Noname manuscript No.
(will be inserted by the editor)

Achieving Starvation-Freedom in Multi-Version Transactional Memory Systems*

Ved Prakash Chaudhary · Chirag Juyal · Sandeep Kulkarni · Sweta Kumari · Sathya Peri**

Received: date / Accepted: date

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. Client programs use STMs by issuing transactions. STM ensures that transaction either commits or aborts. A transaction aborted due to conflicts is typically re-issued with the expectation that it will complete successfully in a subsequent incarnation. However, many existing STMs fail to provide starvation freedom, i.e., in these systems, it is possible that concurrency conflicts may prevent an incarnated transaction from committing. To overcome this limitation, we systematically derive a novel starvation free algorithm for multi-version STM. Our algorithm can be used either with the case where the number of versions is unbounded and garbage collection is used or where only the latest K versions are maintained, KSFTM. We have demonstrated that our proposed algorithm performs better than existing state-of-the-art STMs.

Keywords Software Transactional Memory System · Concurrency Control · Starvation-Freedom · Opacity · Local Opacity · Multi-Version

* A preliminary version of this paper appeared in 8th International Conference On Networked Systems (NETYS 2019). A part of this work was submitted in IIT Hyderabad, India towards the fulfillment of Ph.D. thesis requirement by an author Sweta Kumari.

** Author sequence follows the lexical order of last names.

Ved Prakash Chaudhary
Department of CSE, Indian Institute of Technology, Hyderabad E-mail: cs14mtech11019@iith.ac.in

Chirag Juyal
Department of CSE, Indian Institute of Technology, Hyderabad E-mail: cs17mtech11014@iith.ac.in

Sandeep Kulkarni
Department of Computer Science, Michigan State University, USA E-mail: sandeep@cse.msu.edu

Sweta Kumari
Department of Computer Science, Technion, Israel E-mail: sweta@cs.technion.ac.il

Sathya Peri
Department of CSE, Indian Institute of Technology, Hyderabad E-mail: sathya_p@cs.iith.ac.in

1 Introduction

STMs [16, 28] 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 writes 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: *stm-begin* which begins a transaction, *stm-read* which reads a *transactional object* or *t-object*, *stm-write* which writes to a *t-object*, *stm-tryC* which tries to commit the transaction. Typical code for using STMs is as shown in Algorithm 1 which shows how an insert of a concurrent linked-list library is implemented using STMs.

Correctness: Several *correctness-criteria* have been proposed for STMs such as opacity [13], local opacity [21, 22]. All these *correctness-criteria* require that all the transactions including the aborted ones appear to execute sequentially in an order that agrees with the order of non-overlapping transactions. Unlike the correctness-criteria for traditional databases, such as serializability, strict-serializability [25], 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 by transaction that get aborted later such as divide-by-zero, infinite-loops, crashes etc. in the application due to concurrent executions. This additional requirement on aborted transactions is a fundamental requirement of STMs which differentiates STMs from databases as observed by Guerraoui & Kapalka [13]. Thus in this paper, we focus on optimistic executions with the *correctness-criterion* being *local opacity* [22].

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.

```
retry = 0;
while (true) do
    id = stm-begin (retry);
    ...
    v = stm-read(id, x);   ▷ reads value of x as v
    ...
    stm-write(id, x, v');   ▷ writes a value v' to x
    ...
    ret = stm-tryC(id);   ▷ stm-tryC can return commit or abort
    if (ret == commit) then break;
    else retry++;
    end if
end while
```

Starvation Freedom: In the execution shown in Algorithm 1, there is a possibility that the transaction which a thread tries to execute gets aborted again and again. Every time, it executes the transaction, say T_i, T_i conflicts with some other transaction and hence gets aborted. In other words, the thread is effectively starved because it is not able to commit T_i successfully.

A well known blocking progress condition associated with concurrent programming is starvation-freedom [18, chap 2], [17]. 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

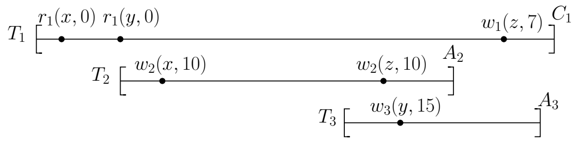

Fig. 1: Limitation of Single-version Starvation Free Algorithm

fair underlying scheduler with bounded termination) and T_i is not *parasitic*, i.e., T_i will try to commit given a chance then T_i will eventually commit. Parasitic transactions [4] will not commit even when given a chance to commit possibly because they are caught in an infinite loop or some other error.

Wait-freedom is another interesting progress condition for STMs in which every transaction commits regardless of the nature of concurrent transactions and the underlying scheduler [17]. But it was shown by Guerraoui and Kapalka [4] 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 while assuming that the data sets of the transactions are *not* known in advance.

Related work on the starvation-free STMs: Starvation-freedom in STMs has been explored by a few researchers such as Gramoli et al. [12], Waliullah and Stenstrom [30], Spear et al. [29]. Most of these systems work by assigning priorities to transactions. In case of a conflict between two transactions, the transaction with lower priority is aborted. They ensure that every aborted transaction, on being retried a sufficient number of times, will eventually have the highest priority and hence will commit. We denote such an algorithm as *single-version starvation-free STM* or *SV-SFTM*.

Although *SV-SFTM* guarantees starvation-freedom, it can still abort many transactions spuriously. Consider the case where a transaction T_i has the highest priority. Hence, as per *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.

Fig 1 illustrates this problem. Consider the execution: $r_1(x,0)r_1(y,0)w_2(x,10)w_2(z,10)w_3(y,15)w_1(z,7)$. It has three transactions T_1, T_2 and T_3. Let T_1 have 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's write operations are in conflict with T_1's read operations. Since T_1 has higher priority and has not committed yet, T_2 and T_3 have to *abort*. If these transactions are retried and again conflict with T_1 (while it is still live), they will have to *abort* again. Thus, any transaction with priority lower than T_1 and conflicts with it has to abort. It is as if T_1 has locked the t-objects x, y and does not allow any other transaction, write to these t-objects and to *commit*.

Multi-version starvation-free STM: A key limitation of single-version STMs is limited concurrency. As shown above, it is possible that one long transaction conflicts with several transactions causing them to abort. This limitation can be overcome by using multi-version STMs where we store multiple versions of the data item (either unbounded versions with garbage collection, or bounded versions where the oldest version is replaced when the number of versions exceeds the bound).

Several multi-version STMs have been proposed in the literature [20, 23, 11, 26] that provide increased concurrency. But none of them provide starvation-freedom. Suppose the execution shown in Fig 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 $T_1T_2T_3$ or $T_1T_3T_2$. Thus multiple versions can help with starvation-freedom without sacrificing on concurrency. This motivated us to develop a multi-version starvation-free STM system.

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

This paper addresses this gap by developing a starvation-free algorithm for bounded MVSTMs. Our approach is different from the approach used in *SV-SFTM* to provide starvation-freedom in single version STMs (the policy of aborting lower priority transactions in case of conflict) as it does not work for MVSTMs. As part of the derivation of our final starvation-free algorithm, we consider an algorithm *PKTO* (*Priority-based K-version Timestamp Order*) that considers this approach and show that it is insufficient to provide starvation freedom.

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 each t-object and can range from 1 to ∞. To the best of our knowledge, this is the first starvation-free MVSTM. We develop *KSFTM* algorithm in a step-wise manner starting from MVTO [20] *(Multi-Version Timestamp Order)* as follows:
  - First, in Section 3.3, we use the standard idea to provide higher priority to older transactions. Specifically, we propose 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* (Section 3.4), 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* (Section 3.5) that preserves the starvation-freedom, strict-serializability, and local opacity. Our algorithm works on the assumption that any transaction that is not deadlocked, terminates (commits or aborts) in a bounded time.
- Our experiments (Section 4) show that *KSFTM* gives an average speedup on the worst-case time to commit of a transaction by a factor of 1.22, 1.89, 23.26, and

13.12 times over *PKTO*, *SV-SFTM*, NOrec STM [8] and ESTM [10] respectively for counter application. *KSFTM* performs 1.5 and 1.44 times better than *PKTO* and *SV-SFTM* but 1.09 times worse than NOrec for low contention KMEANS application of STAMP [24] benchmark whereas *KSFTM* performs 1.14, 1.4, and 2.63 times better than *PKTO*, *SV-SFTM* and NOrec for LABYRINTH application of STAMP benchmark which has high contention with long-running transactions.
Summary of Differences with Chaudhary et. al [6]:
- We perform a few more experiments (see Section 4 and Appendix A.9) to analyze the performance of proposed *KSFTM* with state-of-the-art STMs. We have analyzed the following:
  - Max-time analysis on low contention for counter application.
  - Identify the optimal value of K and C for *KSFTM* and *PKTO*.
  - Average time analysis on STAMP benchmark.
  - Calculates the number of aborts on low as well as high contention.
  - Average time analysis and memory consumption on the variants of *PKTO* and *KSFTM*.
- We have included the detailed related work section in Appendix A.2 (due to lack of space).
- We rigorously prove the safety and liveness of our proposed *KSFTM* in Appendix A.7 and Appendix A.8, respectively.

2 System Model and Preliminaries

Following [14, 22], we assume a system of n processes/threads, $p_1, \ldots, p_n$ 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*: $stm\text{-}begin()$ 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, *stm-tryC*$()$ that tries to commit the transaction and returns $commit\ \mathscr{C}$ if it succeeds. Otherwise, *stm-tryA*$()$ that aborts the transaction and returns $abort\ \mathscr{A}$. For the sake of presentation simplicity, we assume that the values taken as arguments by *stm-write*$()$ are unique.

Operations *stm-read*$()$ and *stm-tryC*$()$ may return $\mathscr{A}$, in which case we say that the operations *forcefully abort*. Otherwise, we say that the operations have *successfully* executed. Each operation is equipped with a unique transaction identifier. A transaction T_i starts with the first operation and completes when any of its operations return $\mathscr{A}$ or $\mathscr{C}$. We denote any operation that returns $\mathscr{A}$ or $\mathscr{C}$ as *terminal operations*. Hence, operations $stm\text{-}tryC()$ and $stm\text{-}tryA()$ 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: $r_k(x, v)$, $r_k(x, \mathscr{A})$, $w_k(x, v)$, $stm\text{-}tryC_k(\mathscr{C})$ (or c_k for short), $stm\text{-}tryC_k(\mathscr{A})$, $stm\text{-}tryA_k(\mathscr{A})$ (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 $\overline{H}$, by inserting $stm\text{-}tryA_k(\mathscr{A})$ immediately after the last event of every transaction $T_k \in H.live$. But for $stm\text{-}tryC_i$ 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 $\langle SH.evts, <_{SH}\rangle$ and is defined as: (1) $<_{SH} \subseteq <_H$; (2) $SH.evts \subseteq H.evts$; (3) If an event of a transaction $T_k \in H.txns$ is in SH then all the events of T_k in H should also be in SH.

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

Valid and legal history: A successful read $r_k(x, v)$ (i.e., $v \neq \mathscr{A}$) in 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, $\langle r_k(x, v)$ is valid $\Leftrightarrow \exists T_j : (c_j <_H r_k(x, v)) \wedge (w_j(x, v) \in T_j.evts) \wedge (v \neq \mathscr{A})\rangle$. The history H is valid if all its successful read operations are valid.

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

Opacity and Strict Serializability: Two histories H and H' are *equivalent* if they have the same set of events. Now a history H is said to be *opaque* [13] 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.

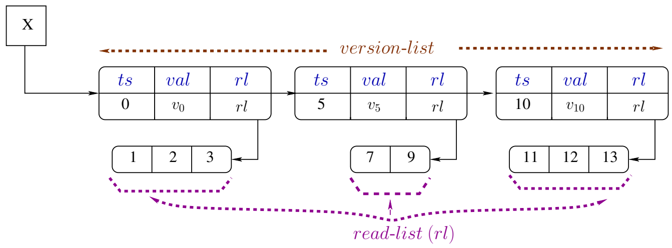

Fig. 2: Data Structures for Maintaining Versions

Along same lines, a valid history H is said to be *strictly serializable* if $H.subhist(H.committed)$, only committed transactions of H 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.

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 [13]. Opacity, on the other hand, considers the correctness of *aborted* transactions as well. But the restrictions of opacity cause the throughput to decrease substantially. Another correctness-criterion for STMs is local opacity [21, 22] which achieves very similar goals as opacity but not as restrictive as opacity.

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

A history H is said to be *locally-opaque* [21, 22] if all the sub-histories in H.subhistSet are opaque. In local opacity, aborted or live transaction can not cause another transaction to abort. It was shown that local opacity [21, 22] 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 again the vice-versa need not be true.

Another correctness criterion is TMS1 [9, 1], similar to local opacity by considering multiple sequential histories for correctness of a history. But it differs from local opacity that the response event could include aborted transactions as well.

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 ∞. When K is 1, it boils down to single-version starvation-free STM. If K is ∞, then *KSFTM* uses unbounded versions and needs a separate garbage collection mechanism to delete old versions like other MVSTMs proposed in the literature [20, 23]. We denote *KSFTM* using unbounded versions as *UVSFTM* and the version with garbage collection as *UVSFTM-GC*.

To explain the intuition behind the *KSFTM* algorithm, we start with the modification of MVTO [2, 20] algorithm and then make a sequence of modifications to it to arrive at *KSFTM* algorithm. The rest of the section is organized as follows. In Section 3.1, we define starvation freedom and identify assumptions made in the paper. Section 3.2 discusses data structures for all the algorithms developed in this section. Section 3.3 develops *PKTO* that adds the approach of providing priority to older transactions in MVTO algorithm. We show why this is insufficient to provide starvation freedom in multi-version setting. Section 3.4 identifies a key idea that can help in providing starvation freedom. Unfortunately, using this idea alone is insufficient as it can violate strict-serializability and consequently local opacity. Section 3.5 describes *KSFTM* algorithm that simultaneously maintains correctness, strict-serializability and local opacity while providing starvation-freedom.

3.1 Starvation-Freedom Explanation

This section starts with the definition of starvation-freedom. Then we describe the assumption about the scheduler for our algorithm to satisfy starvation-freedom.
Definition 1 Starvation-Freedom: A STM system is said to be starvation-free if a thread invoking a non-parasitic transaction T_i gets the opportunity to retry T_i on every abort, due to the presence of a fair scheduler, then T_i will eventually commit.

As explained by Herlihy & Shavit [17], a fair scheduler implies that no thread is forever delayed or crashed. Hence with a fair scheduler, we get that if a thread acquires locks then it will eventually release the locks. Thus a thread cannot block out other threads from progressing.
Assumption about Scheduler: In order for starvation-free algorithm *KSFTM* (described in Section 3.5) to work correctly, we make the following assumption about the fair scheduler:

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

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

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

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

3.2 Algorithm Preliminaries

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

Transaction Invocation: Transactions are invoked by the 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 be different. However, in this case, the thread Th_x will also pass the CTS value of the first incarnation (T_i) to the STM system. By this, Th_x informs the STM that, T_j is not a new invocation but is an incarnation of T_i. The CTS values are obtained by incrementing a global atomic counter G_Count.

