Achieving Starvation-Freedom with Greater Concurrency in Multi-Version Object-based Transactional Memory Systems*

Chirag Juyal[1], Sandeep Kulkarni[2], Sweta Kumari[1], Sathya Peri[1], and Archit Somani[1‡]
[1] Department of Computer Science & Engineering, IIT Hyderabad, Kandi, Telangana, India
(cs17mtech11014, cs15resch01004, sathya_p,
cs15resch01001)@iith.ac.in
[2] Department of Computer Science, Michigan State University, MI, USA
sandeep@cse.msu.edu

Abstract. To utilize the multi-core processors properly concurrent programming is needed. The main challenge is to design a correct and efficient concurrent program. Software Transactional Memory Systems (STMs) provide ease of multi-threading to the programmer without worrying about concurrency issues as deadlock, livelock, priority inversion, etc. Most of the STMs work on read-write operations known as RWSTMs. Some STMs work at higher-level operations and ensure greater concurrency than RWSTMs. Such STMs are known as Single-Version Object-based STMs (SVOSTMs). The transactions of SVOSTMs can return *commit* or *abort*. Aborted SVOSTMs transactions retry. But in the current setting of SVOSTMs, transactions may *starve*. So, we propose a *Starvation-Freedom* in *SVOSTM* as *SF-SVOSTM* that satisfies the correctness criteria *conflict-opacity*. Databases and STMs say that maintaining multiple versions corresponding to each shared data-item (or key) reduces the number of aborts and improves the throughput. So, to achieve greater concurrency further, we propose *Starvation-Freedom* in *Multi-Version OSTM* as *SF-MVOSTM* algorithm. The number of versions maintains by SF-MVOSTM either be unbounded with garbage collection as SF-MVOSTM-GC or bounded with latest K-versions as SF-KOSTM. SF-MVOSTM satisfies the correctness criteria as *local opacity* and shows the performance benefits as compared with state-of-the-art STMs.

Keywords: Software Transactional Memory Systems, Concurrency Control, Starvation-Freedom, Multi-version, Opacity, Local Opacity

1 Introduction

In the era of multi-core processors, we can exploit the cores by concurrent programming. But developing an efficient concurrent program while ensuring the correctness is difficult. Software Transactional Memory Systems (STMs) are a convenient programming interface to access the shared memory concurrently while removing the concurrency responsibilities from the programmer. STMs ensure that consistency issues such as deadlock, livelock, priority inversion, etc will not occur. It provides a high-level abstraction to the programmer with the popular correctness criteria opacity [1], local opacity [2] which consider all the transactions (a piece of code) including aborted one as well in the equivalent serial history. This property makes it different from correctness criteria of

‡Author sequence follows the lexical order of last names.

*This research is partially supported by IMPRINT India project 6918F & gift from Intel, USA.

database serializability, strict-serializability [3] and ensures even aborted transactions read correct value in STMs which prevent from divide-by-zero, infinite loop, crashes, etc. Another advantage of STMs is composability which ensures the effect of multiple operations of the transaction will be atomic. This paper considers the optimistic execution of STMs in which transactions are writing into its local log until the successful validation.

A traditional STM system invokes following methods:(1) *STM_begin*(): begins a transaction T_i with unique timestamp i. (2) *STM_read*$_i(k)$ (or $r_i(k)$): T_i reads the value of key k from shared memory. (3) *STM_write*$_i(k, v)$ (or $w_i(k, v)$): T_i writes the value of k as v locally. (4) *STM_tryC*$_i$*()*: on successful validation, the effect of T_i will be visible to the shared memory and T_i returns commit otherwise (5) *STM_tryA*$_i$*()*: T_i returns abort. These STMs are known as *read-write STMs (RWSTMs)* because it is working at lower-level operations such as read and write.

Herlihy et al. [4], Hassan et al. [5], and Peri et al. [6] have shown that working at higher-level operations such as insert, delete and lookup on the linked-list and hash table gives better concurrency than RWSTMs. STMs which work on higher-level operations are known as *Single-Version Object-based STMs (SVOSTMs)* [6]. It exports the following methods: (1) *STM_begin()*: begins a transaction T_i with unique timestamp i same as RWSTMs. (2) *STM_lookup*$_i(k)$ (or $l_i(k)$): T_i lookups key k from shared memory and returns the value. (3) *STM_insert*$_i(k, v)$ (or $i_i(k, v)$): T_i inserts a key k with value v into its local memory. (4) *STM_delete*$_i(k)$(or $d_i(k)$): T_i deletes key k. (5) *STM_tryC*$_i$*()*: the actual effect of *STM_insert()* and *STM_delete()* will be visible to the shared memory after successful validation and T_i returns commit otherwise (6) *STM_tryA*$_i$*()*: T_i returns abort.

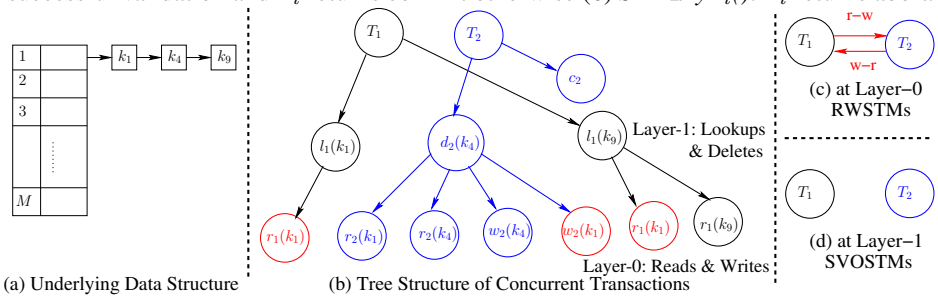


Fig. 1: Advantage of SVOSTMs over RWTSMs

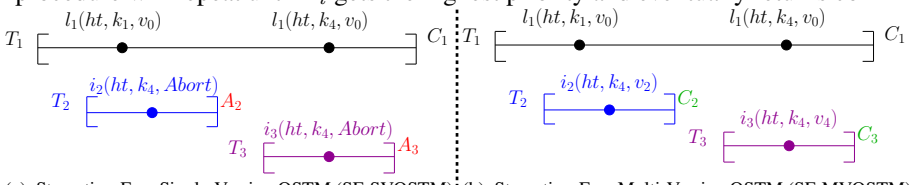
Motivation to work on SVOSTMs: Fig 1 represents the advantage of SVOSTMs over RWSTMs while achieving greater concurrency and reducing the number of aborts. Fig 1.(a) depicts the underlying data structure as a hash table (or ht) with M buckets and bucket 1 stores three keys k_1, k_4 and k_9 in the form of the list. Thus, to access k_4, a thread has to access k_1 before it. Fig 1.(b) shows the tree structure of concurrent execution of two transactions T_1 and T_2 with RWSTMs at layer-0 and SVOSTMs at layer-1 respectively. Consider the execution at layer-0, T_1 and T_2 are in conflict because write operation of T_2 on key k_1 as $w_2(k_1)$ is occurring between two read operations of T_1 on k_1 as $r_1(k_1)$. Two transactions are in conflict if both are accessing the same key k and at least one transaction performs write operation on k. So, this concurrent execution cannot be atomic as shown in Fig 1.(c). To make it atomic either T_1 or T_2 has to return abort. Whereas execution at layer-1 shows the higher-level operations $l_1(k_1)$, $d_2(k_4)$ and $l_1(k_9)$ on different keys k_1, k_4 and k_9 respectively. All the higher-level operations

are isolated to each other so the tree can be pruned [7, Chap 6] from layer-0 to layer-1 and both the transactions return commit with equivalent serial schedule T_1T_2 or T_2T_1 as shown in Fig 1.(d). Hence, some conflicts of RWSTMs does not matter at SVOSTMs which reduce the number of aborts and improve the concurrency using SVOSTMs.

