$ . Here,  $T_k$  read from  $T_j$ . So, From Line 17 of  $read(k, x)$ ,  $G\_ttl_k < vrt_i \rightarrow G\_ttl_k < G\_ttl_i$  from Line 20 of  $read(k, x)$ ,  $G\_ttl_k > vrt_j$ . As we have assumed  $G\_valid_j$  is true so definitely it will execute the Line 99  $tryC(T_j)$ ,  $G\_ttl_j = vrt_j$ .  
Hence,  $G\_ttl_j < G\_ttl_k < G\_ttl_i$ .  
Therefore,  $G\_ttl_j < G\_ttl_k < G\_ttl_i$ .

**Theorem 13** Any history  $H$  in  $gen(KSFTM)$  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 12, 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  $G\_ttl_i < G\_ttl_j$ . While considering transitive case for  $k$  transactions  $T_1, T_2, T_3 \dots T_k$  such that  $G\_valid$  flags of all the transactions are true. if there is an edge from  $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \dots \rightarrow T_k$  then  $G\_ttl_1 < G\_ttl_2 < G\_ttl_3 < \dots < G\_ttl_k$ .

Now, considering our assumption,  $MVSG(H, \ll)$  has cycle so,  $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \dots \rightarrow T_k \rightarrow T_1$  that implies  $G\_ttl_1 < G\_ttl_2 < G\_ttl_3 < \dots < G\_ttl_k < G\_ttl_1$ .

Hence from above assumption,  $G\_ttl_1 < G\_ttl_1$  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 sequential because operations of the transactions are overlapping. The correctness of STMs is defined on sequential history, in order to show history generated by our algorithm is correct we have to consider sequential history. We have enough information to order the overlapping methods, after ordering the operations will have equivalent sequential 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 14** 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 assumed 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 15** 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 10, it returns the largest  $t_s$  value less than  $G\_wts_i$  that has already been committed and return the value successfully. If such version created by transaction  $T_j$  found then  $T_i$  read from  $T_j$ . Otherwise, if there is no version whose WTS is less than  $T_i$ 's WTS, then  $T_i$  returns abort.

Now, consider the base case read( $i, x$ ) is the first transaction  $T_1$  and none of the transactions has been created a version then as we have assumed, 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  $t_s$  value less than  $G\_wts_i$  that has already been committed. Hence,  $(H, \alpha)$  is valid.

**Lemma 16** 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  $\prec_{(H, \alpha)}^{RT} = \prec_{(H, \beta)}^{RT}$ .

**Proof.** Consider a history  $H$   $\text{gen}(\text{KSFTM})$  such that two transactions  $T_i$  and  $T_j$  are in real time order which respects  $M\_Order_H$  i.e.  $\text{try}C_i < \text{tbegin}_j$ . As  $\alpha$  and  $\beta$  are linearizations of  $H$  so,  $\text{try}C_i <_{(H,\alpha)} \text{tbegin}_j$  and  $\text{try}C_i <_{(H,\beta)} \text{tbegin}_j$ . Hence in both the cases of linearizations,  $T_i$  committed before begin of  $T_j$ . So,  $\prec_{(H,\alpha)}^{RT} = \prec_{(H,\beta)}^{RT}$ .

**Lemma 17** Any history  $H$   $\text{gen}(\text{KSFTM})$  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  $H$   $\text{gen}(\text{KSFTM})$  so, from Lemma 15  $(H, \alpha)$  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  $(\bar{H}, \alpha)$ . As we know  $\beta$  is a linearization of  $H$  so  $(\bar{H}, \beta)$  is equivalent to some legal t-sequential history  $S$ . From the definition of local opacity  $\prec_{(H,\alpha)}^{RT} \subseteq \prec_S^{RT}$ . From Lemma 16,  $\prec_{(H,\alpha)}^{RT} = \prec_{(H,\beta)}^{RT}$  that implies  $\prec_{(H,\beta)}^{RT} \subseteq \prec_S^{RT}$ . 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.

**Theorem 18** Any history generated by  $\text{KSFTM}$  is locally-opaque.

**Proof.** For proving this, we consider a sequential history  $H$  generated by  $\text{KSFTM}$ . We define the version order  $\ll_{\text{vrt}}$ : for two versions  $v_i, v_j$  it is defined as

$$(v_i \ll_{\text{vrt}} v_j) \equiv (v_i.\text{vrt} < v_j.\text{vrt})$$

Using this version order  $\ll_{\text{vrt}}$ , we can show that all the sub-histories in  $H.\text{subhistSet}$  are acyclic.

Since the histories generated by  $\text{KSFTM}$  are locally-opaque, we get that they are also strict-serializable.

**Corollary 19** Any history generated by  $\text{KSFTM}$  is strict-serializable.

## 6 Proof of Liveness

**Proof Notations:** Let  $\text{gen}(\text{KSFTM})$  consist of all the histories accepted by  $\text{KSFTM}$  algorithm. In the follow sub-section, we only consider histories that are generated by  $\text{KSFTM}$  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  $\text{KSFTM}$ . Once it executes  $\text{tbegin}$  method, its ITS, CTS, WTS values do not change. Thus, we denote them as  $\text{its}_i, \text{cts}_i, \text{wts}_i$  respectively for  $T_i$ . In case the context of the history  $H$  in which the transaction executing is important, we denote these variables as  $H.\text{its}_i, H.\text{cts}_i, H.\text{wts}_i$  respectively.