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

We now state our assumptions about transactions in the system.

Assumption 2 *We assume that in the absence of other concurrent conflicting transactions, every transaction will commit. In other words, (a) if a transaction T_i is executing in a system where other concurrent conflicting transactions are not present then T_i will not self-abort. (b) Transactions are not parasitic (explained in Section 1).*

If transactions self-abort or behave in parasitic manner 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.

In all our algorithms, for each t-object, the algorithms maintain multiple versions in form of *version-list* (or *vlist*). Similar to MVTO [20], each version of a t-object is a tuple denoted as *vTuple* and consists of three fields: (1) timestamp characterizing the transaction that created the version, (2) value, and (3) a list, *read-list* (or rl) consisting of transaction ids (or CTSs) that read from this version.

Fig 2 illustrates this structure. For a t-object x, we use the notation $x[t]$ to access the version with timestamp t. Depending on the algorithm considered, the fields of this structure change.

We assume that the STM system exports the following methods for a transaction T_i: (1) $stm\text{-}begin(t)$ where t is provided by the invoking thread, Th_x. From our earlier assumption, it is the CTS of the first incarnation or $null$ if Th_x is invoking this transaction for the first time. This method returns a unique timestamp to Th_x which is the CTS/id of the transaction. (2) $stm\text{-}read_i(x)$ tries to read t-object x. It returns either value v or $\mathscr{A}$. (3) $stm\text{-}write_i(x, v)$ operation that updates a t-object x with value v locally. It returns ok. (4) $stm\text{-}tryC_i()$ tries to commit the transaction and returns $\mathscr{C}$ if it succeeds. Otherwise, it returns $\mathscr{A}$.

Correctness Criteria: For ease of exposition, we initially consider strict-serializability as *correctness-criterion* to illustrate the correctness of the algorithms. Subsequently, we consider a stronger property, local opacity that is more suitable for STMs.

3.3 Priority-based MVTO Algorithm

In this subsection, we describe a modification to the multi-version timestamp ordering (MVTO) algorithm [2, 20] to ensure that it provides preference to transactions that have low ITS, i.e., transactions that have been in the system for a longer time. We denote the basic algorithm which maintains unbounded versions as *Priority-based MVTO* or *PMVTO* (akin to the original MVTO). We denote the variant of *PMVTO* that maintains K versions as *PKTO* and the unbounded versions variant with garbage collection as *PMVTO-GC*.

While providing higher priority to older transactions suffices to provide starvation-freedom in *SV-SFTM*, we note that *PKTO* is not starvation free. The reason that demonstrates why *PKTO* is not starvation free forms our basis of designing SFMVTO that provides starvation-freedom (described in Section 3.4).

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

stm-$begin(t)$**:** A unique timestamp ts is allocated to T_i which is its CTS (i from our assumption). The timestamp ts is generated by atomically incrementing the global counter G_Count. 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 of x), i.e., there exists no version of x with timestamp k such that $j < k < i$. After reading this version of x, T_i is stored in $x[j]$'s read-list. If no such version exists then T_i is *aborted*.

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

stm-$tryC$: This operation consists of three steps. In Step 1, it checks whether T_i can be *committed*. In Step 2, it performs the necessary tasks to mark T_i as a *committed* transaction and in Step 3, T_i 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 = CTS of T_k). In this scenario, the value read by T_k would be incorrect (w.r.t strict-serializability) if T_i is allowed to create a new version. In this case, we say that the transactions T_i and T_k are in *conflict*. So, we do the following: (i) if T_k has already *committed* then T_i is *aborted*; (ii) Suppose T_k is live and its_k is less than its_i. Then again T_i is *aborted*; (iii) If T_k is still live with its_i less than its_k then T_k is *aborted*.

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

2. After Step 1, we have verified that it is ok for T_i to commit. Now, we have to create a version of each t-object x in the $wset$ of T_i. This is achieved as follows:
   (a) T_i creates a $vTuple$ $\langle i, wset_i.x.v, null\rangle$. In this tuple, i (CTS of T_i) is the timestamp of the new version; $wset_i.x.v$ is the value of x is in T_i's $wset$, and the read-list of the $vTuple$ is $null$.
   (b) 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 [19]. 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. The detailed pseudocode along with the description can be found in Appendix A.3 and arxiv[5]. We have the following correctness property of *PKTO*.

Property 1 Any history generated by the *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.

While *PKTO* (and *PMVTO*) satisfies strict-serializability, it fails to prevent starvation. The key reason is that if transaction T_j conflicts with T_k and T_k has already committed, then T_j must be aborted. This is true even if T_j is the oldest transaction in the system. Furthermore, next incarnation of T_j may have to be aborted by another transaction T'_k. This cannot be prevented as conflict between T_j and T'_k may not be detected before T'_k has committed. A detailed illustration of starvation in *PKTO* is shown in Appendix A.4.

3.4 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 $stm\text{-}tryC$ operation). We use r_{50} and r_{60} to denote their

S. No.	Sequence	Possible actions by *PKTO*
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

read operations while w_{50} and w_{60} to denote their $stm\text{-}tryC$ 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 $stm\text{-}tryC$ operation (due to optimistic execution) which is the final operation of a transaction.

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 the 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 stm-tryC in *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 the 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 should 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.,

$$wts_i = cts_i + C * (cts_i - its_i); \tag{1}$$

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 $stm\text{-}read$, $stm\text{-}write$ and $stm\text{-}tryC$ operations of *PKTO*. In SFKTO, a transaction T_i uses wts_i to read a version in $stm\text{-}read$. Similarly, T_i uses wts_i in $stm\text{-}tryC$ 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.

SFKTO algorithms ensures starvation-freedom in presence of a fair scheduler that satisfies Assumption 1 (bounded-termination). 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. Then it cannot be aborted. But SFKTO and its variant SFMVTO do not satisfy strict-serializability which is illustrated in Appendix A.5.

3.5 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 local opacity) as well as starvation-freedom. To achieve this, we first understand why the initial algorithm, *PKTO* satisfies strict-serializability. This is because CTS was used to create the ordering among committed transactions. CTS is based on real-time ordering. In contrast, SFKTO uses WTS which may not correspond to the real-time, as WTS may be significantly larger than CTS as shown by history $H1$ in Fig 3.

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_Count) catches up to wts_i. This will ensure that the value of WTS will also become the same as the real-time thereby guaranteeing strict-serializability. However, this is unacceptable, as in practice, it would require transaction T_i locking all the variables it plans to update and wait. This will adversely affect the performance of the STM system.

We can allow the transaction T_i to commit before its wts_i has caught up with the actual time if it does not violate the real-time ordering. Thus, to ensure that the notion of real-time order is respected by transactions in the course of their execution in 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. [27] in the context of multi-version STMs. Several other researchers have used this idea since then such as Guerraoui et al. [15], Crain et al. [7] etc.

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

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

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

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

Finally, when a transaction T_i commits: T_i records its commit time (or $comTime_i$) by getting the current value of G_Count 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).

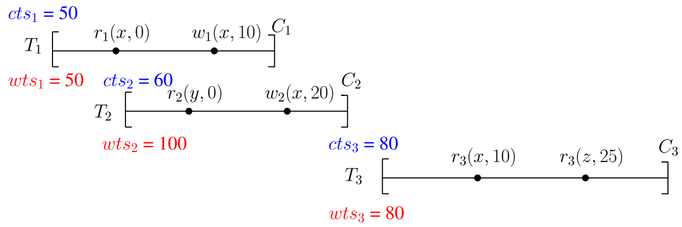

Fig. 3: Correctness of *KSFTM* Algorithm

We illustrate this technique with the history $H1$ shown in Fig 3. When T_1 starts its $cts_1 = 50, tltl_1 = 50, tutl_1 = \infty$. Now when T_1 commits, suppose G_Count is 70. Hence, $tutl_1$ reduces to 70. Next, when T_2 commits, suppose $tutl_2$ reduces to 75 (the current value of G_Count). As T_1, T_2 have accessed a common t-object x in a conflicting manner, $tltl_2$ is incremented to a value greater than $tutl_1$, say 71. Next, when T_3 begins, $tltl_3$ is assigned cts_3 which is 80 and $tutl_3$ is initialized to ∞. When T_3 reads 10 from T_1, which is $r_3(x, 10)$, $tutl_3$ is reduced to a value less than $tltl_2(= 71)$, say 70. But $tltl_3$ is already at 80. Hence, the limits of T_3 have crossed and thus causing T_3 to abort. The resulting history consisting of only committed transactions T_1T_2 is strict-serializable.

Based on this idea, we next develop a variation of 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 $stm\text{-}begin$, $stm\text{-}read$, $stm\text{-}write$ and $stm\text{-}tryC$ 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.
$stm\text{-}begin(t)$**:** A unique timestamp ts is allocated to T_i which is its CTS (i from our assumption) which is generated by atomically incrementing the global counter G_Count. 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 as: $tltl_i = cts_i$, $tutl_i = \infty$.
$stm\text{-}read(x)$**:** Transaction T_i reads from a version of x with timestamp j such that j is the largest timestamp less than wts_i (among the versions x), i.e. there exists no version k such that $j < k < wts_i$ is true. If no such j exists then T_i is aborted. Otherwise, after reading this version of x, T_i is stored in j's rl. Then we modify tltl, tutl as follows:

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

$stm\text{-}write(x, v)$**:** T_i stores this write to value x locally in its $wset_i$.
$stm\text{-}tryC$: 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 $stm\text{-}read$). If there exists no version of x with a timestamp less than wts_i then T_i is aborted. This is similar to Step 1b of the $stm\text{-}tryC$ 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 $stm\text{-}tryC$ of *PKTO* algorithm.
   (c) Next, we must ensure that T_i's tltl and tutl are updated correctly w.r.t to other concurrently executing transactions. To achieve this, we adjust tltl, tutl as follows: (i) Let j be the ts of the largest version smaller than wts_i. Then $tltl_i = max(tltl_i, x[j].\texttt{vrt} + 1)$. Next, for each reading transaction, T_r in $x[j].read\text{-}list$, we again set, $tltl_i = max(tltl_i, tutl_r + 1)$. (ii) Similarly, let p be the ts of the smallest version larger than wts_i. Then, $tutl_i =$

$min(tutl_i, x[p].\texttt{vrt} - 1)$. (Note that we don't have to check for the transactions in the read-list of $x[p]$ as those transactions will have tltl higher than $x[p].\texttt{vrt}$ due to $stm\text{-}read$.) (iii) Finally, we get the commit time of this transaction from G_Count: $comTime_i = G\_Count.add\&Get(incrVal)$ where $incrVal$ is any constant ≥ 1. Then, $tutl_i = min(tutl_i, comTime_i)$. After performing these updates, abort T_i if tltl and tutl have crossed, i.e., $tltl_i > tutl_i$.

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

Step 1c.(iii) of $stm\text{-}tryC$ 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). The detailed pseudo code along with the description can be found in Appendix A.6. 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. Proof of below theorems appear in Appendix A.7 and Appendix A.8, respectively.

Theorem 1 *Any history generated by KSFTM is strict-serializable and locally-opaque.*

Theorem 2 *KSFTM algorithm ensures starvation-freedom.*

4 Experimental Evaluation

For performance evaluation of *KSFTM* with the state-of-the-art STMs, we have implemented our proposed algorithms that are, *PKTO* [6], *SV-SFTM* [12, 30, 29] along with *KSFTM* in C++ [2] computer language. We have used the available implementations of NOrec STM [8], ESTM [10], and MVTO[20] which were originally developed in C++ as well. Although, only the proposed *KSFTM* and *SV-SFTM* provide starvation-freedom, we have also compared their performances with non-starvation free STMs in order to analyze their performance in practice.

Experimental system: The experimental system is a 2-socket Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz with 14 cores per socket and 2 hyper-threads per core thus resulting in a total of 56 logical threads. Each core has a private 32KB L1 cache and 256 KB L2 cache. The machine has 32GB of RAM and runs Ubuntu 16.04.2 LTS. In our implementation, all threads have the same base priority and we use the default Linux scheduling algorithm. This satisfies the Assumption 1 (bounded-termination) about the scheduler. We have ensured that there are no parasitic transactions [3] in our experiments.

Methodology: Here we have considered two different applications: **(1)** *Counter application* - In this, each thread invokes a single transaction which performs 10 reads/writes

[2] Code is available here: https://github.com/PDCRL/KSFTM

operations on randomly chosen t-objects. A thread continues to invoke a transaction until it successfully commits. We have gauged the performance of our proposed algorithm under both *low* as well as *high contention*. For *low contention* we have taken lower number of threads ranging from 1 to 64 while each thread performs 10 random read/write operations on 1000 t-objects. On the other hand, for *high contention* we have taken large number of threads ranging from 50 to 250 where each thread performs read/write operation over a set of 5 t-objects. We have performed our tests on three workloads stated as: (W1) Li - Lookup intensive: 90% read, 10% write, (W2) Mi - Mid intensive: 50% read, 50% write and (W3) Ui - Update intensive: 10% read, 90% write. This application is undoubtedly very flexible as it allows us to examine performance by tweaking different parameters (refer Appendix A.10 for details).

(2) *Two benchmarks from STAMP suite* [24] - (a) We have considered KMEANS which is a low contention application with short running transactions and hence has a smaller chance of thread starvation. The number of data points were chosen as 2048 with 16 dimensions and total clusters as 5. (b) LABYRINTH is another application that we have considered from the suite. LABYRINTH is a high contention application with long-running transactions where the chances of a thread starving is high. We have taken a grid of size 64x64x3 and the number of paths to route as 48.

To study starvation in various algorithms, we have considered *max-time*, which is the maximum time taken by a transaction among all the transactions in a given experiment to commit since its first invocation. This includes time taken by all the aborted incarnations of the transaction to execute as well. To reduce the effect of outliers, we have taken the average of max-time in ten runs as the final result for all the experiments.

