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,
(cs17mtech11014, cs15resch01004, sathya_p, cs15resch01001)@iith.ac.in

[2] Department of Computer Science, Michigan State University
sandeep@cse.msu.edu

Introduction on STM: To utilize the cores of multicore processors, synchronization and communication among them 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 such as locking, races and deadlocks etc. Concurrently executing transactions access shared memory through the interface provided by the *STMs*.

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 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* (or r) which reads from a buffer, *t_write* (or w) 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*.

On the other hand, ***Object-based STMs*** *or OSTMs* operate on higher level objects rather than read & write operations on memory locations. It was shown in databases that object-based schedulers provide greater concurrency than read-write based systems. So, Herlihy et al. [1], Harris et al. [2], and [3, Chap 6] extended this concept to STMs. We have consider an *OSTM*[c] using `hash table` [4]. It exports the following methods: (1) *t_begin* which begins a transaction, (2) *t_insert* (or i) which inserts a value for a given key, (3) *t_delete* (or d) which deletes the value associated with the given key and returns the current value of the key, (4) *t_lookup* (or l) which looks up the value associated with the given key and (5) *tryC* which validates the operations of the transaction.

Figure 1 a) represents a `hash table` which contains nodes with keys $\langle k_2\ k_5\ k_7\ k_8 \rangle$ between two sentinel nodes -∞ and + ∞. We denote the *RWSTMs* and *OSTM*s operations as layer-0 (or leaves) and layer-1 respectively in the form of transactional forest shown

* The initial work was accepted in AADDA workshop, $ICDCN'18$ as **work in progress.**

** Author sequence follows the lexical order of last names.

[c] *OSTM* has been accepted as a full paper in Netys-18.

in Figure 1 b). Suppose transactions T_1 and T_2 are concurrently executing. Consider the history at layer-0 (while ignoring higher-level operations), denoted as $H0$. It is not opaque [5] because between the two reads of k_5 by T_1, T_2 writes to k_5. In order to ensure opacity for $H0$, one of the transactions among T_1 or T_2 would be aborted.

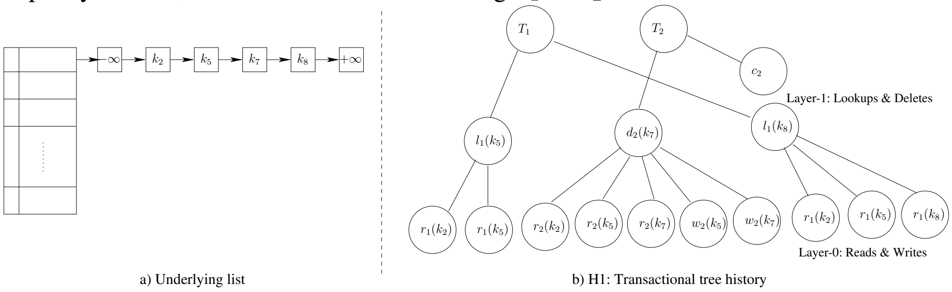

Fig. 1: Motivational example of OSTMs Vs RWSTMs

On the other hand, consider the history $H1$ at layer-1 consisting of l and d operations (while ignoring the underlying read/write operations), since they do not overlap (referred to as pruning in [3, Chap 6]). These methods operate on different keys (k_5, k_7 and k_8), so they are not conflicting and can be re-ordered either way. Thus, we get that $H1$ is opaque [5] with T_1T_2 (or T_2T_1) being an equivalent serial history. Hence, *OSTM* reduces number of aborts and provides greater concurrency.

Objective of *MV-OSTM* : It was observed in databases and STMs that storing multiple versions in *RWSTMs* provides better concurrency [6]. Maintaining multiple versions can ensure that more read operations succeed because the reading operation will have an appropriate version to read. So, we motivated to develop a multi-version *OSTM* as *MV-OSTM*. It exports following methods: (1) *t_begin* which begins a transaction, (2) *t_insert* (or i) which inserts a version for a given key, (3) *t_delete* (or d) which deletes a version associated with the given key and returns the value of the current version, (4) *t_lookup* (or l) which looks up an appropriate version associated with the given key and (5) *tryC* which validates the operations of the transaction.

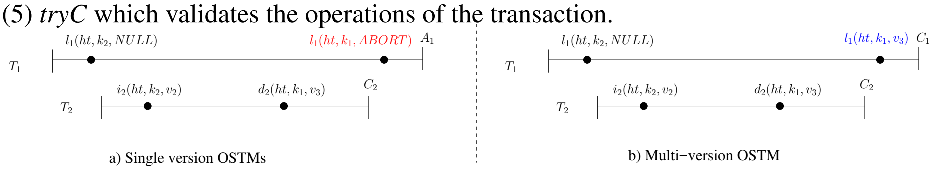

Fig. 2: Advantages of multi-version over single-version *OSTM*s

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 performs a *t_lookup* on key k_2. But due to absence of key k_2 in 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 ht. Then T_2 deletes the key k_1 from 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 STMs has to return 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 serial and hence not 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, consider the example shown in Figure 2 b), 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, it will return v_3 (as previous value) and the history is opaque with the equivalent serial history being T_1T_2.

Thus *MV-OSTM*s[d] reduce number of aborts and achieve greater concurrency than *OSTM*s while ensuring the compositionality. 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. Currently, we have developed *MV-OSTM* with the ∞ number of versions for each key. So, we worked on garbage collection method to delete the unwanted versions of a key (omitted for lack of space). Our **contributions** are as follows: a) 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. b) *MV-OSTM* will never aborts a lookup only transaction because of ∞ versions. c) We have developed the *garbage collection* method to delete unwanted versions from *MV-OSTM*. d) *MV-OSTM* satisfies *opacity*.

Theorem 1 *Any history H generated by MV-OSTM is opaque iff it produces acyclic graph of H.*

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 provides the greater concurrency in terms of number of abort with the help of multiple version and composability. 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. Herlihy, M., Koskinen, E.: Transactional boosting: a methodology for highly-concurrent transactional objects. In: PPoPP, ACM (2008) 207–216
2. Harris, T., et al.: Abstract nested transactions. (2007)
3. Weikum, G., Vossen, G.: Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann (2002)
4. Peri, S., Singh, A., Somani, A.: Efficient means of achieving composability using transactional memory. CoRR **abs/1709.00681** (2017)
5. Guerraoui, R., Kapalka, M.: On the Correctness of Transactional Memory. In: PPoPP, ACM (2008) 175–184
6. Kumar, P., Peri, S., Vidyasankar, K.: A TimeStamp Based Multi-version STM Algorithm. In: ICDCN. (2014) 212–226

[d] Find technical report link of this paper at http://arxiv.org/abs/1709.00681.