The other variables that a transaction maintains are:  $\text{ttl}$ ,  $\text{tutl}$ ,  $\text{lock}$ ,  $\text{valid}$ ,  $\text{state}$ . These values change as the execution proceeds. Hence, we denote them as:  $H.\text{ttl}_i, H.\text{tutl}_i, H.\text{lock}_i, H.\text{valid}_i, H.\text{state}_i$ . These represent the values of  $\text{ttl}$ ,  $\text{tutl}$ ,  $\text{lock}$ ,  $\text{valid}$ ,  $\text{state}$  after the execution of last event in  $H$ . Depending on the context, we sometimes ignore  $H$  and denote them only as:  $\text{lock}_i, \text{valid}_i, \text{state}_i, \text{ttl}_i, \text{tutl}_i$ .

We approximate the system time with the value of  $\text{tCntr}$ . We denote the sys-time of history  $H$  as the value of  $\text{tCntr}$  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  $H.\text{its}_i = H.\text{its}_j$ . 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,

$$H.\text{incarSet}(T_i) = \{T_j | (T_i = T_j) \vee (H.\text{its}_i = H.\text{its}_j)\}$$

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,  $H.\text{incarSet}(T_i) \in H.\text{txns}$ .

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  $T_j$  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 20** 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 21** 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 22** 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  $T_i$ . 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 23** 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 24** 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 25** 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 26** Consider a 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 27** 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

$$H1.wts_i < H2.wts_j \quad (2)$$

The definition of WTS of  $T_i$  is:  $H1.wts_i = H1.cts_i + C * (H1.cts_i - H1.its_i)$ . Combining this Eq.(2), we get that

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

**Lemma 28** 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  $T_2$ .

On the other hand if  $T_i$  is aborted, as seen in Observation 25 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 29** 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,  $\langle T_i, H1 : (T_i \in H.txns) \implies (H.cts_i \leq H.sys-time) \rangle$ .

**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.,  $(H.cts_i \leq H.sys-time)$

From this lemma, we get the following corollary which is the converse of the lemma statement

**Corollary 30** 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,  $\langle T_i, H1, H2 : (H1 \sqsubset H2) \wedge (T_i \notin H1.txns) \wedge (T_i \in H2.txns) \implies (H2.cts_i > H1.sys-time) \rangle$ .

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

**Lemma 31** 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 t-objects in a pre-defined order and then lock relevant G\_locks again in a predefined order. Since all the locks are obtained in the same order, it can be seen that the methods of *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 t-objects in the write-set of  $T_i$ . Moreover, since we assume that the underlying scheduler is fair, we can see that no thread gets swapped out infinitely. Finally, since we assume that all the locks are fair, it can be seen all the methods terminate in finite time.

**Theorem 32** Every transaction either commits or aborts in finite time.

**Proof.** This theorem comes directly from the Lemma 31. 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 33** 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 34** Consider a history  $H1$  with two transactions  $T_i, T_j$ . Let  $T_i$  be in  $H1.live$ . Suppose  $T_j$ 's WTS is greater or equal to  $T_i$ 's WTS. Then ITS of  $T_j$  is less than  $its_i + 2 * L$ . Formally,  $\langle H, T_i, T_j : (\{T_i, T_j\} \subseteq H.txns) \wedge (T_i \in H.live) \wedge (H.wts_j \geq H.wts_i) \implies (H.its_i + 2L \geq H.its_j) \rangle$ .

**Proof.** Since  $T_i$  is live in  $H1$ , from Corollary 33, 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.

$$\begin{aligned} wts_j - wts_i &= (cts_j + C * (cts_j - its_j)) - (cts_i + C * (cts_i - its_i)) \\ &= (C + 1)(cts_j - cts_i) - C(its_j - its_i) \\ &\leq (C + 1)L - C(its_j - its_i) \quad [\because cts_j \leq cts_i + L] \\ &= 2 * L + its_i - its_j \quad [\because C = 1] \end{aligned}$$

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

From the above implication we get that,  $\langle wts_j \geq wts_i \rangle \implies \langle its_i + 2L \geq its_j \rangle$ .

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

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

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

**Lemma 36** 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,  $\langle H1 \sqsubseteq H2 \rangle \wedge (T_i \in H1.live) \wedge (T_i \in H2.live) \wedge (H1.itsEnabled(T_i)) \implies (H2.itsEnabled(T_i))$ .

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

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

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

**Lemma 37** 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,  $\langle H, \{T_i, T_j\} \in H.txns : (tryC_i <_H term-op_j) \implies (comTime_i + incrVal \leq comTime_j) \rangle$ .

**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  $G\_ttl$  and  $comTime$  of a transaction  $T_i$ .

**Lemma 38** Consider a history  $H$  with a transaction  $T_i$  in it. Then in  $H$ ,  $ttl_i$  will be less than or equal to  $comTime_i$ . Formally,  $\langle H, \{T_i\} \in H.txns : (H.ttl_i \leq H.comTime_i) \rangle$ .

**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  $(ttl_i \leq comTime_i)$ . On executing Line 61,  $comTime_i$  gets assigned to some finite value and it does not change after that.