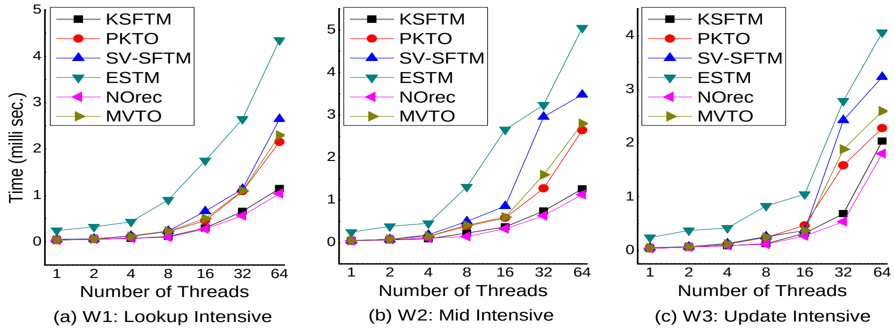

Fig. 4: Max-time analysis on workload $W1$, $W2$, $W3$ for low contention

Results Analysis: Fig 4 and Fig 5 illustrate the max-time analysis of *KSFTM* over state-of-the-art STMs for the counters application on workloads $W1$, $W2$, and $W3$ under low and high contentions, respectively. For *KSFTM* and *PKTO*, we have chosen the value of K as 5 and C as 0.1 as optimal results have been obtained on these parameters as shown in Fig 7. Fig 4 and Fig 5 show that *KSFTM* performs best for all the three workloads except NOrec STMs on low contention. *KSFTM* gives an average speedup on max-time by a factor of 1.74, 2.07, 4.48, 0.95, and 2.41 under low contention and by a factor of 1.22, 1.89, 23.26, 13.12, and 1.49 under high contention over *PKTO*, *SV-SFTM*, ESTM, NOrec STM, and MVTO respectively. We have observed that under low contention, NOrec is slightly better than *KSFTM* but this is a trade-off we pay for ensuring starvation-freedom, while *KSFTM* performs best under high contention.

Fig 6(a) shows analysis of max-time for KMEANS while Fig 6(b) shows for LABYRINTH. In this analysis we have not considered ESTM as the integrated STAMP code for ESTM is not publicly available. For KMEANS, *KSFTM* performs 1.5, 1.44, and 1.67 times better than *PKTO*, *SV-SFTM*, and MVTO. But, NOrec performs 1.09 times better than *KSFTM*. This is because KMEANS has short running transactions with low contention and hence a feeble chance of thread starvation. As a result, the commit time of the transactions is also low.

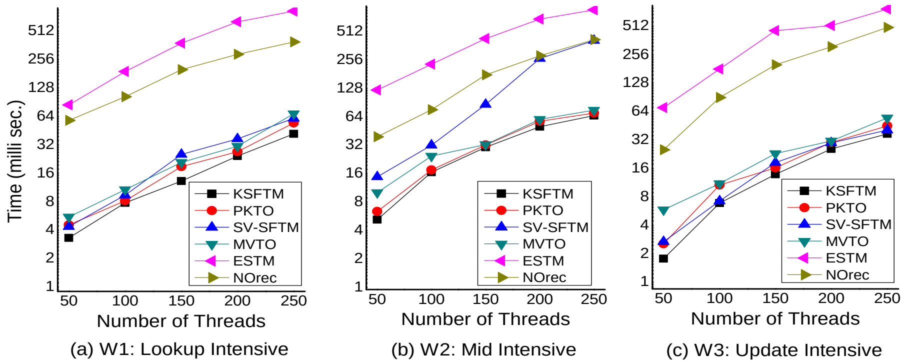

Fig. 5: Max-time analysis on workload $W1$, $W2$, $W3$ for high contention

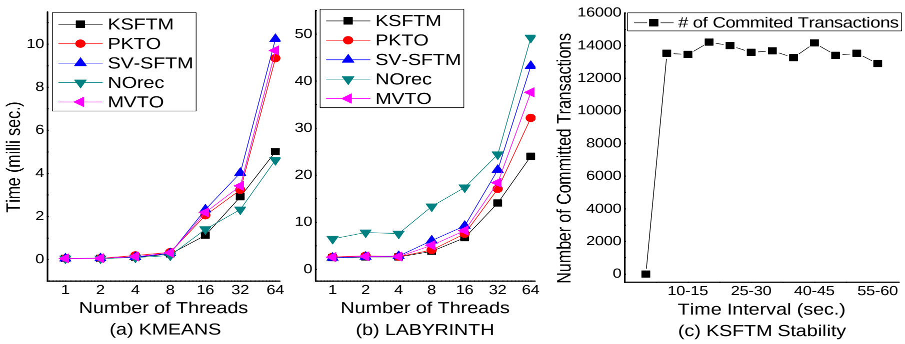

Fig. 6: Max-time analysis on KMEANS, LABYRINTH and KSFTM's Stability

On the other hand for LABYRINTH, *KSFTM* again performs the best. It performs 1.14, 1.4, 2.63, and 1.37 times better than *PKTO*, *SV-SFTM*, NOrec, and MVTO respectively. This is because LABYRINTH has high contention with long-running transactions which can lead to starvation of threads with high probability. This result in longer commit times for transactions.

Fig 6(c) shows the stability of *KSFTM* over time for the counter application. Here we have fixed the number of threads to 32, K as 5, C as 0.1, t-objects as 1000, along with 5 seconds warm-up period on $W1$ workload. Each thread invokes transactions until its time-bound of 60 seconds expires. We have performed the experiments on number of transactions committed in the increments of 5 seconds. The experiment shows that over time *KSFTM* is stable which helps to hold the claim that *KSFTM*'s performance will continue in same manner if time is increased to higher orders.

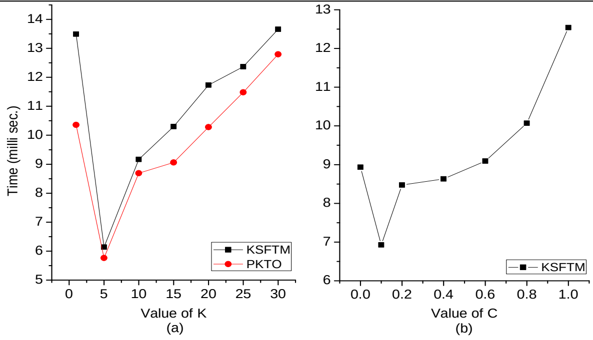

Fig. 7: Optimal value of K and C for *KSFTM*

Optimal value of *K* and constant *C*: To identify the best value of K for *KSFTM*, we ran an experiment with varying the value of K while keepomg the number of threads as 64 on workload $W1$. We observed that the optimal value of K in *KSFTM* is 5 as shown in Fig 7.(a) for counter application. Similarly, the experiments showed that the optimal value of K as 5 for *PKTO* on the same parameters. 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 ran our experiments on workload $W1$, for 64 threads and have observed the optimal value of C as 0.1, shown in Fig 7 (b) for counter application. We have executed several other experiments to study various parameters such as average time analysis on STAMP benchmark, abort counts, average time analysis, and memory consumption by the variants of PKTO and KSFTM in Appendix A.9.

5 Conclusion

We proposed *KSFTM*, a multi-version STM, which ensures starvation-freedom while maintaining K versions for each t-objects. It uses two insights to ensure starvation-freedom in the context of MVSTMs: (1) using ITS to ensure that older transactions are given a higher priority, and (2) using WTS to ensure that conflicting transactions do not commit too quickly before the older transaction could commit. We show *KSFTM* satisfies strict-serializability [25] and local opacity [21, 22]. Our experiments show that *KSFTM* performs better than starvation-free state-of-the-arts STMs as well as non-starvation free STMs under long-running transactions with high contention workloads.

References

1. Attiya H, Gotsman A, Hans S, Rinetzky N (2014) Safety of Live Transactions in Transactional Memory: TMS is Necessary and Sufficient. In: DISC, pp 376–390
2. Bernstein PA, Goodman N (1983) Multiversion Concurrency Control: Theory and Algorithms. ACM Trans Database Syst
3. Bushkov V, Guerraoui R (2015) Liveness in transactional memory pp Transactional Memory. Foundations, Algorithms, Tools, and Applications, 32–49.
4. Bushkov V, Guerraoui R, Kapalka M (2012) On the liveness of transactional memory. In: ACM Symposium on PODC 2012
5. Chaudhary VP, Juyal C, Kulkarni SS, Kumari S, Peri S (2017) Starvation freedom in multi-version transactional memory systems. CoRR abs/1709.01033
6. Chaudhary VP, Juyal C, Kulkarni SS, Kumari S, Peri S (2019) Achieving starvation-freedom in multi-version transactional memory systems. In: NETYS

7. Crain T, Imbs D, Raynal M (2011) Read invisibility, virtual world consistency and probabilistic permissiveness are compatible. In: ICA3PP
8. Dalessandro L, Spear MF, Scott ML (2010) NOrec: Streamlining STM by Abolishing Ownership Records. PPoPP 2010
9. Doherty S, Groves L, Luchangco V, Moir M (2009) Towards Formally Specifying and Verifying Transactional Memory. In: REFINE
10. Felber P, Gramoli V, Guerraoui R (2017) Elastic transactions. J Parallel Distrib Comput 100(C):103–127
11. Fernandes SM, Cachopo J (2011) Lock-free and Scalable Multi-version Software Transactional Memory. PPoPP 2011
12. Gramoli V, Guerraoui R, Trigonakis V (2012) TM2C: A Software Transactional Memory for Many-cores. EuroSys 2012
13. Guerraoui R, Kapalka M (2008) On the Correctness of Transactional Memory. In: PPoPP 2008
14. Guerraoui R, Kapalka M (2010) Principles of Transactional Memory, Synthesis Lectures on Distributed Computing Theory. Morgan and Claypool
15. Guerraoui R, Henzinger T, Singh V (2008) Permissiveness in Transactional Memories. In: DISC 2008
16. Herlihy M, BMoss JE (1993) Transactional memory: Architectural Support for Lock-Free Data Structures. SIGARCH Comput Archit News 21(2)
17. Herlihy M, Shavit N (2011) On the nature of progress. OPODIS 2011
18. Herlihy M, Shavit N (2012) The Art of Multiprocessor Programming, Revised Reprint, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA
19. Herlihy MP, Wing JM (1990) Linearizability: a correctness condition for concurrent objects. ACM Trans Program Lang Syst 12(3)
20. Kumar P, Peri S, Vidyasankar K (2014) A TimeStamp Based Multi-version STM Algorithm. In: ICDCN, pp 212–226
21. Kuznetsov P, Peri S (2014) Non-interference and Local Correctness in Transactional Memory. In: ICDCN, pp 197–211
22. Kuznetsov P, Peri S (2017) Non-interference and local correctness in transactional memory. Theor Comput Sci 688
23. Lu L, Scott ML (2013) Generic multiversion STM. In: DISC 2013
24. Minh CC, Chung J, Kozyrakis C, Olukotun K (2008) STAMP: stanford transactional applications for multi-processing. In: IISWC 2008
25. Papadimitriou CH (1979) The serializability of concurrent database updates. J ACM 26(4)
26. Perelman D, Byshevsky A, Litmanovich O, Keidar I (2011) SMV: Selective Multi-Versioning STM. In: DISC, pp 125–140
27. Riegel T, Felber P, Fetzer C (2006) A lazy snapshot algorithm with eager validation. In: DISC 2006
28. Shavit N, Touitou D (1995) Software Transactional Memory. In: PODC
29. Spear MF, Dalessandro L, Marathe VJ, Scott ML (2009) A comprehensive strategy for contention management in software transactional memory. PPoPP
30. Waliullah MM, Stenström P (2009) Schemes for Avoiding Starvation in Transactional Memory Systems. Concurrency and Computation: Practice and Experience

Appendix

The appendix section is organized as follows:

Section No.	Section Name
Appendix A	Supplements of the Paper
Appendix A.2	Detailed Related Work
Appendix A.3	Pseudo code of *PKTO*
Appendix A.4	Illustration of Starvation in Priority-based MVTO Algorithm
Appendix A.5	The drawback of SFKTO
Appendix A.6	Data Structures and Pseudocode of *KSFTM*
Appendix A.7	Graph Characterization of Local Opacity and *KSFTM* Correctness
Appendix A.8	Proof of Liveness of *KSFTM*
Appendix A.9	Detailed Experimental Evaluation
Appendix A.10	Pseudo code of Counter Application

A Supplements of the Paper

A.1 Missing Notations

Here we define deadlock-freedom in the context of transactions. First, we define it for methods and then extend it to transactions.

Deadlock-Freedom w.r.t method execution: As per the definition of Herlihy & Shavit [17], a method m of a concurrent object is deadlock-free in the following setting: if multiple threads invoke m concurrently then at least one thread will get a response.

Deadlock-Freedom w.r.t transaction execution: We extend the definition of deadlock-freedom to transaction execution. This definition is similar in spirit to starvation-freedom definition of transactions in Section 3.1 and extends the deadlock-freedom given above. Consider the following model: given a set of threads with each thread invoking a transaction. If every time a transaction aborts, the corresponding thread invokes another incarnation of the same transaction. The STM system with this model is said to be *deadlock-free* if some transaction invoked by a thread Th_i successfully commits eventually (possibly after multiple invocations by thread Th_i).

A.2 Detailed Related Work

Discussion on STM Correctness: In Section 2, we discussed about strict-serializability, opacity, local opacity. TMS1 [9, 1] is another interesting correctness-criterion which unlike opacity does not require a single sequential history equivalent to the original history. TMS1 requires that each response should be explained by a sequential history including a subset of the transactions. In this sense, TMS1 is similar to local opacity by considering multiple sequential histories for correctness of a history. But it differs from local opacity that the response event could include aborted transactions whereas local opacity does not involve aborted transaction while considering correctness of a transaction.

Discussion on Multiple Versions and Progress Conditions: Several STM systems have been proposed in the literature. Among them, Elastic STM (ESTM) [10], NOrec STM [8] are popular STMs that execute read/write primitive operations on *transaction objects* or *t-objects*. We represent these STMs as *Read-Write STMs* or *RWSTMs*. ESTM [10] is an appealing alternative to the traditional transactional model which offers better performance than traditional RWSTMs. ESTM is favorable for the search structure like the list, hash-table in shared memory.

Ownership-record-free (NOrec) [8] is another popular STM which ensures low overhead and high scalability. It acquires a global versioned lock when updating the shared memory. Each transaction maintains a read log and snapshot timestamp taken from the global versioned lock whenever a transaction begins. Write of the transaction is occurring directly into its redo-log with a hashing scheme to save the search time. ESTM [10] and NOrec [8] are non-starvation free STMs.

Starvation-freedom in STMs has been explored by a few researchers in literature such as Gramoli et al. [12], Waliullah and Stenstrom [30], Spear et al. [29]. Gramoli et al. [12] proposed a distributed contention manager, *FairCM*, for the transactional memory system that ensures the starvation-freedom for multi-core systems. FairCM used the eager conflict detection technique and visible read to prevent the repetitive abort of the same transaction.

Waliullah and Stenstrom [30] stated that the commit of unordered transactions on a demand-driven basis (commit arbitration policies) in software transactional memory systems are prone to starvation. So, they proposed a scheme by assigning priorities to transactions to avoid starvation. The starvation-freedom is achieved at the cost of modest complexity to the baseline protocol while reducing the wasted computation of roll-back.

Spear et al. [29] proposed a comprehensive strategy for contention management to avoid starvation in software transactional memory systems. It detects the conflicts fairly with invisible reads and lazy acquire of ownership to deal with livelock. The idea is based on extendable timestamps and assigning the priorities to the transactions, and minimizes the unnecessary aborts.

Most of these systems [12, 30, 29] work by assigning priorities to transactions. In case of a conflict between two transactions, the transaction with lower priority is aborted. They ensure that every aborted transaction, on being retried a sufficient number of times, will eventually have the highest priority and hence will commit. We denote such an algorithm as *single-version starvation-free STM* or *SV-SFTM*.

