

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

Chirag Juyal¹, Sandeep Kulkarni², Sweta Kumari¹, Sathya Peri¹ and Archit Somani^{1†}

Department of Computer Science and Engineering, IIT Hyderabad, India¹

Department of Computer Science, Michigan State University, MI, USA²

{cs17mtech11014, cs15resch01004, sathya_p, cs15resch01001}@iiith.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 works 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) *stm_begin()*: It begins a transaction T_i with unique timestamp i . (2) *stm_read_i(k)*: T_i reads the value of k from shared memory. (3) *stm_write_i(k,v)*: T_i writes the value of k as v locally. (4) *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) *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) *stm_begin()*: It begins a transaction T_i with unique timestamp i . (2) *stm_lookup_i(k)* (or $l(k)$): T_i lookups k from shared memory and return its value. (3) *stm_insert_i(k,v)* (or $i(k,v)$): T_i inserts a key k with value v into its local memory. (4) *stm_delete_i(k)*(or $d(k)$): T_i deletes key k . (5) *stm_tryC_i()*: The actual effect of *stm_insert()* and *stm_delete()* will come into shared memory after successful validation and transaction returns commit otherwise returns abort using (6) *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_9 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 $l_1(k_9)$ on different keys k_1, k_4 and k_9 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_1T_2 or T_2T_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 starve 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 *stm_begin()* as ITS which is equal to CTS as well. On abort T_i gets new timestamp as CTS but it will retain 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 $T_1T_2T_3$ and ensure the starvation-freedom. We implemented three variants of SF-MVOSTM and compare its performance: (1) SF-MVOSTM without *garbage collection* (or gc) (2) SF-MVOSTM with gc: It deletes the unwanted versions from version list of keys. (3) Finite version of SF-OSTM as SF-NOSTM which stores finite say N number of versions corresponding to each key. After creation of N version, $N + 1$ version replaces the oldest version. Proposed SF-MVOSTM, SF-MVOSTM with gc, SF-NOSTM ensure the progress guarantee as starvation-freedom and correctness criteria as local opacity [13, 14].

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, \dots, 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

¹It receives Best Student Paper Award in SSS-2018.

level (or lower 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. In this paper, transactions work on object level (or higher level). Transaction of the system at object level invokes $stm_begin()$, $stm_lookup_i(k)$ (or $l(k)$), $stm_insert_i(k,v)$ (or $i(k,v)$), $stm_delete_i(k)$ (or $d(k)$), $stm_tryC_i()$, and $stm_tryA_i()$ as defined in Section 1. For the sake of presentation simplicity, we assume that the values taken as arguments by t_write operations are unique. A transaction T_i begins with unique timestamp i using $stm_begin()$ and completes with any of its operation which returns either commit as C or abort as \mathcal{A} . Transaction cannot invoke any more operation after returning C or \mathcal{A} . Any operation that returns C or \mathcal{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.

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.

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_i(ht, k, val)* or *STM_lookup_i(ht, k, val)* on key *k*.

```

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

```

Algorithm 2 *tryC(T_i)*: Validate the upd_methods of the transaction and return *commit*.

```

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

```

Algorithm 3 *rv_Validation()*

```

64: procedure rv_Validation()
65:   if ( $(preds[0].mark) \vee (currs[1].mark) \vee (preds[0].BL \neq$ 
66:    $currs[1]) \vee (preds[1].RL \neq currs[0])$ ) then return  $\langle false \rangle$ .
67:   end if
68: end procedure

```

Algorithm 4 *tryC_Validation()*

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

REFERENCES

- [1] BUSHKOV, V., AND GUERRAOUI, R. Liveness in transactional memory. 32–49.
- [2] BUSHKOV, 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 KOSKINEN, 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., KUMARI, 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 - 20th International Symposium, SSS 2018, Tokyo, Japan, November 4-7, 2018, Proceedings* (2018), 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).

