Obtaining Progress Guarantee and Greater Concurrency in Multi-Version Object Semantics

Chirag Juyal¹, Sandeep Kulkarni², Sweta Kumari¹, Sathya Peri¹ and Archit Somani¹†
Department of Computer Science and Engineering, IIT Hyderabad, India¹
Department of Computer Science, Michigan State University, MI, USA²
{cs17mtech11014, cs15resch01004, sathya_p, cs15resch01001}@iith.ac.in¹
sandeep@cse.msu.edu²

Software Transactional Memory Systems (STMs) provides ease of multithreading to the programmer without worrying about concurrency issues such as deadlock, livelock, priority inversion, etc. Most of the STMs work on read-write operations known as RWSTMs. Some STMs work at high-level operations and ensure greater concurrency than RWSTMs. Such STMs are known as Object-Based STMs (OSTMs). The transactions of OSTMs can return commit or abort. Aborted OSTMs transactions retry. But in the current setting of OSTMs, transactions may starve. So, we proposed a Starvation-Free OSTM (SF-OSTM) which ensures starvation-freedom while satisfying the correctness criteria as opacity.

Databases, RWSTMs and OSTMs say that maintaining multiple versions corresponding to each key reduces the number of aborts and improves the throughput. So, to achieve the greater concurrency, we proposed Starvation-Free Multi-Version OSTM (SF-MVOSTM) which ensures starvation-freedom while storing multiple version corresponding to each key and satisfies the correctness criteria as local opacity. To show the performance benefits, We implemented three variants of SF-MVOSTM and compare its performance with state-of-the-art STMs.

Additional Key Words and Phrases: Starvation-Freedom, Concurrency, Multi-Version, Software Transactional Memory System, Co-Opacity

1 INTRODUCTION

Concurrency control using locks has various issues such as composability, difficult to reproduce and debug. So, an alternative to locks is Software Transactional Memory Systems (STMs). It access the shared memory while removing the concurrency responsibilities from the programmer. STMs internally use locks carefully and ensure that consistency issues such as deadlock, livelock, priority inversion etc will not occur. It provides high level abstraction to the programmer for concurrent section and ensures the consistency.

There are two types of STMs available in literature. (1) Pessimistic STMs which shows the effect of the operation of a transaction immediately and on inconsistency it rollback and transaction returns abort. (2) Optimistic STMs in which transactions are writing into its local log until the successful validation so, rollback is not required. A traditional optimistic STM system invokes following methods:(1) \textit{stm\_begin}(): It begins a transaction \( T_i \) with unique timestamp \( i \). (2) \textit{stm\_read\_i}(k): \( T_i \) reads the value of \( k \) from shared memory. (3) \textit{stm\_write\_i}(k,v): \( T_i \) writes the value of \( k \) as \( v \) locally. (4) \textit{stm\_tryC\_i}(): On successful validation the effect of transaction will be visible to the shared memory and transaction returns commit otherwise returns abort using (5) \textit{stm\_tryA\_i}(). These STMs are known as read-write STMs (RWSTMs) because its working at low-level operations such as read and write.

*This paper is eligible for Best Student Paper award as Chirag, Sweta & Archit are full-time Ph.D. student.
†Author sequence follows lexical order of last names
Herlihy et al. [8], Hassan et al. [7], and Peri et al. [17] have shown that working at high-level operations such as insert, delete and lookup on hash table gives better concurrency than RWSTMs. STMs which works on high-level operations are known as object-based STMs (OSTMs) [17]. It exports following methods: (1) \(\text{stm\_begin}(i)\): It begins a transaction \(T_i\) with unique timestamp \(i\). (2) \(\text{stm\_lookup}(k)\) (or \(i(k)\)): \(T_i\) lookups \(k\) from shared memory and return its value. (3) \(\text{stm\_insert}(k,v)\) (or \(i(k,v)\)): \(T_i\) inserts a key \(k\) with value \(v\) into its local memory. (4) \(\text{stm\_delete}(k)\) (or \(d(k)\)): \(T_i\) deletes key \(k\). (5) \(\text{stm\_tryC}(i)\): The actual effect of \(\text{stm\_insert}\)() and \(\text{stm\_delete}\)() will come into shared memory after successful validation and transaction returns commit otherwise returns abort using (6) \(\text{stm\_tryA}(i)\).

