An Innovative Approach for Achieving Composability in Concurrent Systems using Multi-Version Object Based STMs*

Sandeep Kulkarni[2], Sweta Kumari[1], Sathya Peri[1], and Archit Somani[1]

[1]Department of Computer Science Engineering, IIT Hyderabad
[2]Department of Computer Science, Michigan State University

Abstract

In the modern era of multicore processors, utilizing multiple cores properly is a tedious job. Synchronization and communication among processors involve high cost. Software transaction memory systems (STMs) addresses this issues and provide better concurrency in which programmer need not have to worry about consistency issues. Several big-data applications which deal large amounts of data can benefit from Transactional Memory Systems.

In this paper, we introduce a new STM system as multi-version object based STM (*MV-OSTM*) which is the fusion of object based STM with multiple versions. As the name suggests *MV-OSTM*, works on a higher level and keeping the multiple versions corresponding to each key. Presently, we have developed *MV-OSTM* with the unlimited number of versions corresponding to each key. To overcome traversal overhead, it performs the garbage collection method to delete the unwanted versions corresponding to the key. It provides greater concurrency while reducing the number of aborts. It ensures composability by making the transaction as atomic. In the proposed algorithm, k is the input parameter and the value of it will be decided by the programmer and depends on the application. Programmer can tune the value of k from 1 to ∞. If k equal to 1 then it will boil down to single version object based STM (*OSTM*) and if k equal to ∞ then it will be equivalent to multi-version *OSTM* with ∞ versions.

MV-OSTM satisfies correctness-criteria as opacity. For a given version order of keys, if any history H generated by *MV-OSTM* produces acyclic graph then H is opaque. The progress condition of the proposed *MV-OSTM* is *multi-version permissiveness* or *mv-permissiveness* which never aborts a transaction which is having return-value method only. 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.

1 Introduction

Software Transaction Memory Systems (*STMs*) are a convenient programming interface for a programmer to access shared memory without worrying about concurrency issues [7, 19]. Concurrently executing transactions access shared memory through the interface provided by the *STMs*. Thus with STMs, the programmer can focus on harnessing optimum parallelism from the application instead of worrying about the locking, races and deadlocks. Transactional Memory Systems can benefit several big-data applications which deal large amounts of data and parallelism to process them.

Another advantage of STMs is that they facilitate composability of concurrent programs with great ease. Different concurrent operations that need to be composed to form a single atomic unit is simply achieved by encapsulating all these operations as a single transaction. Composition of concurrent programs is a very nice feature which makes STMs very appealing to use by programmers.

Most of the *STMs* proposed in the literature are specifically based on read/write primitive operations (or methods) on memory buffers (or memory registers). These *STMs* typically export the following methods: *t_begin* which begins a transaction, *t_read* which reads from a buffer, *t_write* which writes onto a buffer, *tryC* which validates the operations of the transaction and tries to commit. If validation is successful then it returns commit otherwise STMs export *tryA* which returns abort. We refer to these as *Read-Write STMs or RWSTMs*. As a part of the validation, the STMs typically check for *conflicts* among the operations. Two operations are said to be conflicting if at least one of them is a write (or update) operation. Normally, the order of two conflicting operations can not be commutated.

On the other hand, *Object-based STM or OSTM* operate on higher level objects rather than read & write operations on memory locations. They include more complicated operations such as enq/deq on queue objects, push/pop on stack objects etc.

