An Efficient Approach to Achieve Compositionality using Optimized Multi-Version Object Based Transactional Systems [1]

Chirag Juyal[1], Sandeep Kulkarni[2], Sweta Kumari[3], Sathya Peri[1] and Archit Somani[32]

Department of Computer Science & Engineering, IIT Hyderabad, India[1]
`cs17mtech11014@iith.ac.in, sathya_p@cse.iith.ac.in`
Department of Computer Science, Michigan State University, MI, USA[2]
`sandeep@cse.msu.edu`
Department of Computer Science, Technion, Israel[3]
`(sweta, archit)@cs.technion.ac.il`

Abstract

In the modern era of multi-core systems, the main aim is to utilize the cores properly. This utilization can be done by concurrent programming. But developing a flawless and well-organized concurrent program is difficult. Software Transactional Memory Systems (STMs) are a convenient programming interface which assist the programmer to access the shared memory concurrently without worrying about consistency issues such as priority-inversion, deadlock, livelock, etc. Another important feature that STMs facilitate is compositionality of concurrent programs with great ease. It composes different concurrent operations in a single atomic unit by encapsulating them in a transaction.

Many STMs available in the literature execute read/write primitive operations on memory buffers. We represent them as *Read-Write STMs* or *RWSTMs*. Whereas, there exist some STMs (transactional boosting and its variants) which work on higher level operations such as insert, delete, lookup, etc. on a hash-table. We refer these STMs as Object Based STMs or OSTMs.

[1] A preliminary version of this paper appeared in 20th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2018) and awarded with the **Best Student Paper Award**. A poster version of this work received **Best Poster Award** in NETYS 2018.

[2] 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.

The literature of databases and RWSTMs say that maintaining multiple versions ensures greater concurrency. This motivates us to maintain multiple versions at higher level with object semantics and achieves greater concurrency. So, this paper proposes the notion of *Optimized Multi-version Object Based STMs* or *OPT-MVOSTMs* which encapsulates the idea of multiple versions in OSTMs to harness the greater concurrency efficiently. For efficient memory utilization, we develop two variants of *OPT-MVOSTMs*. First, *OPT-MVOSTM* with garbage collection (or *OPT-MVOSTM-GC*) which uses unbounded versions but performs garbage collection scheme to delete the unwanted versions. Second, finite version *OPT-MVOSTM* (or *OPT-KOSTM*) which maintains at most K versions by replacing the oldest version when $(K+1)^{th}$ version is created by the current transaction.

We propose the *OPT-MVOSTMs* for hash-table and list objects as *OPT-HT-MVOSTM* and *OPT-list-MVOSTM*, respectively. For memory utilization, we propose two variants of both the algorithms as *OPT-HT-MVOSTM-GC*, *OPT-HT-KOSTM* and *OPT-list-MVOSTM-GC*, *OPT-list-KOSTM*, respectively. *OPT-HT-KOSTM* performs best among its variants and outperforms state-of-the-art hash-table based STMs (HT-OSTM, ESTM, RWSTM, HT-MVTO, HT-MVTO-GC, HT-KSTM) by a factor of 3.62, 3.95, 3.44, 2.75, 2.11, 1.85 for workload W1 (90% lookup, 5% insert and 5% delete), 1.44, 2.36, 4.45, 9.84, 8.91, 7.42 for workload W2 (50% lookup, 25% insert and 25% delete), and 2.11, 4.05, 7.84, 12.94, 11.45, 10.70 for workload W3 (10% lookup, 45% insert and 45% delete), respectively. Similarly, *OPT-list-KOSTM* performs best among its variants and outperforms state-of-the-art list based STMs (list-OSTM, Trans-list, Boosting-list, NOrec-list, list-MVTO, list-MVTO-GC, list-KSTM) by a factor of 2.56, 25.38, 23.57, 27.44, 13.34, 8.65, 5.99 for W1, 1.51, 20.54, 24.27, 29.45, 24.89, 22.34, 19.78 for W2, and 2.91, 32.88, 28.45, 40.89, 173.92, 156.67, 124.89 for W3, respectively. We rigorously proved that *OPT-MVOSTM*s satisfy opacity and ensure that transaction with lookup only methods will never return abort while maintaining unbounded versions.
Keywords: Object based Software Transactional Memory Systems, Optimized, Lazyrb-list, Hash-Table, List, Keys, Multi-version, Compositionality, Opacity

---

1. Introduction

Nowadays, multi-core systems are in trend which necessitated the need for concurrent programming to exploit the cores appropriately. However, developing the correct and efficient concurrent programs is difficult. Software Transactional Memory Systems (STMs) are a convenient programming interface which assist the programmer to access the shared memory concurrently using multiple threads without worrying about issues such as deadlock, livelock, priority-inversion, etc. STMs facilitate an additional feature - compositionality of concurrent programs with great ease which makes it more attractive to use. Different concurrent operations that need to be composed to form a single atomic unit is achieved by encapsulating them in a transaction. In this paper, we explain various types of STMs such as read-write STMs (or RWSTMs), object based STMs (or OSTMs) present in the literature along with the benefits of OSTMs over RWSTMs. After that we explain the advantages of multi-version OSTMs (or MVOSTMs) over single-version OSTMs. *MVOSTM*s maintain multiple versions for each *transaction object* which aid in improving the concurrency further, i.e., they allow more transactions to commit [1].

Read-Write STMs: There are several popular STMs in the literature such as ESTM [2], NOrec [3] which execute read/write operations on *transaction objects* or *t-objects*. We represent these STMs as *Read-Write STMs* or *RWSTMs*. RWSTMs typically export following methods: (1) *t_begin*: which begins a transaction with a unique identity, (2) *t_read* (or r): which reads the value of t-object from shared memory, (3) *t_write* (or w): which writes the new value to t-object in its local memory, (4) *tryC*: which validates the values written to t-objects by the transaction and tries to commit. If all the updates made by the transaction is consistent then updates reflect to the shared memory and transaction returns commit, and (5) *tryA*: which returns abort on any inconsistency.

Object based STMs: There are some STMs proposed in the literature which execute higher level operations such as push/pop on stacks, insert/delete/lookup on hash-table etc. We represent these STMs as *Object based STMs* or *OSTMs*. The concept of Boosting by Herlihy et al. [4], the optimistic variant by Hassan et al. [5] and recently *HT-OSTM* system by Peri et al. [6] are some examples that demonstrate the performance benefits achieved by *OSTM*s. Peri et al. [6] showed that *OSTM*s provide greater concurrency

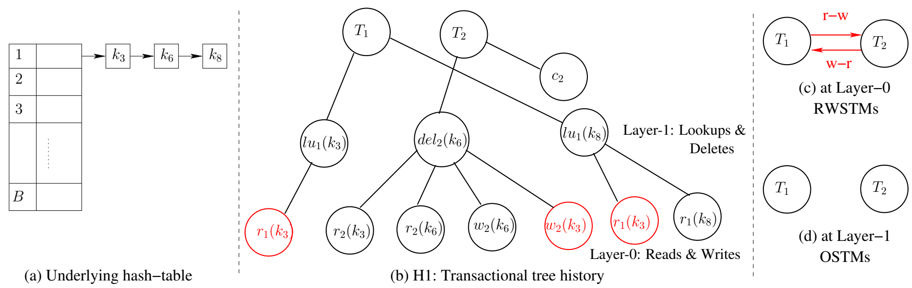


Figure 1: Advantages of OSTMs over RWSTMs

and allow more transactions to commit than RWSTMs while reducing the number of aborts.

Benefits of *OSTM*s over *RWSTM*s: To illustrate the benefits of *OSTM*s, we consider a hash-table based STM system which exports insert (or ins), lookup (or lu) and delete (or del) methods. 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 $\langle k_3,\ k_6,\ k_8\rangle$. 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_3 and k_8 while T_2 performs a delete on k_6. The delete on key k_6 generates read on the keys k_3, k_6 and writes the keys k_6, k_3 assuming that delete is performed similar to delete operation in lazy-list [7]. The lookup on k_3 generates read on k_3 while the lookup on k_8 generates read on k_3, k_8. Note that in this execution k_6 has already been deleted by the time lookup on k_8 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* [8]. This is because, between the two reads of k_3 by T_1, T_2 writes to k_3. It can be seen that if history $H0$ is input to an *RWSTM*s one of the transactions between T_1 or T_2 would be aborted to ensure opacity [8]. 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 [9,

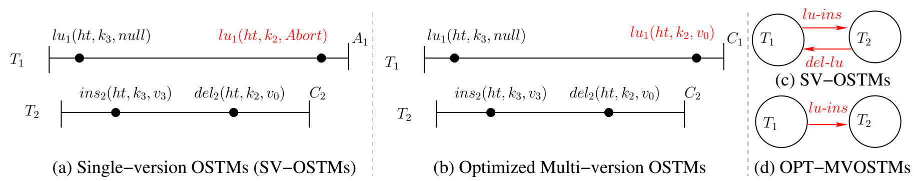


Figure 2: Advantages of optimized multi-version over single-version *OSTM*

Chap 6]). These methods work on distinct keys (k_3, k_6, and k_8). They do not overlap and are not conflicting. So, they can be re-ordered in either way. Thus, $H1$ is opaque [8] 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 abort any of T_1 or T_2. This shows that *OSTM*s can reduce the number of aborts and provide greater concurrency. As explained earlier, this was exploited by some *OSTM*s proposed such as Boosting by Herlihy et al. [4], the optimistic variant by Hassan et al. [5] and HT-OSTM system by Peri et al. [6].

Multi-Version Object Based STMs: It was observed in databases and *RWSTM*s that by storing multiple versions for each t-object, greater concurrency can be obtained which in turn allows more transactions to commit [10]. Specifically, maintaining multiple versions can ensure that more read operations succeed because the reading operation will obtain an appropriate version to read.

Having observed the benefits provided by *OSTM*s over *RWSTM*s, in this paper we combine the notion of multiple versions for each t-object with *OSTM*s to harness greater concurrency. In this paper, we propose and analyze *Optimized Multi-version Object Based STMs* or *OPT-MVOSTMs* while rigorously proving its correctness. Our goal is to analyze the benefit of *OPT-MVOSTMs* over both single-version *OSTM*s and multi-version *RWSTM*s.

The potential benefit of *OPT-MVOSTMs* over *OSTM*s and multi-version *RWSTM*s: We now illustrate the advantage of *OPT-MVOSTMs* as compared to single-version *OSTM*s (*SV-OSTM*s) using the hash-table object with B buckets 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_3. But due to the absence of key k_3 in ht, it obtains a value of

$null$. Then T_2 invokes ins method on the same key k_3 and inserts the value v_3 in ht. Then T_2 deletes the key k_2 from ht and returns v_0 implying that some other transaction had previously inserted v_0 into k_2. The second method of T_1 is lu on the key k_2. 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 $null$ for k_2, 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_2.

In an *OPT-MVOSTMs* based on hash-table, denoted as *OPT-HT-MVOSTM*, whenever a transaction inserts or deletes a key k, a new version is created. Consider the above example with an *OPT-HT-MVOSTM*, as shown in Figure 2 (b). Even after T_2 deletes k_2, the previous value of v_0 is still retained. Thus, when T_1 invokes lu on k_2 after the delete on k_2 by T_2, *OPT-HT-MVOSTM* returns 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, *OPT-MVOSTM* reduces the number of aborts and achieve greater concurrency than *SV-OSTM*s while ensuring the compositionality. We believe that the benefit of *OPT-MVOSTM* over multi-version *RWSTM* is similar to *SV-OSTM* over single-version *RWSTM* as explained above. In this paper, we have considered the hash-table and list based *OPT-MVOSTMs* as *OPT-HT-MVOSTM* and *OPT-list-MVOSTM*, respectively. If the bucket size B of hash-table is set to one then hash-table based *OPT-MVOSTM* boils down to the list based *OPT-MVOSTM*.

OPT-HT-MVOSTM and *OPT-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 *OPT-MVOSTMs* to delete the unwanted versions of a key, denoted as *OPT-MVOSTM-GC*. Garbage collection gave an average performance gain of 16% over *OPT-MVOSTM* without garbage collection in the best case. Thus, the overhead of garbage collection scheme is less than the performance improvement due to improved memory usage. (2) Placing a limit of K on the number versions in *OPT-MVOSTM*, resulting in *OPT-KOSTM*. This gave an average performance gain of 24% over *OPT-MVOSTM* without garbage collection in the best case.

