An Innovative Approach to Achieve Compositionality Efficiently using Multi-Version Object Based Transactional 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. The rise of multi-core systems has necessitated the need for concurrent programming. However, developing correct, efficient concurrent programs is notoriously difficult. Software Transactional Memory Systems (STMs) are a convenient programming interface for a programmer to access shared memory without worrying about concurrency issues. Another advantage of STMs is that they facilitate compositionality of concurrent programs with great ease. Different concurrent operations that need to be composed to form a single atomic unit is achieved by encapsulating them in a single transaction.

Most of the STMs proposed in the literature are based on read/write primitive operations on memory buffers. We denote them as *Read-Write STMs* or *RWSTMs*. On the other hand, there have been some STMs that have been proposed (transactional boosting and its variants) that work on higher level operations such as hash-table insert, delete, lookup, etc. We call them Object STMs or OSTMs.

It was observed in databases that storing multiple versions in RWSTMs provides greater concurrency. In this paper, we combine both these ideas for harnessing greater concurrency in STMs - multiple versions with objects semantics. We propose the notion of *Multi-version Object STMs* or *MVOSTMs*. Specifically, we introduce and implement *MVOSTM* for the hash-table object, denoted as *HT-MVOSTM* and list object, *list-MVOSTM*. These objects export insert, delete and lookup methods within the transactional framework. We also show that both these *MVOSTM*s satisfy opacity and ensure that transaction with lookup only methods do not abort if unbounded versions are used.

Experimental results show that *list-MVOSTM* outperform almost two to twenty fold speedup than existing state-of-the-art list based STMs (Trans-list, Boosting-list, NOrec-list, list-MVTO, and list-OSTM). Similarly, *HT-MVOSTM* shows a significant performance gain of almost two to nineteen times over the existing state-of-the-art hash-table based STMs (ESTM, RWSTMs, HT-MVTO, and HT-OSTM).

[§]Author sequence follows the lexical order of last names. All the authors can be contacted at the addresses given above. Archit Somani's phone number: +91 - 7095044601.

[*]A poster version of this work received **best poster award** in NETYS-2018. An initial version of this work was accepted as **work in progress** in AADDA workshop, $ICDCN - 2018$.

[†]This paper is a regular submission: (1) Eligible for the best student paper award. Chirag Juyal, Sweta Kumari, and Archit Somani are full-time students.

1 Introduction

The rise of multi-core systems has necessitated the need for concurrent programming. However, developing correct concurrent programs without compromising on efficiency is a big challenge. Software Transactional Memory Systems (STMs) are a convenient programming interface for a programmer to access shared memory without worrying about concurrency issues. Another advantage of STMs is that they facilitate compositionality of concurrent programs with great ease. Different concurrent operations that need to be composed to form a single atomic unit is achieved by encapsulating them in a single transaction. Next, we discuss different types of STMs considered in the literature and identify the need to develop multi-version object STMs proposed in this paper.

Read-Write STMs: Most of the STMs proposed in the literature (such as NOrec [1], ESTM [2]) are based on read/write operations on *transaction objects* or *t-objects*. We denote them as *Read Write STMs* or *RWSTMs*. These STMs typically export following methods: (1) *t_begin*: begins a transaction, (2) *t_read* (or r): reads from a t-object, (3) *t_write* (or w): writes to a t-object, (4) *tryC*: validates and tries to commit the transaction by writing values to the shared memory. If validation is successful, then it returns commit. Otherwise, it returns abort.

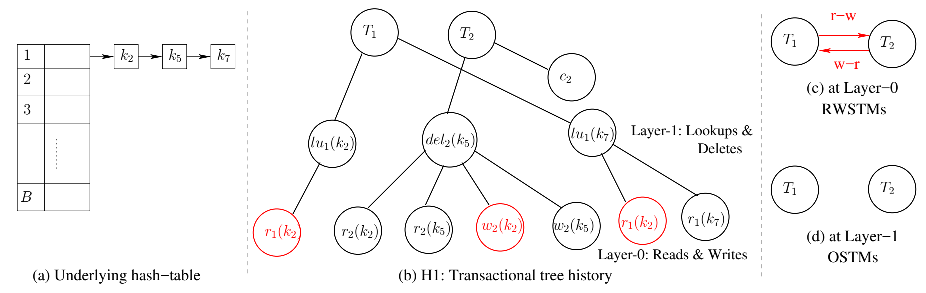

Fig. 1: Advantages of OSTMs over RWSTMs

Object STMs: Some STMs have been proposed that work on higher level operations such as hash-table. We call them *Object STMs* or *OSTMs*. It has been shown that *OSTM*s provide greater concurrency. The concept of Boosting by Herlihy et al. [3], the optimistic variant by Hassan et al. [4] and more recently *HT-OSTM* system by Peri et al. [5] are some examples that demonstrate the performance benefits achieved by *OSTM*s.

Benefit of *OSTM*s over *RWSTM*s: We now illustrate the advantage of *OSTM*s by considering a hash-table based STM system. We assume that the operations of the hash-table are insert (or ins), lookup (or lu) and delete (or del). Each hash-table consists of B buckets with the elements in each bucket arranged in the form of a linked-list. Figure 1(a) represents a hash-table with the first bucket containing keys $\langle k_2,\ k_5,\ k_7\rangle$. Figure 1 (b) shows the execution by two transaction T_1 and T_2 represented in the form of a tree. T_1 performs lookup operations on keys k_2 and k_7 while T_2 performs a delete on k_5. The delete on key k_5 generates read on the keys k_2, k_5 and writes the keys k_2, k_5 assuming that delete is performed similar to delete operation in lazy-list [6]. The lookup on k_2 generates read on k_2 while the lookup on k_7 generates read on k_2, k_7. Note that in this execution k_5 has already been deleted by the time lookup on k_7 is performed.

In this execution, we denote the read-write operations (leaves) as layer-0 and lu, del methods as layer-1. Consider the history (execution) at layer-0 (while ignoring higher-level operations), denoted as $H0$. It can be verified this history is not opaque [7]. This is because between the two reads of k_2 by T_1, T_2 writes to k_2. It can be seen that if history $H0$ is input to a *RWSTM*s one of the transactions between T_1 or T_2 would be aborted to ensure opacity [7]. The Figure 1 (c) shows the presence of a cycle in the conflict graph of $H0$.