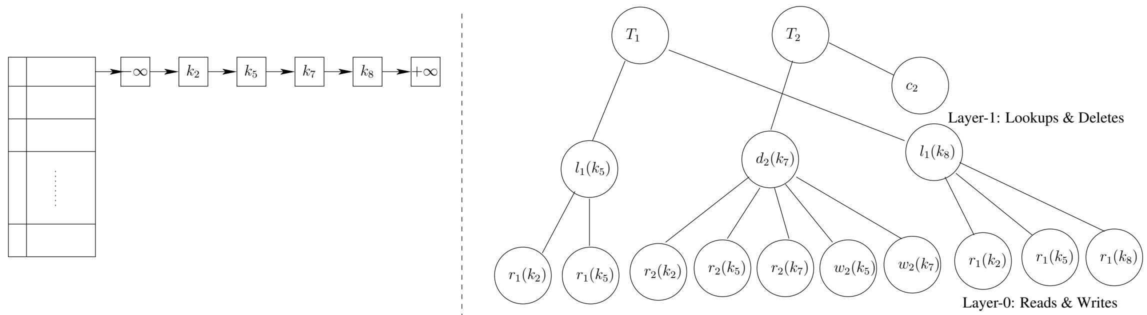
It was shown in databases that object-based schedulers provide greater concurrency than read-write based systems [20, Chap 6]. Harris et al. [3], Hassan et al [4], Herlihy et al.[17, 8] extended this concept to STMs. *OSTM*s achieve greater concurrency by milking the richer semantics of object level operations. In this paper, we consider `hash table` *OSTM* implemented using list. We assume that the `hash table` object supports insert, delete and lookup operations on ⟨key, value⟩ pairs. We show the correctness of the resulting *OSTM* by showing it is opaque[2].

*work in progress

Now, we explain how *OSTM*s provide greater concurrency than *RWSTMs* using `hash table`. Consider an *OSTM* operating on the `hash table` object exports the following methods: (1) *t_begin* which begins a transaction (same as in *RWSTMs*), (2) *t_insert* which inserts a value for a given key, (3) *t_delete* which deletes the value associated with the given key and returns the current value of the key, (4) *t_lookup* which looks up the value associated with the given key and (5) *tryC* which validates the operations of the transaction.

We denote *t_insert*, *t_delete* as update methods since as the name suggests, they update the shared memory. Along the same lines, we denote *t_lookup*, *t_delete* as rv_method as they return the current value of the key on which the method operates on. Thus it can be see that *t_delete* is both an update as well as rv_method. STMs being optimistic in nature, the affect of update methods takes place only upon the commit of the transactions. On being aborted, all the updates by the transaction are discarded.

An intuitive way to implement the `hash table` object is using a collection of lists. All the keys that hash onto the same *bucket* are chained into the same list. Each element of the list stores the ⟨key, value⟩ pair. The elements of the list are sorted by their keys similar to the set implementations discussed in [6, Chap 9]. Figure 1 a) shows this implementation. It can be seen that the underlying list is a concurrent data-structure (DS) manipulated by multiple transactions (and hence threads). So we have adopted the lazy-list approach [5] to implement the operations of the list denoted as: *list_ins*, *list_del* and *list_lookup* (referred as contains in [5]). Thus when a transaction invokes *t_insert*, *t_delete* and *t_lookup* methods, the STM internally invokes the *list_ins*, *list_del* and *list_lookup* methods respectively.



a) Underlying list b) H1: Transactional tree history

Figure 1: Motivational example for OSTMs Vs RWSTMs

Consider an instance of list in one of the chains of `hash table` which contains nodes with keys $\langle k_2\ k_5\ k_7\ k_8\rangle$ as shown in Figure 1 a). Suppose transactions T_1 and T_2 are concurrently executing *t_lookup*$_1(k_5)$, *t_delete*$_2(k_7)$ and *t_lookup*$_1(k_8)$ as shown in Figure 1 b). For simplicity, we refer to nodes of the list by their keys and we abbreviate *t_delete*, *t_lookup*, *t_insert*, *commit* and *abort* as d, l, i, c and a respectively.

In this setting, suppose a transaction T_1 of *OSTM* invokes methods *t_lookup* on the keys k_5, k_8. This would internally cause the *OSTM* to invoke *list_lookup* method on keys $\langle k_2, k_5\rangle$ and $\langle k_2, k_5, k_7, k_8\rangle$ respectively. Concurrently, suppose transaction T_2 invokes the method *t_delete* on key k_7 between the two *t_lookup*s of T_1. This would cause, *OSTM* to invoke *list_del* method of list on k_7. Since, we are using lazy-list approach on the underlying list, *list_del* involves pointing the next field of element k_5 to k_8 and marking element k_7 as deleted. Thus *list_del* of k_7 would execute the following sequence of read/write level operations- $r(k_2)r(k_5)r(k_7)w(k_5)w(k_7)$ where $r(k_5), w(k_5)$ denote read & write on the element k_5 with some value respectively. The execution of *OSTM* denoted as a *history* can be represented as a transactional forest as shown in Figure 1 b). Here the execution of each transaction is a tree.