Experimental results show that *OPT-HT-KOSTM* performs best among its variants and outperforms state-of-the-art hash-table based STMs (HT-OSTM, ESTM, RWSTM, HT-MVTO, HT-MVTO-GC, HT-KSTM) by a factor of 3.62, 3.95, 3.44, 2.75, 2.11, 1.85 for workload W1 (90% lookup, 5% insert and 5% delete), 1.44, 2.36, 4.45, 9.84, 8.91, 7.42 for workload W2 (50% lookup, 25% insert and 25% delete), and 2.11, 4.05, 7.84, 12.94, 11.45, 10.70 for workload W3 (10% lookup, 45% insert and 45% delete), respectively. Similarly, *OPT-list-KOSTM* performs best among its variants and outperforms state-of-the-art list based STMs (list-OSTM, Trans-list, Boosting-list, NOrec-list, list-MVTO, list-MVTO-GC, list-KSTM) by a factor of 2.56, 25.38, 23.57, 27.44, 13.34, 8.65, 5.99 for W1, 1.51, 20.54, 24.27, 29.45, 24.89, 22.34, 19.78 for W2, and 2.91, 32.88, 28.45, 40.89, 173.92, 156.67, 124.89 for W3, respectively. 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.

Contributions of the paper:

- We propose a new notion of optimized multi-version objects based STM system as *OPT-MVOSTM* in Section 5. In this paper, we develop it for list and hash-table objects as *OPT-list-MVOSTM* and *OPT-HT-MVOSTM*, respectively.

- For efficient space utilization in *OPT-MVOSTMs* with unbounded versions, we develop *Garbage Collection* for *OPT-MVOSTM* (i.e. *OPT-MVOSTM-GC*) and bounded version *OPT-MVOSTM* (i.e. *OPT-KOSTM*).

- Section 7 shows that *OPT-list-MVOSTM* and *OPT-HT-MVOSTM* satisfy standard correctness-criterion of STMs, *opacity* [8].

- Experimental analysis of both *OPT-list-MVOSTM* and *OPT-HT-MVOSTM* with state-of-the-art STMs are present in Section 8. Proposed *OPT-list-MVOSTM* and *OPT-HT-MVOSTM* provide greater concurrency, allow more transactions to commit, and reduce the number of aborts as compared to *MVOSTMs*, *SV-OSTM*s, single-version *RWSTM*s and, multi-version *RWSTM*s while maintaining multiple versions corresponding to each key.

Roadmap: The paper is organized as follows. We describe related work and system model in Section 2 and Section 3, respectively. In Section 4, we formally define the graph characterization of opacity. Section 5 represents the *OPT-MVOSTMs* design and data structure. Section 6 shows the working of *OPT-HT-MVOSTMs* and its algorithms. We formally prove the correctness of *OPT-MVOSTMs* in Section 7. In Section 8 we show the experimental evaluation of *OPT-MVOSTMs* with state-of-art-STMs. Finally, we conclude in Section 9.

2. Related Work

Several STM systems have been proposed in the literature such as Elastic STM (ESTM) [2], NOrec [3] which executes read/write primitive operations on *transaction objects* or *t-objects*. We represent these STMs as *Read-Write STMs* or *RWSTMs*. ESTM [2] is an appealing alternative to the traditional transactional model which offers better performance than traditional RWSTMs. ESTM is favorable for the search structure like the hash-table in shared memory.

Ownership-record-free (NOrec) [3] is another popular STM which ensures low overhead and high scalability. It acquires a global versioned lock when updating the shared memory. Each transaction maintains a read log and snapshot timestamp taken from the global versioned lock whenever a transaction begins. Write of the transaction is occurring directly into its redo-log with a hashing scheme to save the search time [11].

Some RWSTMs of the literature allow a transaction to read the older values instead of the latest value while maintaining multiple versions corresponding to each t-object [10, 12, 13, 14, 15, 16]. We represent them as *Multi-version RWSTMs*. It has been shown that Multi-version RWSTMs reduce the number of aborts and provide greater concurrency than RWSTMs. Multiple versions give the flexibility to read from the previous version and guarantee that read-only transactions always return commit. But, maintaining multiple versions require more memory.

To overcome the memory requirement [10, 12, 15] perform a *garbage collection*

(GC) mechanism to delete the unwanted versions. Cachopo and Silva [12] proposed versioned boxes to keep the history of values. They maintain the values as long as some active transactions accesses it. Otherwise they commit a new value to the versioned box and delete the previous value. To track the usability of older versions, they maintain live transactions in a priority queue, sort the transactions by their unique ids, and use a cleanup thread to detach the terminated (committed/aborted) transactions from the queue. The cleanup thread waits until the transaction T_i at the head of the queue terminates. It deletes all the versions created before T_i, if T_i is created a version successfully. Otherwise, it deletes all the obsolete versions except the previous closest version of T_i. Similarly, a few multi-version RWSTMs [10, 15] performed the garbage collection.

Selective multi-versioning (SMV) [15] keeps the versions as long as it is useful for some reading transaction and garbage collects the version when none of the transactions read from it. SMV suggested managing the memory through a special GC thread for a periodic interval to dispose of obsolete versions. If a GC thread calls frequently then it interrupts the other methods of the system and if it calls rarely then lots of versions may get accumulated which will increase the search time to read the correct version from the version list. Multi-version timestamp ordering (MVTO) based STM [10] is another popular Multi-Version RWSTMs for hash-table and list as *HT-MVTO* and *list-MVTO*, respectively. To delete the unwanted versions, they applied *garbage collection* on *HT-MVTO* and *list-MVTO* and proposed *HT-MVTO-GC* and *list-MVTO-GC*, respectively. Later, they said that maintaining finite versions may optimize the solution and proposed K version STM (KSTM) for hash-table and list as *HT-KSTM* and *list-KSTM*, respectively.

All the above described STM systems work on read-write primitive operations and synchronized via read/write conflicts, which may cause benign conflicts [11]. A few STMs available in the literature which executes higher level operations such as insert, delete, lookup on hash-table. We represent these STMs as *Object based STMs* or *OSTMs*. Transactional Boosting [4] proposed a methodology to synchronize using object semantics. It converts a large class of highly-concurrent linearizable objects into highly-concurrent transactional objects. Transactional boosting is a pessimistic STM that works on higher level operations rather than read-write primitives and uses abstract locks. It provides abstract data type level conflicts checking rather than memory-level conflicts checking which prevent spurious conflicts and ensure greater concurrency.

Due to the pessimistic behavior, transactional boosting needs to perform the inverse operations when an inconsistency occurs. This possibly can make it difficult to be adapted by all concurrent data structures. Optimistic Transactional Boosting (OTB) [5] is an optimistic methodology to convert a large class of highly-concurrent linearizable objects into highly-concurrent transactional objects. It composes multiple operations into one atomic execution using a transaction optimistically so avoids the need for inverse operations unlike transactional boosting. It enables an easy way to integrate with the STM framework and implements highly concurrent and optimized data structures. It packages the benefits of concurrent data structures, transactional boosting, and transactional memory systems altogether. Transactional list (Trans-list) [17] proposed a new optimistic methodology for transforming high-performance lock-free list to high-performance lock-free transactional list without revamping the list design.

Recently, an optimistic OSTM system [6] has been proposed by Peri et al. for two data structures hash-table and list as *HT-OSTM* and *list-OSTM*, respectively. It provides greater concurrency than other OSTM systems [4, 17] and *RWSTMs* while reducing the number of aborts. Later, this work gets motivated by storing multiple versions corresponding to each t-object in databases and *RWSTM*s and proposed *MVOSTM* [1] which achieves greater concurrency further. To delete the unwanted versions and utilize the memory *MVOSTM* performs *garbage collect (GC)* and proposed *MVOSTM-GC*. To utilize the memory further, *MVOSTM* maintains finite say K versions corresponding to each t-object and proposed *KOSTM* which performs best among its variants. They proposed all variants for hash-table and list as *HT-MVOSTM*, *HT-MVOSTM-GC*, *HT-KOSTM* and *list-MVOSTM*, *list-MVOSTM-GC*, *list-KOSTM*, respectively [1].

3. System Model

Our assumption follows [6, 18] in which the system consists of a finite set of p processes, $p_1, \ldots, p_n$, accessed by a finite number of n threads in a completely asynchronous fashion and communicates each other using shared keys (or objects). The threads invoke higher level methods on the shared objects and get 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 $\mathscr{V}$. We assume that number of keys are finite. 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 use *read-modify-write()* primitives such as *getAndIncrement()* in *t_begin()* to assign the unique id to each transaction and *CompareAndSwap()* in lock implementation. We assume that all the locks are fair.

We denote *t_insert* and *t_delete* as *update* methods (or upd_method or up) since both of these change the underlying data structure. We denote *t_delete* and *t_lookup* as *return-value methods (or rv_method or rvm)* 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.

Formally, we denote a method m by the tuple $\langle evts(m), <_m \rangle$. Here, $evts(m)$ are all the events invoked by m and the $<_m$ a total order among these events.

Transactions: Following the notations used in database multi-level transactions[9], 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 $methods(T_i)$. We assume that for any method m in $methods(T_i)$, $evts(m)$ is a subset of $evts(T_i)$ and $<_m$ is a subset of $<_{T_i}$. We assume that if a transaction has invoked a method, then it does not invoke a new method until it gets the response of the previous one. Thus all the methods of a transaction can be ordered by $<_{T_i}$. Formally, $\langle \forall m_p, m_q \in methods(T_i) : (m_p <_{T_i} m_q) \vee (m_q <_{T_i} m_p)) \rangle$, here m_p and m_q are p_{th} and q_{th} methods of T_i, respectively.

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 and denoted as $live(H) = txns(H) - committed(H) - aborted(H)$. On the other hand, the set of *terminated* transactions are those which have either committed or aborted and is denoted by $term(H) = committed(H) \cup aborted(H)$.

The relation between the events of transactions & histories is analogous to the relation between methods & transactions. We assume that for any transaction T in $txns(H)$, $evts(T)$ is a subset of $evts(H)$ and $<_T$ is a subset of $<_H$. Formally, $\langle \forall T \in txns(H) : (evts(T) \subseteq evts(H)) \wedge (<_T \subseteq <_H) \rangle$.

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 *OPT-MVOSTM*.

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 [19, 20]) 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. We assume all the methods of the transactions are *complete* in a history.

Consider a sequential history H. Let $m_{ij}(ht, k, v/nil)$ be the first method of T_i in H operating on the key k as $H.firstKeyMth(\langle ht, k\rangle, T_i)$, where m_{ij} stands for j^{th} method of i^{th} transaction. For a method $m_{ix}(ht, k, v)$ which is not the first method on $\langle ht, k\rangle$ of T_i in H, we denote its previous method on k of T_i as $m_{ij}(ht, k, v) = H.prevKeyMth(m_{ix}, T_i)$.

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* [21] or *t-sequential* [20] 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.

Valid Histories: A rv_method (*t_delete* and *t_lookup*) rvm_{ij} on key k is valid if it returns the value updated by any of the previously committed transaction that updated key k. A history H is said to valid if all the rv_methods of H are valid.

Legal Histories: We define the *legality* of rv_methods on sequential histories which we use to define correctness criterion as opacity [8]. Consider a sequential history H having a rv_method *rvm*$_{ij}(ht, k, v)$ (with $v \neq null$) as j^{th} method belonging to transaction T_i. We define this *rvm* method to be *legal* if:

Rule 1 If the *rvm*$_{ij}$ is not the first method of T_i to operate on $\langle ht, k\rangle$ and m_{ix} is the previous method of T_i on $\langle ht, k\rangle$. Formally, *rvm*$_{ij} \neq H.firstKeyMth(\langle ht, k\rangle, T_i)$ $\wedge(m_{ix}(ht, k, v') = H.prevKeyMth(\langle ht, k\rangle, T_i))$ (where v' could be null). Then,

(a) If $m_{ix}(ht, k, v')$ is a *t_insert* method then $v = v'$.

(b) If $m_{ix}(ht, k, v')$ is a *t_lookup* method then $v = v'$.

(c) If $m_{ix}(ht, k, v')$ is a *t_delete* method then $v = null$.

In this case, we denote m_{ix} as the last update method of *rvm*$_{ij}$, i.e., $m_{ix}(ht, k, v') = H.lastUpdt($*rvm*$_{ij}(ht, k, v))$.

Rule 2 If *rvm*$_{ij}$ is the first method of T_i to operate on $\langle ht, k\rangle$ and v is not null. Formally, *rvm*$_{ij}(ht, k, v) = H.firstKeyMth(\langle ht, k\rangle, T_i) \wedge (v \neq null)$. Then,

(a) There is a *t_insert* method *t_insert*$_{pq}(ht, k, v)$ in $methods(H)$ such that T_p committed before *rvm*$_{ij}$. Formally, $\langle\exists$*t_insert*$_{pq}(ht, k, v) \in methods(H)$: *tryC*$_p \prec_H^{MR}$ *rvm*$_{ij}\rangle$.

(b) There is no other update method up_{xy} of a transaction T_x operating on $\langle ht, k\rangle$ in $methods(H)$ such that T_x committed after T_p but before *rvm*$_{ij}$. Formally, $\langle\nexists up_{xy}(ht, k, v'') \in methods(H)$: *tryC*$_p \prec_H^{MR}$ *tryC*$_x \prec_H^{MR}$ *rvm*$_{ij}\rangle$.