Now, consider the history $H1$ at layer-1 consists of lu, and del methods, while ignoring the read/write operations since they do not overlap (referred to as pruning in [8, Chap 6]). These methods work on distinct keys (k_2, k_5, and k_7). They do not overlap and are not conflicting. So, they can be re-ordered in either way. Thus, $H1$ is opaque [7] with equivalent serial history T_1T_2 (or T_2T_1) and the corresponding conflict graph shown in Figure 1 (d). Hence, a hash-table based *OSTM* system does not have to abort either of T_1 or T_2. This shows that *OSTM*s can reduce the number of aborts and provide greater concurrency.

Multi-Version Object STMs: Having seen the advantage achieved by *OSTM*s (which was exploited in some works such as [3], [4], [5]), in this paper we propose and evaluate *Multi-version Object STMs* or *MVOSTMs*. Our work is motivated by the observation that in databases and *RWSTM*s by storing multiple versions for each t-object, greater concurrency can be obtained [9]. Specifically, maintaining multiple versions can ensure that more read operations succeed because the reading operation will have an appropriate version to read. Our goal is to evaluate the benefit of *MVOSTM*s over both multi-version *RWSTM*s as well as single version *OSTM*s.

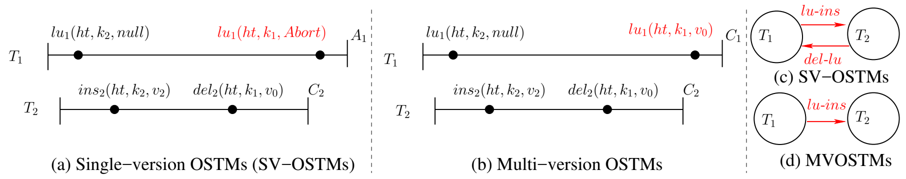

Fig. 2: Advantages of multi-version over single version *OSTM*

Potential benefit of *MVOSTM*s over *OSTM*s and multi-version *RWSTM*s: We now illustrate the advantage of *MVOSTM*s as compared to single-version *OSTM*s (*SV-OSTM*s) using hash-table object having the same operations as discussed above: ins, lu, del. Figure 2 (a) represents a history H with two concurrent transactions T_1 and T_2 operating on a hash-table ht. T_1 first tries to perform a lu on key k_2. But due to the absence of key k_2 in ht, it obtains a value of $null$. Then T_2 invokes ins method on the same key k_2 and inserts the value v_2 in ht. Then T_2 deletes the key k_1 from ht and returns v_0 implying that some other transaction had previously inserted v_0 into k_1. The second method of T_1 is lu on the key k_1. With this execution, any *SV-OSTM* system has to return abort for T_1's lu operation to ensure correctness, i.e., opacity. Otherwise, if T_1 would have obtained a return value v_0 for k_1, then the history would not be opaque anymore. This is reflected by a cycle in the corresponding conflict graph between T_1 and T_2, as shown in Figure 2 (c). Thus to ensure opacity, *SV-OSTM* system has to return abort for T_1's lookup on k_1.

In an *MVOSTM* based on hash-table, denoted as *HT-MVOSTM*, whenever a transaction inserts or deletes a key k, a new version is created. Consider the above example with a *HT-MVOSTM*, as shown in Figure 2 (b). Even after T_2 deletes k_1, the previous value of v_0 is still retained. Thus, when T_1 invokes lu on k_1 after the delete on k_1 by T_2, *HT-MVOSTM* return v_0 (as previous value). With this, the resulting history is opaque with equivalent serial history being T_1T_2. The corresponding conflict graph is shown in Figure 2 (d) does not have a cycle.

Thus, *MVOSTM* reduces the number of aborts and achieve greater concurrency than *SV-OSTM*s while ensuring the compositionality. We believe that the benefit of *MVOSTM* over multi-version *RWSTM* is similar to *SV-OSTM* over single-version *RWSTM* as explained above.

MVOSTM is a generic concept which can be applied to any data structure. In this paper, we have considered the list and hash-table based *MVOSTM*s, *list-MVOSTM* and *HT-MVOSTM* respectively. Experimental results of list-MVOSTM outperform almost two to twenty fold speedup than existing state-of-the-art STMs used to implement a list: Trans-list [10], Boosting-list [3], NOrec-list [1] and *SV-OSTM* [5] under high contention. Similarly, *HT-MVOSTM* shows significant performance gain almost two to nineteen times better than existing state-of-the-art STMs used to implement a hash-table: ESTM [2], NOrec [1] and *SV-OSTM* [5]. To the best of our knowledge, this is the first work to explore the idea of using multiple versions in *OSTM*s to achieve greater concurrency.

HT-MVOSTM and *list-MVOSTM* use an unbounded number of versions for each key. To address this issue, we develop two variants for both hash-table and list data structures (or DS): (1) A garbage collection method in *MVOSTM* to delete the unwanted versions of a key, denoted as *MVOSTM-GC*. Garbage collection gave a performance gain of 15% over *MVOSTM* without garbage collection in the best case. Thus, the overhead of garbage collection is less than the performance improvement due to improved memory usage. (2) Placing a limit of K on the number versions in *MVOSTM*, resulting in *KOSTM*. This gave a performance gain of 22% over *MVOSTM* without garbage collection in the best case.

Contributions of the paper:

- We propose a new notion of multi-version objects based STM system, *MVOSTM*. Specifically develop it for list and hash-table objects, *list-MVOSTM* and *HT-MVOSTM* respectively.
- We show *list-MVOSTM* and *HT-MVOSTM* satisfy *opacity* [7], standard correctness-criterion for STMs.
- Our experiments show that both *list-MVOSTM* and *HT-MVOSTM* provides greater concurrency and reduces the number of aborts as compared to *SV-OSTM*s, single-version *RWSTM*s and, multi-version *RWSTM*s. We achieve this by maintaining multiple versions corresponding to each key.
- For efficient space utilization in *MVOSTM* with unbounded versions we develop *Garbage Collection* for *MVOSTM* (i.e. *MVOSTM-GC*) and bounded version *MVOSTM* (i.e. *KOSTM*).