Although *SV-SFTM* guarantees starvation-freedom, it can still abort many transactions spuriously. Consider the case where a transaction T_i has the highest priority. Hence, as per *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. we illustrated the problem in Fig 1 of Section 1. To address this limitation, we motivated from the literature of multi-version STMs [20, 23, 11, 26] that allows more transactions to commit and reduces the number of aborts as compared to single-version STMs or *SVSTMs*. We denote such STMs as *multi-version STMs* or *MVSTMs*. It allows to read from the previous version and guarantees that read-only transaction never returns abort.

Selective multi-versioning (SMV) [26] maintains multiple versions corresponding to each object which reduces the number of aborts of long-running read-only transactions. (SMV) keeps the versions as long as it is useful for some reading transaction and garbage collects the version when none of the transactions read from it. SMV suggested managing the memory through a special garbage collection (GC) thread for a periodic interval to dispose of obsolete versions.

Multi-version timestamp ordering (MVTO) [20] is another popular timestamp-based MVSTM system that satisfies correctness criteria as opacity [13]. It was shown that MVTO [20] achieves greater concurrency than SVSTMs and maintains at least as many versions as the number of live transactions. It provides a garbage collection mechanism to delete the unwanted versions. Although MVSTMs theoretically provide greater concurrency, they suffer from the cost of garbage collection.

None of these MVSTMs [20, 23, 11, 26] provide starvation-freedom. MVTO algorithm provides an idea that multiple versions can help with starvation-freedom without sacrificing on concurrency which motivated us to develop a multi-version starvation-free STM system. So, we propose a multi-version starvation-free STM system as *K-version starvation-free STM* or *KSFTM* [6] that maintains bounded versions, where the number of versions is bounded to be at most K. By maintaining bounded versions, we don't have to incur any cost of garbage collection although theoretically we compromise on concurrency provided.

A.3 Pseudocode of *PKTO*

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

```
G_Count = 1;
for all x in 𝒯 do                          ▷ All the t-objects used by the STM System
    add ⟨0, 0, nil⟩ to x.vl;                ▷ T_0 is initializing x
end for;
```

S. No.	STMs	Number of versions	Safety	Liveness
1.	NOrec [8]	Single version	Opacity	Non-starvation-free
2.	ESTM [10]	Single version	Opacity	Non-starvation-free
3.	SV-SFTM [12, 30, 29]	Single version	Serializability	Starvation-freedom
4.	MVTO [20]	Multiple versions	Opacity	Non-starvation-free
5.	PKTO	K versions	Strict Serializability & Local opacity	Non-starvation-free
6.	SFKTO	K versions	None	Starvation-freedom
7.	KSFTM	K versions	Strict Serializability & Local opacity	Starvation-freedom

Table 2: Comparison of the various STMs

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

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

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

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

Algorithm 5 $stm\text{-}write_i(x, val)$: A Transaction T_i writes into local memory

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

Algorithm 6 $stm\text{-}tryC()$: Returns ok on commit else return Abort

▷ The following check is an optimization which needs to be performed again later
lock G_lock_i;
if $(G\_valid_i == F)$ **then**
    return abort(i);
end if
unlock G_lock_i;
$largeRL = allRL = nil$; ▷ Initialize larger read list (largeRL), all read list (allRL) to nil
for all $x \in wset_i$ **do**
    lock x in pre-defined order;
    ▷ findLTS: returns the version with the largest `ts` value less than G_cts_i. If no such version exists, it returns nil.
    $prevVer = findLTS(G\_cts_i, x)$; ▷ prevVer: largest version smaller than G_cts_i
    if $(prevVer == nil)$ **then** ▷ There exists no version with `ts` value less than G_cts_i
        lock G_lock_i; return abort(i);
    end if
    ▷ **getLar**: obtain the list of reading transactions of $x[prevVer].rl$ whose G_cts is greater than G_cts_i
    $largeRL = largeRL \cup getLar(G\_cts_i, x[prevVer].rl)$;
end for ▷ $x \in wset_i$
$relLL = largeRL \cup T_i$; ▷ Initialize relevant Lock List (relLL)
for all $(T_k \in relLL)$ **do**
    lock G_lock_k in pre-defined order; ▷ Note: Since T_i is also in $relLL$, G_lock_i is also locked
end for
▷ Verify if G_valid_i is false
if $(G\_valid_i == F)$ **then**
    return abort(i);
end if
$abortRL = nil$ ▷ Initialize abort read list (abortRL)
▷ Among the transactions in T_k in $largeRL$, either T_k or T_i has to be aborted
for all $(T_k \in largeRL)$ **do**
    if $(isAborted(T_k))$ **then** ▷ Transaction T_k can be ignored since it is already aborted or about to be aborted
        continue;
    end if
    if $(G\_its_i < G\_its_k) \wedge (G\_state_k ==$ `live`$)$ **then**
        ▷ Transaction T_k has lower priority and is not yet committed. So it needs to be aborted
        $abortRL = abortRL \cup T_k$; ▷ Store T_k in abortRL
    else ▷ Transaction T_i has to be aborted
        return abort(i);
    end if
end for
▷ Store the current value of the global counter as commit time and increment it
$comTime = G\_Count.get\&Inc()$;
for all $T_k \in abortRL$ **do** ▷ Abort all the transactions in abortRL
    $G\_valid_k = F$;
end for
▷ Having completed all the checks, T_i can be committed
for all $(x \in wset_i)$ **do**
    $newTuple = \langle G\_cts_i, wset_i[x].val, nil\rangle$; ▷ Create new v_tuple: G_cts, val, `rl` for x
    if $(|x.vl| > k)$ **then**
        replace the oldest tuple in $x.$`vl` with $newTuple$; ▷ $x.$`vl` is ordered by timestamp
    else
        add a $newTuple$ to $x.vl$ in sorted order;
    end if
end for ▷ $x \in wset_i$
G_state_i = `commit`;
unlock all variables;
return $\mathscr{C}$;

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

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

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

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

A.4 Illustration of Stravation in Priority-based MVTO Algorithm

As discussed in the main paper, *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 $stm\text{-}read(x)$ method if it does not find any other version of x to read from. This can happen if all the versions of x present have a timestamp greater than cts_i. (2) While executing Step 1a(i) of the $stm\text{-}tryC$ 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*. (3) On executing Step 1b of the $stm\text{-}tryC$ method, T_i does not find a previous version. Hence, it does not know which transactions it can conflict with.

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

We illustrate this problem in *PKTO* with Fig 8. Here transaction T_{26}, with ITS 26 is the lowest among all the live transactions, starves due to Step 1a.(i) of the $stm\text{-}tryC$. *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.

A.5 The drawback of SFKTO

Although the SFKTO satisfies starvation-freedom, it, unfortunately, does not satisfy strict-serializability and hence local opacity as well. 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 Fig 9: $r_1(x,0)r_2(y,0)w_1(x,10)C_1w_2(x,20)C_2r_3(x,10)r_3(z,25)C_3$ 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.

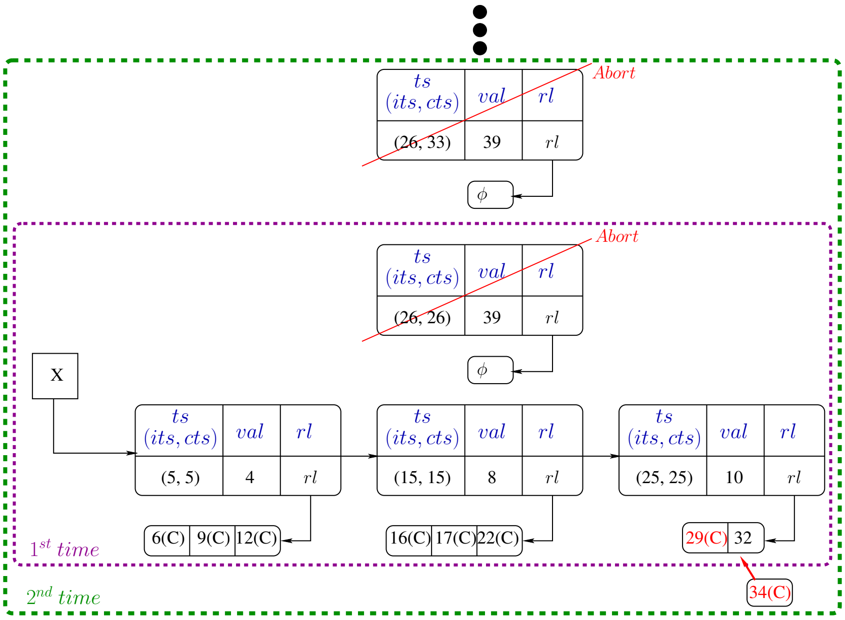

Fig. 8: Representation of execution under *PKTO*

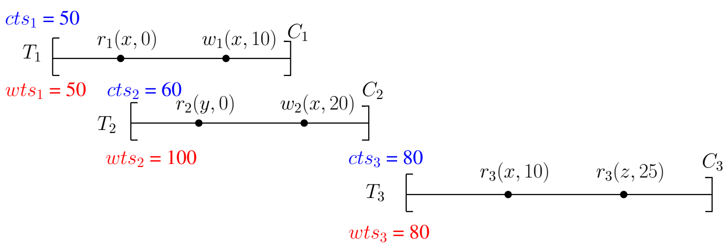

Fig. 9: Correctness of SFKTO Algorithm

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 $H1$ is $T_1T_3T_2$ and it is legal as well. 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.

A.6 Data Structures and Pseudocode of *KSFTM*

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

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

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

For each transaction T_i we maintain the following shared timestamps:

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

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

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

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

```
G_Count = 1;                                   ▷ Global Transaction Counter
for all x in 𝒯 do                              ▷ All the t-objects used by the STM System
    /* T_0 is creating the first version of x: ts = 0, val = 0, rl = nil, vrt = 0 */
    add ⟨0, 0, nil, 0⟩ to x.vl;
end for;
```

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

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

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

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

Algorithm 12 $stm\text{-}write_i(x, val)$: A Transaction T_i writes into local memory

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

Algorithm 13 $stm\text{-}tryC()$: Returns ok on commit else return Abort

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

51: **for all** $(ver \in prevVL)$ **do**
52: x = t-object of ver;
53: $G\_tltl_i = max(G\_tltl_i, x[ver].\texttt{vrt} + 1)$;
54: **end for**
55: ▷ Ensure that $\texttt{vutl}_i$ is less than $\texttt{vrt}$ of versions in $nextVL$
56: **for all** $(ver \in nextVL)$ **do**
57: x = t-object of ver;
58: $G\_tutl_i = min(G\_tutl_i, x[ver].\texttt{vrt} - 1)$;
59: **end for**
60: ▷ Store the current value of the global counter as commit time and increment it
61: $comTime_i = G\_Count.add\&Get(incrVal)$; ▷ $incrVal$ can be any constant ≥ 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** $(G\_tltl_i > G\_tutl_i)$ **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 and 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: $\texttt{ts}, val, \texttt{rl}, \texttt{vrt}$ 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 $x.\texttt{vl}$ with $newTuple$; ▷ $x.\texttt{vl}$ is ordered by $\texttt{ts}$
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 = $\texttt{commit}$;
107: unlock all variables;
108: return $\mathscr{C}$;

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

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

1: $G\_valid_i = F$; G_state_i = `abort`;
2: unlock all variables locked by T_i;
3: return $\mathscr{A}$;

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 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*.

A.7 Graph Characterization of Local Opacity and *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 [20] 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 $mv(H, \ll)$.

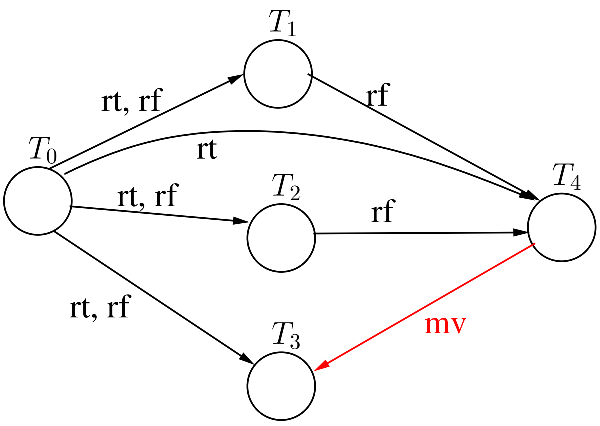

Fig. 10: $OPG(H2, \ll_{H2})$

Using the construction, the $OPG(H2, \ll_{H2})$ for history $H2$ and $\ll_{H2}$ is shown in Fig 10. 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 $r_4(z)$.

Kumar et al [20] 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 3 *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 4 *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,*
$\langle(H$ is locally-opaque$) \Leftrightarrow (\forall sh \in H.subhistSet, \exists \ll_{sh}: OPG(sh, \ll_{sh})$ 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 3. 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 3. □

Lemma 1 *Consider a history H in $gen(KSFTM)$ with two transactions T_i and T_j such that both their G_valid flags are true. there is an edge from $T_i \rightarrow T_j$ then $G\_tltl_i < G\_tltl_j$.*

Proof There are three types of possible edges in MVSG.

1. Real-time edge: Since, transaction T_i and T_j are in real time order so $comTime_i < G\_cts_j$. As we know from Lemma 14 $(G\_tltl_i \leq comTime_i)$. So, $(G\_tltl_i \leq CTS_j)$.
   We know from $stm\text{-}begin(its)$ method, $G\_tltl_j = G\_cts_j$.
   Eventually, $G\_tltl_i < G\_tltl_j$.
2. Read-from edge: Since, transaction T_i has been committed and T_j is reading from T_i so, from Line 99 $stm\text{-}tryC(T_i)$, $G\_tltl_i = \texttt{vrt}_i$.
   and from Line 20 STM $read(j, x)$, $G\_tltl_j = max(G\_tltl_j,$
   $x[curVer].\texttt{vrt} + 1) \Rightarrow (G\_tltl_j > \texttt{vrt}_i) \Rightarrow (G\_tltl_j > G\_tltl_i)$
   Hence, $G\_tltl_i < G\_tltl_j$.
3. Version-order edge: Consider a triplet $w_j(x_j) r_k(x_j) w_i(x_i)$ in which there are two possibilities of version order:
   (a) i $\ll$ j $\Longrightarrow G\_wts_i < G\_wts_j$
   There are two possibilities of commit order:

i. $comTime_i <_H comTime_j$: Since, T_i has been committed before T_j so $G\_tltl_i = \texttt{vrt}_i$. From Line 53 of $stm\text{-}tryC(T_j)$, $\texttt{vrt}_i < G\_tltl(j)$.
Hence, $G\_tltl_i < G\_tltl_j$.

ii. $comTime_j <_H comTime_i$: Since, T_j has been committed before T_i so $G\_tltl_j = \texttt{vrt}_j$. From Line 58 of $stm\text{-}tryC(T_i)$, $G\_tutl_i < \texttt{vrt}_j$. As we have assumed G_valid_i is true so definitely it will execute the Line 84 $stm\text{-}tryC(T_i)$ i.e. $G\_tltl_i = G\_tutl_i$.
Hence, $G\_tltl_i < G\_tltl_j$.