It can be seen that  $ttl_i$  gets initialized to  $cts_i$  in Line 4 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  $cts_i (= ttl_i \text{ initially}) < comTime_i$ . Then  $ttl_i$  gets updated on Line 20 of `read`, Line 53 and Line 84 of `tryC` methods. Let us analyze them case by case assuming that  $ttl_i$  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  $ttl_i$  updated. Here  $ttl_i$  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,

$$ttl_i = 1 + x[j].vrt \quad (3)$$

It can be seen that  $x[j].vrt$  is same as  $ttl_j$  when  $T_j$  executed Line 99 of `tryC`. Further,  $ttl_j$  in turn is same as  $tutl_j$  due to Line 84 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

$$x[j].vrt = ttl_j = tutl_j \leq comTime_j \quad (4)$$

It is clear that from the above discussion that  $T_j$  executed `tryC` method before  $T_i$  terminated (i.e.  $tryC_j <_{H1} term-op_i$ ). From Eq.(3) and Eq.(4), we get

$$ttl_i \leq 1 + comTime_j \xrightarrow{incrVal \geq 1} ttl_i \leq incrVal + comTime_j \xrightarrow{\text{Lemma 37}} ttl_i \leq comTime_i$$

2. Line 53 of `tryC` method: The reasoning in this case is very similar to the above case.
3. Line 84 of `tryC` method: In this line,  $ttl_i$  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  $ttl_i \leq comTime_i$ . It can be seen that the reasoning here is similar in part to Case 1.

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

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 39** 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  $x[nextVer].vrt = tttl_j$  (from Line 99). From Lemma 38, we get that  $tttl_j \leq comTime_j$ . This implies that  $\alpha < comTime_j$ . Now, we have that  $T_j$  has already committed before the termination of  $T_i$ . Thus from Lemma 37, we get that  $comTime_j < comTime_i$ . Hence, we have that,

$$\alpha < comTime_i \quad (5)$$

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  $comTime_i = tutl_i$ . From Eq.(5), we can ignore this case. (ii) Suppose the value of  $tutl_i$  before executing Line 62 was  $\alpha$ . Then from Eq.(5) we get that  $tutl_i$  remains at  $\alpha$  on execution of Line 62. This implies that a transaction  $T_j$  committed such that  $wts_j > wts_i$ . The following lemma connects the  $G\_tutl$  of a committed transaction  $T_j$  and  $comTime$  of a transaction  $T_i$  that commits later.

**Lemma 40** 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  $H2$ ,  $tutl_j$  is less than or equal to  $comTime_i$ . Formally,  $\langle H1, H2 \in gen(KSFTM), \{T_i, T_j\} \subseteq H1, H2.trans : (H1 \sqsubseteq H2) \wedge (T_j \in H1.committed) \wedge (T_i \in H1.live) \implies (H2.tutl_j < H2.comTime_i) \rangle$ .

**Proof.** As observed in the previous proof of Lemma 38, 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 38, we get that  $tutl_j \leq comTime_j$ . This implies that  $tutl_j < comTime_i$ .

In the following sequence of lemmas, we identify the condition by when a transaction will commit.

**Lemma 41** 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,  $\langle H1, H3, T_i : (H1 \sqsubset H3) \wedge (T_i \in H1.live) \wedge (H1.valid_i = True) \wedge (H1.itsEnabled(T_i)) \wedge (T_i \in H3.aborted) \rangle \implies \langle \exists H2, T_j : (H1 \sqsubseteq H2 \sqsubset H3) \wedge (T_i \in H2.live) \wedge (T_j \in H2.trans) \wedge (H2.wts_i < H2.wts_j) \wedge (T_j \in H3.committed) \rangle$ .

**Proof.** To show this lemma, w.l.o.g we assume that  $T_i$  on executing either read or tryC in  $H2$  (which could be same as  $H1$ ) gets aborted resulting in  $H3$ . Thus, we have that  $T_i$  is live in  $H2$ . Here  $T_i$  is itsEnabled in  $H1$ . From Lemma 36, we get that  $T_i$  is itsEnabled in  $H2$  as well.

Let us sequentially consider all the lines where a  $T_i$  could abort. In  $H2$ ,  $T_i$  executes one of the following lines and is aborted in  $H3$ . We start with 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, 74 where  $T_i$  is added to  $T_j$ 's abortRL set. Later in Line 94,  $T_i$ 's valid flag is set to false. Note that  $T_i$ 's valid is true (after the execution of the last event) in  $H1$ . Thus,  $T_i$ 's valid flag must have been set to false in an extension of  $H1$ , which we again denote as  $H2$ .

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

- (b) Line 15: This line is executed in  $H2$  if there exists no version of  $x$  whose  $\text{ts}$  is less than  $T_i$ 's WTS. This implies that all the versions of  $x$  have  $\text{ts}$  greater than  $\text{wts}_i$ . Thus the transactions that created these versions have WTS greater than  $\text{wts}_i$  and have already committed in  $H2$ . Let  $T_j$  create one such version. Hence, we have that  $\langle (T_j \in H2.\text{committed}) \implies (T_j \in H3.\text{committed}) \rangle$  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  $\text{tutl}_i$ . It is initialized in `begin` method to  $\infty$ . As long as it is  $\infty$ , these limits cannot cross each other. Later,  $\text{tutl}_i$  is updated in Line 17 of `read` method, Line 58 & Line 62 of `tryC` method. Suppose  $\text{tutl}_i$  gets decremented to some value  $\alpha$  by one of these lines.  
Now there are two cases here: (1) Suppose  $\text{tutl}_i$  gets decremented to  $\text{comTime}_i$  due to Line 62 of `tryC` method. Then from Lemma 38, we have  $\text{tutl}_i \leq \text{comTime}_i = \text{tutl}_i$ . Thus in this case,  $T_i$  will not abort. (2)  $\text{tutl}_i$  gets decremented to  $\alpha$  which is less than  $\text{comTime}_i$ . Then from Lemma 39, we get that there is a committed transaction  $T_j$  in  $H2.\text{committed}$  such that  $\text{wts}_j > \text{wts}_i$ . This implies that  $T_j$  is in  $H3.\text{committed}$ .
- (f) Line 76: This case is similar to Case 1a, i.e., Line 3.
- (g) Line 79 : In this case,  $T_k$  is in  $T_i$ 's smallRL and is committed in  $H1$ . And, from this case, we have that