2 Building System Model

The basic model we consider is adapted from Peri et al. [5]. We assume that our system consists of a finite set of P processors, accessed by a finite number of n threads that run in a completely asynchronous manner and communicate using shared objects. The threads communicate with each other by invoking higher-level methods on the shared objects and getting corresponding responses. Consequently, we make no assumption about the relative speeds of the threads. We also assume that none of these processors and threads fail or crash abruptly.

Events and Methods: We assume that the threads execute atomic *events* and the events by different threads are (1) read/write on shared/local memory objects, (2) method invocations (or inv) event and responses (or rsp) event on higher level shared-memory objects.

Within a transaction, a process can invoke layer-1 methods (or operations) on a *hash-table* t-object. A hash-table(ht) consists of multiple key-value pairs of the form $\langle k, v\rangle$. The keys and values are respectively from sets $\mathscr{K}$ and $\mathscr{V}$. The methods that a thread can invoke are: (1) *t_begin*$_i$: begins a transaction and returns a unique id to the invoking thread. (2) *t_insert*$_i(ht, k, v)$: transaction T_i inserts a value v onto key k in ht. (3) *t_delete*$_i(ht, k, v)$: transaction T_i deletes the key k from the hash-table ht and returns the current value v for T_i. If key k does not exist, it returns $null$. (4) *t_lookup*$_i(ht, k, v)$: returns the current value v for key k in ht for T_i. Similar to *t_delete*, if the key k does not exist then *t_lookup* returns $null$. (5) *tryC*$_i$: which tries to commit all the operations of T_i and (6) *tryA*$_i$: aborts T_i. We assume that each method consists of an inv and rsp event.

We denote *t_insert* and *t_delete* as *update* methods (or upd_method) since both of these change the underlying data structure. We denote *t_delete* and *t_lookup* as *return-value methods (or rv_method)* as these operations return values from ht. A method may return ok if successful or $\mathscr{A}$(abort) if it sees an inconsistent state of ht.

Transactions: Following the notations used in database multi-level transactions [8], we model a transaction as a two-level tree. The *layer-0* consist of read/write events and *layer-1* of the tree consists of methods invoked by a transaction.

Having informally explained a transaction, we formally define a transaction T as the tuple $\langle evts(T), <_T\rangle$. Here $evts(T)$ are all the read/write events at *layer-0* of the transaction. $<_T$ is a total order among all the events of the transaction.

We denote the first and last events of a transaction T_i as $T_i.firstEvt$ and $T_i.lastEvt$. Given any other read/write event rw in T_i, we assume that $T_i.firstEvt <_{T_i} rw <_{T_i} T_i.lastEvt$. All the methods of T_i are denoted as $methods(T_i)$.

Histories: A *history* is a sequence of events belonging to different transactions. The collection of events is denoted as $evts(H)$. Similar to a transaction, we denote a history H as tuple $\langle evts(H), <_H\rangle$ where all the events are totally ordered by $<_H$. The set of methods that are in H is denoted by $methods(H)$. A method m is *incomplete* if inv(m) is in $evts(H)$ but not its corresponding response event. Otherwise, m is *complete* in H.

Coming to transactions in H, the set of transactions in H are denoted as $txns(H)$. The set of committed (resp., aborted) transactions in H is denoted by $committed(H)$ (resp., $aborted(H)$). The set of *live* transactions in H are those which are neither committed nor aborted. On the other hand, the set of *terminated* transactions are those which have either committed or aborted.

We denote two histories H_1, H_2 as *equivalent* if their events are the same, i.e., $evts(H_1) = evts(H_2)$. A history H is qualified to be *well-formed* if: (1) all the methods of a transaction T_i in H are totally ordered, i.e. a transaction invokes a method only after it receives a response of the previous method invoked by it (2) T_i does not invoke any other method after it received an $\mathscr{A}$ response or after *tryC*(ok) method. We only consider *well-formed* histories for *OSTM*.

A method m_{ij} (j^{th} method of a transaction T_i) in a history H is said to be *isolated* or *atomic* if for any other event e_{pqr} (r^{th} event of method m_{pq}) belonging to some other method m_{pq} of transaction T_p either e_{pqr} occurs before inv(m_{ij}) or after rsp(m_{ij}).

Sequential Histories: A history H is said to be *sequential* (term used in [11, 12]) if all the methods in it are complete and isolated. From now onwards, most of our discussion would relate to sequential histories.

Since in sequential histories all the methods are isolated, we treat each method as a whole without referring to its inv and rsp events. For a sequential history H, we construct the *completion* of H, denoted $\overline{H}$, by inserting *tryA*$_k(\mathscr{A})$ immediately after the last method of every transaction $T_k \in live(H)$. Since all the methods in a sequential history are complete, this definition only has to take care of completed transactions.

Real-time Order and Serial Histories: Given a history H, $<_H$ orders all the events in H. For two complete methods m_{ij}, m_{pq} in $methods(H)$, we denote $m_{ij} \prec_H^{MR} m_{pq}$ if rsp$(m_{ij}) <_H$inv(m_{pq}). Here MR stands for method real-time order. It must be noted that all the methods of the same transaction are ordered. Similarly, for two transactions T_i, T_p in $term(H)$, we denote $(T_i \prec_H^{TR} T_p)$ if $(T_i.lastEvt <_H T_p.firstEvt)$. Here TR stands for transactional real-time order.

We define a history H as *serial* [13] or *t-sequential* [12] if all the transactions in H have terminated and can be totally ordered w.r.t $\prec_{TR}$, i.e. all the transactions execute one after the other without any interleaving. Intuitively, a history H is serial if all its transactions can be isolated. Formally, $\langle(H \text{ is serial}) \implies (\forall T_i \in txns(H) : (T_i \in term(H)) \wedge (\forall T_i, T_p \in txns(H) : (T_i \prec_H^{TR} T_p) \vee (T_p \prec_H^{TR} T_i))\rangle$. Since all the methods within a transaction are ordered, a serial history is also sequential.

Legal Histories: A rv_method m_{ij} on key k is legal if it returns the value updated the latest committed transaction that updated key k. A history H is said to be legal, if all the rv_methods of H are legal. More details on legality are explained in the accompanying technical report [14].