(b) j ≪ i $\Longrightarrow G\_wts_j < G\_wts_i$
Again, there are two possibilities of commit order:

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

A. $G\_wts_k > G\_wts_i$: That means T_k is in largeRL of T_i. From Line 45 to Line 47of $stm\text{-}tryC(i)$, 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 1. Otherwise, as T_j has already been committed and later T_i will execute the Line 99 $stm\text{-}tryC(T_i)$, Hence, $G\_tltl_j < G\_tltl_i$.

B. $G\_wts_k < G\_wts_i$: That means T_k is in smallRL of T_i. From Line 17 of $read(k, x)$, $G\_tutl_k < \texttt{vrt}_i$ and from Line 20 of $read(k, x)$, $G\_tltl_k > \texttt{vrt}_j$. Here, T_j has already been committed so, $G\_tltl_j = \texttt{vrt}_j$. As we have assumed G_valid_i is true so definitely it will execute the Line 99 $stm\text{-}tryC(T_i)$, $G\_tltl_i = \texttt{vrt}_i$.
So, $G\_tutl_k < G\_tltl_i$ and $G\_tltl_k > G\_tltl_j$. While considering G_valid_k flag is true $\rightarrow G\_tltl_k < G\_tutl_k$.
Hence, $G\_tltl_j < G\_tltl_k < G\_tutl_k < G\_tltl_i$.
Therefore, $G\_tltl_j < G\_tltl_k < G\_tltl_i$.

ii. $comTime_i <_H comTime_j$: Since, T_i has been committed before T_j so, $G\_tltl_i = \texttt{vrt}_i$. From Line 58 of $stm\text{-}tryC(T_j)$, $G\_tutl_j < \texttt{vrt}_i$ i.e. $G\_tutl_j < G\_tltl_i$. Here, T_k read from T_j. So, From Line 17 of $read(k, x)$, $G\_tutl_k < \texttt{vrt}_i \rightarrow G\_tutl_k < G\_tltl_i$ from Line 20 of $read(k, x)$, $G\_tltl_k > \texttt{vrt}_j$. As we have assumed G_valid_j is true so definitely it will execute the Line 99 $stm\text{-}tryC(T_j)$, $G\_tltl_j = \texttt{vrt}_j$.
Hence, $G\_tltl_j < G\_tltl_k < G\_tutl_k < G\_tltl_i$.
Therefore, $G\_tltl_j < G\_tltl_k < G\_tltl_i$. □

Theorem 5 *Any history H gen(KSFTM) is local opaque iff for a given version order ≪ H, MVSG(H,≪) is acyclic.*

Proof We are proving it by contradiction, so Assuming MVSG(H,≪) has cycle. From Lemma 1, For any two transactions T_i and T_j such that both their G_valid flags are true and if there is an edge from $T_i \rightarrow T_j$ then $G\_tltl_i < G\_tltl_j$. While considering transitive case for k transactions $T_1, T_2, T_3 ... T_k$ such that G_valid flags of all the transactions are true. if there is an edge from $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow .... \rightarrow T_k$ then $G\_tltl_1 < G\_tltl_2 < G\_tltl_3 < .... < G\_tltl_k$.
Now, considering our assumption, MVSG(H,≪) has cycle so, $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow .... \rightarrow T_k \rightarrow T_1$ that implies $G\_tltl_1 < G\_tltl_2 < G\_tltl_3 < .... < G\_tltl_k < G\_tltl_1$.
Hence from above assumption, $G\_tltl_1 < G\_tltl_1$ but this is impossible. So, our assumption is wrong.
Therefore, MVSG(H,≪) 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_Count through operations op_i and op_j respectively. If res(op_i) $<_H$ inv(op_j) then op_i and op_j are in real-time order in H. So, the *M_Order$_H$* is $op_i \rightarrow op_j$.

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

1. $read_i(x)$ and $read_j(x)$: If $read_i(x)$ acquires the lock on x before $read_j(x)$ then the *M_Order$_H$* is $op_i \rightarrow op_j$.
2. $read_i(x)$ and $stm\text{-}tryC_j()$: If they are accessing common t-objects then, let say $read_i(x)$ acquires the lock on x before $stm\text{-}tryC_j()$ then the *M_Order$_H$* is $op_i \rightarrow op_j$. Now if they are not accessing common t-objects but sharing G_lock then, let say $read_i(x)$ acquires the lock on G_lock_i

before $stm\text{-}tryC_j()$ acquires the lock on $relLL$ (which consists of G_lock_i and G_lock_j) then the *M_Order*$_H$ is $op_i \rightarrow op_j$.

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

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

1. $stm\text{-}begin_i$ and $stm\text{-}begin_j$: Both the $stm\text{-}begin$ are accessing shared counter variable G_Count. If $stm\text{-}begin_i$ executes $G\_Count.get\&Inc()$ before $stm\text{-}begin_j$ then the *M_Order*$_H$ is $op_i \rightarrow op_j$.
2. $stm\text{-}begin_i$ and $stm\text{-}tryC(j)$: If $stm\text{-}begin_i$ executes $G\_Count.get\&Inc()$ before $stm\text{-}tryC(j)$ 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_Count counter: If two operation op_i and op_j are overlapping but accessing common G_Count counter instead of *t-object*. Let say op_i access G_Count 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 2 *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: ($Fpu_i(\alpha)$ $<$ $Lpl\_op_j(\alpha)$) $\cup$ ($access.G\_Count_i < access.G\_Count_j$) $\cup$ ($Fpu\_op_i(\alpha) < access.G\_Count_j$) $\cup$ ($access.G\_Count_i < Lpl\_op_j(\alpha)$). Here, α can either be t-object or G_valid.*

Proof we consider proof by induction, So we assummed there exist a path from op_1 to op_n and there is an edge between op_n to op_{n+1}. As we described, while constructing 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, ($access.G\_Count_n < Ll\_op_{n+1}(\alpha)$). 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, ($Ll\_op_n(\alpha)$) $<$ ($access.G\_Count_n$) $<$ ($Fu\_op_n(\alpha)$).
      Hence, ($Fu\_op_1(\alpha) < Ll\_op_{n+1}(\alpha)$)
      iii. ($access.G\_Count_1$) $<$ ($access.G\_Count_n$):
      Here, ($access.G\_Count_n$) $<$ $Ll\_op_{n+1}(\alpha)$).
      So, ($access.G\_Count_1$) $<$ ($access.G\_Count_n$) $<$ $Ll\_op_{n+1}(\alpha)$).
      Hence, ($access.G\_Count_1$) $<$ $Ll\_op_{n+1}(\alpha)$).

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

v. $(access.G\_Count_1) < Ll\_op_n(\alpha))$:
Here, $(Fu\_op_n(\alpha) < Ll\_op_{n+1}(\alpha))$.
So, $(access.G\_Count_1) < Ll\_op_n(\alpha)) < (Fu\_op_n(\alpha) < Ll\_op_{n+1}(\alpha))$. Hence, $(access.G\_Count_1) < Ll\_op_{n+1}(\alpha))$.

vi. $(access.G\_Count_1) < Ll\_op_n(\alpha))$: Here, $(access.G\_Count_n < Ll\_op_{n+1}(\alpha))$. 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, $(Ll\_op_n(\alpha)) < (access.G\_Count_n) < (Fu\_op_n(\alpha))$.
Hence, $(access.G\_Count_1) < Ll\_op_{n+1}(\alpha))$.

(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, $(access.G\_Count_n) < (access.G\_Count_{n+1})$.
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, $(Ll\_op_n(\alpha)) < (access.G\_Count_n) < (Fu\_op_n(\alpha))$.
Hence, $(Fu\_op_1(\alpha) < (access.G\_Count_{n+1})$

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

iii. $(access.G\_Count_1) < (access.G\_Count_n)$:
Here, $(access.G\_Count_n) < (access.G\_Count_{n+1})$.
So, $(access.G\_Count_1) < (access.G\_Count_n) < (access.G\_Count_{n+1})$.
Hence, $(access.G\_Count_1) < (access.G\_Count_{n+1})$.

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

v. $(access.G\_Count_1) < Ll\_op_n(\alpha))$: Here, $(access.G\_Count_n) < (access.G\_Count_{n+1})$.
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, $(Ll\_op_n(\alpha)) < (access.G\_Count_n) < (Fu\_op_n(\alpha))$.
Hence, $(access.G\_Count_1) < (access.G\_Count_{n+1})$.

vi. $(access.G\_Count_1) < Ll\_op_n(\alpha))$:
Here, $(Fu\_op_n(\alpha) < (access.G\_Count_{n+1})$.
So, $(access.G\_Count_1) < Ll\_op_n(\alpha)) < (Fu\_op_n(\alpha) < (access.G\_Count_{n+1})$.
Hence, $(access.G\_Count_1) < (access.G\_Count_{n+1})$.

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) $(access.G\_Count_1) < (access.G\_Count_n)$: 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, $(Ll\_op_n(\alpha)) < (access.G\_Count_n) < (Fu\_op_n(\alpha))$.
Hence, $(access.G\_Count_1) < Ll\_op_{n+1}(\alpha))$.

(c) $(Fu\_op_1(\alpha) < (access.G\_Count_n)$:
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, $(Ll\_op_n(\alpha)) < (access.G\_Count_n) < (Fu\_op_n(\alpha))$.
Hence, $(Fu\_op_1(\alpha) < Ll\_op_{n+1}(\alpha))$.
(d) $(access.G\_Count_1) < Ll\_op_n(\alpha))$:
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, $(access.G\_Count_n) < (access.G\_Count_{n+1})$. 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, $(Ll\_op_n(\alpha)) < (access.G\_Count_n) < (Fu\_op_n(\alpha))$.
   Hence, $(Fu\_op_1(\alpha) < (access.G\_Count_{n+1})$.
   (b) $(access.G\_Count_1) < (access.G\_Count_n)$:
   Here, $(access.G\_Count_n) < (access.G\_Count_{n+1})$. Ref 1.(b).iii.
   (c) $(Fu\_op_1(\alpha) < (access.G\_Count_n)$: Here, $(access.G\_Count_n) < (access.$ $G\_Count_{n+1})$. Ref 1.(b).iv.
   (d) $(access.G\_Count_1) < Ll\_op_n(\alpha))$: Here, $(access.G\_Count_n) < (access.$ $G\_Count_{n+1})$. Ref 1.(b).v

Therefore, OPG(H, M_Order) produced by KSFTM is acyclic. □

Lemma 3 *Any history H gen(KSFTM) with α linearization such that it respects M_Order_H then (H, α) is valid.*

Proof From the definition of *valid history*: If all the read operations of H is reading from the previously committed transaction T_j then H is valid.
In order to prove H is valid, we are analyzing the read(i,x). so, from Line 10, it returns the largest `ts` value less than G_wts_i that has already been committed and return the value successfully. If such version created by transaction T_j found then T_i read from T_j. Otherwise, if there is no version whose WTS is less than T_i's WTS, then T_i returns abort.
Now, consider the base case read(i,x) is the first transaction T_1 and none of the transactions has been created a version then as we have assummed, there always exist T_0 by default that has been created a version for all t-objects. Hence, T_1 reads from committed transaction T_0.
So, all the reads are reading from largest `ts` value less than G_wts_i that has already been committed. Hence, (H, α) is valid. □

Lemma 4 *Any history H gen(KSFTM) with α and β linearization such that both respects M_Order_H i.e. $M\_Order_H \subseteq \alpha$ and $M\_Order_H \subseteq \beta$ then $\prec^{RT}_{(H,\alpha)} = \prec^{RT}_{(H,\beta)}$.*

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

Lemma 5 *Any history H gen(KSFTM) with α and β linearization such that both respects M_Order_H i.e. $M\_Order_H \subseteq \alpha$ and $M\_Order_H \subseteq \beta$ then (H, α) is local opaque iff (H, β) is local opaque.*

Proof As α and β are linearizations of history H gen(KSFTM) so, from Lemma 3 (H, α) and (H, β) are valid histories.

Now assuming (H, α) is local opaque so we need to show (H, β) is also local opaque. Since (H, α) is local opaque so there exists legal t-sequential history S (with respect to each aborted transactions and last committed transaction while considering only committed transactions) which is equivalent to ($\overline{H}$, α). As we know β is a linearization of H so ($\overline{H}$, β) is equivalent to some legal t-sequential history S. From the definition of local opacity $\prec^{RT}_{(H,\alpha)} \subseteq \prec^{RT}_{S}$. From Lemma 4, $\prec^{RT}_{(H,\alpha)} = \prec^{RT}_{(H,\beta)}$ that implies $\prec^{RT}_{(H,\beta)} \subseteq \prec^{RT}_{S}$. Hence, (H, β) is local opaque.

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

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

Theorem 6 *Any history generated by KSFTM is locally-opaque.*

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

Since the histories generated by *KSFTM* are locally-opaque, we get that they are also strict-serializable.

Corollary 1 *Any history generated by KSFTM is strict-serializable.*

A.8 Proof of Liveness of *KSFTM*

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

Consider a transaction T_i in a history H generated by *KSFTM*. Once it executes stm-begin method, its ITS, CTS, WTS values do not change. Thus, we denote them as its_i, cts_i, wts_i respectively for T_i. In case the context of the history H in which the transaction executing is important, we denote these variables as $H.its_i, H.cts_i, H.wts_i$ respectively.

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

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

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

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

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

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

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

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

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

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

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

Observation 7 *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 8 *Consider a transaction T_i in a history H with its incarCt being true in $H1$. Let $H2$ be a extension of $H1$ with a transaction T_j in it. Suppose T_j is an incarnation of T_i. Then T_j's incarCt is true in $H2$. Formally,* $\langle H1, H2, T_i, T_j : (H1 \sqsubseteq H2) \wedge (H1.incarCt(T_i)) \wedge (T_j \in H2.txns) \wedge (T_i \in H2.incarSet(T_j)) \implies (H2.incarCt(T_j))\rangle$.

Lemma 6 *Consider a history $H1$ with a strict extension $H2$. Let T_i and T_j be two transactions in $H1$ and $H2$ respectively. Let T_j not be in $H1$. Suppose T_i's incarCt is true. Then ITS of T_i cannot be the same as ITS of T_j. Formally,* $\langle H1, H2, T_i, T_j : (H1 \sqsubset H2) \wedge (H1.incarCt(T_i)) \wedge (T_j \in H2.txns) \wedge (T_j \notin H1.txns) \implies (H1.its_i \neq H2.its_j)\rangle$.

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

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

Now we show the liveness with the following observations, lemmas and 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 9 *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 10 *Given two histories $H1$ and $H2$ 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 and committed transactions.