Starvation-Freedom: For long-running transactions along with high conflicts, starvation can occur in SVOSTMs. So, SVOSTMs should ensure the progress guarantee as *starvation-freedom* [8, chap 2]. SVOSTMs 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 [9] with bounded termination) and T_i is not *parasitic*, i.e., if scheduler will give a fair chance to T_i to commit then T_i will eventually return commit. If a transaction gets a chance to commit, still it is not committing because of the infinite loop or some other error such transactions are known as Parasitic transactions [10].

We explored another well known non-blocking progress guarantee *wait-freedom* for STM which ensures every transaction commits regardless of the nature of concurrent transactions and the underlying scheduler [11]. However, Guerraoui and Kapalka [10, 12] showed that achieving wait-freedom is impossible in dynamic STMs in which data-items (or keys) of transactions are not known in advance. So in this paper, we explore the weaker progress condition of *starvation-freedom* for SVOSTM while assuming that the keys of the transactions are not known in advance.

Related work on Starvation-free STMs: Some researchers Gramoli et al. [13], Waliullah and Stenstrom [14], Spear et al. [15], Chaudhary et al. [9] have explored starvation-freedom in RWSTMs. Most of them assigned priority to the transactions. On conflict, higher priority transaction returns commit whereas lower priority transaction returns abort. On every abort, a transaction retries a sufficient number of times, will eventually get the highest priority and returns commit. We inspired by this research and propose a novel *Starvation-Free SVOSTM (SF-SVOSTM)* which assigns the priority to the transaction on conflict. In SF-SVOSTM whenever a conflicting transaction T_i aborts, it retries with T_j which has higher priority than T_i. To ensure the starvation-freedom, this procedure will repeat until T_i gets the highest priority and eventually returns commit.



(a). Starvation-Free Single-Version OSTM (SF-SVOSTM) (b). Starvation-Free Multi-Version OSTM (SF-MVOSTM)

Fig. 2: Benefits of Starvation-Free Multi-Version OSTM over SF-SVOSTM

Motivation to Propose Starvation-Freedom in Multi-Version OSTM: In SF-SVOSTM, if the highest priority transaction becomes slow (for some reason) then it may cause several other transactions to abort and bring down the progress of the system. Fig 2.(a). demonstrates this in which the highest priority transaction T_1 became slow so, it is forcing the conflicting transactions T_2 and T_3 to abort again and again until T_1 commits. Database, RWSTMs [16–19] and SVOSTMs [20] say that maintaining multiple versions corresponding to each key reduces the number of aborts and improves throughput.

So, this paper proposes the first *Starvation-Free Multi-Version OSTM (SF-MVOSTM)* which maintains multiple versions corresponding to each key. Fig 2.(b). shows the bene-

fits of using SF-MVOSTM in which T_1 lookups from the older version with value v_0 created by transaction T_0 (assuming as initial transaction) for key k_1 and k_4. Concurrently, T_2 and T_3 create the new versions for key k_4. So, all the three transactions commit with equivalent serial schedule $T_1T_2T_3$. So, SF-MVOSTM improves the concurrency than SF-SVOSTM while reducing the number of aborts and ensures the starvation-freedom.
Contributions of the paper: We propose two Starvation-Free OSTMs as follows:

- Initially, we propose Starvation-Freedom for Single-Version OSTM as SF-SVOSTM which satisfies correctness criteria as *conflict-opacity (or co-opacity)* [6].
- To achieve the greater concurrency further, we propose Starvation-Freedom for Multi-Version OSTM as SF-MVOSTM in Section 3 which maintains multiple versions corresponding to each key and satisfies the correctness as *local opacity* [2].
- We propose SF-SVOSTM and SF-MVOSTM for *hash table* and *linked-list* data structure describe in SubSection3.2 but its generic for other data structures as well.
- SF-MVOSTM works for unbounded versions with *Garbage Collection (GC)* as SF-MVOSTM-GC which deletes the unwanted versions from version list of keys and for bounded/finite versions as SF-KOSTM which stores finite say latest K number of versions corresponding to each key k. So, whenever any thread creates $(K+1)^{th}$ version of key, it replaces the oldest version of it. The most challenging task is achieving starvation-freedom in bounded version OSTM because say, the highest priority transaction relies on the oldest version that has been replaced. So, in this case, highest priority transaction has to return abort and hence make it harder to achieve starvation-freedom unlike the approach follow in SF-SVOSTM. Thus, in this paper, we propose a novel approach SF-KOSTM which bridges the gap by developing starvation-free OSTM while maintaining bounded number of versions.
- Section 4 shows that SF-KOSTM is best among all proposed Starvation-Free OSTMs (SF-SVOSTM, SF-MVOSTM, and SF-MVOSTM-GC) for both hash table and linked-list data structure. Proposed hash table based SF-KOSTM (HT-SF-KOSTM) performs 3.9x, 32.18x, 22.67x, 10.8x and 17.1x average speedup on *max-time* for a transaction to commit than state-of-the-art STMs HT-KOSTM [20], HT-SVOSTM [6], ESTM [21], RWSTM [7, Chap. 4], and HT-MVTO [16] respectively. Proposed list based SF-KOSTM (list-SF-KOSTM) performs 2.4x, 10.6x, 7.37x, 36.7x, 9.05x, 14.47x, and 1.43x average speedup on *max-time* for a transaction to commit than state-of-the-art STMs list-KOSTM [20], list-SVOSTM [6], Trans-list [22], Boosting-list [4], NOrec-list [23], list-MVTO [16], and list-KSFTM [9] respectively.