In this execution, we denote the read-write operations (leaves) as layer-0 and *t_lookup*, *t_delete* 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[2]. This is because between the two reads of k_5 by T_1, T_2 writes to k_5. It can be seen that if history $H0$ is input to a *RWSTMs* one of the transactions among T_1 or T_2 would be aborted to ensure correctness (in this case opacity[2]).

On the other hand consider the history $H1$ at layer-1 consisting of *t_lookup*, *t_delete* methods while ignoring the underlying read/write operations. We ignore the underlying read & write operations since they do not overlap (referred to as pruning in [20, Chap 6]). Since these methods operate on different keys, they are not conflicting and can be re-ordered either way. Thus, we get that $H1$ is opaque[2] with T_1T_2 (or T_2T_1) being an equivalent serial history.

The important idea in the above argument is ignoring lower-level operations since they do not overlap. Harris et al. referred to it as *benign-conflicts*[3]. This history clearly shows the advantage of considering STMs with higher level operations in this case they are *t_insert*, *t_delete* and *t_lookup*. With object level modeling of histories, we get a higher number of acceptable schedules than read/write model. This is because not all conflicts at the lower level matter at the higher level. Thus, *OSTM* reduces number of aborts and provides greater concurrency which can greatly benefit composition of operations of higher level objects. These ideas have been explored in Harris et al. [3], Hassan et al [4], Herlihy et al.[17, 8].

It must be noted, there are instances where the conflicts at lower level do matter at the higher level. Thus *OSTM*s must be carefully designed to ensure correctness while not reducing the efficiency.

Motivational example of *MV-OSTM* : It was observed in databases and STMs that storing multiple versions in *RWSTMs* provides better concurrency [10, 16]. Maintaining multiple versions can ensure that more read operations succeed because the reading operation will have an appropriate version to read. This motivated us to consider multiple versions of objects with *OSTM*s.

We consider an example to motivate the advantage of having multiple object versions. Figure 2 a) represents a history H with two concurrent transactions T_1 and T_2 operating on a `hash table`. T_1 first performs a *t_lookup* on key k_2. But due to absence of key k_2 in `hash table` ht, its gets $NULL$. After that suppose T_2 invokes *t_insert* method on the same key k_2 and inserts the value v_2 in `hash table` ht. Then T_2 deletes the key k_1 from `hash table` ht and returns v_3 implying that some other transaction had previously inserted v_3 into k_1. The second method of T_1 is *t_lookup* on the key k_1. In this case the STM system has to returns abort to ensure correctness, i.e., opacity. If T_1 obtained a return value of $NULL$ for k_1, then the history will not be opaque.

In order to improve concurrency, we can use multiple versions for each key. Whenever a transaction inserts or deletes, a new version is created. Hence, in the above example even after T_2 deletes k_1, the previous value of v_3 is still retained. Thus, when T_1 invokes *t_lookup* on k_1 after the delete on k_1 by T_2, if the return value is v_3 (the old value) then the history is opaque. In this case, the equivalent serial history being T_1T_2. This is shown in Figure 2 b). Thus by using multiple versions for each key, we get higher number *t_lookup* methods can commit.