$$H2.\text{tutl}_i \leq H2.\text{tutl}_k \quad (6)$$

From the assumption of this case, we have that  $T_k$  commits before  $T_i$ . Thus, from Lemma 40, we get that  $\text{comTime}_k < \text{comTime}_i$ . From Lemma 38, we have that  $\text{tutl}_k \leq \text{comTime}_k$ . Thus, we get that  $\text{tutl}_k < \text{comTime}_i$ . Combining this with the inequality of this case Eq.(6), we get that  $\text{tutl}_i < \text{comTime}_i$ .

Combining this inequality with Lemma 39, we get that there is a transaction  $T_j$  in  $H2.\text{committed}$  and  $H2.\text{wts}_j > H2.\text{wts}_i$ . This implies that  $T_j$  is in  $H3.\text{committed}$  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 notions of enabled, we in turn define a few other auxiliary notions. We start with *affectSet*,

$$H.\text{affectSet}(T_i) = \{T_j | (T_j \in H.\text{txns}) \wedge (H.\text{its}_j < H.\text{its}_i + 2 * L)\}$$

From the description of *KSFTM* algorithm and Lemma 34, it can be seen that a transaction  $T_i$ 's commit can depend on committing of transactions (or their incarnations) which have their ITS less than ITS of  $T_i + 2*L$ , which is  $T_i$ 's *affectSet*. We capture this notion of dependency for a transaction  $T_i$  in a history  $H$  as *commit dependent set* or *cds* as: the set of all transactions  $T_j$  in  $T_i$ 's *affectSet* that do not any incarnation that is committed yet, i.e., not yet have their *incarCt* flag set as true. Formally,

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

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

**Definition 42** 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,

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

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 43** 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 44** 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

$$H1.cts_i \geq H1.its_i + 2 * L \quad (7)$$

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

$$H2.its_k < H2.its_j + 2 * L \quad (8)$$

Since  $T_i$  and  $T_j$  are incarnations of each other, their ITS are the same. Combining this with Eq.(8), we get that

$$H2.its_k < H1.its_i + 2 * L \quad (9)$$

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

- No incarnation of  $T_k$  is in  $H1$ : This implies that  $T_k$  starts afresh after  $H1$ . Since  $T_k$  is not in  $H1$ , from Corollary 30 we get that

$$\begin{aligned} H2.cts_k &> H1.sys-time \xrightarrow[H2.cts_k = H2.its_k]{T_k \text{ starts afresh}} H2.its_k > H1.sys-time \xrightarrow[H1.sys-time \geq H1.cts_i]{(T_i \in H1) \wedge \text{Lemma 29}} H2.its_k > \\ H1.cts_i &\xrightarrow{Eq.(7)} H2.its_k > H1.its_i + 2 * L \xrightarrow[H1.its_i = H2.its_j]{H1.its_i = H2.its_j} H2.its_k > H2.its_j + 2 * L \end{aligned}$$

But this result contradicts with Eq.(8). Hence, this case is not possible.

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

$$H1.its_l = H2.its_k \quad (10)$$

Now combing this result with Eq.(9), we get that  $H1.its_l < H1.its_i + 2 * L$ . This implies that  $T_l$  is in *affectSet* of  $T_i$  in  $H1$ . Since  $T_i$  is *cdsEnabled*, we get that  $T_l$ 's *incarCt* must be true.

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

Thus, we have that,  $(H1 \sqsubset H2) \wedge (H1.incarCt(T_l)) \wedge (T_k \in H2.txns) \wedge (T_k \notin H1.txns)$ . Combining these with Lemma 22, we get that  $(H1.its_l \neq H2.its_k)$ . But this result contradicts Eq.(10). Hence, this case is also not possible.

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

**Lemma 45** Consider two histories  $H1, H2$  where  $H2$  is an extension of  $H1$ . Let  $T_i, T_j, T_k$  be three transactions such that  $T_i$  is in  $H1.txns$  while  $T_j, T_k$  are in  $H2.txns$ . Suppose we have that (1)  $cts_i$  is greater than  $its_i + 2 * L$  in  $H1$ ; (2)  $T_i$  is an incarnation of  $T_j$ ; (3)  $T_k$  is in  $affectedSet$  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,  $\langle H1, H2, T_i, T_j, T_k : (H1 \sqsubseteq H2) \wedge (T_i \in H1.txns) \wedge (\{T_j, T_k\} \in H2.txns) \wedge (H1.cts_i > H1.its_i + 2 * L) \wedge (T_i \in H2.incarSet(T_j)) \wedge (T_k \in H2.affectedSet(T_j)) \implies (\exists T_l : (T_l \in H2.incarSet(T_k)) \wedge (T_l \in H1.txns)) \rangle$