2 System Model and Preliminaries

This section follows the notion and definition described in [12, 20], we assume a system of n processes/threads, $th_1, \ldots, th_n$ that run in a completely asynchronous manner and communicate through a set of *keys* $\mathscr{K}$ (or *transaction-objects*). We also assume that none of the threads crash or fail abruptly. In this paper, a thread executes higher-level methods on $\mathscr{K}$ via atomic *transactions* $T_1, \ldots, T_n$ and receives the corresponding response.
Events and Methods: Threads execute the transactions with higher-level methods (or operations) which internally invoke multiple read-write (or lower-level) operations known as *events* (or $evts$). Transaction T_i of the system at read-write level invokes *STM_begin()*, *STM_read$_i$(k)*, *STM_write$_i$(k,v)*, *STM_tryC$_i$()* and *STM_tryA$_i$()* as defined

in Section 1. We denote a method m_{ij} as the j^{th} method of T_i. Method *invocation* (or inv) and *response* (or rsp) on higher-level methods are also considered as an event.

A thread executes higher-level operations on $\mathscr{K}$ via transaction T_i are known as *methods* (or $mths$). T_i at object level (or higher-level) invokes *STM_begin()*, *STM_lookup$_i(k)$ (or $l_i(k)$)*, *STM_insert$_i(k, v)$ (or $i_i(k, v)$)*, *STM_delete$_i(k)$ (or $d_i(k)$)*, *STM_tryC$_i()$*, and *STM_tryA$_i()$* methods described in Section 1. Here, *STM_lookup()*, and *STM_delete()* return the value from underlying data structure so, we called these methods as *return value methods (or $rv\_methods$)*. Whereas, *STM_insert()*, and *STM_delete()* are updating the underlying data structure after successful *STM_tryC()* so, we called these methods as *update* methods (or $upd\_methods$).

Transactions: We follow multi-level transactions [7] model which consists of two layers. Layer 0 (or lower-level) composed of read-write operations whereas layer 1 (or higher-level) comprises of object-level methods which internally calls multiple read-write events. Formally, we define a transaction T_i at higher-level as the tuple $\langle evts(T_i), <_{T_i} \rangle$, here $<_{T_i}$ represents the total order among all the events of T_i. Transaction T_i cannot invoke any more operation after returning commit ($\mathscr{C}$) or abort ($\mathscr{A}$). Any operation that returns $\mathscr{C}$ or $\mathscr{A}$ are known as *terminal operations* represented as $Term(T_i)$. The transaction which neither committed nor aborted is known as *live transactions* (or $trans.live$).

Histories: A history H consists of multiple transactions, a transaction calls multiple methods and each method internally invokes multiple read-write events. So, a history is a collection of events belonging to the different transactions is represented as $evts(H)$. Formally, we define a history H as the tuple $\langle evts(H), <_H \rangle$, here $<_H$ represents the total order among all the events of H. If all the method invocation of H match with the corresponding response then such history is known as *complete history* denoted as $\overline{H}$. Suppose total transactions in H is $H.trans$, in which number of committed and aborted transactions are $H.committed$ and $H.aborted$ then the *incomplete history* or *live history* is defined as: $H.incomp$ = $H.live$ = $(H.trans$ - $H.committed$ - $H.aborted)$. This paper considers only *well-form history* which ensures (1) the response of the previous method has received then only the transaction T_i can invoke another method. (2) transaction can not invoke any other method after receiving the response as $\mathscr{C}$ or $\mathscr{A}$.

Due to lack of space, we define other useful notions and definitions used in this paper such as sequential histories [2], real-time order and serial history [3], valid and legal history [20], sub-histories [9], conflict-opacity [6], opacity [1], strict-serializability [3], local opacity [2] formally in accompanying technical report [24].

3 The Proposed SF-KOSTM Algorithm

In this section, we propose *Starvation-Free K-version OSTM (SF-KOSTM)* algorithm which maintains K number of versions corresponding to each key. The value of K is application dependent and may vary from 1 to ∞. When K is equal to 1 then SF-KOSTM boils down to *Starvation-Free Single-Version OSTM (SF-SVOSTM)*. When K is ∞ then SF-KOSTM maintains unbounded versions corresponding to each key known as *Starvation-Free Multi-Version OSTM (SF-MVOSTM)* algorithm. To delete the unused version from the version list of SF-MVOSTM, we develop a separate Garbage Collection (GC) method [16] and propose SF-MVOSTM-GC. In this paper, we propose SF-SVOSTM and all the variants of SF-KOSTM *(SF-MVOSTM, SF-MVOSTM-GC,*

SF-KOSTM) for two data structures *hash table* and *linked-list* but it is generic for other data structures as well.

SubSection3.1 describes the definition of *starvation-freedom* followed by our assumption about the scheduler that helps us to achieve starvation-freedom in SF-KOSTM. SubSection3.2 explains the design and data structure of SF-KOSTM. SubSection3.3 shows the working of SF-KOSTM algorithm.

3.1 Description of Starvation-Freedom

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

Herlihy & Shavit [11] defined the fair scheduler which ensures that none of the thread will crash or delayed forever. Hence, any thread Th_i acquires the lock on the shared data-items while executing transaction T_i will eventually release the locks. So, a thread will never block other threads to progress. To satisfy the starvation-freedom for SF-KOSTM, we assumed bounded termination for the fair scheduler.

Assumption 1 ***Bounded-Termination:*** *For any transaction T_i, invoked by a thread Th_i, the fair system scheduler ensures, in the absence of deadlocks, Th_i is given sufficient time on a CPU (and memory, etc) such that T_i terminates ($\mathscr{C}$ or $\mathscr{A}$) in bounded time.*

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

Assumption 2 *We assume, if other concurrent conflicting transactions do not exist in the system then every transaction will commit. i.e. (a) If a transaction T_i is executing in the system with the absence of other conflicting transactions then T_i will not self-abort. (b) Transactions of the system are non-parasitic as explained in Section 1.*

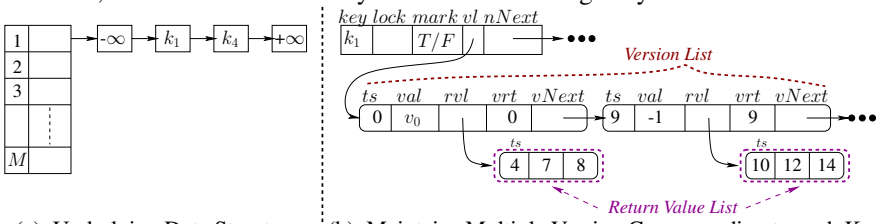
If transactions self-abort or parasitic then ensuring starvation-freedom is impossible.