In this case, we denote *tryC*$_p$ as the last update method of *rvm*$_{ij}$, i.e., *tryC*$_p(ht, k, v)$= $H.lastUpdt($*rvm*$_{ij}(ht, k, v))$.

Rule 3 If *rvm*$_{ij}$ is the first method of T_i to operate on $\langle ht, k\rangle$ and v is null. Formally, *rvm*$_{ij}(ht, k, v) = H.firstKeyMth(\langle ht, k\rangle, T_i) \wedge (v = null)$. Then,

(a) There is *t_delete* method *t_delete*$_{pq}(ht, k, v')$ in $methods(H)$ such that T_p committed before *rvm*$_{ij}$. Formally, $\langle \exists$*t_delete*$_{pq}$ $(ht, k, v') \in methods(H)$: *tryC*$_p \prec_H^{MR}$ *rvm*$_{ij}\rangle$. Here v' could be null.

(b) There is no other update method up_{xy} of a transaction T_x operating on $\langle ht, k\rangle$ in $methods(H)$ such that T_x committed after T_p but before *rvm*$_{ij}$. Formally, $\langle \nexists up_{xy}(ht, k, v'') \in methods(H)$: *tryC*$_p \prec_H^{MR}$ *tryC*$_x \prec_H^{MR}$ *rvm*$_{ij}\rangle$.

In this case, we denote *tryC*$_p$ as the last update method of *rvm*$_{ij}$, i.e., *tryC*$_p(ht, k, v)$ $= H.lastUpdt($*rvm*$_{ij}(ht, k, v))$.

We assume that when a transaction T_i operates on key k of a hash-table ht, the result of this method is stored in *local logs* of T_i, $txLog_i$ for later methods to reuse. Thus, only the first rv_method operating on $\langle ht, k\rangle$ of T_i accesses the shared memory. The other rv_methods of T_i operating on $\langle ht, k\rangle$ do not access the shared memory and they see the effect of the previous method from the *local logs*, $txLog_i$. This idea is utilized in Rule 1. With reference to Rule 2 and Rule 3, it is possible that T_x could have aborted before *rvm*$_{ij}$.

Coming to *t_insert* methods, since a *t_insert* method always returns ok as they always create a new version if node already present. Otherwise, *t_insert* creates a node along with version and always takes effect on the ht. Thus, we denote all *t_insert* methods as legal and only give legality definition for rv_method. We denote a sequential history H as *legal* or *linearized* if all its *rvm* methods are legal. We formally prove the legality of the proposed *OPT-MVOSTMs* in Section 7.

Opacity: It is a *correctness-criteria* for STMs [8]. 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., $\prec_H^{TR} \subseteq \prec_S^{TR}$.

Finally, we show that histories generated by *OPT-MVOSTMs* satisfy correctness criteria as opaque.

4. Graph Characterization of Opacity

To prove that an STM system satisfies opacity, it is useful to consider graph characterization of histories. In this section, we describe the graph characterization of Guerraoui and Kapalka [18] modified for sequential histories.

Consider a history H which consists of multiple version for each t-object. The graph characterization uses the notion of *version order*. Given H and a t-object k, we define a version order for k as any (non-reflexive) total order on all the versions of k ever created by committed transactions in H. It must be noted that the version order may or may not be the same as the actual order in which the versions of k are generated in H. A version order of H, denoted as $\ll_H$ is the union of the version orders of all the t-objects in H.

Consider the history $H3$ as shown in Figure 3 : $lu_1(k_{x,0}, null), lu_2(k_{x,0}, null), lu_1(k_{y,0}, null), lu_3(k_{z,0}, null), ins_1(k_{x,1}, v_{11}), ins_3(k_{y,3}, v_{31}), ins_2(k_{y,2}, v_{21}), ins_1(k_{z,1}, v_{12}), c_1, c_2, lu_4(k_{x,1}, v_{11}), lu_4(k_{y,2}, v_{21}), ins_3(k_{z,3}, v_{32}), c_3, lu_4(k_{z,1}, v_{12}), lu_5(k_{x,1}, v_{11}), lu_6(k_{y,2}, v_{21}), c_4, c_5, c_6$. Using the notation that a committed transaction T_i writing to k_x creates a version $k_{x,i}$, a possible version order for $H3$ $\ll_{H3}$ is: $\langle k_{x,0} \ll k_{x,1}\rangle, \langle k_{y,0} \ll k_{y,2} \ll k_{y,3}\rangle, \langle k_{z,0} \ll k_{z,1} \ll k_{z,3}\rangle$.

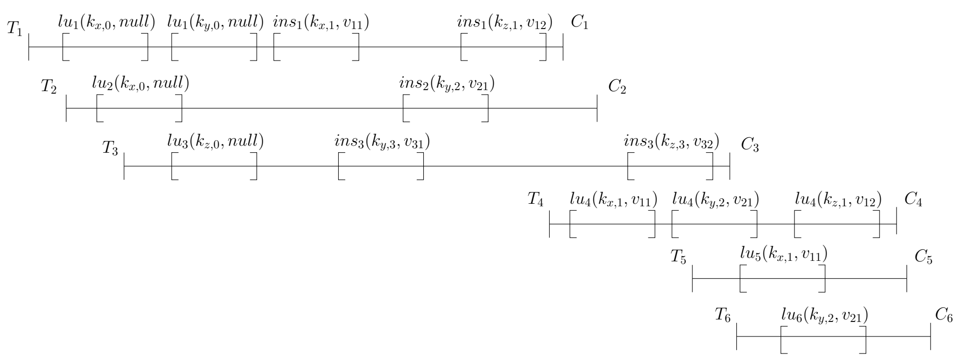


Figure 3: History $H3$ in time line view

We define the graph characterization based on a given version order. Consider a history H and a version order $\ll$. We then define a graph (called opacity graph) on H using $\ll$, denoted as $OPG(H, \ll) = (V, E)$. The vertex set V consists of a vertex for each transaction T_i in $\overline{H}$. The edges of the graph are of three kinds and are defined as follows:

1. real-time (*rt*) edges: If the 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. return value-from (*rvf*) edges: If T_j invokes rv_method on key k_1 from T_i which has already been committed in H, then there exists 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 $ins_i(k_{1,i}, v_{i1}) <_H c_i <_H rvm_j(k_{1,i}, v_{i1})$. If T_i is having upd_method as delete on the same key k_1 then $del_i(k_{1,i}, null) <_H c_i <_H rvm_j(k_{1,i}, null)$. We denote set of such edges as $rvf(H)$.
3. multi-version (*mv*) edges: This is based on version order. Consider a triplet with successful methods as $up_i(k_{1,i}, u)$, $rvm_j(k_{1,i}, u)$, $up_k(k_{1,k}, v)$, 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 $mv(H, \ll)$.

We now show that if a version order $\ll$ exists for a history H such that it is acyclic, then H is opaque.

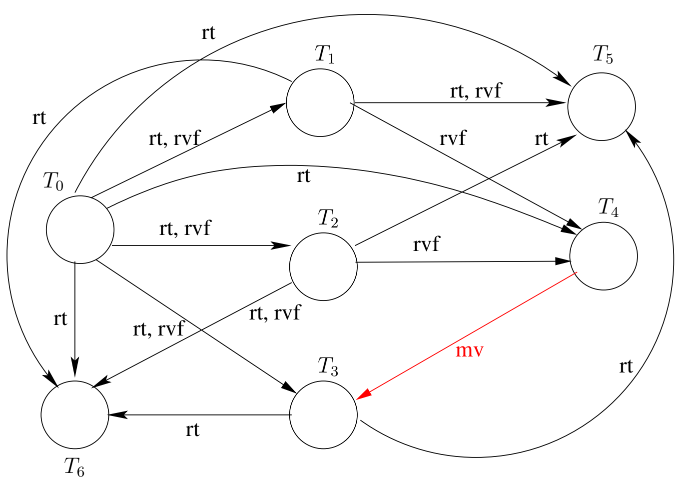


Figure 4: $OPG(H3, \ll_{H3})$

Using this construction, the $OPG(H3, \ll_{H3})$ for history $H3$ and $\ll_{H3}$ is given above is shown in Figure 4. The edges are annotated. The only mv edge from T_4 to T_3 is because of t-objects k_y, k_z. T_4 lookups value v_{12} for k_z from T_1 whereas T_3 also inserts v_{32} to k_z and commits before $lu_4(k_{z,1}, v_{12})$.

Given a history H and a version order $\ll$, consider the graph $OPG(\overline{H}, \ll)$. While considering the rt edges in this graph, we only consider the real-time relation of H and not $\overline{H}$. It can be seen that $\prec_H^{RT} \subseteq \prec_{\overline{H}}^{RT}$ but with this assumption, $rt(H) = rt(\overline{H})$. Hence, we get the following property,

Property 1. *The graphs $OPG(H, \ll)$ and $OPG(\overline{H}, \ll)$ are the same for any history H and $\ll$.*

Definition 1. *For a t-sequential history S, we define a version order $\ll_S$ as follows: For two version $k_{x,i}, k_{x,j}$ created by committed transactions T_i, T_j in S, $\langle k_{x,i} \ll_S k_{x,j} \Leftrightarrow T_i <_S T_j \rangle$.*

Now we show the correctness of our graph characterization using the following lemmas and theorem.

Lemma 2. *Consider a legal t-sequential history S. Then the graph $OPG(S, \ll_S,)$ is acyclic.*

Proof: We numerically order all the transactions in S by their real-time order by using a function *ord*. For two transactions T_i, T_j, we define $ord(T_i) < ord(T_j) \Leftrightarrow T_i <_S T_j$. Let us analyze the edges of $OPG(S, \ll_S,)$ one by one:

- rt edges: It can be seen that all the rt edges go from a lower ord transaction to a higher ord transaction.

- rvf edges: If T_j lookups k_x from T_i in S then T_i is a committed transaction with $ord(T_i) < ord(T_j)$. Thus, all the rvf edges from a lower ord transaction to a higher ord transaction.

- mv edges: Consider a successful rv_method $rvm_j(k_x, u)$ and a committed transaction T_k writing v to k_x where $u \neq v$. Let c_i be $rvm_j(k_x, u)$'s lastWrite. Thus, $up_i(k_{x,i}, u) \in evts(T_i)$. Thus, we have that $ord(T_i) < ord(T_j)$. Now there are two cases w.r.t T_i: (1) Suppose $ord(T_k) < ord(T_i)$. We now have that $T_k \ll T_i$. In this case, the mv edge is from T_k to T_i. (2) Suppose $ord(T_i) < ord(T_k)$ which implies that $T_i \ll T_k$. Since S is legal, we get that $ord(T_j) < ord(T_k)$. This case also implies that there is an edge from $ord(T_j)$ to $ord(T_k)$. Hence, in this case as well the mv edges go from a transaction with lower ord to a transaction with higher ord.

Thus, in all the three cases the edges go from a lower ord transaction to higher ord transaction. This implies that the graph is acyclic. □

Lemma 3. *Consider two histories H, H' that are equivalent to each other. Consider a version order $\ll_H$ on the t-objects created by H. The mv edges $mv(H, \ll_H)$ induced by $\ll_H$ are the same in H and H'.*

Proof: Since the histories are equivalent to each other, the version order $\ll_H$ is applicable to both of them. It can be seen that the mv edges depend only on events of the history and version order $\ll$. It does not depend on the ordering of the events in H. Hence, the mv edges of H and H' are equivalent to each other. □

Using these lemmas, we prove the following theorem.

Theorem 4. *A valid history H is opaque iff there exists a version order $\ll_H$ such that $OPG(H, \ll_H)$ is acyclic.*

Proof: (if part): Here we have a version order $\ll_H$ such that $G_H = OPG(H, \ll)$ is acyclic. Now we have to show that H is opaque. Since the G_H is acyclic, a topological sort can be obtained on all the vertices of G_H. Using the topological sort, we can generate a t-sequential history S. It can be seen that S is equivalent to $\overline{H}$. Since S is obtained by a topological sort on G_H which maintains the real-time edges of H, it can be seen that S respects the rt order of H, i.e $\prec_H^{RT} \subseteq \prec_S^{RT}$.

Similarly, since G_H maintains return value-from (rvf) order of H, it can be seen that if T_j lookups k_x from T_i in H then T_i terminates before $lu_j(k_x)$ and T_j in S. Thus, S is valid. Now it remains to be shown that S is legal. We prove this using contradiction. Assume that S is not legal. Thus, there is a successful rv_method *rvm*$_j(k_x, u)$ such that its lastWrite in S is c_k and T_k updates value $v(\neq u)$ to k_x, i.e *up*$_k(k_{x,k}, v) \in evts(T_k)$. Further, we also have that there is a transaction T_i that inserts u to k_x, i.e *up*$_i(k_{x,i}, u) \in evts(T_i)$. Since S is valid, as shown above, we have that $T_i \prec_S^{RT} T_k \prec_S^{RT} T_j$.