**Proof.**

This proof is similar to the proof of Lemma 44. We are given that

$$H1.cts_i \geq H1.its_i + 2 * L \quad (11)$$

We now show this proof through contradiction. Suppose no incarnation of  $T_k$  is in  $H1.txns$ . This implies that  $T_k$  must have started afresh in some history  $H3$  which is an extension of  $H1$ . Also note that  $H3$  could be same as  $H2$  or a prefix of it, i.e.,  $H3 \sqsubseteq H2$ . Thus, we have that

$$\begin{aligned} H3.its_k > H1.sys-time &\xrightarrow{\text{Lemma 29}} H3.its_k > H1.cts_i \xrightarrow{\text{Eq.(11)}} H3.its_k > H1.its_i + 2 * L \xrightarrow{H1.its_i = H2.its_j} \\ H3.its_k > H2.its_j + 2 * L &\xrightarrow[\text{Observation 24}]{H3 \sqsubseteq H2} H2.its_k > H2.its_j + 2 * L \xrightarrow[\text{definition}]{affectedSet} T_k \notin H2.affectedSet(T_j) \end{aligned}$$

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

**Lemma 46** 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  $affectedSet$  of  $T_i$  in  $H1$  is same as the  $affectedSet$  of  $T_j$  in  $H2$ . Formally,  $\langle H1, H2, T_i, T_j : (H1 \sqsubseteq H2) \wedge (H1.cdsEnabled(T_i)) \wedge (T_j \in H2.txns) \wedge (T_i \in H2.incarSet(T_j)) \implies ((H1.affectedSet(T_i) = H2.affectedSet(T_j))) \rangle$

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

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

$$T_k \in H1.txns \quad (12)$$

Combining Eq.(12) with Observation 24, we get that

$$T_k \in H2.txns \quad (13)$$

From the definition of ITS, we get that

$$H1.its_k = H2.its_k \quad (14)$$

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

$$H1.its_i = H2.its_j \quad (15)$$

From the definition of  $affectedSet$ , we get that,

$$H1.its_k < H1.its_i + 2 * L \xrightarrow{\text{Eq.(14)}} H2.its_k < H1.its_i + 2 * L \xrightarrow{\text{Eq.(15)}} H2.its_k < H2.its_j + 2 * L$$

Combining this result with Eq.(13), we get that  $T_k \in H2.affectedSet(T_j)$ .

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

Here, we have that  $(H1 \sqsubseteq H2) \wedge (H1.cdsEnabled(T_i)) \wedge (T_i \in H2.incarSet(T_j)) \wedge (T_k \in H2.affectedSet(T_j))$ . Thus from Lemma 44, we get that  $T_k \in H1.txns$ . Now, this case is similar to the above case. It can be seen that Equations 12, 13, 14, 15 hold good in this case as well.

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

$$H2.its_k < H2.its_j + 2 * L \xrightarrow{\text{Eq.(14)}} H1.its_k < H2.its_j + 2 * L \xrightarrow{\text{Eq.(15)}} H1.its_k < H1.its_i + 2 * L$$

Combining this result with Eq.(12), we get that  $T_k \in H1.affectedSet(T_i)$ .

Next we explore how a  $cdsEnabled$  transaction remains  $cdsEnabled$  in the future histories once it becomes true.

**Lemma 47** 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

$$H1.cts_i \geq H2.its_i + 2 * L \quad (16)$$

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

$$\begin{aligned} H2.cts_j &> H1.cts_i \\ &> H1.its_i + 2 * L && \text{[From Eq.(16)]} \\ &> H2.its_j + 2 * L && [its_i = its_j] \end{aligned}$$

Thus we get that  $cts_j > its_j + 2 * L$ . We have that  $T_j$  is live in  $H2$ . In order to show that  $T_j$  is  $cdsEnabled$  in  $H2$ , it only remains to show that  $cds$  of  $T_j$  in  $H2$  is empty, i.e.,  $H2.cds(T_j) = \phi$ . The  $cds$  becomes empty when all the transactions of  $T_j$ 's  $affectedSet$  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 46, we get that  $H1.affectedSet(T_i) = H2.affectedSet(T_j)$ .

Now, consider a transaction  $T_k$  in  $H2.affectedSet(T_j)$ . From the above result, we get that  $T_k$  is also in  $H1.affectedSet(T_i)$ . Since  $T_i$  is  $cdsEnabled$  in  $H1$ , i.e.,  $H1.cdsEnabled(T_i)$  is true, we get that  $H1.incarCt(T_k)$  is true. Combining this with Observation 21, we get that  $T_k$  must have its  $incarCt$  as true in  $H2$  as well, i.e.,  $H2.incarCt(T_k)$ . This implies that all the transactions in  $T_j$ 's  $affectedSet$  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 48** For any transaction  $T_i$  in  $H$ , we have that  $wts_i$  is less than or equal to  $H.maxWTS(T_i)$ . Formally,  $H.wts_i \leq H.maxWTS(T_i)$ .

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

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

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

**Lemma 49** 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  $affectedSet$  of  $T_i$  in  $H1$  is same as  $affectedSet$  of  $T_j$  in  $H2$ . Then the  $affWTS$  of  $T_i$  in  $H1$  is same as  $affWTS$  of  $T_j$  in  $H2$ . Formally,  $\langle H1, H2, T_i, T_j : (H1 \sqsubseteq H2) \wedge (T_i \in H1.txns) \wedge (T_j \in H2.txns) \wedge (H1.affectedSet(T_i) = H2.affectedSet(T_j)) \implies (H1.affWTS(T_i) = H2.affWTS(T_j)) \rangle$ .