3.2 Design and Data Structure of SF-KOSTM Algorithm

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

To achieve the *Starvation-Freedom* in *K-version Object-based STM (SF-KOSTM)*, we use chaining hash table (or ht) as an underlying data structure where the size of the hash table is *M* buckets as shown in Fig 3.(a) and we propose HT-SF-KOSTM. Hash table with bucket size one becomes the linked-list data structure for SF-KOSTM represented as list-SF-KOSTM. The representation of SF-KOSTM is similar to MVOSTM [20]. Each bucket stores multiple nodes in the form of linked-list between the two sentinel nodes *Head*(-∞) and *Tail*(+∞). Fig 3.(b) illustrates the structure of each node as $\langle$*key, lock, mark, vl, nNext*$\rangle$. Where *key* is the unique value from the range of [1 to $\mathscr{K}$] stored in the increasing order between the two sentinel nodes similar to linked-list based concurrent

set implementation [25, 26]. The *lock* field is acquired by the transaction before updating (inserting or deleting) on the node. *mark* is the boolean field which says anode is deleted or not. If *mark* sets to true then node is logically deleted else present in the hash table. Here, the deletion is in a lazy manner similar to concurrent linked-list structure [25]. The field *vl* stands for version list. SF-KOSTM maintains the finite say latest K-versions corresponding to each key to achieving the greater concurrency as explained in Section 1. Whenever $(K+1)^{th}$ version created for the key then it overwrites the oldest version corresponding to that key. If K is equal to 1, i.e., version list contains only one version corresponding to each key which boils down to *Starvation-Free Single-Version OSTM (SF-SVOSTM)*. So, the data structure of SF-SVOSTM is same as SF-KOSTM with one version. The field *nNext* points to next available node in the linked-list. From now onwards, we will use the term key and node interchangeably.



(a). Underlying Data Structure (b). Maintains Multiple Version Corresponding to each Key

Fig. 3: Design and Data Structure of SF-KOSTM

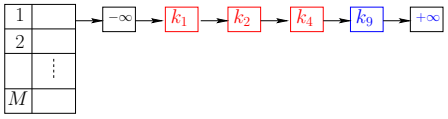


Fig. 4: Searching k_9 over *lazy-list*

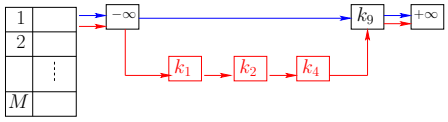


Fig. 5: Searching k_9 over $rblazy\text{-}list$

The structure of the *vl* is ⟨*ts, val, rvl, vrt, vNext*⟩ as shown in Fig 3.(b). *ts* is the unique timestamp assigned by the *STM_begin*(). If the value (*val*) is *nil* then version is created by the *STM_delete*() otherwise *STM_insert*() creates a version with not *nil* value. To satisfy the correctness criteria as *local opacity*, *STM_delete*() also maintains the version corresponding to each key with *mark* field as $true$. It allows the concurrent transactions to lookup from the older version of the marked node and returns the value as not *nil*. *rvl* stands for *return value list* which maintains the information about lookup transaction that has lookups from a particular version. It maintains the timestamp (*ts*) of rv_methods (*STM_lookup*() or *STM_delete*()) transaction in it. *vrt* stands for *version real-time* which helps to maintain the *real-time order* among the transactions. *vNext* points to the next available version in the version list.

Maintaining the deleted node along with the live (not deleted) node will increase the traversal time to search a particular node in thelist. Consider Fig 4, where red color depicts the deleted node $\langle k_1, k_2, k_4 \rangle$ and blue color depicts the live node $\langle k_9 \rangle$. When any method of SF-KOSTM searches the key k_9 then it has to traverse the deleted nodes $\langle k_1, k_2, k_4 \rangle$ as well before reach to k_9 that increases the traversal time.

This motivated us to modify the lazy-list structure of a node to form a skip list based on red and blue links. We called it as a *red-blue lazy-list* or *rblazy-list*. This idea has been explored by Peri et al. in SVOSTMs [6]. *rblazy-list* maintains two-pointer corresponding to each node such as red link (RL) and blue link (BL). Where BL points

to the live node and RL points to live node as well as a deleted node. Let us consider the same example as discussed above with this modification, key k_9 is directly searched from the head of the list with the help of BL as shown in Fig 5. In this case, traversal time is efficient because any method of SF-KOSTM need not traverse the deleted nodes. To maintain the RL and BL in each node we modify the structure of *lazy-list* as ⟨*key, lock, mark, vl, RL, BL, nNext*⟩ and called it as *rblazy-list*.

3.3 Working of SF-KOSTM Algorithm

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

SF-KOSTM invokes *STM_begin*(), *STM_lookup*(), *STM_delete*(), *STM_insert*(), and *STM_tryC*() methods. *STM_lookup*() and *STM_delete*() work as rv_methods() which lookup the value of key k from shared memory and return it. Whereas *STM_insert*() and *STM_delete*() work as upd_methods() that modify the value of k in shared memory. We propose optimistic SF-KOSTM, so, upd_methods() first update the value of k in transaction local log $txLog$ and the actual effect of upd_methods() will be visible after successful *STM_tryC*(). Now, we explain the functionality of each method as follows:
***STM_begin*(): ** When a thread Th_i invokes transaction T_i for the first time (or first incarnation), *STM_begin*() assigns a unique timestamp known as *current timestamp* (cts) using an atomic global counter $(gcounter)$. If T_i gets aborted then thread Th_i executes it again with the new incarnation of T_i, say T_j with the new cts until T_i commits but retains its initial cts as *initial timestamp* (its). Th_i uses its to inform the SF-KOSTM system that whether T_i is a new invocation or an incarnation. If T_i is the first incarnation then its and cts are same as cts_i so, Th_i maintains $\langle its_i, cts_i \rangle$. If T_i gets aborted and retries with T_j then Th_i maintains $\langle its_i, cts_j \rangle$.

By assigning priority to the lowest its transaction (i.e. transaction have been in the system for a longer time) in *Single-Version OSTM*, *Starvation-Freedom* can easily be achieved as explained in Section 1. The detailed working of *Starvation-Free Single-Version OSTM (SF-SVOSTM)* is in accompanying technical report [24]. But achieving *Starvation-Freedom* in finite *K-versions OSTM (SF-KOSTM)* is challenging. Though the transaction T_i has lowest its but T_i may return abort because of finite versions T_i did not find a correct version to lookup from or overwrite a version. Table 1 shows the key insight to achieve the starvation-freedom in finite K-versions OSTM. Here, we considered two transaction T_{10} and T_{20} with cts 10 and 20 that performs *STM_lookup()* (or l) and *STM_insert()* (or i) on same key k. We assume that a version of k exists with cts 5, so, *STM_lookup()* of T_{10} and T_{20} find a previous version to lookup and never return abort. Due to the optimistic execution in SF-KOSTM, effect of *STM_insert()* comes after successful *STM_tryC()*, so *STM_lookup()* of a transaction comes before effect of its *STM_insert()*. Hence, a total of six permutations are possible as defined in Table 1. We can observe from Table 1 that in some cases T_{10} returns abort. But if T_{20} gets the lowest its then T_{20} never returns abort. This ensures that a transaction with lowest its and highest cts will never return abort. But achieving highest cts along with lowest its is a bit difficult because new transactions are keep on coming with higher cts using $gcounter$. So, to achieve the highest cts, we introduce a new timestamp as *working timestamp (wts)* which is significantly larger than cts.