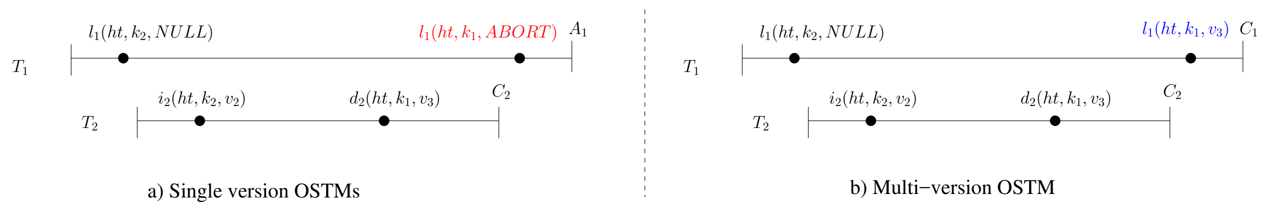


Figure 2: Advantages of multi version over single version *OSTM*

Thus to reduce the number of aborts and achieving greater concurrency we propose *MV-OSTM*. 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. This we believe can in turn ensure greater number of successful composed operations.

Currently, we have developed *MV-OSTM* with the ∞ number of versions for each key. So, we need garbage collection method to delete the unwanted versions of a key. Our contributions are as follows:

- We have proposed a new STM as *MV-OSTM* which providing the greater concurrency with the help of multiple versions to reduce the number of aborts and its composable too.
- *MV-OSTM* ensures the progress condition as *multi-version permissiveness* or *mv-permissiveness* [18]. A mv-permissive *MV-OSTM* system never aborts a return-value only transaction. In that sense *return-value method* will never return abort because of ∞ versions.
- We have developed the *garbage collection* method to delete old & unwanted versions from *MV-OSTM*.
- *MV-OSTM* satisfies *opacity*.

Roadmap. We describe our system model in Section 2. Section 3 represents the graph characterization for *MV-OSTM*. Section 4 describes the design along with data structure and pcode of *MV-OSTM* algorithm. We conclude in Section 5 followed by future direction. Finally in technical report[a], we describe graph characterization of opacity, detailed description of data structure, missing pcode and garbage collection method.

2 System Model and Preliminaries

The basic model we consider is adapted from Kuznetsov et.al, [11] and Lev-Ari et. al.[14, 15]. It comprises of n processes, $p_1, \ldots, p_n$ that access a collection of shared *t-objects/keys* via atomic *transactions*. A process is accessed by a thread and internally, a threads may

[a]

invoke one or more atomic transactions. Transactions consists of multiple operations on *keys*. Key is a container of data. Transaction T_i is accessing keys to perform the operations of `hash table` (*t_begin*, *t_lookup*, *t_insert*, *t_delete* and *tryC*).

We are assuming the version created by each transaction on each keys are unique. Let say, if transaction T_j has created a version on key k_i then the version corresponding to the key is represented as k_{ij}.

Events: Lower level operations of STMs (*t_begin*(), *t_read*(), *t_write*(), *tryC*()) are events. We assume that events are atomic.

Methods: A method consists of multiple events including invocation (inv) and response (res). So, it is a higher level operation on `hash table` (ht) invoked by a transaction T_i on any key k. Method can be as follows: *init*(), *t_begin*$_i$(), *t_lookup*$_i$(), *t_insert*$_i$(),*t_delete*$_i$() and *tryC*$_i$() as explained in Section 1. Consider a method m composed of multiple events as $evts(m)$ then m should have total order among all the events $evts(m)$ invoked by it. Formally, $\langle evts(m), <_m\rangle$. As *t_insert* and *t_delete* are modifying the underlying data-structure so, we represent it as *update* methods (or upd_method or up). Methods, *t_delete* and *t_lookup* returns the values from `hash table` ht, so we represent it as *return-value* method (or rv_method or rvm).

Transactions: As defined in database multi-level transactions [20], it modeled as a two-layer tree. The *layer-0* comprises of low level operations as read/write events. Consider a transaction T_i composed of multiple events as $evts(T_i)$ then T_i should have total order among all the events $evts(T_i)$ invoked by it. Formally, $\langle evts(T_i), <_{T_i}\rangle$.

The *layer-1* of the tree consists of methods invoked by transaction at higher level. A transaction can invoked multiple methods. Consider a transaction T_i composed of multiple methods as $methods(T_i)$ then T_i should have total order among all the methods $methods(T_i)$ invoked by it. Formally, $\langle methods(T_i), <_{T_i}\rangle$.