Now in $\ll_H$, if $k_{x,k} \ll_H k_{x,i}$ then there is an edge from T_k to T_i in G_H. Otherwise ($k_{x,i} \ll_H k_{x,k}$), there is an edge from T_j to T_k. Thus, in either case, T_k can not be in between T_i and T_j in S contradicting our assumption. This shows that S is legal.

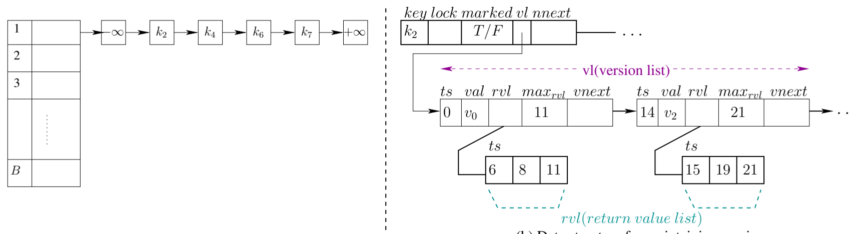
(Only if part): Here we are given that H is opaque and we have to show that there exists a version order $\ll$ such that $G_H = OPG(H, \ll)(= OPG(\overline{H}, \ll)$, Property 1) is acyclic. Since H is opaque there exists a legal t-sequential history S equivalent to $\overline{H}$ such that it respects real-time order of H. Now, we define a version order for S, $\ll_S$ as in Definition 1. Since the S is equivalent to $\overline{H}$, $\ll_S$ is applicable to $\overline{H}$ as well. From Lemma 2, we get that $G_S = OPG(S, \ll_S)$ is acyclic. Now consider $G_H = OPG(\overline{H}, \ll_S)$. The vertices of G_H are the same as G_S. Coming to the edges,

- rt edges: We have that S respects real-time order of H, i.e $\prec_H^{RT} \subseteq \prec_S^{RT}$. Hence, all the rt edges of H are a subset of S.
- rvf edges: Since $\overline{H}$ and S are equivalent, the return value-from relation of $\overline{H}$ and S are the same. Hence, the rvf edges are the same in G_H and G_S.
- mv edges: Since the version-order and the operations of the H and S are the same, from Lemma 3 it can be seen that $\overline{H}$ and S have the same mv edges as well.

Thus, the graph G_H is a subgraph of G_S. Since we already know that G_S is acyclic from Lemma 2, we get that G_H is also acyclic. □

5. *OPT-MVOSTMs* Design and Data Structure

This section describes the design and data structure of optimized *MVOSTMs* (or *OPT-MVOSTMs*). Here, we propose hash-table and list based *OPT-MVOSTMs* as *OPT-HT-MVOSTM* and *OPT-list-MVOSTM*, respectively. *OPT-HT-MVOSTM* is a hash-table



(a) Underlying hash−table

(b) Data structure for maintaining versions

Figure 5: Optimized *HT-MVOSTM* design

based *OPT-MVOSTM* that explores the idea of multiple versions in *OSTM*s for hash-table object to achieve greater concurrency. The design of *OPT-HT-MVOSTM* is similar to *HT-MVOSTM* [1] 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 5 (a), similar to any linked-list based concurrent set implementation [7, 22]. 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 [7]. 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 5 (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* [7]. Given a node n in the linked-list of bucket B with key k, we denote its fields as $n.key$ (or $k.key$), $n.lock$ (or $k.lock$), $n.marked$ (or $k.marked$), $n.vl$ (or $k.vl$), $n.nnext$ (or $k.nnext$).

The structure of each version in the vl of a key k is $\langle ts,\ val,\ rvl,\ max_{rvl},\ vnext\rangle$ as shown in Figure 5 (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 $k.vl[j].ts,\ k.vl[j].val,\ k.vl[j].rvl,\ k.vl[j].max_{rvl},\ k.vl[j].vnext$.

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*$_i(ht, k, v)$ then val will be $null$. In this case, as per the algorithm, the node of key k will also be marked. *OPT-HT-MVOSTM* algorithm does not immediately physically remove deleted keys from the hash-table. The need for this is explained below. Thus an 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 first optimization in *OPT-HT-MVOSTM* to reduce the traversal time of rvl, we have used max_{rvl} which contains the maximum ts of the transaction that executed rv_method on this version. The field $vnext$ points to the next available version of that key.

In order to increase the efficiency and utilize the memory properly, we propose two variants of *OPT-HT-MVOSTM* as follows: First, we apply garbage collection (or GC) on the versions and propose *OPT-HT-MVOSTM-GC*. It maintains unbounded versions in vl (the length of the list) while deleting the unwanted versions using garbage collection scheme. Second, we propose *OPT-HT-KOSTM* which maintains the bounded number of versions such as K and improves the efficiency further. 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.

We propose *OPT-list-MVOSTMs* while considering the bucket size as 1 in *OPT-HT-MVOSTM*. Along with this, we propose two variants of *OPT-list-MVOSTM* as *OPT-list-MVOSTM-GC* and *OPT-list-KOSTM* which applies the garbage collection scheme in unbounded versions and bounded K versions for list based object, respectively, similar to *OPT-HT-MVOSTM*.

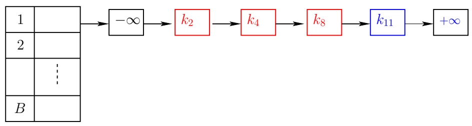


Figure 6: Searching k_{11} over *lazy-list*

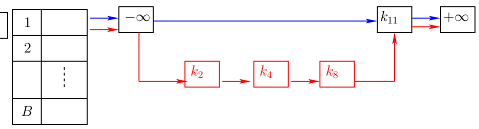


Figure 7: Searching k_{11} over $lazyrb\text{-}list$

Marked Version Nodes: *OPT-HT-MVOSTM* stores keys even after they have been deleted (the version of the 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 $lu_1(ht, k_2, v_0)$ as shown in Figure 2 (b). Due to the presence of previous version v_0, *OPT-HT-MVOSTM* returns this earlier version v_0 for $lu_1(ht, k_2, v_0)$ method. Whereas, it is not possible for *HT-OSTM* to return the version v_0 because k_1 has been removed from the system by delete method of higher timestamp transaction T_2 than T_1. 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 the live keys (or unmarked node) in a lazy-list will increase the traversal time to access unmarked nodes. Consider Figure 6, in which there are four keys $\langle k_2, k_4, k_8, k_{11} \rangle$ present in the list. Here $\langle k_2, k_4, k_8 \rangle$ are marked (or deleted) nodes while k_{11} is unmarked. Now, consider accessing the key k_{11} by *OPT-HT-MVOSTM* as a part of one of its methods. Then *OPT-HT-MVOSTM* would have to unnecessarily traverse the marked nodes to reach key k_{11}.

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 *OSTM*s [6]. $lazyrb\text{-}list$ consists of nodes with two links, red link (or RL) and blue link (or BL). The node which is 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_{11}. It can be seen that k_{11} can be accessed much more quickly through BL as shown in Figure 7. Using the idea of $lazyrb\text{-}list$, we have modified the structure of each node as $\langle$ *key, lock, marked, vl, RL, BL* $\rangle$. Further, for a bucket B, we denote its linked-list as $B.lazyrb\text{-}list$.

6. Working of *OPT-HT-MVOSTM*

OPT-HT-MVOSTM exports *t_begin*, *t_insert*, *t_delete*, *t_lookup*, and *tryC* methods as explained in Section 3. Among them *t_delete*, *t_lookup* are return-value methods (or rv_methods) while *t_insert*, *t_delete* are update methods (or 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 the local log, *txLog*. If all the updates by a transaction are consistent in tryC then it transfers all its updates to the shared memory to be visible to the other transactions in the system. We now explain the working of each method as follows:

***t_begin()*:** A thread invokes a new transaction T_i using this method. The transaction T_i local log $txLog_i$ is initialized at Line 2. This method returns a unique id to the invoking thread by incrementing an atomic counter at Line 3. 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.

Algorithm 1 *t_begin()*: It provides the local log and unique id to each transaction.

```
1: procedure t_begin()
2:     txLog ← new txLog().                  ▷ Initialize the local log of transaction
3:     t_id ← get&inc(counter).   ▷ Get the unique transaction id (t_id) while incrementing the counter atomically
4:     return t_id.
5: end procedure
```

rv_methods: It can be either *t_delete*(ht, k, v) or *t_lookup*(ht, k, v). Both these methods return the current value of key k. Algorithm 2 gives the high level overview of these methods. First, the algorithm checks to see if the given key is already in the local log, $txLog_i$ of T_i (Line 7). 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 *OPT-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. If the bucket size B of hash-table becomes 1 then hash-table based OPT-MVOSTMs boils down to the list based OPT-MVOSTMs. 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 [7]. 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 12 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: $(preds[0].BL = currs[1])$ && $(preds[1].RL = currs[0])$ at Line 74.

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 [7].

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 n for k as: $\langle key = k, lock = false, marked = true, vl = ver, nnext = \phi \rangle$ and insert it into $B_k.lazyrb\text{-}list$ such that it is accessible only via RL. This node will have a single-version ver as $\langle ts = 0, val = null, rvl = i, max_{rvl} = i, vnext = \phi \rangle$. 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_2, the previous value of v_0 is still retained. Thus, when T_1 invokes lu on k_2 after the delete on k_2 by T_2, *OPT-HT-MVOSTM* will return v_0 (as previous value). Hence, each rv_method will find a version to read while maintaining unbounded versions corresponding to each key k. $marked$ field sets to true because it access by RL only. In rvl and max_{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 are 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 24). max_{rvl} maintains the maximum timestamp among all rv_methods read from this version at Line 26. Then release the locks, update the local log $txLog_i$ in Line 29 and return the value stored in $ver_j.val$ in Line 31.

Algorithm 2 *rv_method:* It can be either *t_delete*$_i(ht, k, v)$ or *t_lookup*$_i(ht, k, v)$ on key k that maps to bucket B_k of hash-table ht.

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

***t_insert*()**: This is another optimization done in *OPT-HT-MVOSTMs* to identify the early abort which prevents the work done by aborted transactions and saves time. The actual effect of the *t_insert*() comes after the successful tryC method. First, t_insert() searches the key k in the local log, $txLog_i$ of T_i at Line 34. If k does not exist in the $txLog_i$ then it identifies the appropriate location ($pred$ and $curr$) of key k using BL and RL (Line 35) in the lazyrb-list of B_k without acquiring any locks similar to rv_method explained above.

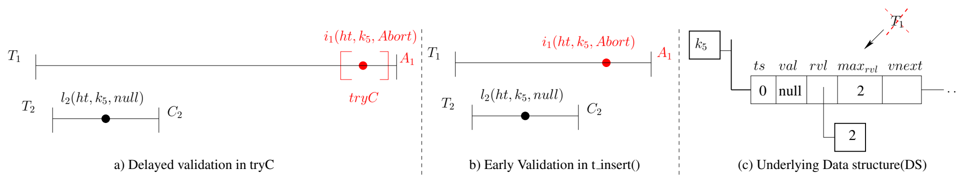


Figure 8: Advantage of early validation in t_insert()

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 37 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) at Line 85. If max_{rvl} of ver_j is greater than timestamp i at Line 86 then it returns $Abort$ in Line 38.

tryC_Validation() in *t_insert*() identifies the early abort of invalid transaction. The advantage of doing the early validation to save the significant computation of long running transaction which will abort in the future. Consider Figure 8 where two transaction T_1 and T_2 working on key k_5. In Figure 8 (a), T_1 aborts in tryC (delayed validation) because higher timestamp T_2 committed. But in Figure 8 (b), T_1 validates the t_insert() instantly by looking into the max_{rvl} of k_5 as shown in Figure 8 (c) and save its computation and returns abort.

Algorithm 3 *t_insert():* Actual insertion happens in the tryC.

```
33: procedure t_insert()
34:     if (k ∉ txLog_i) then
35:         Search in lazyrb-list to identify the preds[] and currs[] for k using BL and RL in bucket B_k.
36:         Acquire the locks on preds[] and currs[] in increasing order.
37:         if (! tryC_Validation()) then
38:             return Abort.                                        ▷ Release the locks
39:         end if
40:         Release the locks.
41:     else
42:         Update the local log.
43:     end if
44: end procedure
```

upd_methods: It can be either *t_insert*(ht, k, v) or *t_delete*(ht, k, v). 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*. Algorithm 4 represents the high level overview of *tryC*.

Initially, to avoid deadlocks, the 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 50) 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 52 as explained in t_insert().

Algorithm 4 *tryC(T_i)*: Validate the upd_methods of the transaction and then commit.