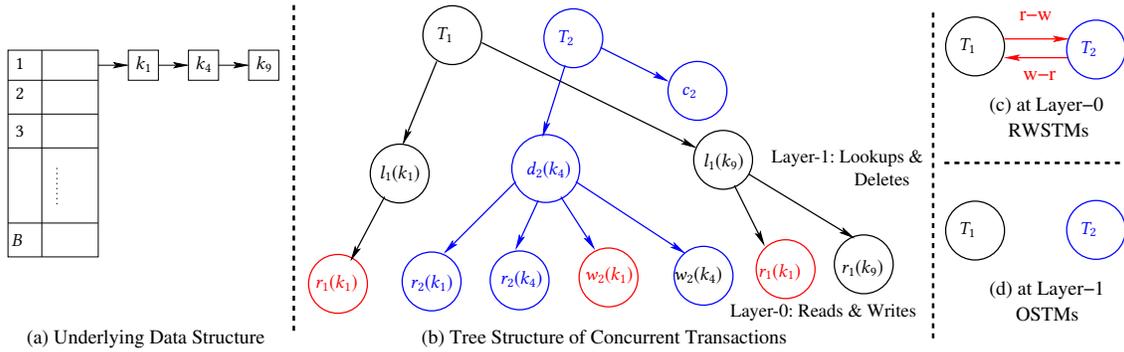


Fig. 1. Advantage of OSTMs over RWTSMs

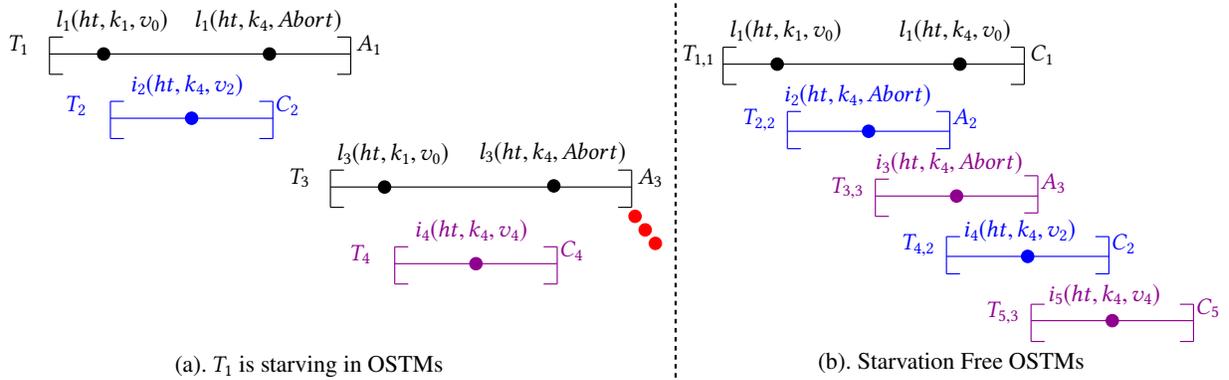


Fig. 2. Advantage of SF-OSTM over OSTMs

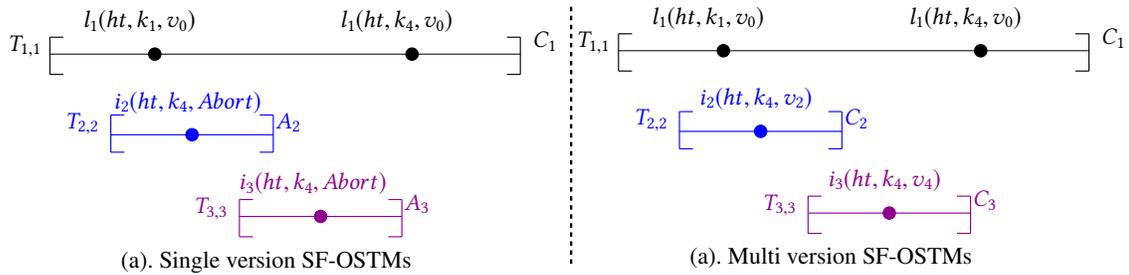


Fig. 3. Benefits of MVOSTM over Single version SF-OSTM

4 DESIGNATED FIGURES

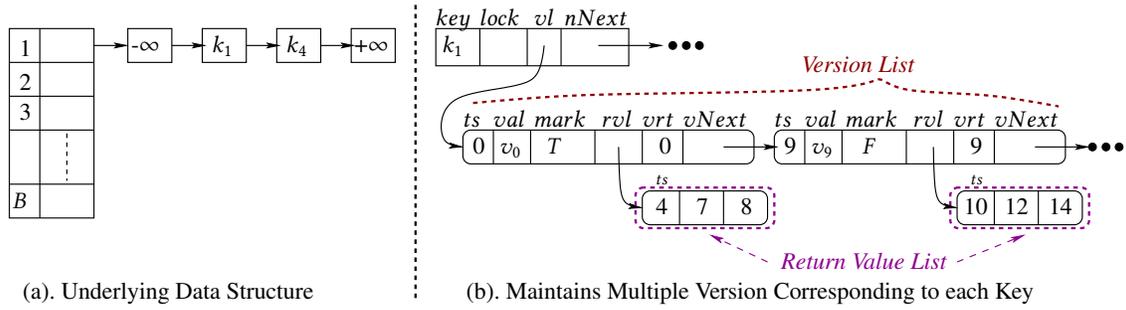


Fig. 4. Data Structure and Design of SF-MVOSTM

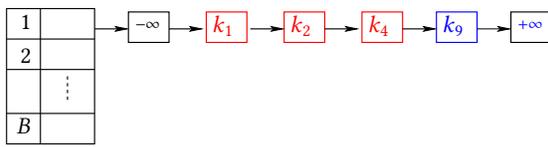


Fig. 5. Searching k_9 over *lazy-list*

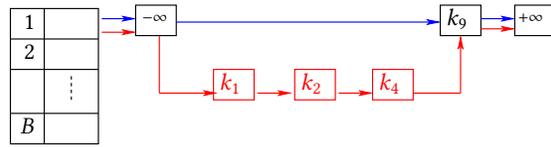


Fig. 6. Searching k_9 over *rblazy-list*