Opacity: It is a *correctness-criteria* for STMs [7]. A sequential history H is said to be opaque if there exists a serial history S such that: (1) S is equivalent to $\overline{H}$, i.e., $evts(\overline{H}) = evts(S)$ (2) S is legal and (3) S respects the transactional real-time order of H, i.e., $\prec_H^{TR} \subseteq \prec_S^{TR}$.

3 *HT-MVOSTM* Design and Data Structure

HT-MVOSTM is a hash-table based *MVOSTM* that explores the idea of using multiple versions in *OSTM*s for hash-table object to achieve greater concurrency. The design of

HT-MVOSTM is similar to *HT-OSTM* [5] consisting of B buckets. All the keys of the hash-table in the range $\mathscr{K}$ are statically allocated to one of these buckets.

Each bucket consists of linked-list of nodes along with two sentinel nodes *head* and *tail* with values -∞ and +∞ respectively. The structure of each node is as $\langle key,\ lock,\ marked,\ vl,\ nnext\rangle$. The key is a unique value from the set of all keys $\mathscr{K}$. All the nodes are stored in increasing order in each bucket as shown in Figure 3 (a), similar to any linked-list based concurrent set implementation [6, 15]. In the rest of the document, we use the terms key and node interchangeably. To perform any operation on a key, the corresponding $lock$ is acquired. $marked$ is a boolean field which represents whether the key is deleted or not. The deletion is performed in a lazy manner similar to the concurrent linked-lists structure [6]. If the $marked$ field is true then key corresponding to the node has been logically deleted; otherwise, it is present. The vl field of the node points to the version list (shown in Figure 3 (b)) which stores multiple versions corresponding to the key. The last field of the node is $nnext$ which stores the address of the next node. It can be seen that the list of keys in a bucket is as an extension of *lazy-list* [6]. Given a node n in the linked-list of bucket B, we denote its fields as $n.key(k.key)$, $n.lock(k.lock)$, $n.marked(k.marked)$, $n.vl(k.vl)$, $n.nnext(k.nnext)$.

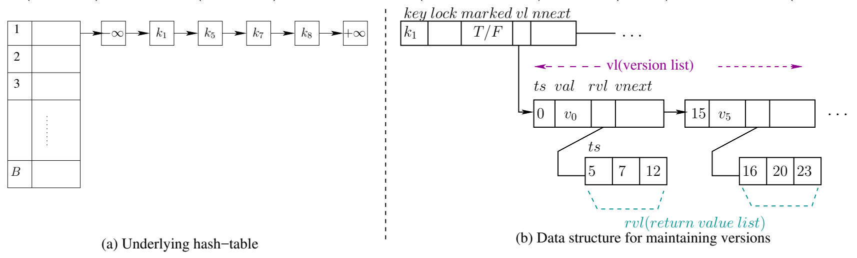

Fig. 3: *HT-MVOSTM* design

The structure of each version in the vl of a key k is $\langle ts,\ val,\ rvl,\ vnext\rangle$ as shown in Figure 3 (b). The field ts denotes the unique timestamp of the version. In our algorithm, every transaction is assigned a unique timestamp when it begins which is also its id. Thus ts of this version is the timestamp of the transaction that created it. All the versions in the vl of k are sorted by ts. Since the timestamps are unique, we denote a version, ver of a node n with key k having $ts\ j$ as $n.vl[j].ver$ or $k.vl[j].ver$. The corresponding fields in the version as $k.vl[j].ts,\ k.vl[j].val,\ k.vl[j].rvl,\ k.vl[j].vnext$.

The field val contains the value updated by an update transaction. If this version is created by an insert method $t\_insert_i(ht, k, v)$ by transaction T_i, then val will be v. On the other hand, if the method is $t\_delete_i(ht, k)$ with the return value v, then val will be $null$. In this case, as per the algorithm, the node of key k will also be marked. *HT-MVOSTM* algorithm does not immediately physically remove deleted keys from the hash-table. The need for this is explained below. Thus a rv_method (*t_delete* or *t_lookup*) on key k can return $null$ when it does not find the key or encounters a $null$ value for k.

The rvl field stands for *return value list* which is a list of all the transactions that executed rv_method on this version, i.e., those transactions which returned val. The field $vnext$ points to the next available version of that key.

Number of versions in vl (the length of the list) as per *HT-MVOSTM* can be bounded or unbounded. It can be bounded by having a limit on the number of versions such as

K. Whenever a new version ver is created and is about to be added to vl, the length of vl is checked. If the length becomes greater than K, the version with lowest ts (i.e., the oldest) is replaced with the new version ver and thus maintaining the length back to K. If the length is unbounded, then we need a garbage collection scheme to delete unwanted versions for efficiency.

Marked Nodes: *HT-MVOSTM* stores keys even after they have been deleted (nodes which have $marked$ field as true). This is because some other concurrent transactions could read from a different version of this key and not the $null$ value inserted by the deleting transaction. Consider for instance the transaction T_1 performing *t_lookup*(ht, k) as shown in Figure 2 (b). Due to the presence of previous version v_0, *HT-MVOSTM* could return this earlier version v_0 for *t_lookup*(ht, k) method. Whereas, it is not possible for *HT-OSTM* to return the version v_0 because k has been removed from the system after the delete by T_2. In that case, T_1 would have to be aborted. Thus as explained in Section 1, storing multiple versions increases the concurrency.

To store deleted keys along with live keys (or unmarked node) in a lazy-list will increase the traversal time to access unmarked nodes. Consider the Figure 4, in which there are four keys $\langle k_5, k_8, k_9, k_{12}\rangle$ present in the list. Here $\langle k_5, k_8, k_9\rangle$ are marked (or deleted) nodes while k_{12} is unmarked. Now, consider an access the key k_{12} as by *HT-MVOSTM* as a part of one of its methods. Then *HT-MVOSTM* would have to unnecessarily traverse the marked nodes to reach key k_{12}.