45: **procedure** $tryC(T_i)$
46: /*Operation name (opn) which could be either *t_insert* or *t_delete* */
47: /*Sort the $keys$ of $txLog_i$ in increasing order.*/
48: **for all** ($opn_i \in txLog_i$) **do**
49: **if** ((opn_i == *t_insert*) \|\| (opn_i == *t_delete*)) **then**
50: Search in $lazyrb\text{-}list$ to identify the $preds[]$ and $currs[]$ for k using BL and RL in bucket B_k.
51: Acquire the locks on $preds[]$ and $currs[]$ in increasing order.
52: **if** (! $tryC\_Validation()$) **then**
53: return $Abort$. ▷ Release the locks
54: **end if**
55: **end if**
56: **end for**
57: **for all** ($opn_i \in txLog_i$) **do**
58: $intraTransValidation()$ modifies the $preds[]$ and $currs[]$ of current operation which would have been updated by the previous operation of the same transaction.
59: **if** ((opn_i == *t_insert*) && ($k \notin B_k.lazyrb\text{-}list$)) **then**
60: Create new node n with k as: ⟨ *key = k, lock = false, marked = false, vl = ver, nnext =* ϕ ⟩.
61: Create two versions ver as: ⟨ *ts=0, val=null, rvl=*ϕ*,* max_{rvl} = ϕ*, vnext=i* ⟩ for T_0 and ⟨ *ts=i, val=v, rvl=*ϕ*,* max_{rvl} = ϕ*, vnext=*ϕ ⟩ for T_i.
62: Insert node n into $B_k.lazyrb\text{-}list$ such that it is accessible via RL as well as BL ▷ $lock$ sets $true$.
63: **else if** (opn_i == *t_insert*) **then**
64: Add the version ver as: ⟨ *ts=i, val=v, rvl=*ϕ*,* max_{rvl}*=*ϕ*, vnext=*ϕ ⟩ into $B_k.lazyrb\text{-}list$ such that it is accessible via RL as well as BL.
65: **end if**
66: **if** (opn_i == *t_delete*) **then**
67: Add the version ver as: ⟨ *ts=i, val=null, rvl=*ϕ*,* max_{rvl}*=*ϕ*, vnext=*ϕ ⟩ into $B_k.lazyrb\text{-}list$ such that it is accessible only via RL.
68: **end if**
69: Update the $preds[]$ and $currs[]$ of opn_i in $txLog_i$.
70: **end for**
71: Release the locks; return $Commit$.
72: **end procedure**

If *tryC_Validation* is successful then each upd_methods exist in $txLog_i$ will take the effect in the shared memory after doing the *intraTransValidation()* in Line 58. 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 are 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 58.

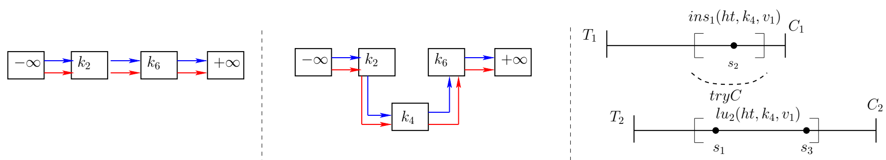
Next, we check if upd_method is *t_insert* and k is in $B_k.lazyrb\text{-}list$. If k is not in B_k, then create a new node n for k as $\langle key = k, lock = false, marked = false, vl = ver, nnext = \phi \rangle$. This node will have two versions ver as $\langle ts = 0, val = null, rvl = \phi, max_{rvl} = \phi, vnext = i \rangle$ for T_0 and $\langle ts = i, val = v, rvl = \phi, max_{rvl} = \phi, vnext = \phi \rangle$ for T_i. T_i is creating a version with timestamp 0 to ensure that rv_methods of other transactions will never abort. For second version, i is the timestamp of the transaction T_i invoking this method; $marked$ field sets to false because the node is inserted in the BL. rvl, max_{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 62). 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 ver as: $\langle ts = i, val = v, rvl = \phi, max_{rvl} = \phi, vnext = \phi \rangle$ and insert the version into $B_k.lazyrb\text{-}list$ such that it is accessible via RL as well as BL (Line 64).

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 ver as $\langle ts = i, val = null, rvl = \phi, max_{rvl} = \phi, vnext = \phi \rangle$ and insert the version into $B_k.lazyrb\text{-}list$ such that it is accessible only via RL (Line 67).

Finally, at Line 69 it updates the $pred$ and $curr$ of opn_i in local log, $txLog_i$. At Line 71 releases the locks on all the $pred$ and $curr$ in increasing order of keys to avoid deadlocks and return $Commit$.

We illustrate the helping methods of rv_method, t_insert(), and upd_method in detail as follows:

rv_Validation(): It is called by the rv_method, t_insert(), and upd_method. It identifies the conflicts among the concurrent methods of different transactions. Consider an example shown in Figure 9, where two concurrent conflicting methods of different transactions are working on the same key k_4. Initially, at stage s_1 in Figure 9 (c) both



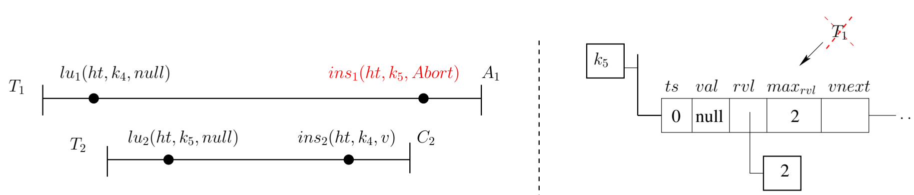
(a) Underlying list at stage s_1 (b) Successful insertion of k_4 at stage s_2 (c) Two concurrent conflicting methods

Figure 9: Illustration of rv_Validation()

the conflicting method optimistically (without acquiring locks) identify the same $pred$ and $curr$ keys for key k_4 from $B_k.lazyrb\text{-}list$ in Figure 9 (a). At stage s_2 in Figure 9 (c), method $ins_1(ht, k_4, v_1)$ of transaction T_1 acquired the lock on $pred$ and $curr$ keys and inserted the node into $B_k.lazyrb\text{-}list$ as shown in Figure 9 (b). After successful insertion by T_1, $pred$ and $curr$ have been changed for $lu_2(ht, k_4)$ at stage s_3 in Figure 9 (c). So, the above modified information is delivered by *rv_Validation* method at Line 74 when $(preds[0].BL \neq curr s[1])$ for $lu_2(ht, k_4)$. After that again it will find the new $pred$ and $curr$ for $lu_2(ht, k_4, v_1)$ and eventually it will commit.

Algorithm 5 *rv_Validation()*: Validate against the conflicting method of different transactions.

```
73: procedure rv_validation()
74:     if ((preds[0].marked)||(currs[1].marked)||(preds[0].BL) ≠ currs[1]||(preds[1].RL) ≠ currs[0]) then
75:         return false.
76:     else
77:         return true.
78:     end if
79: end procedure
```



(a) Opaque history: T_1 Abort (b) Underlying Data structure(DS)

Figure 10: Illustration of tryC_Validation()

tryC_Validation(): It is called by t_insert(), and upd_method in *tryC*. First, it does the *rv_Validation()* in Line 81. If its successful and 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) at Line 85. If max_{rvl} of ver_j is greater than the timestamp of i then the algorithm returns false (in Line 87) and eventually, return $Abort$ in Line 38 or Line 53. Consider an example as shown in Figure 10 (a), where second method $ins_1(ht, k_5)$ of transaction T_1 returns $Abort$ because higher timestamp of conflicting transaction T_2 is already present in the max_{rvl} of version T_0 identified by T_1 in Figure 10 (b).

Algorithm 6 *tryC_Validation()*: It maintains the order among the transactions.

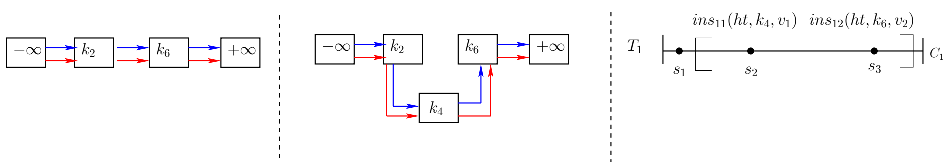
80: **procedure** $tryC\_validation()$
81:     **if** (! $rv\_Validation()$) **then**
82:         Release the locks and retry.
83:     **end if**
84:     **if** (k $\in B_k.lazyrb\text{-}list$) **then**
85:         Identify the version ver_j with $ts = j$ such that j is the largest timestamp smaller than i.
86:         **if** ($ver_j.max_{rvl} > i$) **then**
87:             return $false$.
88:         **end if**
89:     **end if**
90:     return $true$.
91: **end procedure**

Algorithm 7 *intraTransValidation()*: Help the upcoming method of the same transaction.

92: **procedure** $intraTransValidation()$
93:     **if** ($(preds[0].marked) || (preds[0].BL \neq currs[1])$) **then**
94:         **if** (opn_k == Insert) **then**
95:             /*Modify the pred of current transaction T_i with the help of previous transaction T_k*/
96:             $preds[0]_i$ = $preds[0]_k$.BL.   ▷ Set the T_i preds[0] as T_k curs[1]
97:         **else**
98:             $preds[0]_i$ = $preds[0]_k$.   ▷ Set the T_i preds[0] as T_k preds[0]
99:         **end if**
100:     **end if**
101:     **if** (preds[1].RL $\neq$ currs[0]) **then**
102:         $preds[1]_i$ = $preds[1]_k.RL$.   ▷ Set the T_i preds[1] as T_k currs[0]
103:     **end if**
104: **end procedure**

intraTransValidation(): It is called by upd_method in *tryC*. 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 from Line 93 to Line 103. Consider an example as shown in Figure 11, where two upd_methods of transaction T_1 are $ins_{11}(ht, k_4, v_1)$ and $ins_{12}(ht, k_6, v_2)$ in Figure 11 (c). At stage s_1 in Figure 11 (c) both the upd_methods identify the same $pred$ and $curr$

from underlying DS as $B_k.lazyrb\text{-}list$ shown in Figure 11 (a). After the successful insertion done by first upd_method at stage s_2 in Figure 11 (c), key k_4 is part of $B_k.lazyrb\text{-}list$ (Figure 11 (b)). At stage s_3 in Figure 11 (c), $ins_{12}(ht, k_6, v_2)$ identified ($preds[0].BL \neq curr s[1]$) in *intraTransValidation()* at Line 93. So it updates the $preds[0]$ in Line 96 for correct updation in $B_k.lazyrb\text{-}list$.



(a) Underlying list at stage s_1 (b) Successful insertion of k_4 at stage s_2 (c) Two update methods of T_1

Figure 11: Illustration of intraTransValidation()

For efficient memory utilization, we have developed two variants of *OPT-MVOSTM* with Garbage Collection described as follows:

6.1. OPT-MVOSTM-GC: Garbage Collection in OPT-MVOSTM as

We applied the garbage collection (GC) in OPT-MVOSTM at two places: (1) *Version List (VL)* and (2) *Red Link (RL)*.

(1) Version List (VL): *OPT-MVOSTM-GC* performs *garbage collection (GC)* to delete the unwanted versions from VL. This is achieved by deleting obsolete versions whose timestamp is less than the timestamp of the lowest live transaction. We have adapted the existing GC techniques of multi-versioned STMs [10, 12] and used it in our *OPT-MVOSTM*. For this, we maintain a list of live transactions globally in *livelist*, sorted in ascending order by the transaction id. When transaction T_i begins, the thread invoking it, say Th_x, adds T_i to the *livelist* and when T_i terminates (either commits or aborts), Th_x removes its entry from the *livelist*. During the *tryC()* of T_i of Algorithm 4, after successful validation at Line 70, suppose T_i is at the head of the *livelist*, i.e., T_i has the lowest id among all the live transaction and has successfully created a version of a key k. Then T_i (Th_x) invokes garbage collection procedure to delete all obsolete versions of k whose timestamps are less than T_i's id. T_i then similarly deletes old versions from all the keys accessed by it. If T_i is not the lowest live transaction then it does not invoke the garbage collection.

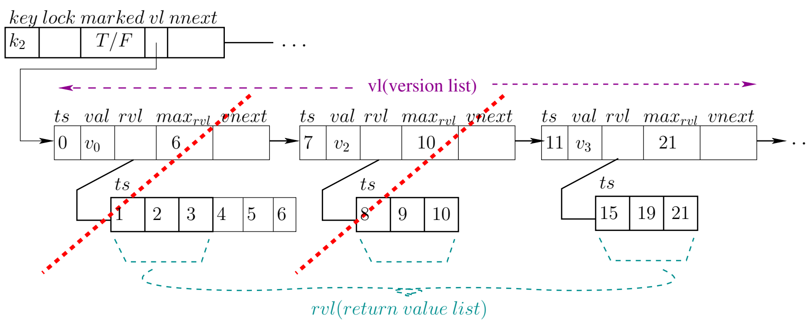


Figure 12: Garbage collection at Version list

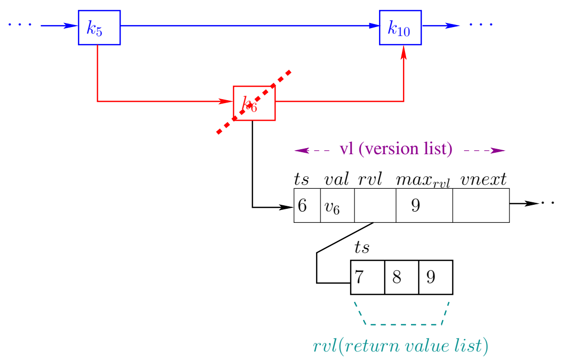


Figure 13: Garbage collection at Red Link

Figure 12 illustrates the garbage collection of the version list. Here key k_2 has three versions created by transaction T_0, T_7, and T_{11}, respectively. Suppose that all the transactions between T_0 through T_{10} have been terminated and T_{11} is the lowest live timestamp transaction. So, T_{11} garbage collects the version created by T_0 and T_7 upon termination.