Histories: It consists of sequence of interleaving events of different transactions. We denote events of history H as $evts(H)$. A history H should have total order among all the events $evts(H)$ invoked by it. Formally, $\langle evts(H), <_H\rangle$. The method of H is represented as $methods(H)$ which is made up of inv(m) and rsp(m).

Sequential Histories: A history H is said to be *sequential* [12, 13]) or *linearized* [9] if all the methods are atomic instead of interval. Consider a sequential history H, let $m_{ij}(ht, k, v/nil)$, where m_{ij} stands for j^{th} method of i^{th} transaction.

Real-time Order & Serial Histories: Two methods m_{ij} and m_{pq} of history H are in real-time order, if rsp(m_{ij}) $<_H$inv(m_{pq}). Similarly, two transactions T_i and T_j are in real-time order, if $(T_i.lastEvt <_H T_j.firstEvt)$. A history H is said to be serial if all the transactions are atomic and totally ordered.

Valid and Legal Histories: A history H is said to valid if all the rv_methods are lookup from previous committed transactions. A history H is said to be legal, if all the rv_methods of H are legal. If T_j invokes rv_method on key k_1 from T_i in H, note that in order for this to happen, T_i must have closest committed before T_j i.e. $c_i <_H$ *rvm*$_j(k_{1,i})$. Such *rvm*$_j$ is consider as legal.

Opacity: It is a *correctness-criteria* for STMs [2]. A 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 Graph Characterisation for MV-OSTM

Graph characterisation of histories is one of best technique to prove the correctness of STMs. So to prove the correctness of *MV-OSTM*, we are taking the help of graph characterization proposed by Kumar et al [10] for proving opacity which is coming from graph characterization by Bernstein et al [1]. We describe graph characterisation for a history H with a given version order $\ll$. Then we define the opacity graph (or $OPG(H, \ll)) = (V, E)$ as follows: each transaction of complete history $\overline{H}$ is considered as a vertex and edges are of three types:

1. *rt*(real-time) edges: If commit of T_i happens before beginning of T_j in H, then there exist a real-time edge from v_i to v_j. We denote set of such edges as $rt(H)$.
2. *rvf*(return value-from) edges: If T_j invokes rv_method on key k_1 from T_i which has already been committed in H, then there exist a return value-from edge from v_i to v_j. If T_i is having upd_method as insert on the same key k_1 then $i_i(k_{1,i}, v_{i1}) <_H c_i <_H$ *rvm*$_j(k_{1,i}, v_{i1})$. If T_i is having upd_method as delete on the same key k_1 then $d_i(k_{1,i}, nil_{i1}) <_H c_i <_H$ *rvm*$_j(k_{1,i}, nil_{i1})$. We denote set of such edges as $rvf(H)$.
3. *mv*(multi-version) edges: This is based on version order. Consider a triplet with successful methods as *up*$_i(k_{1,i}, u)$ *rvm*$_j(k_1, u)$ *up*$_k(k_{1,k}, v)$, where $u \neq v$. As we can observe it from *rvm*$_j(k_{1,i}, u)$, $c_i <_H$ *rvm*$_j(k_{1,i}, u)$. If $k_{1,i} \ll k_{1,k}$ then there exist a multi-version edge from v_j to v_k. Otherwise ($k_{1,k} \ll k_{1,i}$), there exist a multi-version edge from v_k to v_i. We denote set of such edges as $mv(H, \ll)$.