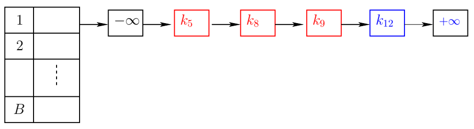

Fig. 4: Searching k_{12} over *lazy-list*

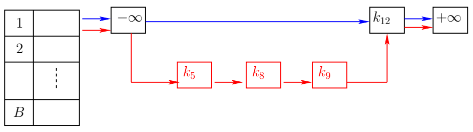

Fig. 5: Searching k_{12} over $lazyrb\text{-}list$

This motivated us to modify the lazy-list structure of nodes in each bucket to form a skip list based on red and blue links. We denote it as *red-blue lazy-list* or *lazyrb-list*. This idea was earlier explored by Peri et al. in developing *OSTMs* [5]. $lazyrb\text{-}list$ consists of nodes with two links, red link (or RL) and blue link (or BL). The node which are not marked (or not deleted) are accessible from the head via BL. While all the nodes including the marked ones can be accessed from the head via RL. With this modification, let us consider the above example of accessing unmarked key k_{12}. It can be seen that k_{12} can be accessed much more quickly through BL as shown in Figure 5. Using the idea of $lazyrb\text{-}list$, we have modified the structure of each node as ⟨ *key, lock, marked, vl, RL, BL* ⟩. Further, for a bucket B, we denote its linked-list as $B.lazyrb\text{-}list$.

4 Working of *HT-MVOSTM*

As explained in Section 2, *HT-MVOSTM* exports *t_begin*, *t_insert*, *t_delete*, *t_lookup*, *tryC* methods. *t_delete*, *t_lookup* are rv_methods while *t_insert*, *t_delete* are upd_methods. We treat *t_delete* as both rv_method as well as upd_method. The rv_methods return the current value of the key. The upd_methods, update to the keys are first noted down in local log, *txLog*. Then in the *tryC* method after validations of these updates are transferred to the shared memory. We now explain the working of rv_method and upd_method. Additional details including pseudocode is in the accompanying technical report [14].

t_begin() : A thread invokes a new transaction T_i using this method. This method returns a unique id to the invoking thread by incrementing an atomic counter. This unique id is also the timestamp of the transaction T_i. For convenience, we use the notation that i is the timestamp (or id) of the transaction T_i. The transaction T_i local log $txLog_i$ is initialized in this method.

rv_methods - *t_delete*$_i(ht, k, v)$ and *t_lookup*$_i(ht, k, v)$: Both these methods return the current value of key k. Algo 1 gives the high-level overview of these methods. First, the algorithm checks to see if the given key is already in the local log, $txLog$ of T_i (Line 2). If the key is already there then the current rv_method is not the first method on k and is a subsequent method of T_i on k. So, we can return the value of k from the $txLog_i$.

If the key is not present in the $txLog_i$, then *HT-MVOSTM* searches into shared memory. Specifically, it searches the bucket to which k belongs to. Every key in the range $\mathscr{K}$ is statically allocated to one of the B buckets. So the algorithms search for k in the corresponding bucket, say B_k to identify the appropriate location, i.e., identify the correct *predecessor* or $pred$ and *current* or $curr$ keys in the lazyrb-list of B_k without acquiring any locks similar to the search in lazy-list [6]. Since each key has two links, RL and BL, the algorithm identifies four node references: two $pred$ and two $curr$ according to red and blue links. They are stored in the form of an array with $preds[0]$ and $currs[1]$ corresponding to blue links; $preds[1]$ and $currs[0]$ corresponding to red links. If both $preds[1]$ and $currs[0]$ nodes are unmarked then the $pred, curr$ nodes of both red and blue links will be the same, i.e., $preds[0] = preds[1]$ and $currs[0] = currs[1]$. Thus depending on the marking of $pred, curr$ nodes, a total of two, three or four different nodes will be identified. Here, the search ensures that $preds[0].key \leq preds[1].key < k \leq currs[0].key \leq currs[1].key$.

Next, the re-entrant locks on all the $pred, curr$ keys are acquired in increasing order to avoid the deadlock. Then all the $pred$ and $curr$ keys are validated by *rv_Validation()* in Line 7 as follows: (1) If $pred$ and $curr$ nodes of blue links are not marked, i.e, $(\neg preds[0].marked)$ && $(\neg currs[1].marked)$. (2) If the next links of both blue and red $pred$ nodes point to the correct $curr$ nodes: $(preds[0].BL = currs[1])$ && $(preds[1].RL = currs[0])$.

If any of these checks fail, then the algorithm retries to find the correct $pred$ and $curr$ keys. It can be seen that the validation check is similar to the validation in concurrent lazy-list [6].

Next, we check if k is in $B_k.lazyrb\text{-}list$. If k is not in B_k, then we create a new node for k as: $\langle key = k, lock = false, marked = false, vl = v, nnext = \phi\rangle$ and insert it into $B_k.lazyrb\text{-}list$ such that it is accessible only via RL since this node is marked (Line 14). This node will have a single version v as: $\langle ts = 0, val = null, rvl = i, vnext = \phi\rangle$. Here invoking transaction T_i is creating a version with timestamp 0 to ensure that rv_methods of other transactions will never abort. As we have explained in Figure 2 (b) of Section 1, even after T_2 deletes k_1, the previous value of v_0 is still retained. Thus, when T_1 invokes lu on k_1 after the delete on k_1 by T_2, *HT-MVOSTM* will return v_0 (as previous value). Hence, each rv_methods will find a version to read while maintaining the infinite version corresponding to each key k. In rvl, T_i adds the timestamp as i in it and $vnext$ is initialized to empty value. Since val is null and the n, this version and the node is not technically inserted into $B_k.lazyrb\text{-}list$.

If k is in $B_k.lazyrb\text{-}list$ then, k is the same as $currs[0]$ or $currs[1]$ or both. Let n be the node of k in $B_k.lazyrb\text{-}list$. We then find the version of n, ver_j which has the timestamp j such that j has the largest timestamp smaller than i (timestamp of T_i). Add i to ver_j's rvl (Line 22). Then release the locks, update the local log $txLog_i$ in Line 24 and return the value stored in $ver_j.val$ in Line 26).

