Achieving Starvation-Freedom in Multi-version Transactional Memory Systems

Ved P. Chaudhary[1(✉)], Chirag Juyal[1(✉)], Sandeep Kulkarni[2(✉)], Sweta Kumari[1(✉)], and Sathya Peri[1(✉)]

[1] CSE Department, IIT Hyderabad, Sangareddy, India
{cs14mtech11019,cs17mtech11014,cs15resch01004,sathya_p}@iith.ac.in
[2] CSE Department, Michigan State University, East Lansing, MI, USA
sandeep@cse.msu.edu

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

1 Introduction

STMs [1,2] 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

Author sequence follows lexical order of last names.

M. F. Atig and A. A. Schwarzmann (Eds.): NETYS 2019, LNCS 11704, pp. 291–310, 2019.
https://doi.org/10.1007/978-3-030-31277-0_20

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 [3], local opacity [4,5]. 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 [6], 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 [3]. Thus in this paper, we focus on optimistic executions with the *correctness-criterion* being *local opacity* [5].

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 [7, chap. 2], [8]. In the context of STMs, starvation-freedom ensures that every aborted transaction that is retried infinitely often eventually commits. It can be defined as: an STM system is said to be *starvation-free* if a thread invoking a transaction T_i gets the opportunity to retry T_i on every abort (due to the presence of a fair underlying scheduler with bounded termination) and T_i is not *parasitic*, i.e., T_i will try to commit given a chance then T_i will eventually commit. Parasitic transactions [9] 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 [8]. But it was shown by Guerraoui and Kapalka [9] 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 in literature such as Gramoli et al. [10], Waliullah and Stenstrom [11], Spear et al. [12]. 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.

Figure 1 illustrates this problem. Consider the execution: $r_1(x,0)r_1(y,0)$ $w_2(x,10)w_2(z,10)w_3(y,15)w_1(z,7)$. 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*.

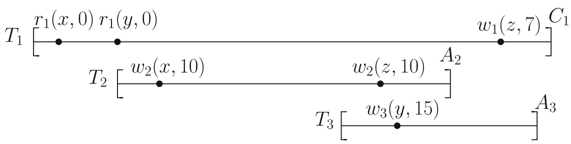

Fig. 1. Limitation of single-version starvation free algorithm

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 [13–16] 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 $T_1 T_2 T_3$ or $T_1 T_3 T_2$. Thus multiple versions can help with starvation-freedom without sacrificing on concurrency. This motivated us to develop a multi-version starvation-free STM system.

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 [13] *(Multi-Version Timestamp Order)* as follows:
  - First, in Subsect. 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*. 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* (Subsect. 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* (Subsect. 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 (Sect. 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 [17] and ESTM [18] 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 [19] 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.

2 System Model and Preliminaries

Following [5,20], 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-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-tryC*() and *stm-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-tryC*$_k(\mathscr{C})$ (or c_k for short), *stm-tryC*$_k(\mathscr{A})$, *stm-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.

Due to lack of space, we define other useful notions used in this paper such as opacity [3], local opacity [4,5], strict-serializability [6] formally in technical report [21].

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 [13,14]. 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 [13,22] 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 Subsect. 3.1, we define starvation freedom and identify assumptions made in the paper. Subsection 3.2 discusses data structures for all the algorithms developed in this section. Subsection 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. Subsection 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. Subsection 3.5 describes *KSFTM* algorithm that simultaneously maintains correctness, strict-serializability and local opacity while providing starvation-freedom.

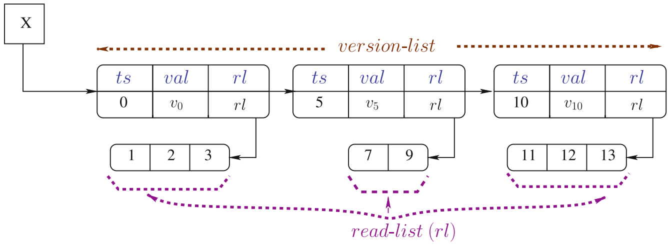

Fig. 2. Data structures for maintaining versions

3.1 Starvation-Freedom Explanation

This section starts with the definition of starvation-freedom. Then we describe the assumption that we make 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 [8], 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 Subsect. 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_tCntr.