(2) Red Link (RL): We did one more optimization in *OPT-MVOSTM* on the marked nodes present in the RL. We delete obsolete nodes in the RL (and not just old versions into them) to make it search efficiently. This is achieved by deleting a marked node from RL whose max_{rvl} of the last version is less than the timestamp of the lowest live transaction. That is, after the successful validation of tryC by T_i if T_i is the lowest live transaction in the *livelist* and performs a delete on key k such that max_{rvl} of the last version of key k is less than the timestamp of T_i then T_i does the GC on it. T_i then similarly garbage collects all the keys deleted by it. Otherwise, T_i does not perform this optimization.

Figure 13 illustrates the garbage collection at the red link. Marked node k_6 have one version created by transaction T_6. Transaction T_7, T_8, and T_9 look up from this version, so max_{rvl} is 9. Consider a transaction say T_{10} with lowest live timestamp transaction greater than the max_{rvl} (here max_{rvl} of k_6 is 9) of k_6 then T_{10} garbage collects the marked node k_6 successfully.

6.2. Finite Version in OPT-MVOSTM as OPT-KOSTM

OPT-KOSTM keeps at most K versions by replacing the oldest version when $(K+1)^{th}$ version is created by a current transaction. As *OPT-KOSTM* has a limited number of versions while *OPT-MVOSTM-GC* can have unbounded versions, the memory consumed by *OPT-KOSTM* is also less than *OPT-MVOSTM-GC*. We have integrated this variant in both hash-table based (*OPT-HT-MVOSTM-GC* and *OPT-HT-KOSTM*) and list based MVOSTMs (*OPT-list-MVOSTM-GC* and *OPT-list-KOSTM*). We observed that these two variants (*OPT-MVOSTM-GC* and *OPT-KOSTM*) increase the performance, concurrency and reduce the number of aborts as compared to OPT-MVOSTM in Section 8.

7. Correctness of *OPT-MVOSTM*

In this section, we will prove that our implementation satisfies opacity. Consider the history H generated by *OPT-MVOSTM* algorithm. Recall that only the *t_begin*, rv_method, t_insert(), upd_method (or $tryC$) access shared memory.

Note that H is not necessarily sequential: the transactional methods can execute in an overlapping manner. To reason about correctness, we have to prove H is opaque. Since we defined opacity for histories which are sequential, we order all the overlapping methods in H to get an equivalent sequential history. We then show that this resulting sequential history satisfies opacity.

We order overlapping methods of H as follows: (1) two overlapping *t_begin* methods based on the order in which they obtain lock over the $counter$; (2) two rv_methods accessing the same key k by their order of unlocking over $\langle preds[0], preds[1], currs[0],$

$curr s[1]\rangle$ of k; (3) an rv_method $rvm_i(k)$ and a $t\_insert_j()$, of a transaction T_j accessing the same key k, are ordered by their order of unlocking over $\langle preds[0], preds[1], curr s[0], curr s[1]\rangle$ of k; (4) an rv_method $rvm_i(k)$ and a *tryC*$_j$, of a transaction T_j which has written to k, are similarly ordered by their order of unlocking over $\langle preds[0], preds[1], curr s[0], curr s[1]\rangle$ of k; (5) two t_insert() methods accessing the same key k by their order of unlocking over $\langle preds[0], preds[1], curr s[0], curr s[1]\rangle$ of k; (6) a $t\_insert_i()$ and a *tryC*$_j$, of a transaction T_j which has written to k, are similarly ordered by their order of unlocking over $\langle preds[0], preds[1], curr s[0], curr s[1]\rangle$ of k; (7) similarly, two *tryC* methods based on the order in which they unlock over $\langle preds[0], preds[1], curr s[0], curr s[1]\rangle$ of same key k.

Combining the real-time order of events with above-mentioned order, we obtain a partial order which we denote as $lockOrder_H$. (It is a partial order since it does not order overlapping rv_methods on different $keys$ or an overlapping rv_method and a *tryC* which do not access any common key).

In order for H to be sequential, all its methods must be ordered. Let α be a total order or *linearization* of methods of H such that when this order is applied to H, it is sequential. We denote the resulting history as $H^\alpha = linearize(H, \alpha)$. We now argue about the validity of histories generated by the algorithm.

Lemma 5. *Consider a history H generated by the OPT-MVOSTM algorithm. Let α be a linearization of H which respects $lockOrder_H$, i.e. $lockOrder_H \subseteq \alpha$. Then $H^\alpha = linearize(H, \alpha)$ is valid.*

Proof: Consider a successful rv_method $rvm_i(k)$ that returns value v. The rv_method first obtains the lock on $\langle preds[0], preds[1], curr s[0], curr s[1]\rangle$ of key k. Thus the value v returned by the rv_method must have already been stored in k's version list by a transaction, say T_j when it successfully returned OK from its *tryC* method. For this to have occurred, T_j must have successfully locked and released $\langle preds[0], preds[1], curr s[0], curr s[1]\rangle$ of k prior to T_i's locking method. Thus from the definition of $lockOrder_H$, we get that *tryC*$_j(ok)$ occurs before $rvm_i(k, v)$ which also holds in α. □

It can be seen that for proving correctness, any linearization of a history H is sufficient as long as the linearization respects $lockOrder_H$. The following lemma formalizes this intuition,

Lemma 6. *Consider a history H. Let α and β be two linearizations of H such that both of them respect $lockOrder_H$, i.e. $lockOrder_H \subseteq \alpha$ and $lockOrder_H \subseteq \beta$. Then, $H^\alpha = linearize(H, \alpha)$ is opaque if $H^\beta = linearize(H, \beta)$ is opaque.*

Proof: From Lemma 5, we get that both H^α and H^β are valid histories. Now let us consider each case

If: Assume that H^α is opaque. Then, we get that there exists a legal t-sequential history S that is equivalent to $\overline{H^\alpha}$. From the definition of H^β, we get that $\overline{H^\alpha}$ is equivalent to $\overline{H^\beta}$. Hence, S is equivalent to $\overline{H^\beta}$ as well. We also have that, $\prec_{H^\alpha}^{RT} \subseteq \prec_S^{RT}$. From the definition of $lockOrder_H$, we get that $\prec_{H^\alpha}^{RT} = \prec_{lockOrder_H}^{RT} = \prec_{H^\beta}^{RT}$. This automatically implies that $\prec_{H^\beta}^{RT} \subseteq \prec_S^{RT}$. Thus H^β is opaque as well.

Only if: This proof comes from symmetry since H^α and H^β are not distinguishable. $\square$

This lemma shows that, given a history H, it is enough to consider one sequential history H^α that respects $lockOrder_H$ for proving correctness. If this history is opaque, then any other sequential history that respects $lockOrder_H$ is also opaque.

Consider a history H generated by *OPT-MVOSTM* algorithm. We then generate a sequential history that respects $lockOrder_H$. For simplicity, we denote the resulting sequential history of *OPT-MVOSTM* as H_{to}. Let T_i be a committed transaction in H_{to} that writes to k (i.e. it creates a new version of k).

To prove the correctness, we now introduce some more notations. We define $H_{to}.stl(T_i, k)$ as a committed transaction T_j such that T_j has the *smallest timestamp larger (or stl)* than T_i in H_{to} that writes to k in H_{to}. Similarly, we define $H_{to}.lts(T_i, k)$ as a committed transaction T_k such that T_k has the *largest timestamp smaller (or lts)* than T_i that writes to k in H_{to}. Using these notations, we describe the following properties and lemmas on H_{to},

Property 7. *Every transaction T_i is assigned a unique numeric timestamp i.*

Property 8. *If a transaction T_i begins after another transaction T_j then $j < i$.*

Lemma 9. *If a transaction T_k looks up key k_x from (a committed transaction) T_j then T_j is a committed transaction updating to k_x with j being the largest timestamp smaller than k. Formally, $T_j = H_{to}.lts(T_k, k_x)$.*

Proof: We prove it by contradiction. So, assume that transaction T_k looks up key k_x from T_i that has committed before T_j so, from Property 8, $i < k$ and $k < j$ i.e. i is not largest timestamp smaller than k. But given statement in this lemma is $i < j < k$ which contradicts our assumption. Hence, T_k looks up key k_x from T_j which is the largest timestamp smaller than k. □

Lemma 10. *Suppose a transaction T_k looks up k_x from (a committed transaction) T_j in H_{to}, i.e. $\{\text{up}_j(k_{x,j}, v), \text{rvm}_k(k_{x,i}, v)\} \in evts(H_{to})$. Let T_i be a committed transaction that updates to k_x, i.e. $\text{up}_i(k_{x,i}, u) \in evts(T_i)$. Then, the timestamp of T_i is either less than T_j's timestamp or greater than T_k's timestamp, i.e. $i < j \oplus k < i$ (where $\oplus$ is XOR operator).*

Proof: We will prove this by contradiction. Assume that $i < j \oplus k < i$ is not true. This implies that, $j < i < k$. But from the implementation of rv_method and *tryC* methods, we get that either transaction T_i is aborted or T_k looks up k from T_i in H. Since neither of them are true, we get that $j < i < k$ is not possible. Hence, $i < j \oplus k < i$. □

To show that H_{to} satisfies opacity, we use the graph characterization developed above in Section 4. For the graph characterization, we use the version order defined using timestamps. Consider two committed transactions T_i, T_j such that $i < j$. Suppose both the transactions write to key k. Then the versions created are ordered as $k_i \ll k_j$. We denote this version order on all the $keys$ created as $\ll_{to}$. Now consider the opacity graph of H_{to} with version order as defined by $\ll_{to}$, $G_{to} = OPG(H_{to}, \ll_{to})$. In the following lemmas, we will prove that G_{to} is acyclic.

Lemma 11. *All the edges in $G_{to} = OPG(H_{to}, \ll_{to})$ are in timestamp order, i.e. if there is an edge from T_j to T_i then the $j < i$.*

Proof: To prove this, let us analyze the edges one by one,

- rt edges: If there is an rt edge from T_j to T_i, then T_j terminated before T_i started. Hence, from Property 8 we get that $j < i$.

- rvf edges: This follows directly from Lemma 9.
- mv edges: The mv edges relate a committed transaction T_k updates to a key k, $up_k(k, v)$; a successful rv_method $rvm_j(k, u)$ belonging to a transaction T_j looks up k updated by a committed transaction T_i, $up_i(k, u)$. Transactions T_i, T_k create new versions k_i, k_k, respectively. According to $\ll_{to}$, if $k_k \ll_{to} k_i$, then there is an edge from T_k to T_i. From the definition of $\ll_{to}$ this automatically implies that $k < i$.

  On the other hand, if $k_i \ll_{to} k_k$ then there is an edge from T_j to T_k. Thus, in this case, we get that $i < k$. Combining this with Lemma 10, we get that $j < k$.

Thus in all the cases, we have shown that if there is an edge from T_j to T_i then the $j < i$.

□

To maintain the correctness of OPT-MVOSTM, we order the equivalent serial history which follows the increasing order of timestamp of transactions.

Theorem 12. *Any history H_{to} generated by OPT-MVOSTM is opaque.*

Proof: From the definition of H_{to} and Lemma 5, we get that H_{to} is valid. We show that $G_{to} = OPG(H_{to}, \ll_{to})$ is acyclic. We prove this by contradiction. Assume that G_{to} contains a cycle of the form, $T_{c1} \rightarrow T_{c2} \rightarrow ..T_{cm} \rightarrow T_{c1}$. From Lemma 11 we get that, $c1 < c2 < ... < cm < c1$ which implies that $c1 < c1$. Hence, a contradiction. This implies that G_{to} is acyclic. Thus from Theorem 4, we get that H_{to} is opaque. □

Now, it is left to show that our algorithm is *live*, i.e., under certain conditions, every operation eventually completes. We have to show that the transactions do not deadlock. This is because all the transactions lock all $\langle preds[0], preds[1], currs[0], currs[1] \rangle$ of $keys$ in a predefined order. As discussed earlier, the STM system orders all $\langle preds[0], preds[1], currs[0], currs[1] \rangle$ of $keys$. We denote this order as accessOr-derand denote it as $\prec_{ao}$. Thus $k_1 \prec_{ao} k_2 \prec_{ao} ... \prec_{ao} k_n$.