Observation 11 *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 12 *Consider an committed transaction T_i in a history $H1$. Then there is no extension of $H1$, in which an incarnation of T_i, T_j is live. Formally,* $\langle H1, T_i : (T_i \in H1.committed) \implies (\nexists T_j, H2 : (H1 \sqsubseteq H2) \wedge (T_j \in H2.live) \wedge (H2.its_i = H2.its_j))\rangle$.

Lemma 7 *Consider a history $H1$ and its extension $H2$. Let T_i, T_j be in $H1, H2$ respectively such that they are incarnations of each other. If WTS of T_i is less than WTS of T_j then CTS of T_i is less than CTS T_j. Formally,* $\langle H1, H2, T_i, T_j : (H1 \sqsubset H2) \wedge (T_i \in H1.txns) \wedge (T_j \in H2.txns) \wedge (T_i \in H2.incarSet(T_j)) \wedge (H1.wts_i < H2.wts_j) \implies (H1.cts_i < H2.cts_j)\rangle$

Proof Here we are given that

$$H1.wts_i < H2.wts_j \tag{2}$$

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

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 11 it will be invoked again or reincarnated with another CTS and WTS. It can be seen that CTS is always increasing. As a result, the WTS is also increasing. Thus eventually the WTS will become greater α. Hence, we have that either an incarnation of T_i will get committed or will eventually have WTS greater than or equal to α. □

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

Lemma 9 *Consider a transaction T_i in a history H. Then, we have that CTS of T_i will be less than or equal to sys-time of H. Formally, $\langle T_i, H1 : (T_i \in H.txns) \implies (H.cts_i \leq H.sys\text{-}time)\rangle$.*

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

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

Corollary 2 *Consider a transaction T_i which is not in a history $H1$ but in an strict extension of $H1$, $H2$. Then, we have that CTS of T_i is greater than the sys-time of H. Formally, $\langle T_i, H1, H2 : (H1 \sqsubset H2) \wedge (T_i \notin H1.txns) \wedge (T_i \in H2.txns) \implies (H2.cts_i > H1.sys\text{-}time)\rangle$.*

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

Lemma 10 *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 stm-tryC method iterate through all the t-objects in the write-set of T_i. Moreover, since we assume that the underlying scheduler is fair, we can see that no thread gets swapped out infinitely. Finally, since we assume that all the locks are fair, it can be seen all the methods terminate in finite time. □

Theorem 13 *Every transaction either commits or aborts in finite time.*

Proof This theorem comes directly from the Lemma 10. 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 3 *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 11 *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, $\langle H, T_i, T_j : (\{T_i, T_j\} \subseteq H.txns) \wedge (T_i \in H.live) \wedge (H.wts_j \geq H.wts_i) \Longrightarrow (H.its_i + 2L \geq H.its_j)\rangle$.*

Proof Since T_i is live in $H1$, from Corollary 3, we get that it terminates before the system time, t_Count becomes $cts_i + L$. Thus, sys-time of history $H1$ did not progress beyond $cts_i + L$. Hence, for any other transaction T_j (which is either live or terminated) in $H1$, it must have started before sys-time has crossed $cts_i + L$. Formally $\langle cts_j \leq cts_i + L \rangle$.

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

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

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

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

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

Lemma 12 *Consider two histories $H1$ and $H2$ with $H2$ being a extension of $H1$. Let a transaction T_i being live in both of them. Suppose T_i is itsEnabled in $H1$. Then T_i is itsEnabled in $H2$ as well. Formally, $\langle H1, H2, T_i : (H1 \sqsubseteq H2) \land (T_i \in H1.live) \land (T_i \in H2.live) \land (H1.itsEnabled(T_i)) \implies (H2.itsEnabled(T_i)) \rangle$.*

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. $\square$

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

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

If T_j terminates after executing Line 61 of stm-tryC method then $comTime_j$ is assigned a value, say β. It can be seen that β will be greater than α by at least $incrVal$ due to the execution of this line. Thus, we have that $\langle \alpha + incrVal \leq \beta \rangle$. $\square$

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

Lemma 14 *Consider a history H with a transaction T_i in it. Then in H, $tltl_i$ will be less than or equal to $comTime_i$. Formally, $\langle H, \{T_i\} \in H.txns : (H.tltl_i \leq H.comTime_i)\rangle$.*

Proof Consider the transaction T_i. In stm-begin method, $comTime_i$ get initialized to ∞. The only place where $comTime_i$ gets modified is at Line 61 of stm-tryC. Thus if T_i gets aborted before this line or if T_i is live we have that $(tltl_i \leq comTime_i)$. On executing Line 61, $comTime_i$ gets assigned to some finite value and it does not change after that.

It can be seen that $tltl_i$ gets initialized to cts_i in Line 4 of stm-begin method. In that line, cts_i reads t_Count and increments it atomically. Then in Line 61, $comTime_i$ gets assigned the value of t_Count after incrementing it. Thus, we clearly get that $cts_i(= tltl_i$ initially$) < comTime_i$. Then $tltl_i$ gets updated on Line 20 of read, Line 53 and Line 84 of stm-tryC methods. Let us analyze them case by case assuming that $tltl_i$ was last updated in each of these methods before the termination of T_i:

1. Line 20 of read method: Suppose this is the last line where $tltl_i$ updated. Here $tltl_i$ gets assigned to 1 + $\texttt{vrt}$ of the previously committed version which say was created by a transaction T_j. Thus, we have the following equation,
$$tltl_i = 1 + x[j].\texttt{vrt} \tag{3}$$
It can be seen that $x[j].\texttt{vrt}$ is same as $tltl_j$ when T_j executed Line 99 of stm-tryC. Further, $tltl_j$ in turn is same as $tutl_j$ due to Line 84 of stm-tryC. From Line 62, it can be seen that $tutl_j$ is less than or equal to $comTime_j$ when T_j committed. Thus we have that
$$x[j].\texttt{vrt} = tltl_j = tutl_j \leq comTime_j \tag{4}$$
It is clear that from the above discussion that T_j executed stm-tryC method before T_i terminated (i.e. $stm\text{-}tryC_j <_{H1} term\text{-}op_i$). From Eq.(3) and Eq.(4), we get
$tltl_i \leq 1 + comTime_j \xrightarrow{incrVal \geq 1} tltl_i \leq incrVal + comTime_j$
$\xrightarrow{Lemma\ 13} tltl_i \leq comTime_i$
2. Line 53 of stm-tryC method: The reasoning in this case is very similar to the above case.
3. Line 84 of stm-tryC method: In this line, $tltl_i$ is made equal to $tutl_i$. Further, in Line 62, $tutl_i$ is made lesser than or equal to $comTime_i$. Thus combing these, we get that $tltl_i \leq comTime_i$. It can be seen that the reasoning here is similar in part to Case 1.

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

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

Lemma 15 *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 stm-begin method to ∞. $tutl_i$ is updated in Line 17 of read method, Line 58 and Line 62 of stm-tryC method. If T_i executes Line 17 of read method and/or Line 58 of stm-tryC method then $tutl_i$ gets decremented to some value less than ∞, say α. Further, it can be seen that in both these lines the value of $tutl_i$ is possibly decremented from ∞ because of $nextVer$ (or ver), a version of x whose $\texttt{ts}$ is greater than T_i's WTS. This implies that some transaction T_j, which is committed in H, must have created $nextVer$ (or ver) and $wts_j > wts_i$.

Next, let us analyze the value of α. It can be seen that $\alpha = x[nextVer/ver].vrt - 1$ where $nextVer/ver$ was created by T_j. Further, we can see when T_j executed stm-tryC, we have that $x[nextVer].vrt = tltl_j$ (from Line 99). From Lemma 14, we get that $tltl_j \leq comTime_j$. This implies that $\alpha < comTime_j$. Now, we have that T_j has already committed before the termination of T_i. Thus from Lemma 13, we get that $comTime_j < comTime_i$. Hence, we have that,
$$\alpha < comTime_i \tag{5}$$
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 stm-tryC method. This is because every transaction executes stm-tryC method only after read method. Further within stm-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 ∞ 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 α. Then from Eq.(5) we get that $tutl_i$ remains at α 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_tltl of a committed transaction T_j and comTime of a transaction T_i that commits later.

Lemma 16 *Consider a history $H1$ with transactions T_i, T_j in it. Suppose T_j is committed and T_i is live in $H1$. Then in any extension of $H1$, say $H2$, $tltl_j$ is less than or equal to $comTime_i$. Formally,* $\langle H1, H2 \in gen(KSFTM), \{T_i, T_j\} \subseteq H1, H2.txns : (H1 \sqsubseteq H2) \wedge (T_j \in H1.committed) \wedge (T_i \in H1.live) \implies (H2.tltl_j < H2.comTime_i)\rangle$.

Proof As observed in the previous proof of Lemma 14, if T_i is live or aborted in $H2$, then its comTime is ∞. In both these cases, the result follows.

If T_i is committed in $H2$ then, one can see that comTime of T_i is not ∞. In this case, it can be seen that T_j committed before T_i. Hence, we have that $comTime_j < comTime_i$. From Lemma 14, we get that $tltl_j \leq comTime_j$. This implies that $tltl_j < comTime_i$. □

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

Lemma 17 *Consider two histories $H1, H3$ such that $H3$ is a strict extension of $H1$. Let T_i be a transaction in $H1.live$ such that T_i itsEnabled in $H1$ and G_valid_i flag is true in $H1$. Suppose T_i is aborted in $H3$. Then there is a history $H2$ which is an extension of $H1$ (and could be same as $H1$) such that (1) Transaction T_i is live in $H2$; (2) there is a transaction T_j that is live in $H2$; (3) $H2.wts_j$ is greater than $H2.wts_i$; (4) T_j is committed in $H3$. Formally,* $\langle H1, H3, T_i : (H1 \sqsubset H3) \wedge (T_i \in H1.live) \wedge (H1.valid_i = True) \wedge (H1.itsEnabled(T_i)) \wedge (T_i \in H3.aborted)) \implies (\exists H2, T_j : (H1 \sqsubseteq H2 \sqsubset H3) \wedge (T_i \in H2.live) \wedge (T_j \in H2.txns) \wedge (H2.wts_i < H2.wts_j) \wedge (T_j \in H3.committed))\rangle$.

Proof To show this lemma, w.l.o.g we assume that T_i on executing either read or stm-tryC in $H2$ (which could be same as $H1$) gets aborted resulting in $H3$. Thus, we have that T_i is live in $H2$. Here T_i is itsEnabled in $H1$. From Lemma 12, we get that T_i is itsEnabled in $H2$ as well.

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

1. stm-tryC:
   (a) Line 3 : This line invokes abort() method on T_i which releases all the locks and returns $\mathscr{A}$ to the invoking thread. Here T_i is aborted because its valid flag, is set to false by some other transaction, say T_j, in its stm-tryC algorithm. This can occur in Lines: 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 `ts` is less than T_i's WTS. This implies that all the versions of x have `tss` greater than wts_i. Thus the transactions that created these versions have WTS greater than wts_i and have already committed in $H2$. Let T_j create one such version. Hence, we have that $\langle (T_j \in H2.committed) \implies (T_j \in H3.committed)\rangle$ since $H3$ is an extension of $H2$.
   (c) Line 34 : This case is similar to Case 1a, i.e., Line 3.
   (d) Line 47 : In this line, T_i is aborted as some other transaction T_j in T_i's largeRL has committed. Any transaction in T_i's largeRL has WTS greater than T_i's WTS. This implies that T_j is already committed in $H2$ and hence committed in $H3$ as well.

(e) Line 64 : In this line, T_i is aborted because its lower limit has crossed its upper limit. First, let us consider $tutl_i$. It is initialized in stm-begin method to ∞. As long as it is ∞, these limits cannot cross each other. Later, $tutl_i$ is updated in Line 17 of read method, Line 58 and Line 62 of stm-tryC method. Suppose $tutl_i$ gets decremented to some value α by one of these lines.
Now there are two cases here: (1) Suppose $tutl_i$ gets decremented to $comTime_i$ due to Line 62 of stm-tryC method. Then from Lemma 14, we have $tltl_i \leq comTime_i = tutl_i$. Thus in this case, T_i will not abort. (2) $tutl_i$ gets decremented to α which is less than $comTime_i$. Then from Lemma 15, we get that there is a committed transaction T_j in $H2.committed$ such that $wts_j > wts_i$. This implies that T_j is in $H3.committed$.

(f) Line 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

$$H2.tutl_i \leq H2.tltl_k \tag{6}$$

From the assumption of this case, we have that T_k commits before T_i. Thus, from Lemma 16, we get that $comTime_k < comTime_i$. From Lemma 14, we have that $tltl_k \leq comTime_k$. Thus, we get that $tltl_k < comTime_i$. Combining this with the inequality of this case Eq.(6), we get that $tutl_i < comTime_i$.
Combining this inequality with Lemma 15, we get that there is a transaction T_j in $H2.committed$ and $H2.wts_j > H2.wts_i$. This implies that T_j is in $H3.committed$ as well.