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

Table 1: Possible Permutations of Methods

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

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

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

Observation 1 *Any transaction T_i with lowest its_i and highest wts_i will never abort.*

Sometimes, the value of wts is significantly larger than cts. So, wts is unable to maintain *real-time order* between the transactions which violates the correctness of SF-KOSTM. To address this issue SF-KOSTM uses the idea of timestamp ranges [27–29] along with $\langle its_i, cts_i, wts_i \rangle$ for transaction T_i in *STM_begin*(). It maintains the *transaction lower timestamp limit ($tltl_i$)* and *transaction upper timestamp limit ($tutl_i$)* for T_i. Initially, $\langle its_i, cts_i, wts_i, tltl_i \rangle$ are the same for T_i. $tutl_i$ would be set as a largest possible value denoted as $+\infty$ for T_i. After successful execution of *rv_methods()* or *STM_tryC*() of T_i, $tltl_i$ gets incremented and $tutl_i$ gets decremented[3] to respect the real-time order among the transactions. *STM_begin*() initializes the *transaction local log* ($txLog_i$) for each transaction T_i to store the information in it. Whenever a transaction starts it atomically sets its *status* to be *live* as a global variable. Transaction *status* can be $\langle$*live, commit, false*$\rangle$. After successful execution of *STM_tryC*(), T_i sets its *status* to be *commit*. If the *status* of the transaction is *false* then it returns *abort*. For more details of *STM_begin*() please refer the accompanying technical report [24].

***STM_lookup*() and *STM_delete*() as rv_methods():** *rv_methods(ht, k, val)* return the value (*val*) corresponding to the key *k* from the shared memory as hash table (*ht*). We show the high-level overview of the rv_methods() in Algo 1. First, it identifies the key k in the transaction local log as $txLog_i$ for transaction T_i. If k exists then it updates the $txLog_i$ and returns the val at Line 3.

If key k does not exist in the $txLog_i$ then before identify the location in share memory rv_methods() check the *status* of T_i at Line 6. If *status* of T_i (or i) is *false* then T_i has to *abort* which says that T_i is not having the lowest its and highest wts among other concurrent conflicting transactions. So, to propose starvation-freedom in SF-KOSTM other conflicting transactions set the *status* of T_i as *false* and force it to *abort*.

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

If the *status* of T_i is not *false* and key k does not exist in the $txLog_i$ then it identifies the location of key k optimistically (without acquiring the locks similar to the *lazy-list* [25]) in the shared memory at Line 8. SF-KOSTM maintains the shared memory in the form of a hash table with M buckets as shown in SubSection3.2, where each bucket stores the keys in *rblazy-list*. Each node contains two pointer $\langle RL, BL \rangle$. So, it identifies the two *predecessors (pred)* and two *current (curr)* with respect to each node. First, it identifies the pred and curr for key k in BL as $\langle preds[0], currs[1] \rangle$. After that it identifies the pred and curr for key k in RL as $\langle preds[1], currs[0] \rangle$. If $\langle preds[1], currs[0] \rangle$ are not marked then $\langle preds[0] = preds[1], currs[1] = currs[0] \rangle$. SF-KOSTM maintains the keys are in increasing order. So, the order among the nodes are $\langle preds[0].key \leq preds[1].key < k \leq currs[0].key \leq currs[1].key \rangle$.

rv_methods() acquire the lock in predefined order on all the identified preds and currs for key k to avoid the deadlock at Line 9 and do the *rv_Validation()* at Line 10. If $\langle preds[0] \vee currs[1] \rangle$ is marked or preds are not pointing to identified currs as $\langle (preds[0].BL \neq currs[1]) \vee (preds[1].RL \neq currs[0]) \rangle$ then it releases the locks from all the preds and currs and identify the new preds and currs for k in shared memory.

Algorithm 1 *rv_methods(ht, k, val):* It can either be *STM_delete*$_i(ht, k, val)$ or *STM_lookup*$_i(ht, k, val)$ on key k by transaction T_i.

1: **procedure** $rv\_methods_i(ht, k, val)$
2: **if** $(k \in txLog_i)$ **then**
3: Update the local log of T_i and return val.
4: **else**
5: /*Atomically check the *status* of its own transaction T_i (or i).*/
6: **if** (*i.status* == *false*) **then** return $\langle abort_i \rangle$.
7: **end if**
8: Identify the *preds[]* and *currs[]* for key k in bucket M_k of *rblazy-list* using BL and RL.
9: Acquire locks on *preds[]* & *currs[]* in increasing order of keys to avoid the deadlock.
10: **if** (!*rv_Validation(preds[], currs[])*) **then**
11: Release the locks and goto Line 8.
12: **end if**
13: **if** $(k \notin M_k.rblazy\text{-}list)$ **then**
14: Create a new node n with key k as: $\langle$*key=k, lock=false, mark=true, vl=ver, nNext=*$\phi\rangle$*./*n is marked*/
15: Create version ver as:$\langle$*ts=0, val=nil, rvl=i, vrt=0, vNext=*$\phi\rangle$.
16: Insert n into M_k*.rblazy-list* s.t. it is accessible only via RLs. /**lock* sets *true**/
17: Release locks; update the $txLog_i$ with k.
18: return $\langle$*val*$\rangle$. /**val* as nil*/
19: **end if**
20: Identify the version ver_j with $ts = j$ such that j is the *largest timestamp smaller (lts)* than i.
21: **if** $(ver_j == nil)$ **then** /*Finite Versions*/
22: return $\langle abort_i \rangle$
23: **else if** $(ver_j.vNext$!= $nil)$ **then**
24: /*$tutl_i$ should be less then *vrt* of next version ver_j*/
25: Calculate $tutl_i$ = min($tutl_i, ver_j.vNext.vrt - 1$).
26: **end if**
27: /*$tltl_i$ should be greater then vrt of ver_j*/
28: Calculate $tltl_i$ = max($tltl_i, ver_j.vrt + 1$).
29: /*If limit has crossed each other then abort T_i*/
30: **if** $(tltl_i > tutl_i)$ **then** return $\langle abort_i \rangle$.
31: **end if**
32: Add i into the rvl of ver_j.
33: Release the locks; update the $txLog_i$ with k and value.
34: **end if**
35: return $\langle ver_j.val \rangle$.
36: **end procedure**

If key k does not exist in the *rblazy-list* of corresponding bucket M_k at Line 13 then it creates a new node n with key k as $\langle$*key=k, lock=false, mark=true, vl=ver, nNext=*$\phi\rangle$ at Line 14 and creates a version (*ver*) for transaction T_0 as $\langle$*ts=0, val=nil, rvl=i, vrt=0, vNext=*$\phi\rangle$ at Line 15. Transaction T_i creates the version of T_0, so, other concurrent conflicting transaction (say T_p) with lower timestamp than T_i, i.e., $\langle p < i \rangle$ can lookup from T_0 version. Thus, T_i save T_p to abort while creating a T_0 version and ensures greater concurrency. After that T_i adds its wts_i in the *rvl* of T_0 and sets the *vrt* 0 as the timestamp of T_0 version. Finally, it inserts the node n into $M_k.rblazy\text{-}list$ such that it is accessible via RL only at Line 16. rv_method() releases the locks and update the $txLog_i$ with key k and value as nil (Line 17). Eventually, it returns the *val* as nil at Line 18.