**Proof.**

From the definition of  $affWTS$ , we get the following equations

$$H.affWTS(T_i) = \max\{H.maxWTS(T_k) | (T_k \in H1.affectedSet(T_i))\} \quad (17)$$

$$H.affWTS(T_j) = \max\{H.maxWTS(T_i) \mid (T_i \in H2.affectSet(T_j))\} \quad (18)$$

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

Here, we are given that  $H1.affectSet(T_i) = H2.affectSet(T_j)$ . Hence, we get that  $T_p$  is also in  $H1.affectSet(T_i)$ . Similarly,  $T_q$  is in  $H2.affectSet(T_j)$  as well. Thus from Equations (17) & (18), we get that

$$H1.maxWTS(T_p) \geq H2.maxWTS(T_q) \quad (19)$$

$$H2.maxWTS(T_q) \geq H1.maxWTS(T_p) \quad (20)$$

Combining these both equations, we get that  $H1.maxWTS(T_p) = H2.maxWTS(T_q)$  which in turn implies that  $H1.affWTS(T_i) = H2.affWTS(T_j)$ .

Finally, using the notion of affWTS and cdsEnabled, we define the notion of *finEnabled*

**Definition 50** 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* in  $H$ ; (3)  $H.wts_j$  is greater than  $H.affWTS(T_i)$ . Formally,

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

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 51** 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,  $(H1, H2, T_i, T_j : (H1 \sqsubseteq H2) \wedge (T_i \in H1.live) \wedge (T_j \in H2.live) \wedge (T_i \in H2.incarSet(T_j)) \wedge (H1.cts_i < H2.cts_j) \wedge (H1.finEnabled(T_i)) \implies (H2.finEnabled(T_j)))$ .

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

$$\langle (H1 \sqsubseteq H2) \wedge (T_i \in H1.live) \wedge (T_j \in H2.live) \wedge (T_i \in H2.incarSet(T_j)) \wedge (H1.cts_i < H2.cts_j) \wedge (H1.cdsEnabled(T_i)) \rangle \quad (21)$$

Combining Eq.(21) with Lemma 47, we get that  $T_j$  is *cdsEnabled* in  $H2$ , i.e.,  $H2.cdsEnabled(T_j)$ . Now, in order to show that  $T_j$  is *finEnabled* in  $H2$  it remains for us to show that  $H2.wts_j > H2.affWTS(T_j)$ .

We are given that  $T_j$  is live in  $H2$  which in turn implies that  $T_j$  is in  $H2.txns$ . Thus changing this in Eq.(21), we get the following

$$\langle (H1 \sqsubseteq H2) \wedge (T_j \in H2.txns) \wedge (T_i \in H2.incarSet(T_j)) \wedge (H1.cts_i < H2.cts_j) \wedge (H1.cdsEnabled(T_i)) \rangle \quad (22)$$

Combining Eq.(22) with Lemma 46 we get that

$$H1.affWTS(T_i) = H2.affWTS(T_j) \quad (23)$$

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

$$H1.wts_i < H2.wts_j \quad (24)$$

Since  $T_i$  is *finEnabled* in  $H1$ , we have that

$$H1.wts_i > H1.affWTS(T_i) \xrightarrow{Eq.(24)} H2.wts_j > H1.affWTS(T_i) \xrightarrow{Eq.(23)} H2.wts_j > H2.affWTS(T_j)$$

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

**Lemma 52** Consider a live transaction  $T_i$  in a history  $H1$ . Suppose  $T_i$  is *finEnabled* in  $H1$  and  $valid_i$  is true in  $H1$ . Then there exists an extension of  $H1$ ,  $H3$  in which  $T_i$  is committed. Formally,  $\langle H1, T_i : (T_i \in H1.live) \wedge (H1.valid_i) \wedge (H1.finEnabled(T_i)) \implies (\exists H3 : (H1 \sqsubseteq H3) \wedge (T_i \in H3.committed)) \rangle$ .

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

Now consider  $T_i$  in  $H1$ :  $T_i$  is live; its valid flag is true; and is *finEnabled*. From the definition of *finEnabled*, we get that it is also *cdsEnabled*. From Lemma 43, we get that  $T_i$  is *itsEnabled* in  $H1$ . Thus from Lemma 41, 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,

$$\langle (\exists H2, T_j : (H1 \sqsubseteq H2 \sqsubseteq H3) \wedge (T_i \in H2.live) \wedge (T_j \in H2.txns) \wedge (H2.wts_i < H2.wts_j) \wedge (T_j \in H3.committed)) \rangle \quad (25)$$

Here, we have that  $H2$  is an extension of  $H1$  with  $T_i$  being live in both of them and  $T_i$  is *finEnabled* in  $H1$ . Thus from Lemma 51, we get that  $T_i$  is *finEnabled* in  $H2$  as well. Now, let us consider  $T_j$  in  $H2$ . From Eq.(25), we get that  $(H2.wts_i < H2.wts_j)$ . Combining this with the observation that  $T_i$  being live in  $H2$ , Lemma 34 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

$$(H2.affWTS(T_i) \geq H2.maxWTS(T_j)) \quad (26)$$

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

$$(H2.wts_i > H2.affWTS(T_i)) \quad (27)$$

Now combining Equations 26, 27 we get,

$$\begin{aligned} H2.wts_i &> H2.affWTS(T_i) \geq H2.maxWTS(T_j) \\ &> H2.affWTS(T_i) \geq H2.maxWTS(T_j) \geq H2.wts_j \quad [\text{From Observation 48}] \\ &> H2.wts_j \end{aligned}$$

But this equation contradicts with Eq.(25). Hence our assumption that  $T_i$  will get aborted in  $H3$  after getting *finEnabled* is not possible. Thus  $T_i$  has to commit in  $H3$ .