Figure 1 represents the advantage of OSTMs over RWSTMs while achieving greater concurrency and reducing the number of aborts. Figure 1.(a) depicts the underlying data structure as hash table with \(B\) buckets and bucket 1 stores three keys \(k_1, k_4\) and \(k_5\) in the form of list. Figure 1.(b) shows the tree structure of concurrent execution of two transactions \(T_1\) and \(T_2\) with RWSTMs at layer-0 and OSTMs 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 operation of \(T_1\) on \(k_1\) as \(r_1(k_1)\). Two transactions are said to be in conflict, if both are accessing the same key \(k\) and at least one transaction performs write operation on \(k\). So, this concurrent execution can’t be atomic as shown Figure 1.(c). To make it atomic either \(T_1\) or \(T_2\) needs to return abort. Whereas execution at layer-1 shows the high-level operations \(l_1(k_1), d_2(k_4)\) and \(l_1(k_5)\) on different keys \(k_1, k_4\) and \(k_5\) respectively. All the high-level operations are isolated to each other so tree can be pruned from layer-0 to layer-1 with equivalent serial schedule \(T_1 T_2\) or \(T_2 T_1\) as shown in Figure 1.(d). Hence, some conflicts of RWSTMs does not matter at OSTMs which leads to reduce the number of aborts and improve the concurrency using OSTMs.

When transactions are short with less conflicts then optimistic OSTMs is better than pessimistic OSTMs [1]. But for long running transactions along with high conflicts, starvation can occur in optimistic OSTMs. So, optimistic OSTMs should ensures the progress guarantee as starvation-freedom [10, chap 2]. An OSTMs 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., If scheduler will give a fair chance to \(T_i\) to commit then \(T_i\) will eventually returns commit. If a transaction gets a chance to commit, still its not committing because of infinite loop or some other error such transactions are known as Parasitic transactions [1].

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 [9]. However, Guerraoui and Kapalka [2, 6] 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 OSTMs while assuming that the keys of the transactions are not known in advance.

**Existing Starvation-free STMs:** There are few researchers Gramoli et al. [4], Waliullah and Stenstrom [19], Spear et al. [18] who explored starvation-freedom in RWSTMs. They are giving the priority to the transaction on conflict. We also inspired with them and proposed *Starvation-Free OSTM* (or SF-OSTM). This is the first paper which explores starvation-freedom in OSTMs. In SF-OSTM whenever a conflicting transaction \(T_i\) aborts, it retries with \(T_j\) which has higher priority than \(T_i\). This procedure will repeat until \(T_i\) gets highest priority and eventually returns commit. Figure 2 represents the starvation in OSTM whereas SF-OSTM ensures starvation-freedom. Figure 2.(a) shows the execution under OSTMs on hash table \(ht\) in which higher timestamp transaction \(T_2\) has already been committed so lower timestamp transaction \(T_1\)
returns abort [17]. \( T_1 \) retries with \( T_3 \) but again higher timestamp transaction \( T_4 \) has been committed which causes \( T_3 \) to abort again. This situation can occur again and again and leads to stave the transaction \( T_1 \). Albeit, SF-OSTMs ensures starvation-freedom while giving the priority to lowest timestamp. Here, each transaction maintains two timestamps, Initial Timestamp (ITS) and Current Timestamp (CTS). Whenever a transaction \( T_i \) starts for the first time, it gets a unique timestamp \( i \) using \( \text{stm}_\text{begin}() \) as ITS which is equal to CTS as well. On abort \( T_i \) gets new timestamp as CTS but it will retains same ITS. Consider the Figure 2.(b) in which \( T_{1,1} \) represents the first incarnation of \( T_1 \) so, CTS equals to ITS as 1. \( T_{1,1} \) conflicts with \( T_{2,2} \) and \( T_{3,3} \). As \( T_{1,1} \) have the lowest ITS so \( T_1 \) gets the priority to execute whereas \( T_{2,2} \) and \( T_{3,3} \) returns abort. On abort, \( T_{2,2} \) and \( T_{3,3} \) retries with new CTS 4 and 5 but with same ITS 2 and 3 respectively. So, due to lowest ITS \( T_{4,2} \) returns commit but \( T_{5,3} \) returns abort and so on. Hence, none of the transaction starves. So, when conflicts occur assigning priority to the lowest ITS transaction ensures the starvation-freedom in OSTMs.