From accessOrder, we get the following property

Property 13. *Suppose transaction T_i accesses shared objects p and q in H. If p is ordered before q in accessOrder, then $lock(p)$ by transaction T_i occurs before $lock(q)$. Formally, $(p \prec_{ao} q) \Leftrightarrow (lock(p) <_H lock(q))$.*

Theorem 14. *OPT-MVOSTM with unbounded versions ensures that rv_methods do not abort.*

Proof: This is self-explanatory with the help of *OPT-MVOSTM* algorithm because each key is maintaining multiple versions in the case of unbounded versions. So rv_method always finds a correct version to read it from. Thus, rv_methods do not $abort$. □

8. Experimental Evaluation

This section describes the experimental analysis of proposed *OPT-MVOSTMs* with the state-of-the-art STMs. We have three major goals in this section: (1) Analyze the performance benefits (both in terms of time and memory) of the optimized multi-version object based STMs (or *OPT-MVOSTMs*) over multi-version object based STMs (or *MVOSTMs*). (2) Evaluate the benefit of *OPT-MVOSTMs* over the single-version object based STMs (or *OSTMs*), and (3) Analyze the benefit of *OPT-MVOSTMs* over single-version and multi-version read-write STMs. We have implemented hash-table and list object as *OPT-HT-MVOSTM* and *OPT-list-MVOSTM* described in Section 6. To reduce the memory usage, we have also developed two variants of our OPT-MVOSTM. One variant implements the garbage collection with unbounded versions for each key called *OPT-MVOSTM-GC* and another variant maintains a threshold of K versions for each key called *OPT-KOSTM*. We have implemented both the variants for hash-table and list (*OPT-HT-MVOSTM-GC, OPT-HT-KOSTM*) and (*OPT-list-MVOSTM-GC, OPT-list-KOSTM*), respectively.

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 [3], Boosting-list [4], Trans-list [17], ESTM [2], and RWSTM directly from the TLDS framework[3]. And the implementation of MVOSTM [1], OSTM [6] and MVTO [10] from our PDCRL library[4]. We have described the basic functionality of all the above defined state-of-the-art STMs in Section 2. We have implemented our algorithms in C++. Each STM algorithm first creates N-threads, each thread in turn spawns a transaction. Each transaction exports *t_begin*, *t_insert*, *t_lookup*, *t_delete* and *tryC* methods as described in Section 6.

Methodology:[5] We have considered three types of workloads: ($W1$) Li - Lookup intensive (90% lookup, 5% insert, and 5% delete), ($W2$) Mi - Mid intensive (50% lookup, 25% insert, and 25% delete), and ($W3$) Ui - Update intensive (10% lookup, 45% insert, and 45% delete). The experiments are conducted by varying the number of threads from 1 to 64 in the power of 2. To have a fair comparison with the state-of-the-art STMs, we have also used the similar setting which has been used by most of them in their published work, where each thread executes a single transaction, and each transaction in turn performs 10 randomly chosen operations (*t_lookup*, *t_delete*, and *t_insert*). In our experiments, we have considered keys in a range of 0-1000 which are randomly chosen by the transactions to operate on. All the experiments of hash-table based STMs are performed on a 5 buckets hash-table, which has a modulo ($\%$) hash function to map a key to its respective bucket. A bucket in the worst case can have a maximum of 200 keys for 1000 keys in a range from 0-1000. The experimental setting for list-based STMs are same as hash-table based STMs except for bucket size as 1 instead of 5 buckets. We have taken an average over 10 results as the final result for each experiment.

Results: Figure 14 represents the performance benefit of all the variants of proposed optimized *MVOSTM* as compared with the different variants of *MVOSTM* for hash-table objects. It shows *OPT-HT-KOSTM* performs best among all the algorithms (OPT-HT-MVOSTM-GC, OPT-HT-MVOSTM, HT-KOSTM, HT-MVOSTM-GC, HT-MVOSTM)

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

by a factor of 1.02, 1.11, 1.05, 1.07, 1.22 for workload W1, 1.06, 1.09, 1.07, 1.08, 1.15 for workload W2, and 1.01, 1.03, 1.02, 1.03, 1.08 for workload W3, respectively. Along with this, Figure 15 shows the abort count of respective algorithms on workload W1, W2, and W3. The number of aborts are almost same for all the algorithms when number of threads are less than 16. But increasing the number of threads beyond 16, the number of aborts are least in *OPT-HT-KOSTM* as compared to others. So, we compare the performance of *OPT-HT-KOSTM* with the state-of-the-art STMs as shown in Figure 16. *OPT-HT-KOSTM* outperforms all the algorithms (HT-OSTM, ESTM, RWSTM, HT-MVTO, HT-MVTO-GC, HT-KSTM) by a factor of 3.62, 3.95, 3.44, 2.75, 2.11, 1.85 for W1, 1.44, 2.36, 4.45, 9.84, 8.91, 7.42 for W2, and 2.11, 4.05, 7.84, 12.94, 11.45, 10.70 for W3, respectively. The corresponding number of aborts are represented in Figure 17. Number of aborts are minimum for *OPT-HT-KOSTM* as compared to state-of-the-art STMs. Specifically, the number of aborts for *OPT-HT-KOSTM* is almost negligible as compared to HT-OSTM on lookup-intensive workload (W1) as shown in Figure 17 (a). This is because, in lookup-intensive workload W1, the number of updates is not too many. As a result, not many versions get created which causes old versions to get over-written rather infrequently. Hence, *OPT-HT-KOSTM* finds a correct version to look up from in most cases.

The performance of optimized list based *MVOSTM* is similar as optimized hash-table based *MVOSTM*. Figure 18 shows the benefit of the different variants of the proposed optimized *MVOSTM* with all variants of *MVOSTM* for list objects. It shows *OPT-list-KOSTM* performs best among all the algorithms (OPT-list-MVOSTM-GC, OPT-list-MVOSTM, list-KOSTM, list-MVOSTM-GC, list-MVOSTM) by a factor of 1.14, 1.24, 1.21, 1.20, 1.35 for W1, 1.06, 1.07, 1.12, 1.13, 1.20 for W2, and 1.09, 1.19, 1.11, 1.17, 1.31 for W3, respectively. Along with this, Figure 19 shows the minimum abort count by *OPT-list-KOSTM* as compared to other algorithms on workload W1, W2, and W3. These experiments have shown us that the proposed *OPT-list-KOSTM* is the most optimal among all the variants of MVOSTM. Hence, next we choose it to compare with the state-of-the-art list based STMs.

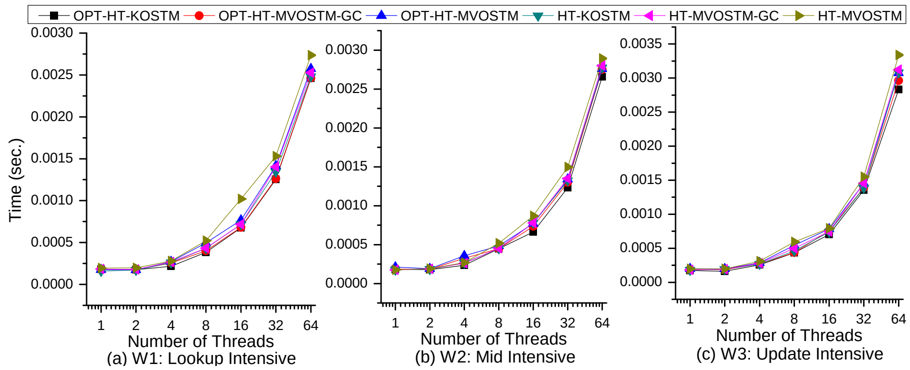


Figure 14: Time comparison among variants of *OPT-HT-MVOSTMs* and *HT-MVOSTMs* on hash-table

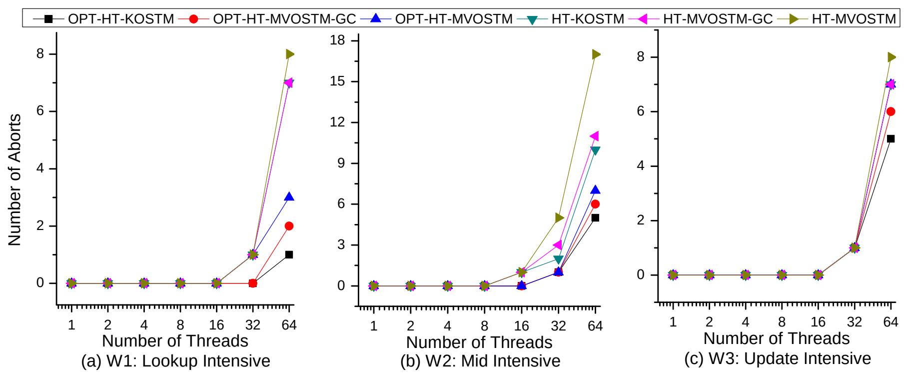


Figure 15: Abort count among variants of *OPT-HT-MVOSTMs* and *HT-MVOSTMs* on hash-table

Figure 20 represents *OPT-list-KOSTM* outperforms all the algorithms (list-OSTM, Trans-list, Boosting-list, NOrec-list, list-MVTO, list-MVTO-GC, list-KSTM) by a factor of 2.56, 25.38, 23.57, 27.44, 13.34, 8.65, 5.99 for W1, 1.51, 20.54, 24.27, 29.45, 24.89, 22.34, 19.78 for W2, and 2.91, 32.88, 28.45, 40.89, 173.92, 156.67, 124.89 for W3, respectively. Similarly, Figure 21 shows that *OPT-list-KOSTM* incurred the least number of aborts as compared to other above mentioned algorithms on the respective workloads.

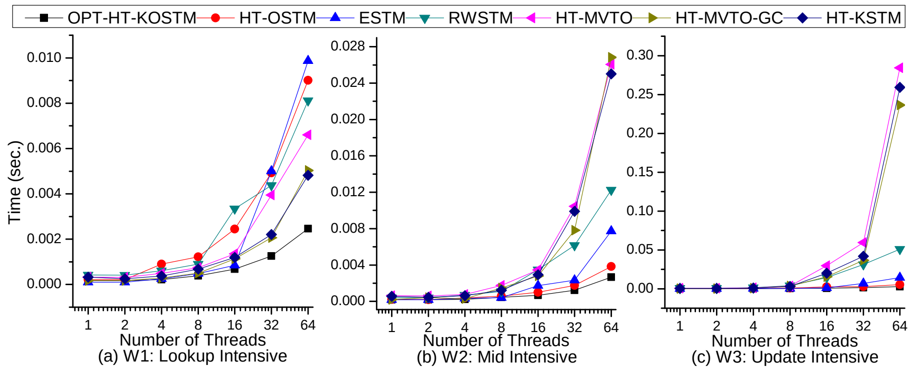


Figure 16: Time comparison of *OPT-HT-KOSTM* and State-of-the-art hash-table based STMs

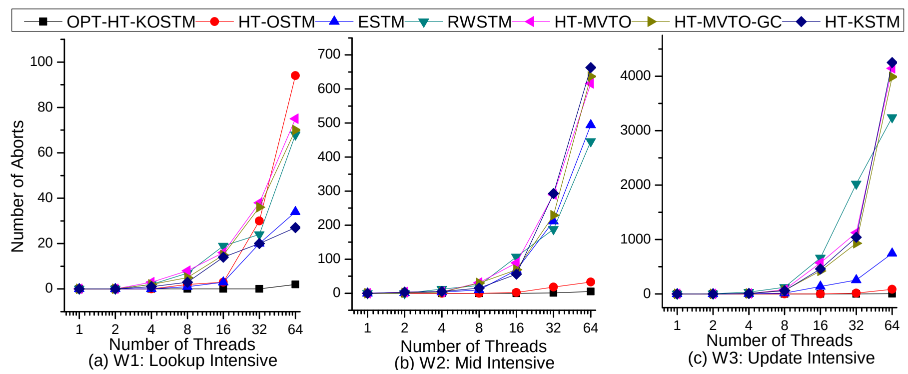


Figure 17: Abort count of *OPT-HT-KOSTM* and State-of-the-art hash-table based STMs

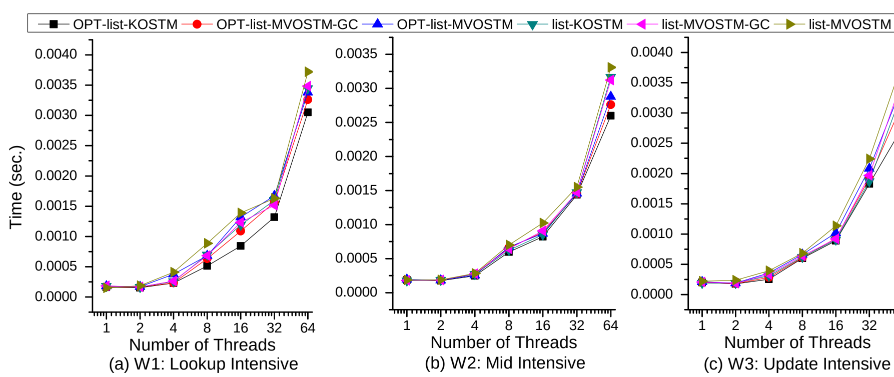


Figure 18: Time comparison among variants of *OPT-list-MVOSTMs* and *list-MVOSTMs* on list