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 Sect. 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 [13], 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.

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

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

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

3.3 Priority-Based MVTO Algorithm

In this subsection, we describe a modification to the multi-version timestamp ordering (MVTO) algorithm [13,22] 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 Subsect. 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_tCntr. If the input t is null, then $cts_i = its_i = ts$ as this is the first incarnation of this transaction. Otherwise, the non-null value of t is assigned as its_i.

stm-read(x): Transaction T_i reads from a version of x in the shared memory (if x does not exist in T_i's local buffer) with timestamp j such that j is the largest timestamp less than i (among the versions 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 [23]. 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 the technical report [21]. We have the following property on the correctness 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 the technical report [21].

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-tryC* operation). We use r_{50} and r_{60} to denote their read operations while w_{50} and w_{60} to denote their *stm-tryC* operations. Here, a read operation will not fail as there is a previous version present.

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

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

Table 1. Permutations of operations

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}

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-read*, *stm-write* and *stm-tryC* operations of *PKTO*. In SFKTO, a transaction T_i uses wts_i to read a version in *stm-read*. Similarly, T_i uses wts_i in *stm-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 the technical report [21].

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

We can allow the transaction T_i to commit before its wts_i has caught up with the actual time if it does not violate the real-time ordering. Thus, to ensure that the notion of real-time order is respected by transactions in the course of their execution in SFKTO, we add extra time constraints. We use the idea of timestamp ranges. This notion of timestamp ranges was first used by Riegel et al. [24] in the context of multi-version STMs. Several other researchers have used this idea since then such as Guerraoui et al. [25], Crain et al. [26] etc.

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

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

[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.

decremented such that it becomes less than $tltl_k$. Again, note that all such T_k are serialized after T_i.

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

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

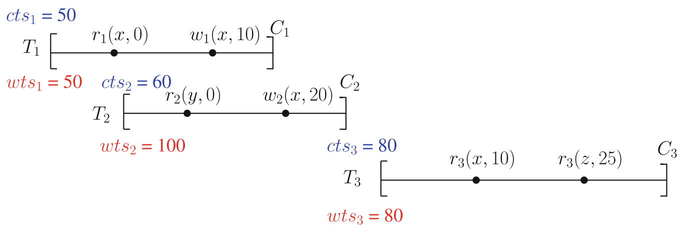

Fig. 3. Correctness of *KSFTM* algorithm

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

Based on this idea, we next develop a variation of 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-begin*, *stm-read*, *stm-write* and *stm-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-begin(t): A unique timestamp ts is allocated to T_i which is its CTS (i from our assumption) which is generated by atomically incrementing the global counter G_tCntr. If the input t is null then $cts_i = its_i = ts$ as this is the first incarnation of this transaction. Otherwise, the non-null value of t is assigned to its_i. Then, WTS is computed by Eq. 1. Finally, tltl and tutl are initialized as: $tltl_i = cts_i$, $tutl_i = \infty$.

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

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

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

stm-tryC : This operation consists of multiple steps:

1. Before T_i can commit, we need to verify that any version it creates is updated consistently. T_i creates a new version with timestamp wts_i. Hence, we must ensure that any transaction that read a previous version is unaffected by this new version. Additionally, creating this version would require an update of tltl and tutl of T_i and other transactions whose read-write set overlaps with that of T_i. Thus, T_i first validates each t-object x in its $wset$ as follows:
   (a) T_i finds a version of x with timestamp j such that j is the largest timestamp less than wts_i (like in *stm-read*). If there exists no version of x with a timestamp less than wts_i then T_i is aborted. This is similar to Step 1b of the *stm-tryC* of $PKTO$ algorithm.
   (b) Among all the transactions that have previously read from j suppose there is a transaction T_k such that $j < wts_i < wts_k$. Then (i) if T_k has already committed then T_i is aborted; (ii) Suppose T_k is live, and its_k is less than its_i. Then again T_i is aborted; (iii) If T_k is still live with its_i less than its_k then T_k is aborted.
   This step is similar to Step 1a of the *stm-tryC* of $PKTO$ algorithm.
   (c) Next, we must ensure that T_i's tltl and tutl are updated correctly w.r.t to other concurrently executing transactions. To achieve this, we adjust tltl, tutl as follows: (i) Let j be the ts of the largest version smaller than wts_i. Then $tltl_i = max(tltl_i, x[j].\texttt{vrt} + 1)$. Next, for each reading transaction,

T_r in $x[j].read\text{-}list$, we again set, $tltl_i = max(tltl_i, tutl_r + 1)$. (ii) Similarly, let p be the ts of the smallest version larger than wts_i. Then, $tutl_i = min(tutl_i, x[p].\texttt{vrt} - 1)$. (Note that we don't have to check for the transactions in the read-list of $x[p]$ as those transactions will have tltl higher than $x[p].\texttt{vrt}$ due to *stm-read*.) (iii) Finally, we get the commit time of this transaction from G_tCntr: $comTime_i = G\_tCntr.add\&Get(incrVal)$ where $incrVal$ is any constant ≥ 1. Then, $tutl_i = min(tutl_i, comTime_i)$. After performing these updates, abort T_i if tltl and tutl have crossed, i.e., $tltl_i > tutl_i$.

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

Step 1c.(iii) of *stm-tryC* ensures that real-time order between transactions that are not in conflict. It can be seen that locks have to be used to ensure that all these methods to execute in a linearizable manner (i.e., atomically). The detailed pseudo code along with the description can be found in accompanying technical report [21]. We get the following nice properties on *KSFTM* with the complete details in [21]. For simplicity, we assumed C and $incrVal$ to be 0.1 and 1 respectively in our analysis. But the proof and the analysis holds for any value greater than 0.

Theorem 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 implemented the the algorithms *PKTO*, *SV-SFTM* [10–12] along with *KSFTM* in C++[2] We used the available implementations of NOrec STM [17], and ESTM [18] developed in C++. Although, only *KSFTM* and *SV-SFTM* provide starvation-freedom, we compared with other STMs as well, to see its performance in practice.

Experimental System: The experimental system is a 2-socket Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60 GHz with 14 cores per socket and 2 hyper-threads (HTs) per core, for a total of 56 threads. Each core has a private 32KB L1 cache and 256 KB L2 cache. The machine has 32 GB of RAM and runs Ubuntu

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

16.04.2 LTS. In our implementation, all threads have the same base priority and we use the default Linux scheduling algorithm. This satisfies the Assumption 1 (bounded-termination) about the scheduler. We ensured that there no parasitic transactions [27] 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 operations on randomly chosen t-objects. A thread continues to invoke a transaction until it successfully commits. To obtain high contention, we have taken large number of threads ranging from 50–250 where each thread performs its read/write operation over a set 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 to the technical report [21] for details). **(2)** Two benchmarks from STAMP suite [19] - (a) We considered KMEANS which has low contention with short running transactions. The number of data points as 2048 with 16 dimensions and total clusters as 5. (b) We then considered LABYRINTH which has high contention with long running transactions. We considered the grid size as 64x64x3 and paths to route as 48.