If the highest priority transaction becomes slow then it may cause several other transactions to abort as shown in Figure 3.(a). Here, transaction \( T_{1,1} \) became slow so, it is forcing the conflicting transactions \( T_{2,2} \) and \( T_{3,3} \) to abort again and again. Database and several STMs at read-write level [3, 12, 15, 16] and object-based level [11] say that maintaining multiple versions corresponding to each key reduces the number of aborts and improves the throughput. OSTMs maintains single version corresponding to each key whereas Multi-Version OSTM (or MV-OSTM) \(^1\) maintains multiple versions corresponding to each key which improves the concurrency further. So, in this paper we propose the first starvation-free OSTM using multiple versions as Starvation-Free Multi-Version OSTMs (SF-MVOSTMs). Figure 3.(b), shows the benefits of execution using SF-MVOSTMs in which \( T_{1,1} \) lookups from the older version created by transaction \( T_{0,0} \) (assuming as initial transaction) for key \( k_1 \) and \( k_4 \). Concurrently, \( T_{2,2} \) and \( T_{3,3} \) create the new versions for key \( k_4 \). So, all the three transactions can commit with equivalent serial schedule as \( T_1 T_2 T_3 \) and ensure the starvation-freedom.

Contributions of the paper are as follows:

- We propose a SV-OSTM which ensures starvation-freedom and correctness criteria as opacity [5].
- To achieve the greater concurrency, we propose SF-MVOSTM which ensures starvation-freedom while storing multiple version corresponding to each key and satisfies the correctness criteria as local opacity.
- We implement three variants of SF-MVOSTM and compare its performance. Result shows that SF-NOSTM performs better among all.

2 SYSTEM MODEL AND PRELIMINARIES

This section follows the notion and definition described in [6, 14], we assume a system of \( n \) processes/threads, \( p_1, \ldots, p_n \) that access a collection of keys (or transaction-objects) via atomic transactions. Each transaction has been identified by a unique identifier. The transaction of the system at read-write

\(^1\)It receives Best Student Paper Award in SSS-2018.
level (or lower level) invokes \textit{stm}\_begin(), \textit{stm}\_read\_{i}(k), \textit{stm}\_write\_{i}(k,v), \textit{stm}\_try\_{C}\_{i}() and \textit{stm}\_try\_{A}\_{i}() as defined in Section 1. In this paper, transactions works on object level (or higher level). Transaction of the system at object level invokes \textit{stm}\_begin(), \textit{stm}\_lookup\_{i}(k) (or \textit{l}(k)), \textit{stm}\_insert\_{i}(k,v) (or \textit{i}(k,v)), \textit{stm}\_delete\_{i}(k) (or \textit{d}(k)), \textit{stm}\_try\_{C}\_{i}(), and \textit{stm}\_try\_{A}\_{i}() as defined in Section 1. For the sake of presentation simplicity, we assume that the values taken as arguments by \textit{t}\_write operations are unique. A transaction \(T_i\) begins with unique timestamp \(i\) using \textit{stm}\_begin() and completes with any of its operation which returns either commit as \(C\) or abort as \(A\). Transaction cannot invoke any more operation after returning \(C\) or \(A\). Any operation that returns \(C\) or \(A\) are known as terminal operations.

3 PROPOSED MECHANISM

3.1 Description of Starvation-Freedom

This subsection describes the definition of starvation-freedom followed by our assumption about the scheduler that helps us to achieve starvation-freedom in OSTMs and MV-OSTMs.

\textit{Definition 3.1. 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 [9] defined the fair scheduler which ensures that none of the thread will crashed or delayed forever. Hence, any thread \(Th_i\) acquires the lock on the data-items while executing transaction \(T_i\) will eventually release the locks. So, a thread will never block another threads to progress. In order to satisfy the starvation-freedom for OSTMs and MV-OSTMs, we assumed bounded termination for fair scheduler.