2. STM read:

(a) Line 7: This case is similar to Case 1a, i.e., Line 3

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

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

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

From the description of *KSFTM* algorithm and Lemma 11, it can be seen that a transaction T_i's commit can depend on committing of transactions (or their incarnations) which have their ITS less than ITS of T_i + $2 * L$, which is T_i's affectSet. We capture this notion of dependency for a transaction T_i in a history H as *commit dependent set* or *cds* as: the set of all transactions T_j in T_i's affectSet that do not any incarnation that is committed yet, i.e., not yet have their incarCt flag set as true. Formally,

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

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

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

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

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

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

Lemma 19 *Consider a transaction T_i which is cdsEnabled in a history $H1$. Consider an extension of $H1$, $H2$ with a transaction T_j in it such that T_i is an incarnation of T_j. Let T_k be a transaction in the affectSet of T_j in $H2$ Then T_k is also in the set of transaction of $H1$. Formally, $\langle H1, H2, T_i, T_j, T_k : (H1 \sqsubseteq H2) \wedge (H1.cdsEnabled(T_i)) \wedge (T_i \in H2.incarSet(T_j)) \wedge (T_k \in H2.affectSet(T_j)) \implies (T_k \in H1.txns)\rangle$*

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

$$H1.cts_i \geq H1.its_i + 2 * L \tag{7}$$

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

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

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

$$H2.its_k < H1.its_i + 2 * L \tag{9}$$

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 2 we get that
  $H2.cts_k > H1.sys\text{-}time \xrightarrow[H2.cts_k = H2.its_k]{T_k \text{ starts afresh}} H2.its_k >$ $H1.sys\text{-}time \xrightarrow[H1.sys\text{-}time \geq H1.cts_i]{(T_i \in H1) \wedge Lemma\ 9} H2.its_k > H1.cts_i \xrightarrow{Eq.(7)} H2.its_k > H1.its_i + 2 * L \xrightarrow{H1.its_i = H2.its_j} H2.its_k > H2.its_j + 2 * L$
  But this result contradicts with Eq.(8). Hence, this case is not possible.
- There is an incarnation of T_k, T_l in $H1$: In this case, we have that

$$H1.its_l = H2.its_k \tag{10}$$

  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 6, 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 20 *Consider two histories $H1, H2$ where $H2$ is an extension of $H1$. Let T_i, T_j, T_k be three transactions such that T_i is in $H1.txns$ while T_j, T_k are in $H2.txns$. Suppose we have that (1) cts_i is greater than $its_i + 2 * L$ in $H1$; (2) T_i is an incarnation of T_j; (3) T_k is in affectSet of T_j in $H2$. Then an incarnation of T_k, say T_l (which could be same as T_k) is in $H1.txns$. Formally, $\langle H1, H2, T_i, T_j, T_k : (H1 \sqsubseteq H2) \wedge (T_i \in H1.txns) \wedge (\{T_j, T_k\} \in H2.txns) \wedge (H1.cts_i > H1.its_i + 2 * L) \wedge (T_i \in H2.incarSet(T_j)) \wedge (T_k \in H2.affectSet(T_j)) \implies (\exists T_l : (T_l \in H2.incarSet(T_k)) \wedge (T_l \in H1.txns))\rangle$*

Proof This proof is similar to the proof of Lemma 19. We are given that

$$H1.cts_i \geq H1.its_i + 2 * L \tag{11}$$

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

$H3.its_k > H1.sys\text{-}time \xrightarrow{Lemma\ 9} H3.its_k > H1.cts_i \xrightarrow{Eq.(11)} H3.its_k > H1.its_i + 2 * L \xrightarrow{H1.its_i = H2.its_j} H3.its_k > H2.its_j + 2 * L \xrightarrow[Observation\ 10]{H3 \sqsubseteq H2} H2.its_k > H2.its_j + 2 * L \xrightarrow[definition]{affectSet} T_k \notin H2.affectSet(T_j)$

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

Lemma 21 *Consider a transaction T_i which is cdsEnabled in a history $H1$. Consider an extension of $H1$, $H2$ with a transaction T_j in it such that T_j is an incarnation of T_i in $H2$. Then affectSet of T_i in $H1$ is same as the affectSet of T_j in $H2$. Formally, $\langle H1, H2, T_i, T_j : (H1 \sqsubseteq H2) \wedge (H1.cdsEnabled(T_i)) \wedge (T_j \in H2.txns) \wedge (T_i \in H2.incarSet(T_j)) \implies ((H1.affectSet(T_i) = H2.affectSet(T_j))) \rangle$*

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

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

$$T_k \in H1.txns \tag{12}$$

Combining Eq.(12) with Observation 10, we get that

$$T_k \in H2.txns \tag{13}$$

From the definition of ITS, we get that

$$H1.its_k = H2.its_k \tag{14}$$

Since T_i, T_j are incarnations we have that .

$$H1.its_i = H2.its_j \tag{15}$$

From the definition of affectSet, we get that,

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

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

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

Here, we have that $(H1 \sqsubseteq H2) \wedge (H1.cdsEnabled(T_i)) \wedge (T_i \in H2.incarSet(T_j)) \wedge (T_k \in H2.affectSet(T_j))$. Thus from Lemma 19, 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 $H2.affectSet(T_j)$, we get that

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

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

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

Lemma 22 *Consider two histories $H1$ and $H2$ with $H2$ being an extension of $H1$. Let T_i and T_j be two transactions which are live in $H1$ and $H2$ respectively. Let T_i be an incarnation of T_j and cts_i is less than cts_j. Suppose T_i is cdsEnabled in $H1$. Then T_j is cdsEnabled in $H2$ as well. Formally, $\langle H1, H2, T_i, T_j : (H1 \sqsubseteq H2) \wedge (T_i \in H1.live) \wedge (T_j \in H2.live) \wedge (T_i \in H2.incarSet(T_j)) \wedge (H1.cts_i < H2.cts_j) \wedge (H1.cdsEnabled(T_i)) \implies (H2.cdsEnabled(T_j)) \rangle$.*

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

$$H1.cts_i \geq H2.its_i + 2 * L \tag{16}$$

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 affectSet in $H2$ have their incarCt as true in $H2$.

Since T_j is live in $H2$, we get that T_j is in $H2.txns$. Here, we have that $(H1 \sqsubseteq H2) \wedge (T_j \in H2.txns) \wedge (T_i \in H2.incarSet(T_j)) \wedge (H1.cdsEnabled(T_i))$. Combining this with Lemma 21, we get that $H1.affectSet(T_i) = H2.affectSet(T_j)$.

Now, consider a transaction T_k in $H2.affectSet(T_j)$. From the above result, we get that T_k is also in $H1.affectSet(T_i)$. Since T_i is cdsEnabled in $H1$, i.e., $H1.cdsEnabled(T_i)$ is true, we get that $H1.incarCt(T_k)$ is true. Combining this with Observation 8, we get that T_k must have its incarCt as true in $H2$ as well, i.e. $H2.incarCt(T_k)$. This implies that all the transactions in T_j's affectSet have their incarCt flags as true in $H2$. Hence the $H2.cds(T_j)$ is empty. As a result, T_j is cdsEnabled in $H2$, i.e., $H2.cdsEnabled(T_j)$. □

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

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

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

Observation 14 *For any transaction T_i in H, we have that wts_i is less than or equal to $H.maxWTS(T_i)$. Formally, $H.wts_i \leq H.maxWTS(T_i)$.*

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

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

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

Lemma 23 *Consider two histories $H1$ and $H2$ with $H2$ being an extension of $H1$. Let T_i and T_j be two transactions which are live in $H1$ and $H2$ respectively. Suppose the affectSet of T_i in $H1$ is same as affectSet of T_j in $H2$. Then the affWTS of T_i in $H1$ is same as affWTS of T_j in $H2$. Formally, $\langle H1, H2, T_i, T_j : (H1 \sqsubseteq H2) \wedge (T_i \in H1.txns) \wedge (T_j \in H2.txns) \wedge (H1.affectSet(T_i) = H2.affectSet(T_j)) \implies (H1.affWTS(T_i) = H2.affWTS(T_j))\rangle$.*

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

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

$$H.affWTS(T_j) = max\{H.maxWTS(T_l) | (T_l \in H2.affectSet(T_j))\} \tag{18}$$

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) and (18), we get that

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

$$H2.maxWTS(T_q) \geq H1.maxWTS(T_p) \tag{20}$$

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

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

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

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

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

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

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

Combining Eq.(21) with Lemma 22, 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

$$\begin{aligned} \langle (H1 \sqsubseteq H2) \wedge (T_j \in H2.txns) \wedge (T_i \in H2.incarSet(T_j)) \\ \wedge(H1.cts_i < H2.cts_j) \wedge (H1.cdsEnabled(T_i))\rangle \end{aligned} \tag{22}$$

Combining Eq.(22) with Lemma 21 we get that

$$H1.affWTS(T_i) = H2.affWTS(T_j) \tag{23}$$

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

$$H1.wts_i < H2.wts_j \tag{24}$$

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

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

Now consider T_i in $H1$: T_i is live; its valid flag is true; and is finEnabled. From the definition of finEnabled, we get that it is also cdsEnabled. From Lemma 18, we get that T_i is itsEnabled in $H1$. Thus from Lemma 17, 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,

$$\begin{aligned}\langle(\exists H2, T_j : (H1 \sqsubseteq H2 \sqsubset H3) \wedge (T_i \in H2.live)\\ \wedge(T_j \in H2.txns) \wedge (H2.wts_i < H2.wts_j)\\ \wedge(T_j \in H3.committed))\rangle\end{aligned} \tag{25}$$

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 24, 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 11 we get that $(H2.its_j \leq H2.its_i + 2 * L)$.

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

$$(H2.affWTS(T_i) \geq H2.maxWTS(T_j)) \tag{26}$$

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

$$(H2.wts_i > H2.affWTS(T_i)) \tag{27}$$

Now combining Equations 26, 27 we get,

$$\begin{aligned}H2.wts_i > H2.affWTS(T_i) \geq H2.maxWTS(T_j)\\ > H2.affWTS(T_i) \geq H2.maxWTS(T_j)\\ \geq H2.wts_j \text{[From Observation 14]}\\ > H2.wts_j\end{aligned}$$

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 26 *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 2 we get the following

$H3.its_k > H1.sys\text{-}time \xrightarrow{Lemma\ 9} H3.its_k > H1.cts_i \implies H3.its_k > H1.its_i + 2 * L \xrightarrow{H1.its_i = H2.its_j} H3.its_k > H2.its_j + 2 * L \xrightarrow[definitions]{affectSet, depIts} H2.its_k \notin H2.depIts(T_j)$

We started with its_k in $H2.depIts(T_j)$ and ended with its_k not in $H2.depIts(T_j)$. Thus, we have a contradiction. Hence, the lemma follows. □

Next we denote the set of committed transactions in T_i's affectSet in H as *cis (commit independent set)*. Formally,

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

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

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

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

Lemma 27 *Consider a transaction T_i which is live in a history $H1$ and cts_i is greater than or equal to $its_i + 2 * L$. If T_i is itsEnabled in $H1$ then there is an extension of $H1$, $H2$ in which an incarnation T_i, T_j (which could be same as T_i), is either committed or cdsEnabled. Formally, $\langle H1, T_i : (T_i \in H1.live) \wedge (H1.cts_i \geq H1.its_i + 2 * L) \wedge (H1.itsEnabled(T_i)) \implies (\exists H2, T_j : (H1 \sqsubset H2) \wedge (T_j \in H2.incarSet(T_i)) \wedge ((T_j \in H2.committed) \vee (H2.cdsEnabled(T_j)))))\rangle$.*

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

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

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

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

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

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

Let α be $H1.partAffWTS(T_i)$. Suppose $H1.wts_i < \alpha$. Then from Lemma 8, we get that there is an extension of $H1$, say $H3$ in which an incarnation of T_i, T_l (which could be same as T_i) is committed or is live in $H3$ and has WTS greater than α. If T_l is committed then $P(k+1)$ is trivially true. So we consider the latter case in which T_l is live in $H3$. In case $H1.wts_i \geq \alpha$, then in the analysis below follow where we can replace T_l with T_i.

Next, suppose T_l is aborted in an extension of $H3$, $H5$. Then from Lemma 17, 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 14 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 20, 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 26, we get that $H4.depIts(T_i)$ is a subset of $H1.depIts(T_i)$. Further, we have that transaction T_m has committed. Thus $H4.its_m$ which was in $H1.depIts(T_i)$ is no longer in $H4.depIts(T_i)$. This implies that $H4.depIts(T_i)$ is a strict subset of $H1.depIts(T_i)$ and hence $|H4.depIts(T_i)| \leq k$.
Since T_i and T_l are incarnations, we get that $H4.depIts(T_i) = H1.depIts(T_l)$. Thus, we get that

$$|H4.depIts(T_i)| \leq k \implies |H4.depIts(T_l)| \leq k \tag{30}$$

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

$$\begin{aligned} H4.cts_l > H4.cts_i \xrightarrow{given} H4.cts_l > H4.its_i + 2 * L \xrightarrow{H4.its_i = H4.its_l} \\ H4.cts_l > H4.its_l + 2 * L \end{aligned} \tag{31}$$

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

Proof In $H1$, suppose $H1.affWTS(T_i)$ is α. From Lemma 8, we get that there is a extension of $H1$, $H2$ with a transaction T_j which is an incarnation of T_i. Here there are two cases: (1) Either T_j is committed in $H2$. This trivially proves the lemma; (2) Otherwise, wts_j is greater than α.
In the second case, we get that

$$\begin{aligned} (T_i \in H1.live) \wedge (T_j \in H2.live) \wedge (H.cdsEnabled(T_i)) \\ \wedge (T_j \in H2.incarSet(T_i)) \wedge (H1.wts_i < H2.wts_j) \end{aligned} \tag{32}$$

Combining the above result with Lemma 7, we get that $H1.cts_i < H2.cts_j$. Thus the modified equation is

$$\begin{aligned} (T_i \in H1.live) \wedge (T_j \in H2.live) \wedge (H1.cdsEnabled(T_i)) \\ \wedge (T_j \in H2.incarSet(T_i)) \wedge (H1.cts_i < H2.cts_j) \end{aligned} \tag{33}$$

Next combining Eq.(33) with Lemma 21, we get that

$$H1.affectSet(T_i) = H2.affectSet(T_j) \tag{34}$$

Similarly, combining Eq.(33) with Lemma 22 we get that T_j is cdsEnabled in $H2$ as well. Formally,

$$H2.cdsEnabled(T_j) \tag{35}$$

Now combining Eq.(34) with Lemma 23, we get that

$$H1.affWTS(T_i) = H2.affWTS(T_j) \tag{36}$$

From our initial assumption we have that $H1.affWTS(T_i)$ is α. From Eq.(36), we get that $H2.affWTS(T_j) = \alpha$. Further, we had earlier also seen that $H2.wts_j$ is greater than α. Hence, we have that $H2.wts_j > H2.affWTS(T_j)$.
Combining the above result with Eq.(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 29 *Consider a history $H1$ with T_i be a transaction in $H1.live$. Then there is an extension of $H1$, $H2$ in which an incarnation of T_i, T_j (which could be same as T_i) is either committed or is itsEnabled. Formally, $\langle H1, T_i : (T_i \in H.live) \implies (\exists T_j, H2 : (H1 \sqsubset H2) \wedge (T_j \in H2.incarSet(T_i)) \wedge (T_j \in H2.committed) \vee (H.itsEnabled(T_i))) \rangle$.*

Proof We prove this lemma by inducting on ITS.

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

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

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

Theorem 15 *Consider a history $H1$ with T_i be a transaction in $H1.live$. Then there is an extension of $H1$, $H2$ in which an incarnation of T_i, T_j is committed. Formally, $\langle H1, T_i : (T_i \in H.live) \implies (\exists T_j, H2 : (H1 \sqsubset H2) \wedge (T_j \in H2.incarSet(T_i)) \wedge (T_j \in H2.committed)) \rangle$.*

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

Consider a live transaction T_i in $H1$. Then from Lemma 29, 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 27, 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 28, 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 25, we get that there is an extension of $H4$ in which T_m, an incarnation of T_l, will be committed. This proves our theorem. □

From this theorem, we get the following corollary which states that any history generated by *KSFTM* is $starvation\text{-}freedom$.

Corollary 4 *KSFTM algorithm ensures starvation-freedom.*

A.9 Detailed Experimental Evaluation

This section explains the additional experiments which we have performed to analyze the performance of our proposed algorithms. Especially, we have performed average time analysis on STAMP benchmark, abort counts for low and high contention counter application, average time analysis, and memory consumption on the variants of PKTO and KSFTM.