Algorithm 1 *rv_method:* Could be either *t_delete*$_i(ht, k, v)$ or *t_lookup*$_i(ht, k, v)$ on key k that maps to bucket B_k.

```
1:  procedure $rv\_method_i(ht, k, v)$
2:      if ($k \in txLog_i$) then
3:          Update the local log and return $val$.
4:      else
5:          Search in lazyrb-list to identify the $preds[]$ and $currs[]$ for $k$ using BL and RL in bucket $B_k$.
6:          Acquire the locks on $preds[]$ and $currs[]$ in increasing order.
7:          if (! $rv\_Validation(preds[], currs[])$) then
8:              Release the locks and goto Line 5.
9:          end if
10:         if ($k \notin B_k.lazyrb\text{-}list$) then
11:             Create a new node $n$ with key $k$ as: ⟨ key = k, lock = false, marked = false, vl = v, nnext = $\phi$⟩.
12:             /* The $vl$ consists of a single element $v$ with $ts$ as $i$ */
13:             Create the version $v$ as: $\langle ts = 0, val = null, rvl = i, vnext = \phi\rangle$.
14:             Insert $n$ into $B_k.lazyrb\text{-}list$ such that it is accessible only via RLs. /* $n$ is marked */
15:             Release the locks; update the $txLog_i$ with $k$.
16:             return $null$.
17:         end if
18:         Identify the version $ver_j$ with $ts = j$ such that $j$ is the largest timestamp smaller than $i$.
19:         if ($ver_j$ == $null$) then
20:             goto Line 11.
21:         end if
22:         Add $i$ into the $rvl$ of $ver_j$.
23:         $retVal = ver_j.val$.
24:         Release the locks; update the $txLog_i$ with $k$ and $retVal$.
25:     end if
26:     return $retVal$.
27: end procedure
```

upd_methods - *t_insert* and *t_delete*: Both the methods create a version corresponding to the key k. The actual effect of *t_insert* and *t_delete* in shared memory will take place in *tryC*. Algo 2 represents the high-level overview of *tryC*.

Initially, to avoid deadlocks, algorithm sorts all the $keys$ in increasing order which are present in the local log, $txLog_i$. In *tryC*, $txLog_i$ consists of upd_methods (*t_insert* or *t_delete*) only. For all the upd_methods (opn_i) it searches the key k in the shared memory corresponding to the bucket B_k. It identifies the appropriate location ($pred$ and $curr$) of key k using BL and RL (Line 25) in the lazyrb-list of B_k without acquiring any locks similar to rv_method explained above.

Next, it acquires the re-entrant locks on all the $pred$ and $curr$ keys in increasing order. After that, all the $pred$ and $curr$ keys are validated by *tryC_Validation* in Line 27 as follows: (1) It does the *rv_Validation()* as explained above in the rv_method. (2) If key k exists in the $B_k.lazyrb\text{-}list$ and let n as a node of k. Then algorithm identifies the version of n, ver_j which has the timestamp j such that j has the largest timestamp smaller than i (timestamp of T_i). If any higher timestamp k of T_k than timestamp i of T_i exist in $ver_j.rvl$ then algorithm returns $Abort$ in Line 28.

If all the above steps are true then each upd_methods exist in $txLog_i$ will take the effect in the shared memory after doing the *intraTransValidation()* in Line 33. If two $upd\_methods$ of the same transaction have at least one common shared node among its recorded $pred$ and $curr$ keys, then the previous upd_method effect may overwrite if the current upd_method of $pred$ and $curr$ keys are not updated according to the updates done by the previous upd_method. Thus to solve this we have *intraTransValidation()* that modifies the $pred$ and $curr$ keys of current operation based on the previous operation in Line 33.

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

```
20: procedure $tryC(T_i)$
21:     /*Operation name ($opn$) which could be either *t_insert* or *t_delete* */
22:     /*Sort the $keys$ of $txLog_i$ in increasing order.*/
23:     for all ($opn_i \in txLog_i$) do
24:         if (($opn_i$ == *t_insert*) || ($opn_i$ == *t_delete*)) then
25:             Search in $lazyrb\text{-}list$ to identify the $preds$[] and $currs$[] for $k$ of $opn_i$ using BL and RL in bucket $B_k$.
26:             Acquire the locks on $preds$[] and $currs$[] in increasing order.
27:             if (! $tryC\_Validation()$) then
28:                 return $Abort$.
29:             end if
30:         end if
31:     end for
32:     for all ($opn_i \in txLog_i$) do
33:         $intraTransValidation()$ modifies the $preds$[] and $currs$[] of current operation which would have been updated by the previous operation of the same transaction.
34:         if (($opn_i$ == *t_insert*) && ($k \notin B_k.lazyrb\text{-}list$)) then
35:             Create new node $n$ with $k$ as: ⟨ *key = k, lock = false, marked = false, vl = v, nnext = $\phi$* ⟩.
36:             Create two versions $v$ as: ⟨ *ts=i, val=v, rvl=$\phi$, vnext=$\phi$* ⟩.
37:             Insert node $n$ into $B_k.lazyrb\text{-}list$ such that it is accessible via RL as well as BL /* $lock$ sets $true$ */.
38:         else if ($opn_i$ == *t_insert*) then
39:             Add the version $v$ as: ⟨ *ts = i, val = v, rvl = $\phi$, vnext = $\phi$* ⟩ into $B_k.lazyrb\text{-}list$ such that it is accessible via RL as well as BL.
40:         end if
41:         if ($opn_i$ == *t_delete*) then
42:             Add the version $i$ as: ⟨ *ts=i, val=null, rvl=$\phi$, vnext=$\phi$* ⟩ into $B_k.lazyrb\text{-}list$ such that it is accessible only via RL.
43:         end if
44:         Update the $preds$[] and $currs$[] of $opn_i$ in $txLog_i$.
45:     end for
46:     Release the locks; return $Commit$.
47: end procedure
```

Next, we check if upd_method is *t_insert* and k is in $B_k.lazyrb\text{-}list$. If k is not in B_k, then create a new node n for k as: $\langle key = k, lock = false, marked = false, vl = v, nnext = \phi\rangle$. This node will have a single version v as: $\langle ts = i, val = v, rvl = \phi, vnext = \phi\rangle$. Here i is the timestamp of the transaction T_i invoking this method; rvl and $vnext$ are initialized to empty values. We set the val as v and insert n into $B_k.lazyrb\text{-}list$ such that it is accessible via RL as well as BL and set the lock field to be $true$ (Line 37). If k is in $B_k.lazyrb\text{-}list$ then, k is the same as $currs[0]$ or $currs[1]$ or both. Let n be the node of k in $B_k.lazyrb\text{-}list$. Then, we create the version v as: $\langle ts = i, val = v, rvl = \phi, vnext = \phi\rangle$ and insert the version into $B_k.lazyrb\text{-}list$ such that it is accessible via RL as well as BL (Line 39).