Consider the history $H4 : l_1(ht, k_{x,0}, NULL) l_2(ht, k_{x,0}, NULL) l_1(ht, k_{y,0}, NULL) l_3(ht, k_{z,0}, NULL) i_1(ht, k_{x,1}, v_{11}) i_3(ht, k_{y,3}, v_{31}) i_2(ht, k_{y,2}, v_{21}) i_1(ht, k_{z,1}, v_{12}) c_1 c_2 l_4(ht, k_{x,1}, v_{11}) l_4(ht, k_{y,2}, v_{21}) i_3(ht, k_{z,3}, v_{32}) c_3 l_4(ht, k_{z,1}, v_{12}) l_5(ht, k_{x,1}, v_{11}), l_6(ht, k_{y,2}, v_{21})$

c_4, c_5, c_6. Using the notation that a committed transaction T_i writing to k_x creates a version $k_{x,i}$, a possible version order for $H4 \ll_{H4}$ is: $\langle k_{x,0} \ll k_{x,1} \rangle, \langle k_{y,0} \ll k_{y,2} \ll k_{y,3} \rangle, \langle k_{z,0} \ll k_{z,1} \ll k_{z,3} \rangle$ shown in Figure 3.

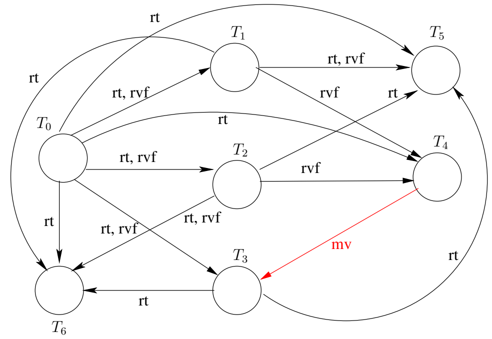


Figure 3: $OPG(H4, \ll_{H4})$

4 MV-OSTM Design and Data Structure

MV-OSTM is a new *STM* that explore the idea of using multiple versions in *OSTM*s to achieve greater concurrency. The idea of *MV-OSTM* has come from multi-version *RWSTMs*. But *RWSTMs* works on a lower level which is prone to more number of aborts. So, we developed *MV-OSTM* in the context of *OSTM* which works on a higher level. *MV-OSTM* stores multiple versions (say, k versions) corresponding to each key that reduces the number of aborts and provides greater concurrency. The value of k can be vary from 1 to ∞. Proposed system gives privilege to the programmer to accept the value of k as any integer. If k is 1, it boils down to *OSTM*. Currently, we have developed *MV-OSTM* with ∞ versions. So, we are performing garbage collection method to delete unwanted version.