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

Results Analysis: Fig. 4 illustrates max-time analysis of *KSFTM* over the above mentioned STMs for the counters application under the workloads $W1$, $W2$ and $W3$ while varying the number of threads from 50 to 250. For *KSFTM* and *PKTO*, we chose the value of K as 5 and C as 0.1 as the best results were obtained with these parameters (refer to the technical report [21] for details). We can see that *KSFTM* performs the best for all the three workloads. *KSFTM* gives an average speedup on max-time by a factor of 1.22, 1.89, 23.26 and 13.12 over *PKTO*, *SV-SFTM*, NOrec STM and ESTM respectively.

Figure 5(a) shows analysis of max-time for KMEANS while Fig. 5(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 and 1.44 times better than *PKTO* and *SV-SFTM*. But, NOrec is performing 1.09 times better than *KSFTM*. This is because KMEANS has short running transactions have low contention. As a result, the commit time of the transactions is also low.

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

Figure 5(c) shows the stability of *KSFTM* algorithm over time for the counter application. Here we fixed the number of threads to 32, K as 5, C as 0.1, t-objects as 1000, along with 5 s warm-up period on $W1$ workload. Each thread invokes transactions until its time-bound of 60 s expires. We performed the experiments on number of transactions committed over time in the increments 5 s. 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.

We have executed several experiments to study various parameters such as average case analysis, number of aborts, effect of garbage-collection, best value of K and optimal value of C. These are explained in detail in the technical report [21].

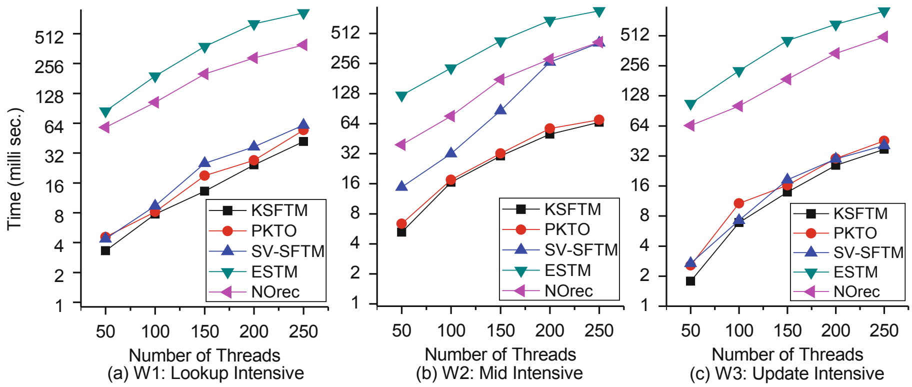

Fig. 4. Performance analysis on workload $W1$, $W2$, $W3$

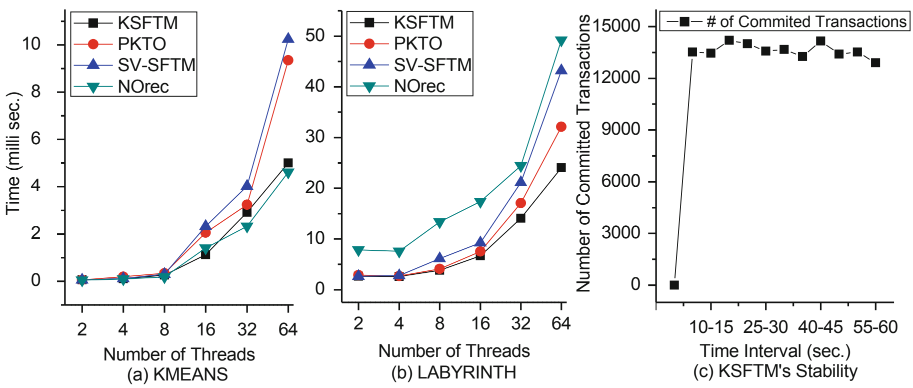

Fig. 5. Performance analysis on KMEANS, LABYRINTH and KSFTM's Stability

5 Conclusion

In this paper, we proposed *KSFTM* 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 [6] and local opacity [4,5]. 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.

Acknowledgments. We are thankful to the anonymous reviewers for carefully reading the paper and providing us valuable suggestions.