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

Now, we show that a transaction that is *itsEnabled* will become *cdsEnabled* or committed. To show this, we introduce a few more notations and definitions. We start with the notion of *depIts* (*dependent-its*) which is the set of 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 53** 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 30 we get the following

$$\begin{aligned} H3.its_k > H1.sys-time &\xrightarrow{\text{Lemma 29}} H3.its_k > H1.cts_i \implies H3.its_k > H1.its_i + 2 * L \xrightarrow{H1.its_i = H2.its_j} \\ H3.its_k > H2.its_j + 2 * L &\xrightarrow[\text{definitions}]{\text{affectSet, depIts}} H2.its_k \notin H2.depIts(T_j) \end{aligned}$$

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 54** 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:

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

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

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

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

$$|H1.depIts(T_i)| = k + 1 \tag{28}$$

Let  $\alpha$  be  $H1.partAffWTS(T_i)$ . Suppose  $H1.wts_i < \alpha$ . Then from Lemma 28, 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 41, 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 38 we get that in  $H4$ ,

$$H4.its_m \leq H4.its_l + 2 * L \tag{29}$$



Eq.(29) implies that  $T_m$  is in  $T_l$ 's affectSet. Here, we have that  $T_l$  is an incarnation of  $T_i$  and we are given that  $H1.cts_i \geq H1.its_i + 2 * L$ . Thus from Lemma 45, we get that there exists an incarnation of  $T_m, T_n$  in  $H1$ .

Combining Eq.(29) with the observations (a)  $T_n, T_m$  are incarnations; (b)  $T_l, T_i$  are incarnations; (c)  $T_i, T_n$  are in  $H1.txns$ , we get that  $H1.its_n \leq H1.its_i + 2 * L$ . This implies that  $T_n$  is in  $H1.affectSet(T_i)$ . Since  $T_n$  is not committed in  $H1$  (otherwise, it is not possible for  $T_m$  to be an incarnation of  $T_n$ ), we get that  $T_n$  is in  $H1.cds(T_i)$ . Hence, we get that  $H4.its_m = H1.its_n$  is in  $H1.depIts(T_i)$ .

From Eq.(28), we have that  $H1.depIts(T_i)$  is  $k+1$ . From Lemma 53, 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

$$|H4.depIts(T_i)| \leq k \implies |H4.depIts(T_l)| \leq k \quad (30)$$

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

$$H4.cts_l > H4.cts_i \xrightarrow{\text{given}} H4.cts_l > H4.its_i + 2 * L \xrightarrow{H4.its_i = H4.its_l} H4.cts_l > H4.its_l + 2 * L \quad (31)$$

We also have that  $T_l$  is live in  $H4$ . Combining this with Equations 30, 31 and given the induction hypothesis that  $P(k)$  is true, we get that there exists a history extension of  $H4, H6$  in which an incarnation of  $T_l$  (also  $T_i$ ),  $T_p$  is either committed or cdsEnabled. This proves the lemma.

**Lemma 55** 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,  $\langle H1, T_i : (T_i \in H1.live) \wedge (H1.cdsEnabled(T_i)) \implies (\exists H2, T_j : (H1 \sqsubset H2) \wedge (T_j \in H2.incarSet(T_i)) \wedge ((T_j \in H2.committed) \vee (H2.finEnabled(T_j)))) \rangle$ .

**Proof.** In  $H1$ , suppose  $H1.affWTS(T_i)$  is  $\alpha$ . From Lemma 28, 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,  $wtss_j$  is greater than  $\alpha$ .

In the second case, we get that

$$(T_i \in H1.live) \wedge (T_j \in H2.live) \wedge (H.cdsEnabled(T_i)) \wedge (T_j \in H2.incarSet(T_i)) \wedge (H1.wtss_i < H2.wtss_j) \quad (32)$$

Combining the above result with Lemma 27, we get that  $H1.cts_i < H2.cts_j$ . Thus the modified equation is

$$(T_i \in H1.live) \wedge (T_j \in H2.live) \wedge (H1.cdsEnabled(T_i)) \wedge (T_j \in H2.incarSet(T_i)) \wedge (H1.cts_i < H2.cts_j) \quad (33)$$

Next combining Eq.(33) with Lemma 46, we get that

$$H1.affectSet(T_i) = H2.affectSet(T_j) \quad (34)$$

Similarly, combining Eq.(33) with Lemma 47 we get that  $T_j$  is cdsEnabled in  $H2$  as well. Formally,

$$H2.cdsEnabled(T_j) \quad (35)$$

Now combining Eq.(34) with Lemma 49, we get that

$$H1.affWTS(T_i) = H2.affWTS(T_j) \quad (36)$$

From our initial assumption we have that  $H1.affWTS(T_i)$  is  $\alpha$ . From Eq.(36), we get that  $H2.affWTS(T_j) = \alpha$ . Further, we had earlier also seen that  $H2.wtss_j$  is greater than  $\alpha$ . Hence, we have that  $H2.wtss_j > H2.affWTS(T_j)$ . Combining the above result with Eq.(35),  $H2.cdsEnabled(T_j)$ , we get that  $T_j$  is finEnabled, i.e.,  $H2.finEnabled(T_j)$ .