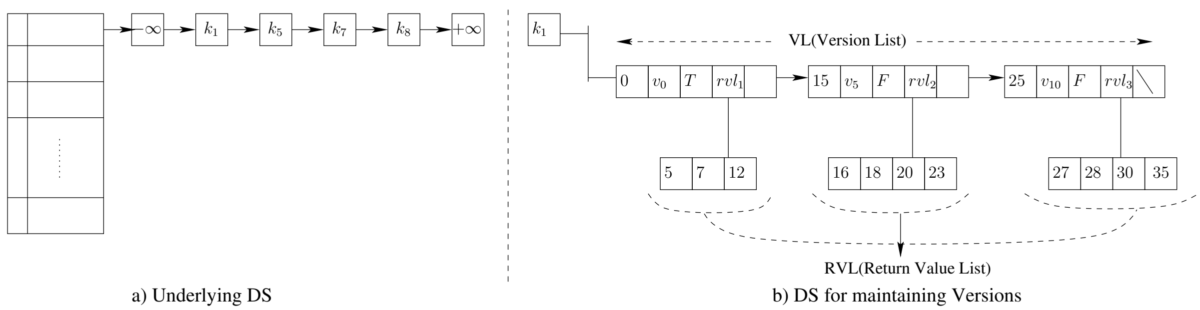


Figure 4: *MV-OSTM* design

We are considering the chaining `hash table` as a underlying data structure where chaining is done via lazy-list refer Figure 4 a). Each bucket of the `hash table` is having two sentinel nodes: $head$ and $tail$. $Head$ and $tail$ are initialized as -∞ and +∞ respectively. Keys $\langle k_1, k_2, ...k_n \rangle$ are added in increasing order in the list between the sentinel nodes. Each key is maintaining the multiple versions in increasing order of timestamp (Figure 4 b)). For each key k_1 of transaction T_i, we maintain $k_1.vl$ (version list) which is a list consisting of version tuples in the form $\langle ts, val, mark, rvl, vnext \rangle$. Description of each field as: ts stands for timestamp which is unique for each transaction, val is the value written by any transaction corresponding to the key, $mark$ is the boolean variable which can be true or false (if the method corresponding to the key is *STM delete()* then the value of $mark$ field will be true, (T) and if the method corresponding to the key is *STM insert()* then the value of $mark$ field will be false, (F)), rvl represents $return\text{-}value\ list$ which is having all the transactions who has performed rv_method on the same key k_1 and $vnext$ is having the information about next available version of the same key k_1. The *MV-OSTM* system consists of the following main methods: *STM init()*, *STM begin()*, *STM insert()*, *STM lookup()*, *STM delete()* and *STM tryC()*.

1. **STM *init*():** This method invokes at the start of the STM system. Initialize the global counter as 1.
2. **STM *begin()*:** It invoked by a thread to being a new transaction T_i. It creates transaction local log and allocate unique id.

3. **STM *insert()*:** Optimistically, actual insertion will happen in the *STM tryC()*. First, it will identify the node corresponding to the key in local log. If the node exists then it just update the local log with useful information like value, operation name and operation status for later use in *STM tryC()*. Otherwise, it will create a local log and update it.

4. **STM *lookup()*:** If *STM lookup()* is not the first method on a particular key means if its a subsequent method of the same transaction on that key then it will search into the local log and return the value and operation status based on the previous operation.

   If *STM lookup()* is the first method on that key then it will identify the location of the node corresponding to the key in the underlying data structure (DS) with the help of *list_lookup()*. If node corresponding to the key is not present in the underlying DS then it will create the node and insert the version of T_0 and add itself into $T_0.rvl$.

   Why do we need to create a version of transaction T_0 by rv_method? This will be clear by the Figure 5, where we have two concurrent transactions T_1 and T_2. History in the Figure 5.a) is not opaque because we cannot come up with any serial order. To make it serial (or opaque) first method $l_2(ht, k_{3,0}, NULL)$ of transaction T_2 have to create the version of T_0 if its not present in the underlying DS and add itself into $T_0.rvl$ (refer Figure 5.c). So in future if any lower timestamp transaction less than T_2 will come then that lower transaction will ABORT (in this case transaction T_1 is aborting in (Figure 5.b))) because higher timestamp already present in the rvl (Figure 5.c)) of the same version. After aborting T_1 we will get the serial history.

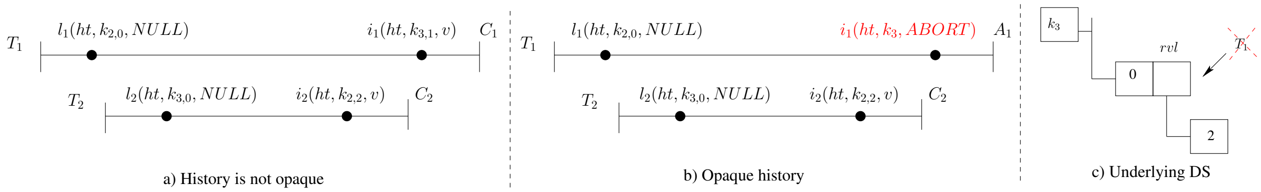


Figure 5: Need of inserting version of T_0 by rv_method to satisfy opacity

5. **STM *delete()*:** *STM delete()* will work same as a *STM lookup()*. The actual deletion will be happen in the *STM tryC()*.

6. **STM *tryC()*:** The actual effect of upd_method (*STM insert()* and *STM delete()*) will take place in *STM tryC()*. It will identify the $pred$ and $curr$ of each upd_method and validate it. If there exist any higher timestamp transaction in the rvl of the $closest\_tuple$ (version with largest timestamp less then itself) of $curr$ then return ABORT.