Subsequently, we check if upd_method is *t_delete* and k is in $B_k.lazyrb\text{-}list$. Let n be the node of k in $B_k.lazyrb\text{-}list$. Then create the version v as: $\langle ts = i, val = null, rvl = \phi, vnext = \phi\rangle$ and insert the version into $B_k.lazyrb\text{-}list$ such that it is accessible only via RL (Line 42).

Finally, at Line 44 it updates the $pred$ and $curr$ of opn_i in local log, $txLog_i$. At Line 46 releases the locks on all the $pred$ and $curr$ in increasing order of keys to avoid deadlocks and return $Commit$.
Now, we have the following properties about *HT-MVOSTM*.

Theorem 1. *Any history generated by HT-MVOSTM is opaque.*

Theorem 2. *HT-MVOSTM with unbounded versions ensures that rv_methods do not return abort.*

Theorem 2 gives us a nice property a transaction with *t_lookup* only methods will not abort.

5 Experimental Evaluation

In this section, we present our experimental results. We have two main goals in this section: (1) evaluating the benefit of multi-version object STMs over the single-version object STMs, and (2) evaluating the benefit of multi-version object STMs over multi-version read-write STMs. We use the *HT-MVOSTM* described in Section 4 as well as the corresponding *list-MVOSTM* which implements the list object. We also consider extensions of these multi-version object STMs to reduce the memory usage. Specifically, we consider a variant that implements garbage collection with unbounded versions and another variant where the number of versions never exceeds a given threshold K.
Experimental system: The Experimental system is a large-scale 2-socket Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz with 14 cores per socket and two hyper-threads (HTs) per core, for a total of 56 threads. Each core has a private 32KB L1 cache and 256 KB L2 cache (which is shared among HTs on that core). All cores on a socket share a 35MB L3 cache. The machine has 32GB of RAM and runs Ubuntu 16.04.2 LTS. All code was compiled with the GNU C++ compiler (G++) 5.4.0 with the build target x86_64-Linux-gnu and compilation option -std=c++1x -O3.

STM implementations: We have taken the implementation of NOrec-list [1], Boosting-list [3], Trans-list [10], ESTM [2], and RWSTM directly from the TLDS framework[3]. And the implementation of OSTM and MVTO published by Sathya Peri, one of the author of this paper. We implemented our algorithms in C++. Each STM algorithm first creates N-threads, each thread, in turn, spawns a transaction. Each transaction exports the following methods as follows: *t_begin*, *t_insert*, *t_lookup*, *t_delete* and *tryC*.
Methodology:[4] We have considered two types of workloads: ($W1$) Li - Lookup intensive (90% lookup, 8% insert and 2% delete) and ($W2$) Ui - Update intensive(10% lookup, 45% insert and 45% delete). The experiments are conducted by varying number of threads from 2 to 64 in power of 2, with 1000 keys randomly chosen. We assume that the hash-table of *HT-MVOSTM* has five buckets and each of the bucket (or list in case of *list-MVOSTM*) can have a maximum size of 1000 keys. Each transaction, in turn, executes 10 operations which include *t_lookup*, *t_delete* and *t_insert* operations. We take an average over 10 results as the final result for each experiment.

[3] https://ucf-cs.github.io/tlds/
[4] Code is available here: https://github.com/PDCRL/MVOSTM

Results: Figure 6 shows *HT-MVOSTM* outperforms all the other algorithms(HT-MVTO, RWSTM, ESTM, HT-OSTM) by a factor of 2.6, 3.1, 3.8, 3.5 for workload type $W1$ and by a factor of 10, 19, 6, 2 for workload type $W2$ respectively. As shown in Figure 6, List based MVOSTM (*list-MVOSTM*) performs even better compared with the existing state-of-the-art STMs (list-MVTO, NOrec-list, Boosting-list, Trans-list, list-OSTM) by a factor of 12, 24, 22, 20, 2.2 for workload type $W1$ and by a factor of 169, 35, 24, 28, 2 for workload type $W2$ respectively. As shown in Figure 7 for both types of workloads, HT-MVOSTM and list-MVOSTM have the least number of aborts.