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

**Lemma 56** 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,  $\langle H1, T_i : (T_i \in H1.live) \implies (\exists T_j, H2 : (H1 \sqsubset H2) \wedge (T_j \in H2.incarSet(T_i)) \wedge (T_j \in H2.committed) \vee (H2.itsEnabled(T_i))) \rangle$ .

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

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

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

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

**Theorem 57** 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,  $\langle H1, T_i : (T_i \in H1.live) \implies (\exists T_j, H2 : (H1 \sqsubset H2) \wedge (T_j \in H2.incarSet(T_i)) \wedge (T_j \in H2.committed)) \rangle$ .

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

Consider a live transaction  $T_i$  in  $H1$ . Then from Lemma 56, we get that there is a history  $H2$ , which is an extension of  $H1$ , in which  $T_j$  an incarnation of  $T_i$  is either committed or *itsEnabled*. If  $T_j$  is *itsEnabled* in  $H2$ , then from Lemma 54, we get that  $T_k$ , an incarnation of  $T_j$ , will be *cdsEnabled* in a extension of  $H2$ ,  $H3$  (assuming that  $T_k$  is not committed in  $H3$ ).

From Lemma 55, we get that there is an extension of  $H3$ ,  $H4$  in which an incarnation of  $T_k$ ,  $T_l$  will be *finEnabled* assuming that it is not committed in  $H4$ . Finally, from Lemma 52, we get that there is an extension of  $H4$  in which  $T_m$ , an incarnation of  $T_l$ , will be committed. This proves our theorem.

## 7 Discussion and Conclusion

In this paper, we propose a  $K$  version *starvation-free* STM system, *KSFTM*. The algorithm ensures that if an *aborted* transaction is retried successively, then it will eventually commit. The algorithm maintains  $K$  versions where  $K$  can range from between one to infinity. For correctness, we show *KSFTM* satisfies strict-serializability [22] and local opacity [19, 20]. To the best of our knowledge, this is the first work to explore *starvation-freedom* with MVSTMs.

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

In this document, we have not considered a transactional solution based on two-phase locking (2PL) and its multi-version variants [28]. With the carefully designed 2PL solution, one can ensure that none of the transactions abort [28]. But this will require advance knowledge of the code of the transactions which may not always be available with the STM library. Without such knowledge, it is possible that a 2PL solution can deadlock and cause further aborts which will, raise the issue of *starvation-freedom* again.

Since we have considered strict-serializable as one of the *correctness-criteria*, this algorithm can be extended to databases as well. In fact, to the best of our knowledge, there has been no prior work on *starvation-freedom* in the context of database concurrency control.

## References

- [1] Utku Aydonat and Tarek S. Abdelrahman. Relaxed Concurrency Control in Software Transactional Memory. *IEEE Trans. Parallel Distrib. Syst.*, 23(7):1312–1325, July 2012.
- [2] Philip A. Bernstein and Nathan Goodman. Multiversion Concurrency Control: Theory and Algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, December 1983.

- [3] Victor Bushkov, Rachid Guerraoui, and Michal Kapalka. On the liveness of transactional memory. In *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 9–18, 2012.
- [4] Tyler Crain, Damien Imbs, and Michel Raynal. Read invisibility, virtual world consistency and probabilistic permissiveness are compatible. In *ICA3PP (1)*, pages 244–257, 2011.
- [5] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: Streamlining stm by abolishing ownership records. *PPoPP* 2010, 2010.
- [6] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic transactions. *J. Parallel Distrib. Comput.*, 100(C):103–127, February 2017.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] Rachid Guerraoui and Michal Kapalka. On the Correctness of Transactional Memory. In *PPoPP*, pages 175–184, New York, NY, USA, 2008. ACM.
- [11] Rachid Guerraoui and Michal Kapalka. *Principles of Transactional Memory, Synthesis Lectures on Distributed Computing Theory*. Morgan and Claypool, 2010.
- [12] Maurice Herlihy. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.
- [13] 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.
- [14] Maurice Herlihy and Nir Shavit. On the nature of progress. In *Proceedings of the 15th International Conference on Principles of Distributed Systems, OPODIS'11*, pages 313–328, Berlin, Heidelberg, 2011. Springer-Verlag.
- [15] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.
- [16] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
- [17] Damien Imbs, José Ramon de Mendivil, and Michel Raynal. Brief announcement: virtual world consistency: a new condition for STM systems. In *PODC*, pages 280–281, New York, NY, USA, 2009. ACM.
- [18] Priyanka Kumar, Sathya Peri, and K. Vidyasankar. A TimeStamp Based Multi-version STM Algorithm. In *ICDCN*, pages 212–226, 2014.
- [19] Petr Kuznetsov and Sathya Peri. Non-interference and Local Correctness in Transactional Memory. In *ICDCN*, pages 197–211, 2014.
- [20] Petr Kuznetsov and Sathya Peri. Non-interference and local correctness in transactional memory. *Theor. Comput. Sci.*, 688:103–116, 2017.
- [21] 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.
- [22] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

- [23] Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. SMV: Selective Multi-Versioning STM. In *DISC*, pages 125–140, 2011.
- [24] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*, pages 284–298, 2006.
- [25] Nir Shavit and Dan Touitou. Software Transactional Memory. In *PODC*, pages 204–213, New York, NY, USA, 1995. ACM.
- [26] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory, 2009.
- [27] M. M. Waliullah and Per Stenström. Schemes for Avoiding Starvation in Transactional Memory Systems. *Concurrency and Computation: Practice and Experience*, 21(7):859–873, 2009.
- [28] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.