1. **Average time analysis by a transaction to commit for STAMP benchmark:** We have performed an experiment to analyze the average time taken by a transaction to commit in two of the applications (KMEANS and LABYRINTH) from STAMP. Fig 11 (a) shows the behavior of the algorithms for KMEANS application which has low contention. We observed that till thread count 32 *NOrec* performs the best, but after 32 thread count, *PKTO* outperforms *NOrec* and the performance of *KSFTM* improves. As there is an overhead involved in achieving starvation-freedom *KSFTM* does not performs best. On the other hand, *KSFTM* performs best for LABYRINTH application which has high contention and long-running transaction as shown in Fig 11 (b).

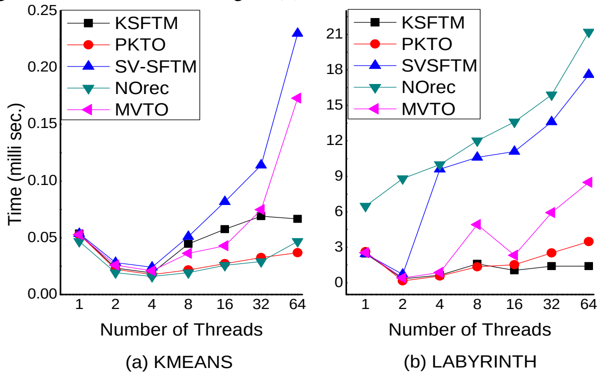

Fig. 11: Average time analysis on KMEANS and LABYRINTH

2. **Abort Count:** We have performed experiment to analyze the abort counts by all the proposed as well as state-of-the art algorithms, under both low as well as high contention for all the three predefined workloads ($W1$, $W2$, and $W3$) on counter application. We observed that, under low contention the number of aborts in *ESTM* and *NOrec* are high as compared to all other algorithms (*KSFTM*, *PKTO*, *SV-SFTM*, MVTO) who have marginally small differences among them as shown in Fig 12. While under high contention *NOrec* has the least number of abort count as shown in Fig 13.

3. **Garbage Collection**: Maintaining 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 required. We have used a global, i.e., across all transactions a list that keeps track of all the live transactions in the system. We call this list as *live-list*. Each transaction at the beginning of its life cycle creates its entry in this *live-list*. Under the optimistic approach of STM, each transaction in the shared memory performs its updates in the $stm\text{-}tryC$ 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 checks if it is the least live timestamp transaction present in the system using *live-list* data structure then the current transaction deletes all the version of that t-object whose timestamp is less than its own and creates a version of it. Otherwise, the transaction does not perform garbage collection or delete any version and look for creating a new version of next t-object in the write set, if at all. Here we have proposed two algorithms that use this garbage collection technique. One of which is a variant of priority-based multi-version timestamp ordering STMs which is *PMVTO-GC* and the other algorithm is a variant of unbounded versions starvation-free STMs which is *UVSFTM-GC*.

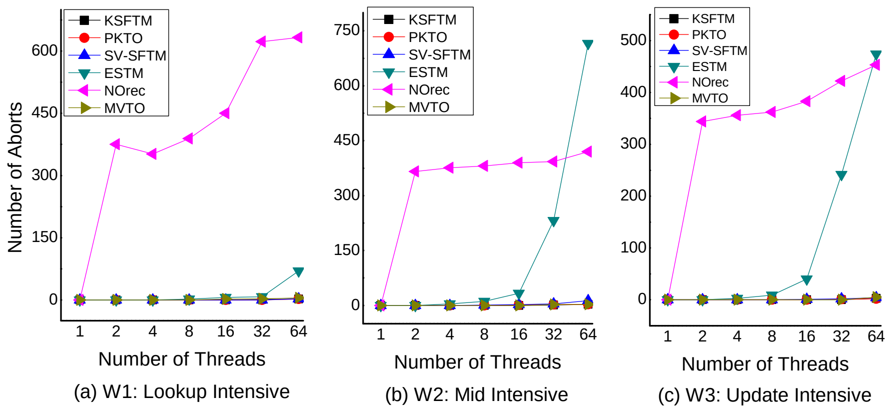

Fig. 12: Abort Count on workload $W1, W2, W3$ for low contention

4. **Variants of *PKTO***: In order to understand and analyze the best performing algorithm among the variants of priority-based multi-version read/write STMs (*PMVTO*, *PMVTO-GC* and *PKTO*) we have performed two experiments. Our first experiment as also shown in Fig 14 have helped us to observe that among all three variants of priority-based multi-version read/write STMs, *PKTO* (priority-based k-version STM) takes the least time when threads are varied from 2^0 to 2^6 on 1000 t-objects. *PKTO* outperforms *PMVTO* and *PMVTO-GC* by a factor of 2 and 1.35. In addition to time efficiency, *PKTO* is also memory efficient as shown in Fig 15. The number of versions created by *PKTO* is least among all the variants. Our experiments have helped us to conclude that *PKTO* is the best variant (among *PMVTO*, *PMVTO-GC* and *PKTO*) of priority-based multi-version read/write STMs.

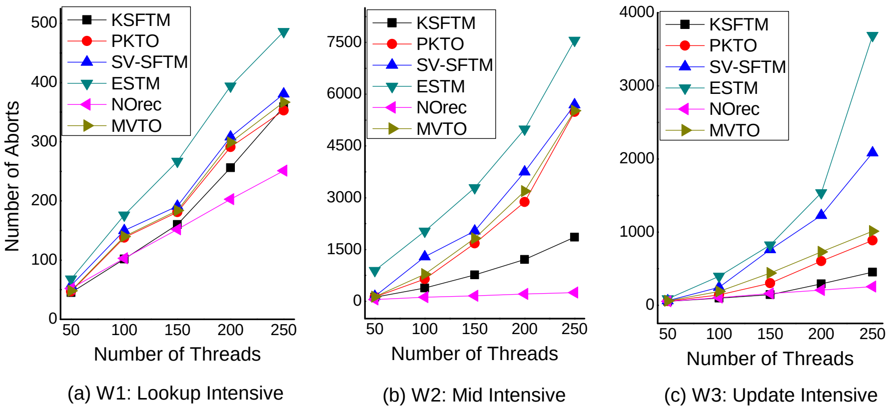

Fig. 13: Abort Count on workload $W1, W2, W3$ for high contention

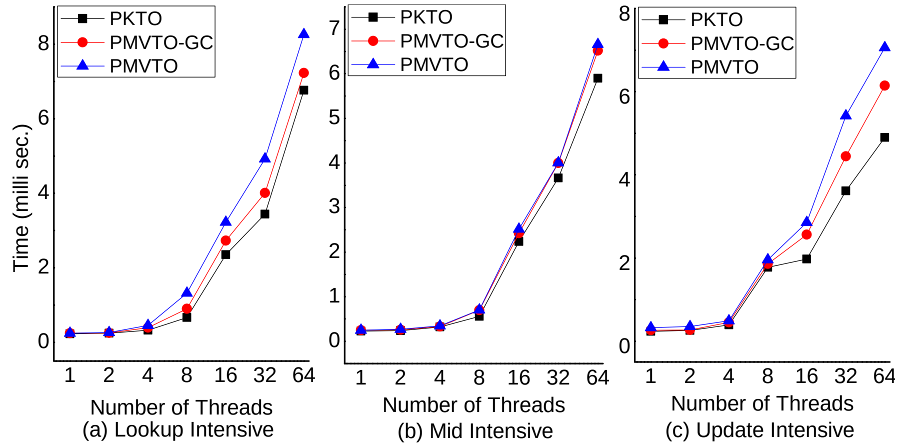

Fig. 14: Average time analysis among variants of *PKTO*

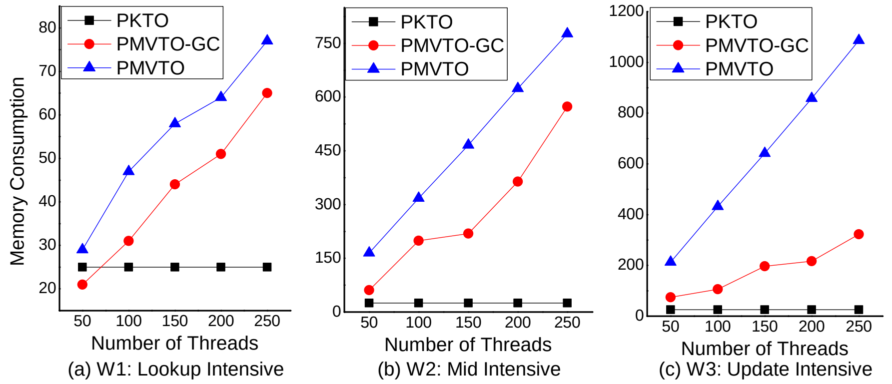

Fig. 15: Memory consumption among variants of *PKTO*

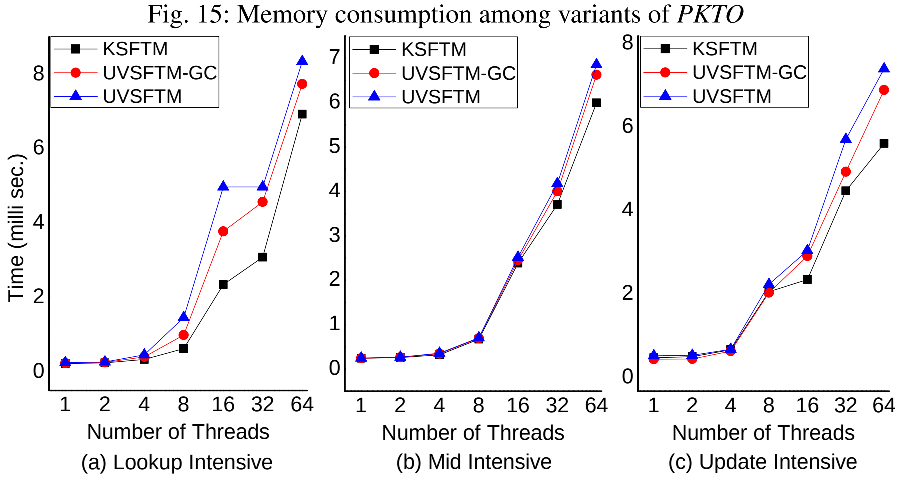

Fig. 16: Average time analysis among variants of *KSFTM*

5. **Variants of *KSFTM***: Similar to our last experiment, we performed experiment to analyze the best time as well as memory efficient variant among all the proposed multi-version starvation-free STMs (*UVSFTM*, *UVSFTM-GC*, *KSFTM*). Our experiments for time and memory as shown in Fig 16 and Fig 17, respectively. In terms of time, *KSFTM* outperforms *UVSFTM* and *UVSFTM-GC* by a factor of 2.1 and 1.5. We can conclude that *KSFTM* performs best among all its variants.

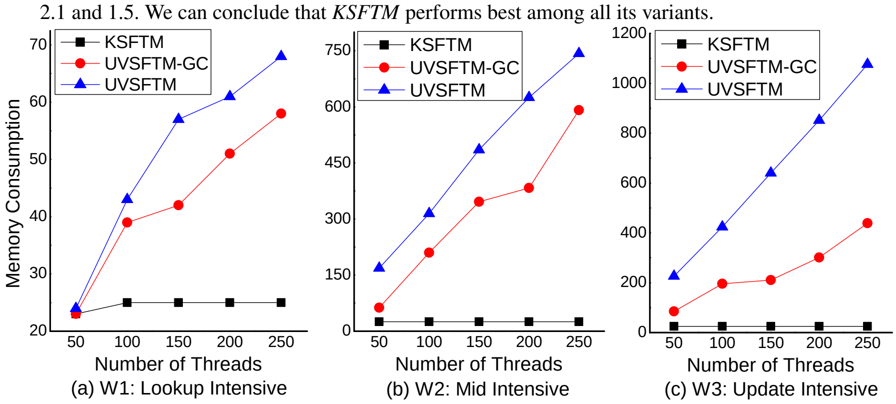

Fig. 17: Memory consumption among variants of *KSFTM*

These results show that maintaining finite versions corresponding to each t-object performs better than maintaining infinite versions and garbage collection on infinite versions corresponding to each t-object.

A.10 Pseudo code of Counter Application

OP_LT_SEED is defined as number of operations per transaction, T_OBJ_SEED is defined as number of transaction objects in the system, TRANS_LT defines the total number of transactions to be executed in the system, and READ_PER is the percentage of read operation which is used to define various workloads.

Algorithm 16 $main()$: The main procedure invoked by counter application

▷ To log abort counts by each thread
$abort\_count$[NUMTHREADS]
▷ To log average time taken by each transaction to commit
$time\_taken$[NUMTHREADS]
▷ To log the time of longest running transaction by each thread, worst case time
$worst\_time$[NUMTHREADS]
for (i = 0 : NUMTHREADS) **do**
    pthread_create(&threads[i], NULL, testFunc_helper,(void∗)args)
end for
for (i = 0 : NUMTHREADS) **do**
    pthread_join(threads[i], &status)
end for
$max\_worst\_time = 0.0$
$total\_abort\_count = 0$
$average\_time_taken = 0$
for (i = 0 : NUMTHREADS) **do**
    if ($max\_worst\_time < worst\_time[i]$) **then**
        $max\_worst\_time = worst\_time[i]$
    end if
    $total\_abort\_count+ = abort\_count[i]$
    $average\_time\_taken+ = time\_taken[i]$
end for

Algorithm 17 $testFunc\_helper()$:Function invoked by threads

$transaction\_count = 0$
while (TRANS_LT) **do**
▷ Log the time at the start of every transaction
  $begin\_time = time\_request()$
▷ Invoke the test function to execute a transaction
  $abort\_count[thread\_id] = test\_function()$
  $transaction\_count + +$
▷ Log the time at the end of every transaction
  $end\_time = time\_request()$
  $time\_taken[thread\_id] + = (end\_time - begin\_time)$
  if $(worst\_time[thread_id] < (end\_time - begin\_time))$ **then**
    $worst\_time[thread_id] = (end\_time - begin\_time)$
  end if
  TRANS_LT -= 1
end while
$time\_taken[thread\_id]$ /= $transaction\_count$

Algorithm 18 $test\_function()$:main test function while executes a transaction

Transaction $*$T = new Transaction;
$T \rightarrow g\_its$ = NIL
$local\_abort\_count$ = 0
label:
while (true) **do**
  if $(T \rightarrow g\_its$!= $NIL)$ **then**
    $its = T \rightarrow g\_its$
    $T = lib \rightarrow stm\text{-}begin(its)$
  else
    $T = lib \rightarrow stm\text{-}begin(T \rightarrow g\_its)$
  end if
  for all (OP_LT_SEED) **do**
    $t\_obj = rand()\%T\_OBJ\_SEED$
    $randVal = rand()\%OP\_SEED$
    if $(randVal <= READ\_PER)$ **then**
      $stm\text{-}read(t\_obj, value)$
      if (value == $ABORTED$) **then**
        $local\_abort\_count$++
        goto label
      end if
    else
      $stm\text{-}write(t\_obj, value)$
    end if
  end for
  if $(lib \rightarrow stm\text{-}tryC() == ABORTED)$ **then**
    $local\_abort\_count$++
    continue
  end if
  break
end while