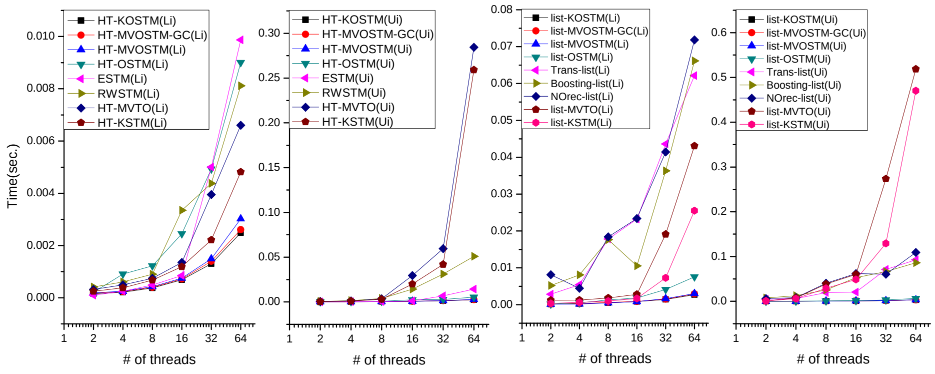

Fig. 6: Performance of *HT-MVOSTM* and *list-MVOSTM*

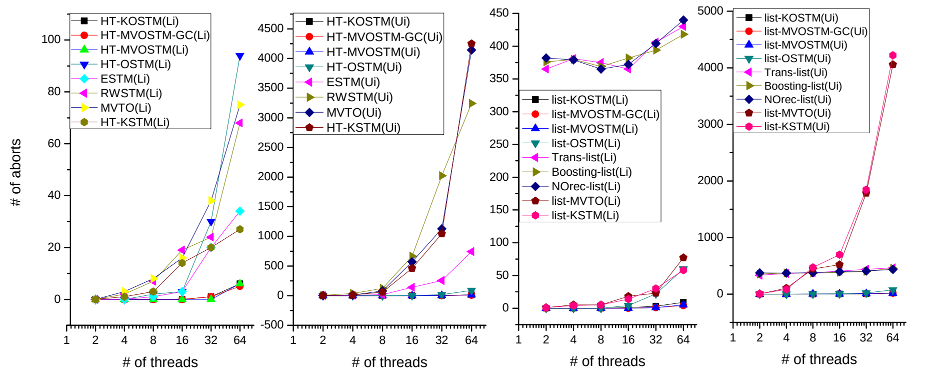

Fig. 7: Aborts of *HT-MVOSTM* and *list-MVOSTM*

MVOSTM-GC and KOSTM: For efficient memory utilization, we develop two variations of *MVOSTM*. The first, *MVOSTM-GC*, uses unbounded versions but performs garbage collection. **This is achieved by deleting non-latest versions whose timestamp is less than the timestamp of the least live transaction.** *MVOSTM-GC* gave a performance gain of 15% over *MVOSTM* without garbage collection in the best case. The second, *KOSTM*, keeps at most K versions by deleting the oldest version when $(K+1)^{th}$ version is created by a current transaction. As *KOSTM* has limited number of versions while *MVOSTM-GC* can have infinite versions, the memory consumed by

KOSTM is 21% less than *MVOSTM*. (Implementation details for both are in the technical report [14].)

We have integrated these variations in both hash-table based (*HT-MVOSTM-GC* and *HT-KOSTM*) and linked-list based MVOSTMs (*list-MVOSTM-GC* and *list-KOSTM*), we observed that these two variations increase the performance, concurrency and reduces the number of aborts as compared to MVOSTM.

Experiments show that these variations outperform the corresponding MVOSTMs. Between these two variations, *KOSTM* perform better than *MVOSTM-GC* as shown in Figure 6 and Figure 7. *HT-KOSTM* helps to achieve a performance speedup of 1.22 and 1.15 for workload type $W1$ and speedup of 1.15 and 1.08 for workload type $W2$ as compared to *HT-MVOSTM* and *HT-MVOSTM-GC* respectively. Whereas *list-KOSTM* (with four versions) gives a speedup of 1.1, 1.07 for workload type $W1$ and speedup of 1.25, 1.13 for workload type $W2$ over the *list-MVOSTM* and *list-MVOSTM-GC* respectively.

6 Conclusion and Future Work

Multi-core systems have become very common nowadays. Concurrent programming using multiple threads has become necessary to utilize all the cores present in the system effectively. But concurrent programming is usually challenging due to synchronization issues between the threads.

In the past few years, several STMs have been proposed which address these synchronization issues and provide greater concurrency. STMs hide the synchronization and communication difficulties among the multiple threads from the programmer while ensuring correctness and hence making programming easy. Another advantage of STMs is that they facilitate compositionality of concurrent programs with great ease. Different concurrent operations that need to be composed to form a single atomic unit is achieved by encapsulating them in a single transaction.

In literature, most of the STMs are *RWSTM*s which export read and write operations. To improve the performance, a few researchers have proposed *OSTM*s [3–5] which export higher level objects operation such as hash-table insert, delete etc. By leveraging the semantics of these higher level operations, these STMs provide greater concurrency. On the other hand, it has been observed in STMs and databases that by storing multiple versions for each t-object in case of *RWSTM*s provides greater concurrency [9, 16].

This paper presents the notion of multi-version object STMs and compares their effectiveness with single version object STMs and multi-version read-write STMs. We find that multi-version object STM provides a significant benefit over both of these for different types of workloads. Specifically, we have evaluated the effectiveness of MVOSTM for the list and hash-table data structure as *list-MVOSTM* and *HT-MVOSTM*. Experimental results of *list-MVOSTM* provide almost two to twenty fold speedup over existing state-of-the-art list based STMs (Trans-list, Boosting-list, NOrec-list, list-MVTO, and list-OSTM). Similarly, *HT-MVOSTM* shows a significant performance gain of almost two to nineteen times better than existing state-of-the-art hash-table based STMs (ESTM, RWSTMs, HT-MVTO, and HT-OSTM).

HT-MVOSTM and *list-MVOSTM* and use unbounded number of versions for each key. To limit the number of versions, we develop two variants for both hash-table and list data-structures: (1) A garbage collection method in *MVOSTM* to delete the unwanted versions of a key, denoted as *MVOSTM-GC*. (2) Placing a limit of k on the number versions in *MVOSTM*, resulting in *KOSTM*. Both these variants gave a performance gain of over 15% over *MVOSTM*.

References

1. 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) 67–78
2. Felber, P., Gramoli, V., Guerraoui, R.: Elastic Transactions. J. Parallel Distrib. Comput. **100**(C) (February 2017) 103–127
3. Herlihy, M., 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) 207–216
4. Hassan, A., Palmieri, R., Ravindran, B.: Optimistic transactional boosting. In: PPoPP. (2014) 387–388
5. Peri, S., Singh, A., Somani, A.: Efficient means of Achieving Composability using Transactional Memory. NETYS '18 (2018)
6. 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
7. Guerraoui, R., Kapalka, M.: On the Correctness of Transactional Memory. In: PPoPP, ACM (2008) 175–184
8. Weikum, G., Vossen, G.: Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann (2002)
9. Kumar, P., Peri, S., Vidyasankar, K.: A TimeStamp Based Multi-version STM Algorithm. In: ICDCN. (2014) 212–226
10. Zhang, D., Dechev, D.: Lock-free Transactions Without Rollbacks for Linked Data Structures. SPAA '16, New York, NY, USA, ACM (2016) 325–336
11. Kuznetsov, P., Peri, S.: Non-interference and local correctness in transactional memory. Theor. Comput. Sci. **688** (2017) 103–116
12. Kuznetsov, P., Ravi, S.: On the cost of concurrency in transactional memory. In: OPODIS. (2011) 112–127
13. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4) (1979) 631–653
14. Juyal, C., Kulkarni, S., Kumari, S., Peri, S., Somani, A.: An Innovative Approach for Achieving Composability in Concurrent Systems using Multi-Version Object Based STMs. CoRR **abs/1712.09803** (2017)
15. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings. (2001) 300–314
16. Perelman, D., Fan, R., Keidar, I.: On Maintaining Multiple Versions in STM. In: PODC. (2010) 16–25