\textbf{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 (either commits or aborts) in bounded time.

In the proposed algorithms, we have considered \(Max\) as the maximum time bound of a transaction \(T_i\) within this either \(T_i\) will return commit or abort due to the absence of deadlock. Approach for achieving the deadlock-freedom is motivated from the literature in which threads executing transaction acquire the locks in increasing order of the keys and releases the locks in bounded time either by committing or aborting the transaction.

3.2 Data Structure and Design of SF-OSTM and SF-MVOSTM
3.3 Working of SF-OSTM and SF-MVOSTM

Algorithm 1 \(rv\_method()\): Could be either \(STM\_delete\)(\(ht, k, val\)) or \(STM\_lookup\)(\(ht, k, val\)) on key \(k\).

\[
\text{1: procedure } rv\_method(ht, k, val) \text{ 16: Release the locks; update the txLog with } k. \\
\text{2: if } (k \in txLog) \text{ then Update the local log and return } val. \text{ 17: return } (val). /*val as null*/ \\
\text{3: else } \text{ 18: end if} \\
\text{4: \quad /*Atomically check the status of its own transaction */} \text{ 19: Identify the version } very \text{ with ts } = j \text{ such that } j \text{ is the} \\
\text{5: if (i.status == false) then return (abort i).} \text{ 20: largest timestamp smaller (ts) than } i. \\
\text{6: end if} \text{ 21: if (ver, vNext != null) then} \\
\text{7: Search in rblazy-list to identify the preds[] and currs[]} \text{ 22: Calculate } \text{tltl}_i = \text{min(} \text{tltl}_i, \text{ver, vrt + 1}. \\
\text{for } k \text{ using BL and RL in bucket } B_k. \text{ 23: end if} \\
\text{8: Acquire locks on preds[] and currs[] in increasing order.} \text{ 24: /*If limit has crossed each other then abort } T_i*/ \\
\text{9: if (try\_Validation(preds[], currs[])) then} \text{ 25: end if} \\
\text{10: Release the locks and goto Line 7.} \text{ 26: end if} \\
\text{11: end if} \text{ 27: if (tltl}_i > \text{tltl}_j) \text{ then return (abort } j). \\
\text{12: if (k \notin B_k.rblazy-list) then} \text{ 28: end if} \\
\text{13: Create a new node with key } k \text{ as: } (\text{key} = k, \text{lock} = \text{false, true, vl} = v, \text{nNext} = \phi). \text{ 29: Add } i \text{ into the } \text{rvl of } ver_j. \\
\text{14: Create version } \text{ver as: } (\text{ts}_i, \text{val} = \text{null, rvl} = i, \text{vrt} = 0, \text{vNext} = \phi). \text{ 30: Release the locks; update the txLog with } k \text{ and value.} \\
\text{15: Insert } n \text{ into } B_k.rblazy-list \text{ such that it is} \text{ 31: end if} \\
\text{16: return (ver}_j, val). \text{ 32: accessible only via RLs.} \\
\text{17: end procedure} \text{ 33: end procedure}
\]

Algorithm 2 \(tryC(T_i)\): Validate the upd\_methods of the transaction and return commit.