   On successful validation of all the upd_methods, the actually effect will be taken place. If the upd_method is insert and node corresponding to the key is part of underlying DS then it creates the new version tuple and add it in increasing order of version list. Otherwise it will create the node with the help of *list_Ins()* and insert the version tuple. If the upd_method is delete and node corresponding to the key is part of underlying DS then it creates the new version tuple and set its mark field as TRUE and add it in increasing order of version list. Otherwise it will create the node with the help of *list_Ins()* and insert the version tuple with mark field TRUE. After successful completion of each upd_method it will release all the locks in the same order of lock acquisition.

Theorem 1 *Any valid history H generated by MV-OSTM algorithm with a given version order $\ll$, if $OPG(H, \ll)$ is acyclic, then H is opaque.*

5 Conclusion and Future Work

STMs is an alternative to provide synchronization and communication among multiple threads without worrying about consistency issues. We have proposed a new STM as *MV-OSTM* which providing the greater concurrency in terms of number of abort with the help of multiple version and composability. It ensures the progress condition as *mv-permissiveness* which will never abort a transaction which is having a return-value method only. In that sense *return-value* method will never return abort because of ∞ versions. To overcome the traversal overhead, we have developed the *garbage collection* method to delete unwanted versions from *MV-OSTM*. It satisfies correctness-criteria as *opacity*.

Further, we want to optimize *MV-OSTM* with limited (say k) number of versions corresponding to each key. Later on, we will implement our proposed *MV-OSTM* with the unlimited and limited number of version and compare its performance.

References

[1] Philip A. Bernstein and Nathan Goodman. Multiversion Concurrency Control: Theory and Algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, December 1983. URL: `http://doi.acm.org/10.1145/319996.319998`, `doi:10.1145/319996.319998`.

[2] Rachid Guerraoui and Michal Kapalka. On the Correctness of Transactional Memory. In *PPoPP*, pages 175–184. ACM, 2008.

[3] Tim Harris et al. Abstract nested transactions. 2007.

[4] Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. On developing optimistic transactional lazy set. In *OPODIS*, pages 437–452. Springer, 2014.

[5] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. *Parallel Processing Letters*, 17(4):411–424, 2007.

[6] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier Science, 2012.

[7] Maurice Herlihy and J. Eliot B.Moss. Transactional memory: Architectural Support for Lock-Free Data Structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[8] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP*, pages 207–216. ACM, 2008.

[9] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[10] Priyanka Kumar, Sathya Peri, and K. Vidyasankar. A TimeStamp Based Multi-version STM Algorithm. In *ICDCN*, pages 212–226, 2014.

[11] Petr Kuznetsov and Sathya Peri. On non-interference of transactions. *CoRR*, abs/1211.6315, 2012.

[12] Petr Kuznetsov and Sathya Peri. Non-interference and local correctness in transactional memory. *Theor. Comput. Sci.*, 688:103–116, 2017.

[13] Petr Kuznetsov and Srivatsan Ravi. On the cost of concurrency in transactional memory. In *OPODIS*, pages 112–127, 2011.

[14] Kfir Lev-Ari, Gregory V. Chockler, and Idit Keidar. On correctness of data structures under reads-write concurrency. In Fabian Kuhn, editor, *DISC*, pages 273–287. Springer, 2014.

[15] Kfir Lev-Ari, Gregory V Chockler, and Idit Keidar. A Constructive Approach for Proving Data Structures' Linearizability. In Yoram Moses, editor, *DISC*, pages 356–370. Springer, 2015.

[16] Li Lu and Michael L. Scott. Generic multiversion STM. In *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, pages 134–148, 2013. URL: `http://dx.doi.org/10.1007/978-3-642-41527-2_10`, `doi:10.1007/978-3-642-41527-2_10`.

[17] Yang Ni, Vijay S Menon, Ali-Reza Adl-Tabatabai, Antony L Hosking, Richard L Hudson, J Eliot B Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPoPP*, pages 68–78. ACM, 2007.

[18] Dmitri Perelman, Rui Fan, and Idit Keidar. On Maintaining Multiple Versions in STM. In *PODC*, pages 16–25, 2010.

[19] Nir Shavit and Dan Touitou. Software Transactional Memory. In *PODC*, pages 204–213, 1995.

[20] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.