If key k exists in the $M_k.rblazy\text{-}list$ then it identifies the current version ver_j with ts = j such that j is the *largest timestamp smaller (lts)* than i at Line 20 and there exists no other version with timestamp p by T_p on same key k such that $\langle j < p < i \rangle$. If ver_j is nil at Line 21 then SF-KOSTM returns $abort$ for transaction T_i because it does not found a version to lookup otherwise it identifies the next version with the help of $ver_j.vNext$. If next version ($ver_j.vNext$ as ver_k) exist then T_i maintains the $tutl_i$ with the minimum of $\langle tutl_i \vee ver_k.vrt - 1 \rangle$ at Line 25 and $tltl_i$ with a maximum of $\langle tltl_i \vee ver_j.vrt + 1 \rangle$ at Line 28 to respect the *real-time order* among the transactions. If $tltl_i$ is greater than $tutl_i$ at Line 30 then transaction T_i returns *abort* (fail to maintains real-time order) otherwise it adds the ts of T_i (wts_i) in the rvl of ver_j at Line 32. Finally, it releases the lock and updates the $txLog_i$ with key k and value as the current version value ($ver_j.val$) at Line 33. Eventually, it returns the value as $ver_j.val$ at Line 35.

***STM_insert*() and *STM_delete*() as upd_methods():** Actual effect of *STM_insert*() and *STM_delete*() come after successful *STM_tryC*(). They create the version corresponding to the key in shared memory. We show the high level view of *STM_tryC*() in Algo 2. First, *STM_tryC*() checks the *status* of the transaction T_i at Line 39. If the *status* of T_i is *false* then T_i returns *abort* with similar reasoning explained above in rv_method().

If the *status* is not false then *STM_tryC*() sort the keys (exist in $txLog_i$ of T_i) of upd_-methods() in increasing order. It takes the method (m_{ij}) from $txLog_i$ one by one and identifies the location of the key k in M_k.*rblazy-list* as explained above in rv_method(). After identifying the preds and currs for k it acquire the locks in predefined order to avoid the deadlock at Line 46 and calls *tryC_Validation()* to validate the methods of T_i.

tryC_Validation() identifies whether the methods of invoking transaction T_i are able to insert or delete a version corresponding to the keys while ensuring the progress guarantee as *starvation-freedom* and maintaining the *real-time order* among the transactions. It does four steps for validation. Step 1: First, it does the *rv_Validation()* as explained in rv_method() above. Step 2: If *rv_Validation()* is successful and key k is exist in the $M_k.rblazy\text{-}list$ then it identifies the current version ver_j with ts = j such that j is the *largest timestamp smaller (lts)* than i. If ver_j is *not exist* then SF-KOSTM returns $abort$ for transaction T_i because it does not found the version to replace. Step 3: If ver_j *exist* then T_i compares its_i with its of other $live$ transactions present in $ver_j.rvl$. If its_i of T_i is less than the its of such transactions then T_i sets the $status$ of all those transactions to be $false$, otherwise, T_i returns $abort$. Step 4: To maintain the *real-time order*, T_i update the $tltl_i$ and $tutl_i$ of it with the help of ver_j and its next version ($ver_j.vNext$) respectively (explained in rv_method() above). Please find the detailed descriptions of *tryC_Validation()* in accompanying technical report [24].

If all the steps of the *tryC_Validation()* is successful then the actual effect of the *STM_insert*() and *STM_delete*() will be visible to the shared memory. At Line 53, *STM_-tryC*() checks for *poValidation()*. When two subsequent methods $\langle m_{ij}, m_{ik} \rangle$ of the same transaction T_i identify the overlapping location of preds and currs in *rblazy-list*. Then *poValidation()* updates the current method m_{ik} preds and currs with the help of previous method m_{ij} preds and currs.

If m_{ij} is *STM_insert*() and key k is not exist in the $M_k.rblazy\text{-}list$ then it creates the new node n with key k as $\langle$*key=k, lock=false, mark=false, vl=ver, nNext=*$\phi\rangle$ at Line 55. Later, it creates a version (ver) for transaction T_0 and T_i as $\langle$ *ts=0, val=nil,*

rvl=i, vrt=0, vNext=i $\rangle$ and $\langle$*ts=i, val=v, rvl=*ϕ*, vrt=i, vNext=*$\phi\rangle$ at Line 56. The T_0 version created by transaction T_i to helps other concurrent conflicting transactions (with lower timestamp than T_i) to lookup from T_0 version. Finally, it inserts the node n into $M_k.rblazy\text{-}list$ such that it is accessible via RL as well as BL at Line 57. If m_{ij} is *STM_insert*() and key k exists in the $M_k.rblazy\text{-}list$ then it creates the new version ver_i as $\langle$*ts=i, val=v, rvl=*ϕ*, vrt=i, vNext=*$\phi\rangle$ corresponding to key k. If the limit of the version reaches to K then SF-KOSTM replaces the oldest version with $(K+1)^{th}$ version which is accessible via RL as well as BL at Line 60.

Algorithm 2 STM_tryC *(T_i)*: Validate the upd_methods() of T_i and returns *commit*.

37: **procedure** *STM_tryC*(T_i)
38: /*Atomically check the *status* of its own transaction T_i (or i)*/
39: **if** (*i.status* == *false*) **then** return $\langle abort_i\rangle$.
40: **end if**
41: /*Sort the $keys$ of $txLog_i$ in increasing order.*/
42: /*Method (m) will be either *STM_insert* or *STM_delete**/
43: **for all** ($m_{ij} \in txLog_i$) **do**
44: **if**(m_{ij}==*STM_insert* ||m_{ij}==*STM_delete*)**then**
45: Identify the *preds[]* & *currs[]* for key k in bucket M_k of *rblazy-list* using BL & RL.
46: Acquire the locks on *preds[]* & *currs[]* in increasing order of keys to avoid deadlock.
47: **if** (! $tryC\_Validation()$) **then**
48: return $\langle abort_i\rangle$.
49: **end if**
50: **end if**
51: **end for**
52: **for all** ($m_{ij} \in txLog_i$) **do**
53: *poValidation()* modifies the *preds[]* & *currs[]* of current method which would have been updated by previous method of the same transaction.
54: **if** ((m_{ij}==*STM_insert*)&&(k$\notin M_k$.rblazy-list)) **then**
55: Create new node n with k as: $\langle$*key=k, lock=false, mark= false, vl=ver, nNext=*$\phi\rangle$.
56: Create first version ver for T_0 and next for i: $\langle$*ts=i, val=v, rvl=*ϕ*, vrt=i, vNext=*$\phi\rangle$.
57: Insert node n into $M_k.rblazy\text{-}list$ such that it is accessible via RL as well as BL.
58: /**lock* sets *true**/
59: **else if** (m_{ij} == *STM_insert*) **then**
60: Add ver: $\langle$*ts=i, val=v, rvl=*ϕ*, vrt=i, vNext=*$\phi\rangle$ into $M_k.rblazy\text{-}list$ & accessible via RL, BL. /**mark=false**/
61: **end if**
62: **if** (m_{ij} == *STM_delete*) **then**
63: Add *ver:*$\langle$*ts=i, val=nil, rvl=*ϕ*, vrt=i, vNext=*$\phi\rangle$ into $M_k.rblazy\text{-}list$ & accessible via RL only. /**mark=true**/
64: **end if**
65: Update *preds[]* & *currs[]* of m_{ij} in $txLog_i$.
66: **end for**
67: Release the locks; return $\langle commit_i\rangle$.
68: **end procedure**