\[
\text{34: procedure } tryC(T_i) \text{ 50: if (op}_i\text{== }\text{STM\_insert} \text{ and } (k \notin B_k.rblazy-list) \text{ then} \\
\text{35: \quad /*Atomically check the status of its own transaction */} \text{ 51: Create new node with key } k \text{ as: } (\text{key} = k, \text{lock} = \text{false, mark} = \text{false}}, \text{vl} = v, \text{vlNext} = \phi). \\
\text{36: if (i.status == false) then return (abort i).} \text{ 52: Create first version } \text{ver for } T_i \text{ and next for } i: (ts = i, \text{val = v, rvl = } \phi, \text{vrt} = i, \text{vNext} = \phi). \\
\text{37: end if} \text{ 53: Insert node } n \text{ into } B_k.rblazy-list \text{ such that it is} \text{ 54: end if} \\
\text{38: \quad /*Sort the keys of txLog with in increasing order.*/} \text{ 55: accessible only via RL as well as BL.*/lock sets true*/} \\
\text{39: \quad /*Operation (op) will be either STM\_insert or STM\_delete */} \text{ 56: else if (op}_i\text{== }\text{STM\_insert} \text{ then} \\
\text{40: for all (op}_i\text{ in txLog) do} \text{ 57: Add } \text{ver: (ts}_i, \text{val = null, rvl = } \phi, \text{vrt} = i, \text{vNext = } \phi) \text{ into} \\
\text{41: \quad if (op}_i\text{ == }\text{STM\_insert} \text{ and (op}_i\text{ == }\text{STM\_delete}) \text{ then} \text{B}_k.rblazy-list \text{ and accessible via RL, BL.*/mark=false*/} \\
\text{42: Search in rblazy-list to identify the preds[] and} \text{ 58: end if} \\
\text{43: Acquire lock on preds[] and currs[] in increasing order.} \text{ 59: if (op}_i\text{ == }\text{STM\_delete} \text{ then} \\
\text{44: if (tryC\_Validation()) then return (abort i).} \text{ 60: Update the preds[] and currs[] of } \text{op}_i \text{ in txLog.} \\
\text{45: end if} \text{ 61: end for} \\
\text{46: end if} \text{ 62: Release the locks; return (commit}_i). \\
\text{47: end for} \text{ 63: end procedure} \\
\text{48: for all (op}_i\text{ in txLog) do} \text{ 64: algorithm 3 \(rv\_Validation()\) } \\
\text{49: \quad polValidation() modifies the preds[] and currs[] of} \text{ 65: if (\text{preds}[0].mark}||\text{currs}[1].mark)||\text{currs[0].BL} \text{ 66: return (true).} \\
\text{50: \text{current operation which would have been updated by the} \text{ 67: end if} \\
\text{51: previous operation of the same transaction.} \text{ 68: end procedure} \\
\text{52: \quad \text{return (false).} \text{ 69: end for}}
\]

Algorithm 3 \(rv\_Validation()\)
Algorithm 4: `tryC_Validation()`

69: procedure `tryC_Validation()`
70: if `true_Validation()` then Release the locks and retry.
71: end if
72: if `(k ∈ B_k) & (rlazy-list)` then
73: Identify the version `ver_j` with `ts = j` such that `j` is the largest timestamp smaller (lts) than `i`.
74: Maintain the list of `ver_j, ver_j.vNext, ver_j.rvl, (ver_j.rvl > i),` and `(ver_j.rvl < i)` as prevVL, nextVL, allRVL, largeRVL, smallRVL respectively for all `k` of `T_i`.
75: if `(k ∈ allRVL)` then /*Includes `i` in allRVL as well*/
76: Lock the status of each `k` in pre-defined order.
77: end if
78: if `(i.status == false)` then return `(false)`, /*abort `i` itself*/
79: end if
80: for all `(k ∈ largeRVL)` do
81: if `(tts_k < tts_j)` & & `(k.status == live)` then
82: Maintain abort list as abortRVL & includes `k` in it.
83: else return `(false)`, /*abort `i` itself*/
84: end if
85: end for
86: for all `(ver ∈ prevVL)` do
87: Calculate `tutl_i = min(tutl_i, ver.vort + 1)`.
88: end for
89: for all `(ver ∈ nextVL)` do
90: Calculate `tutl_i = min(tutl_i, ver.vNext.vrt - 1)`.
91: end for
92: if `(tutl_i > tutl_j)` then return `(false)`, /*abort `i` itself*/
93: end if
94: if `(k ∈ smallRVL)` then
95: if `(tts_k < tts_j)` & & `(k.status == live)` then
96: Includes `k` in abortRVL list.
97: end if
98: else return `(false)`, /*abort `i` itself*/
99: end if
100: end if
101: end if
102: `tutl_i = tutl_i`, /*After this point `i` can’t abort*/
103: for all `(k ∈ smallRVL)` do /*Only for live transactions*/
104: Calculate the `tutl_k = min(tutl_k, tltl_i - 1)`.
105: end for
106: for all `(k ∈ abortRVL)` do
107: Set the status of `k` to be `false`.
108: end for
109: end if
110: return `(true)`.
111: end procedure