References

1. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News **21**(2) (1993)
2. Shavit, N., Touitou, D.: Software transactional memory. In: PODC (1995)
3. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: PPoPP (2008)
4. Kuznetsov, P., Peri, S.: Non-interference and local correctness in transactional memory. In: ICDCN (2014) 197–211
5. Kuznetsov, P., Peri, S.: Non-interference and local correctness in transactional memory. Theor. Comput. Sci. **688**, 103–116 (2017)
6. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4) (1979)
7. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, Revised Reprint, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2012)
8. Herlihy, M., Shavit, N.: On the nature of progress. In: Fernàndez Anta, A., Lipari, G., Roy, M. (eds.) OPODIS 2011. LNCS, vol. 7109, pp. 313–328. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25873-2_22
9. Bushkov, V., Guerraoui, R., Kapalka, M.: On the liveness of transactional memory. In: ACM Symposium on PODC (2012)
10. Gramoli, V., Guerraoui, R., Trigonakis, V.: TM2C: a software transactional memory for many-cores. In: EuroSys (2012)
11. Waliullah, M.M., Stenström, P.: Schemes for avoiding starvation in transactional memory systems. Concurr. Comput. Pract. Exp. **21**(7), 859–873 (2009)
12. Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: A comprehensive strategy for contention management in software transactional memory (2009)
13. Kumar, P., Peri, S., Vidyasankar, K.: A TimeStamp based multi-version STM algorithm. In: Chatterjee, M., Cao, J., Kothapalli, K., Rajsbaum, S. (eds.) ICDCN 2014. LNCS, vol. 8314, pp. 212–226. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-45249-9_14
14. Lu, L., Scott, M.L.: Generic multiversion STM. In: Afek, Y. (ed.) DISC 2013. LNCS, vol. 8205, pp. 134–148. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41527-2_10

15. Fernandes, S.M., Cachopo, J.: Lock-free and scalable multi-version software transactional memory. In: PPoPP (2011)
16. Perelman, D., Byshevsky, A., Litmanovich, O., Keidar, I.: SMV: selective multi-versioning STM. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 125–140. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24100-0_9
17. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: streamlining STM by abolishing ownership records. In: PPoPP (2010)
18. Felber, P., Gramoli, V., Guerraoui, R.: Elastic transactions. J. Parallel Distrib. Comput. **100**(C), 103–127 (2017)
19. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: stanford transactional applications for multi-processing. In: IISWC (2008)
20. Guerraoui, R., Kapalka, M.: Principles of Transactional Memory, Synthesis Lectures on Distributed Computing Theory. Morgan and Claypool, San Rafael (2010)
21. Chaudhary, V.P., Juyal, C., Kulkarni, S.S., Kumari, S., Peri, S.: Starvation freedom in multi-version transactional memory systems. CoRR abs/1709.01033 (2017)
22. Bernstein, P.A., Goodman, N.: Multiversion concurrency control: theory and algorithms. ACM Trans. Database Syst **8**, 465–483 (1983)
23. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
24. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 284–298. Springer, Heidelberg (2006). https://doi.org/10.1007/11864219_20
25. Guerraoui, R., Henzinger, T.A., Singh, V.: Permissiveness in transactional memories. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 305–319. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87779-0_21
26. Crain, T., Imbs, D., Raynal, M.: Read invisibility, virtual world consistency and probabilistic permissiveness are compatible. In: Xiang, Y., Cuzzocrea, A., Hobbs, M., Zhou, W. (eds.) ICA3PP 2011. LNCS, vol. 7016, pp. 244–257. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24650-0_21
27. Bushkov, V., Guerraoui, R.: Liveness in transactional memory. In: Guerraoui, R., Romano, P. (eds.) Transactional Memory. Foundations, Algorithms, Tools, and Applications. LNCS, vol. 8913, pp. 32–49. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-14720-8_2