If m_{ij} is *STM_delete*() and key k exists in the $M_k.rblazy\text{-}list$ then it creates the new version ver_i as $\langle$*ts=i, val=nil, rvl=*ϕ*, vrt=i, vNext=*$\phi\rangle$ which is accessible via RL only at Line 63. At last it updates the preds and currs of each m_{ij} into its $txLog_i$ to help the upcoming methods of the same transactions in *poValidation()* at Line 65. Finally, it releases the locks on all the keys in a predefined order and returns *commit* at Line 67.

Theorem 1. *Any legal history H generated by SF-SVOSTM satisfies co-opacity.*

Theorem 2. *Any valid history H generated by SF-KOSTM satisfies local-opacity.*

Theorem 3. *SF-SVOSTM and SF-KOSTM ensure starvation-freedom in presence of a fair scheduler that satisfies Assumption 1(bounded-termination) and in the absence of parasitic transactions that satisfies Assumption 2.*

Please find the proof of theorems in accompanying technical report [24].

4 Experimental Evaluation

This section represents the experimental analysis of variants of the proposed Starvation-Free Object-based STMs (SF-SVOSTM, SF-MVOSTM, SF-MVOSTM-GC, and SF-KOSTM)[4] for two data structure *hash table* (HT-SF-SVOSTM, HT-SF-MVOSTM, HT-SF-MVOSTM-GC and HT-SF-KOSTM) and *linked-list* (list-SF-SVOSTM, list-SF-MVOSTM, list-SF-MVOSTM-GC and list-SF-KOSTM) implemented in C++. We

[4]Code is available here: https://github.com/PDCRL/SF-MVOSTM.

analyzed that HT-SF-KOSTM and list-SF-KOSTM perform best among all the proposed algorithms. So, we compared our HT-SF-KOSTM with hash table based state-of-the-art STMs HT-KOSTM [20], HT-SVOSTM [6], ESTM [21], RWSTM [7, Chap. 4], HT-MVTO [16] and our list-SF-KOSTM with list based state-of-the-art STMs list-KOSTM [20], list-SVOSTM [6], Trans-list [22], Boosting-list [4], NOrec-list [23], list-MVTO [16], list-KSFTM [9].

Experimental Setup: The system configuration for experiments is 2 socket Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz with 14 cores per socket and 2 hyper-threads per core, a total of 56 threads. A private 32KB L1 cache and 256 KB L2 cache is with each core. It has 32 GB RAM with Ubuntu 16.04.2 LTS running Operating System. Default scheduling algorithm of Linux with all threads have the same base priority is used in our experiments. This satisfies Assumption 1(bounded-termination) of the scheduler and we ensure the absence of parasitic transactions for our setup to satisfy Assumption 2.

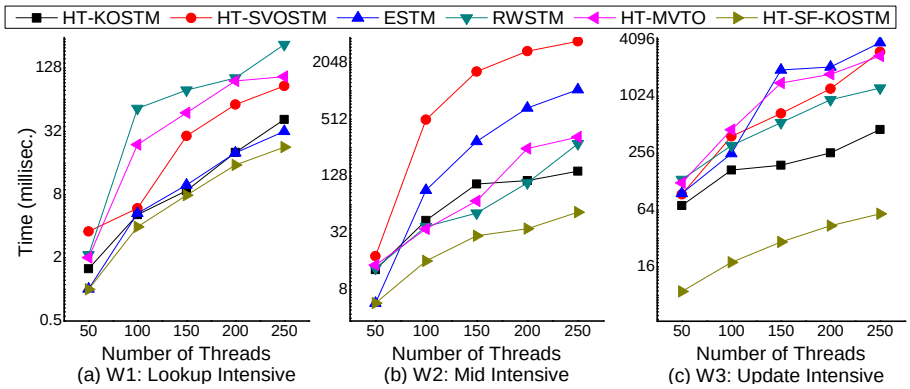


Fig. 6: Performance analysis of SF-KOSTM and State-of-the-art STMs on hash table

Methodology: We have considered three different types of workloads namely, W1 (Lookup Intensive - 5% insert, 5% delete, and 90% lookup), W2 (Mid Intensive - 25% insert, 25% delete, and 50% lookup), and W3 (Update Intensive - 45% insert, 45% delete, and 10% lookup). To analyze the absolute benefit of starvation-freedom, we used a customized application called as the *Counter Application* (refer the pseudo-code in the technical report [24]) which provides us the flexibility to create a high contention environment where the probability of transactions undergoing starvation on an average is very high. Our *high contention* environment includes only 30 shared data-items (or keys), number of threads ranging from 50 to 250, each thread spawns upon a transaction, where each transaction performs 10 operations depending upon the workload chosen. To study starvation-freedom of various algorithms, we have used *max-time* which is the maximum time required by a transaction to finally commit from its first incarnation, which also involves time taken by all its aborted incarnations. We perform each of our experiments 10 times and consider the average of it to avoid the effect of outliers.

Results Analysis: All our results reflect the same ideology as proposed showcasing the benefits of Starvation-Freedom in Multi-Version OSTMs. We started our experiments with *hash table* data structure of bucket size 5 and compared *max-time* for a transaction to commit by proposed HT-SF-KOSTM (best among all the proposed algorithms shown in the technical report [24]) with hash table based state-of-the-art STMs. HT-SF-KOSTM

achieved an average speedup of 3.9x, 32.18x, 22.67x, 10.8x and 17.1x over HT-KOSTM, HT-SVOSTM, ESTM, RWSTM and HT-MVTO respectively as shown in Fig 6.