REFERENCES

[1] BUSHIKOV, V., AND GUERRAOUI, R. Liveness in transactional memory. 32–49.
[2] BUSHIKOV, V., GUERRAOUI, R., AND KAPALKA, M. On the liveness of transactional memory. In *ACM Symposium on Principles of Distributed Computing, PODC ’12, Funchal, Madeira, Portugal, July 16-18, 2012* (2012), pp. 9–18.
[3] FERNANDES, S. M., AND CACHOPO, J. Lock-free and Scalable Multi-version Software Transactional Memory. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming* (New York, NY, USA, 2011), PPoPP ’11, ACM, pp. 179–188.
[4] GRAMOLI, V., GUERRAOUI, R., AND TRIGONAKIS, V. TM2C: A Software Transactional Memory for Many-cores. EuroSys 2012.
[5] GUERRAOUI, R., AND KAPALKA, M. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2008), PPoPP ’08, ACM, pp. 175–184.
[6] GUERRAOUI, R., AND KAPALKA, M. Principles of Transactional Memory, *Synthesis Lectures on Distributed Computing Theory*. Morgan and Claypool, 2010.
[7] HASSAN, A., PALMIERI, R., AND RAVINDRAN, B. Optimistic transactional boosting. In *Ppopp* (2014), pp. 387–388.
[8] HERLIHY, M., AND KOSSIKEN, E. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2008, Salt Lake City, UT, USA, February 20-23, 2008* (2008), pp. 207–216.
[9] HERLIHY, M., AND SHAVIT, N. On the nature of progress. *OPODIS* 2011.
[10] HERLIHY, M., AND SHAVIT, N. *The Art of Multiprocessor Programming, Revised Reprint*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012.
[11] JUYAL, C., KULKARNI, S. S., KUMAR, S., PERI, S., AND SOMANI, A. An innovative approach to achieve compositionality efficiently using multi-version object based transactional systems. In *Stabilization, Safety, and Security of Distributed Systems - 2014 International Symposium, SSS 2014, Tokyo, Japan, November 4-7, 2014, Proceedings* (2014), pp. 284–300.
[12] KUMAR, P., PERI, S., AND VIDYASANKAR, K. A TimeStamp Based Multi-version STM Algorithm. In *ICDCN* (2014), pp. 212–226.
[13] Kuznetsov, P., and Peri, S. Non-interference and Local Correctness in Transactional Memory. In ICDCN (2014), pp. 197–211.

[14] Kuznetsov, P., and Peri, S. Non-interference and local correctness in transactional memory. Theor. Comput. Sci. 688 (2017), 103–116.

[15] Lu, L., and Scott, M. L. Generic Multiversion STM. In Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings (2013), pp. 134–148.

[16] Perelman, D., Byshevsky, A., Litmanovich, O., and Keidar, I. SMV: Selective Multi-Versioning STM. In DISC (2011), pp. 125–140.

[17] Peri, S., Singh, A., and Somani, A. Efficient means of Achieving Composability using Transactional Memory. NETYS '18.

[18] Spear, M. F., Dalessandro, L., Marathe, V. J., and Scott, M. L. A comprehensive strategy for contention management in software transactional memory, 2009.

[19] Waliullah, M. M., and Stenström, P. Schemes for Avoiding Starvation in Transactional Memory Systems. Concurrency and Computation: Practice and Experience (2009).
Fig. 1. Advantage of OSTMs over RWSTMs

(a). T₁ is starving in OSTMs
(b). Tree Structure of Concurrent Transactions

Layer-0: Reads & Writes
T₁, T₂

Layer-1: Lookups & Deletes
T₁, T₂

Fig. 2. Advantage of SF-OSTM over OSTMs

(a). Single version SF-OSTMs
(b). Starvation Free OSTMs

Fig. 3. Benefits of MVOSTM over Single version SF-OSTM

4 DESIGNATED FIGURES
Fig. 4. Data Structure and Design of SF-MVOSTM

Fig. 5. Searching $k_9$ over lazy-list

Fig. 6. Searching $k_9$ over rblazy-list