We further considered another data structure *linked-list* and compared *max-time* for a transaction to commit by proposed list-SF-KOSTM (best among all the proposed algorithms shown in the technical report [24]) with list based state-of-the-arts STMs. list-SF-KOSTM achieved an average speedup of 2.4x, 10.6x, 7.37x, 36.7x, 9.05x, 14.47x, and 1.43x over list-KOSTM, list-SVOSTM, Trans-list, Boosting-list, NOrec-list, list-MVTO, and list-KSFTM respectively as shown in Fig 7. We consider the number of versions in the version list K as 5 and value of C as 0.1.

For additional experiments please refer the technical report [24] which shows the performance of HT-SF-KOSTM and list-SF-KOSTM under *low contention* is slightly lesser than non starvation-free HT-KOSTM and list-KOSTM. It also has plots of abort counts while varying the threads, best value of K and C, $stability$ and *memory consumption*.

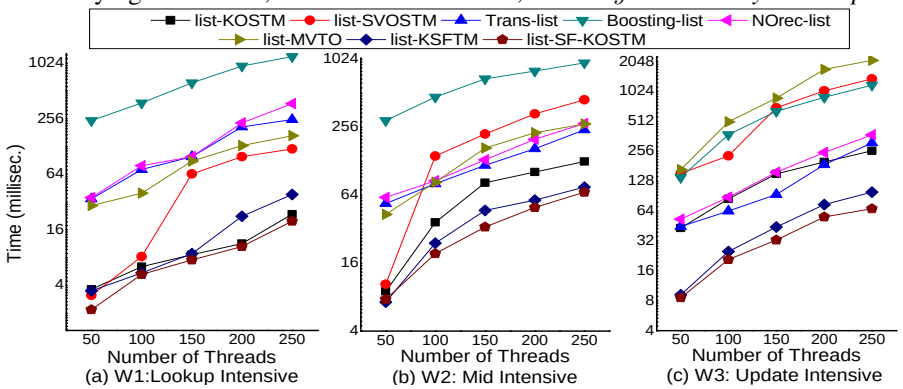


Fig. 7: Performance analysis of SF-KOSTM and State-of-the-art STMs on list

5 Conclusion

We proposed a novel *Starvation-Free K-Version Object-based STM (SF-KOSTM)* which ensure the *starvation-freedom* while maintaining the latest K-versions corresponding to each key and satisfies the correctness criteria as *local-opacity*. The value of K can vary from 1 to ∞. When K is equal to 1 then SF-KOSTM boils down to *Single-Version Starvation-Free OSTM (SF-SVOSTM)*. When K is ∞ then SF-KOSTM algorithm maintains unbounded versions corresponding to each key known as *Multi-Version Starvation-Free OSTM (SF-MVOSTM)*. To delete the unused version from the version list of SF-MVOSTM, we developed a separate Garbage Collection (GC) method and proposed SF-MVOSTM-GC. SF-KOSTM provides greater concurrency and higher throughput using higher-level methods. We implemented all the proposed algorithms for *hash table* and *linked-list* data structure but it is generic for other data structures as well. Results of SF-KOSTM shows significant performance gain over state-of-the-art STMs.

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

References

1. Guerraoui, R., Kapalka, M.: On the Correctness of Transactional Memory. In: PPoPP. (2008)

2. Kuznetsov, P., Peri, S.: Non-interference and local correctness in transactional memory. Theor. Comput. Sci. 2017
3. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4) (1979)
4. Herlihy, M., Koskinen, E.: Transactional boosting: a methodology for highly-concurrent transactional objects. In: PPOPP. (2008)
5. Hassan, A., Palmieri, R., Ravindran, B.: Optimistic transactional boosting. In: PPoPP. (2014)
6. Peri, S., Singh, A., Somani, A.: Efficient means of Achieving Composability using Transactional Memory. NETYS '18 (2018)
7. Weikum, G., Vossen, G.: Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann (2002)
8. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, Revised Reprint. 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2012)
9. Chaudhary, V.P., Juyal, C., Kulkarni, S.S., Kumari, S., Peri, S.: Achieving starvation freedom in multi-version transactional memory systems. NETYS 2019
10. Bushkov, V., Guerraoui, R., Kapalka, M.: On the liveness of transactional memory. In: PODC'12.
11. Herlihy, M., Shavit, N.: On the nature of progress. OPODIS 2011
12. Guerraoui, R., Kapalka, M.: Principles of Transactional Memory, Synthesis Lectures on Distributed Computing Theory. Morgan and Claypool (2010)
13. Gramoli, V., Guerraoui, R., Trigonakis, V.: TM2C: A Software Transactional Memory for Many-cores. EuroSys 2012 (2012)
14. Waliullah, M.M., Stenström, P.: Schemes for Avoiding Starvation in Transactional Memory Systems. Concurrency and Computation: Practice and Experience (2009)
15. Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: A comprehensive strategy for contention management in software transactional memory (2009)
16. Kumar, P., Peri, S., Vidyasankar, K.: A TimeStamp Based Multi-version STM Algorithm. In: ICDCN. (2014)
17. Lu, L., Scott, M.L.: Generic Multiversion STM. In: DISC. (2013) 134–148
18. Fernandes, S.M., Cachopo, J.: Lock-free and Scalable Multi-version Software Transactional Memory. PPoPP '11, New York, NY, USA (2011)
19. Perelman, D., Byshevsky, A., Litmanovich, O., Keidar, I.: SMV: Selective Multi-Versioning STM. In: DISC. (2011) 125–140
20. Juyal, C., Kulkarni, S.S., Kumari, S., Peri, S., Somani, A.: An innovative approach to achieve compositionality efficiently using multi-version object based transactional systems. In: SSS'18.
21. Felber, P., Gramoli, V., Guerraoui, R.: Elastic Transactions. J. Parallel Distrib. Comput. **100**(C) (February 2017)
22. Zhang, D., Dechev, D.: Lock-free Transactions Without Rollbacks for Linked Data Structures. SPAA '16, New York, NY, USA (2016)
23. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: Streamlining STM by Abolishing Ownership Records. In Govindarajan, R., Padua, D.A., Hall, M.W., eds.: PPOPP, ACM (2010)
24. Juyal, C., Kulkarni, S.S., Kumari, S., Peri, S., Somani, A.: Obtaining progress guarantee and greaterconcurrency in multi-version object semantics. CoRR **abs/1904.03700** (2019)
25. Heller, S., Herlihy, M., Luchangco, V., Moir, M., III, W.N.S., Shavit, N.: A Lazy Concurrent List-Based Set Algorithm. Parallel Processing Letters **17**(4) (2007) 411–424
26. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: DISC. (2001)
27. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: DISC 2006.
28. Guerraoui, R., Henzinger, T.A., Singh, V.: Permissiveness in transactional memories. In: DISC. (2008) 305–319
29. Crain, T., Imbs, D., Raynal, M.: Read invisibility, virtual world consistency and probabilistic permissiveness are compatible. In: ICA3PP. (2011) 244–257