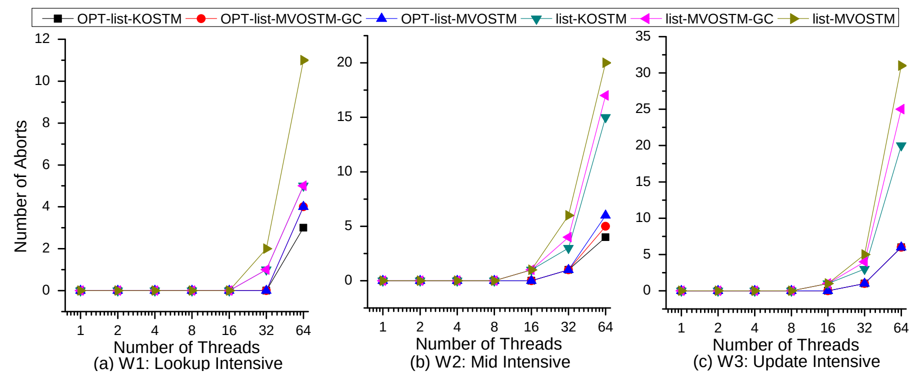


Figure 19: Abort count among variants of *OPT-list-MVOSTMs* and *list-MVOSTMs* on list

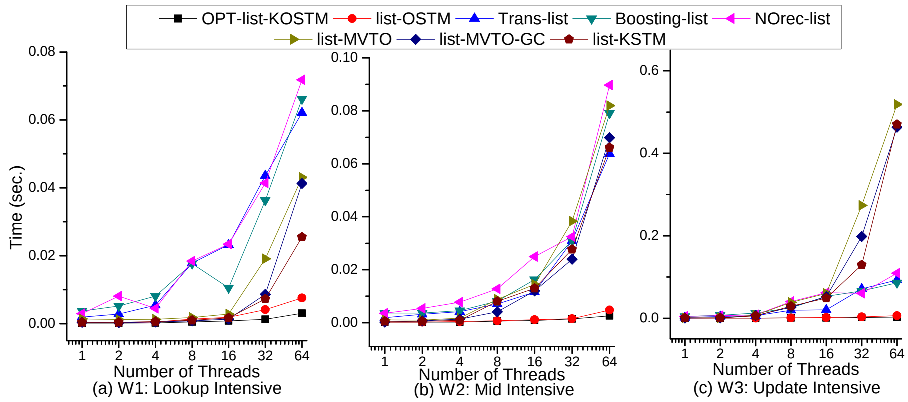


Figure 20: Time comparison of *OPT-list-KOSTM* and State-of-the-art list based STMs

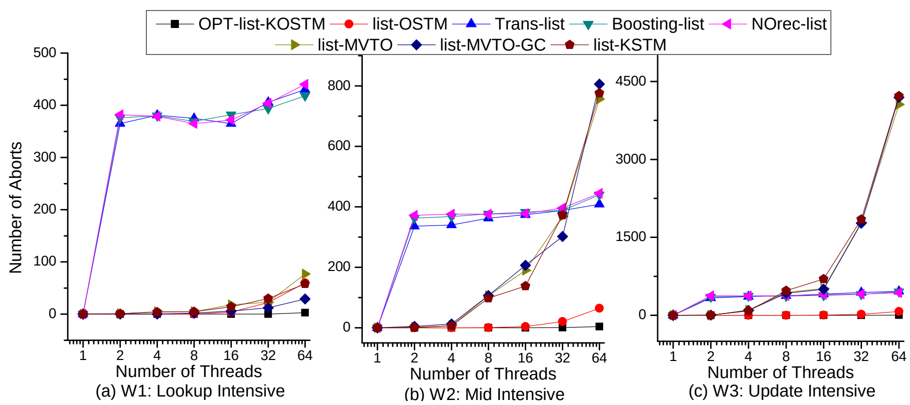


Figure 21: Abort count of *OPT-list-KOSTM* and State-of-the-art list based STMs

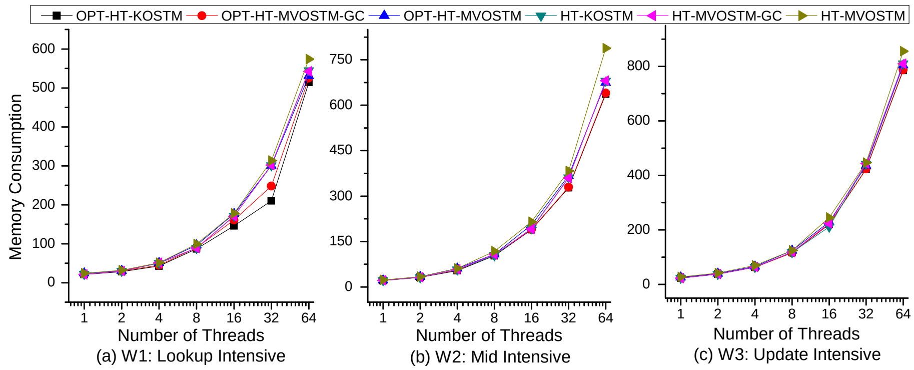


Figure 22: Memory consumption among variants of *OPT-HT-MVOSTMs* and *HT-MVOSTMs* on hash-table

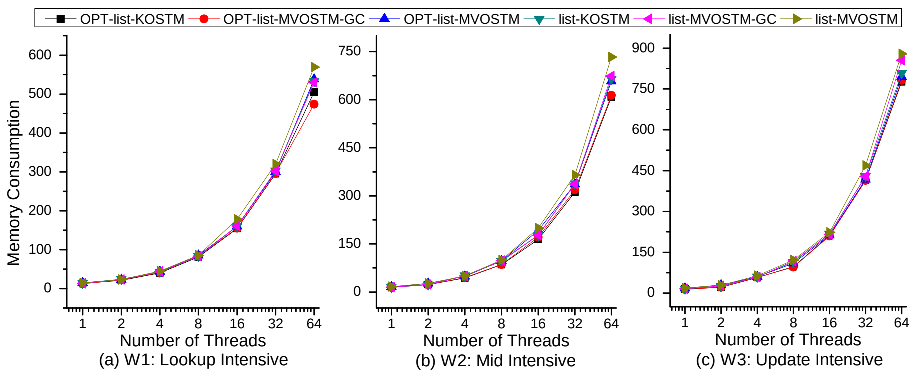


Figure 23: Memory consumption among variants of *OPT-list-MVOSTMs* and *list-MVOSTMs* on list

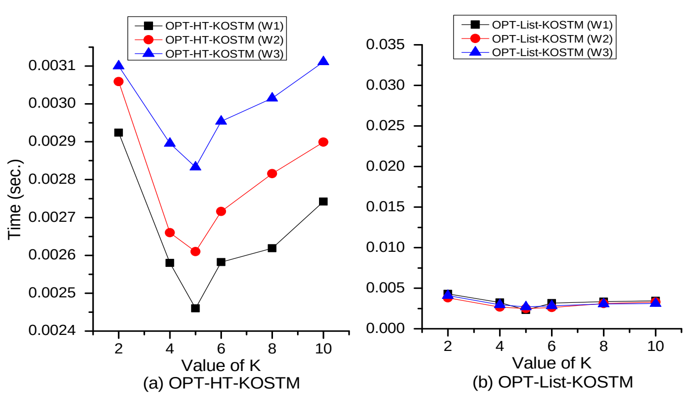


Figure 24: Optimal Value of K for *OPT-HT-KOSTM* and *OPT-list-KOSTM*

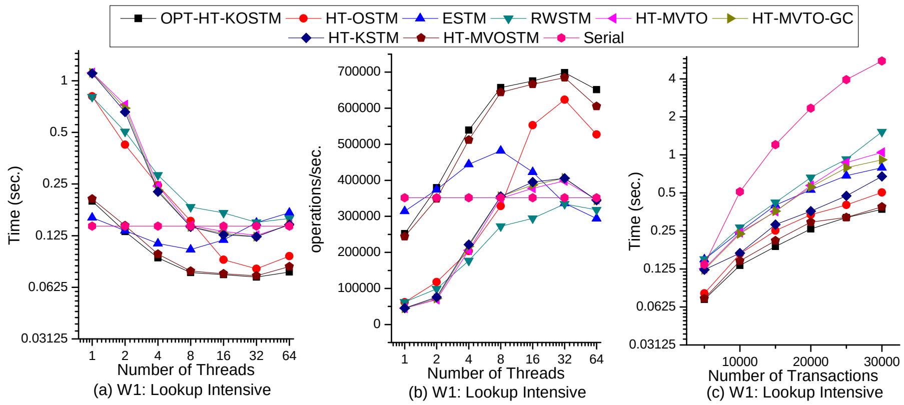


Figure 25: (a) Time taken by varying number of threads (b) Throughput (c) Time taken by varying the number of transactions of *OPT-HT-KOSTM* and State-of-the-art hash-table based STMs

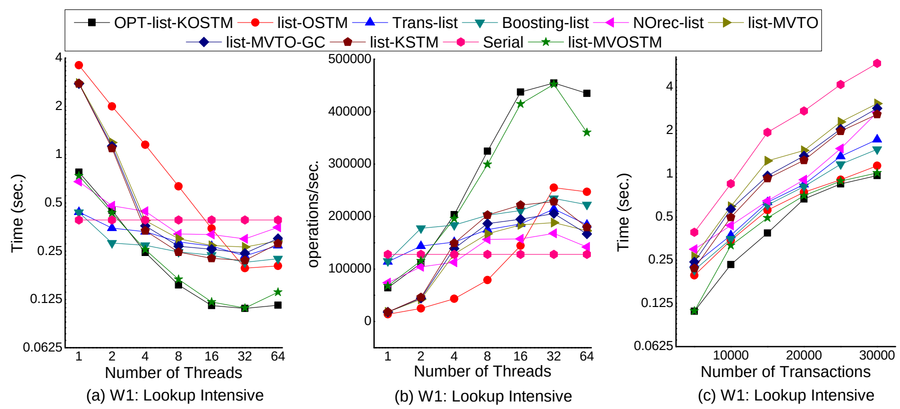


Figure 26: (a) Time taken by varying number of threads (b) Throughput (c) Time taken by varying the number of transactions of *OPT-list-KOSTM* and State-of-the-art list based STMs

We explore the memory consumption of the different variants of *OPT-MVOSTM*. As explained in SubSection6.1 and SubSection6.2, for efficient memory utilization, we develop two variants of *OPT-MVOSTM*: *OPT-MVOSTM-GC* and *OPT-KOSTM*. *OPT-MVOSTM-GC* gave a performance gain of 16% over *OPT-MVOSTM* without garbage collection in the best case which is on workload W1 with 64 number of threads as shown in Figure 22(a). *OPT-KOSTM* showed a performance gain of 24% over *OPT-MVOSTM* without garbage collection in the best case which is on workload W1 with 64 number of threads as shown in Figure 23(a). As *OPT-KOSTM* has a limited number of versions while *OPT-MVOSTM-GC* can have unbounded versions, the memory consumed by *OPT-KOSTM* is also less than *OPT-MVOSTM-GC*. We have integrated these variants in both hash-table based (*OPT-HT-MVOSTM-GC* and *OPT-HT-KOSTM*) and list based MVOSTMs (*OPT-list-MVOSTM-GC* and *OPT-list-KOSTM*), we observed that these two variants increase the performance, concurrency and reduce the number of aborts as compared to OPT-MVOSTM which does not perform garbage collection.

Memory Consumption by OPT-MVOSTM-GC and OPT-KOSTM: As depicted above *OPT-KOSTM* performs better than *OPT-MVOSTM-GC*. Continuing with the comparison of the two variants of *OPT-MVOSTM*, we chose another parameter as *memory consumption*. Here we test for the memory consumed by each variant of the algorithm in creating a version of a key. We count the total versions created, where creating a version increases the counter value by 1 and deleting a version decreases the counter value by 1. We introduced an extra atomic variable to count the number of versions. This atomic variable is not there for the other experiments (time, operations/sec) because of its overhead. We keep counting the number of versions through the atomic variable and return it at the end of the simulation because each thread runs an allocated number of transactions for given keys.

Figure 22 depicts the comparison of memory consumption by all the variants of proposed optimized *MVOSTM* with all variants of *MVOSTM* for hash-table objects. *OPT-HT-KOSTM* consumes minimum memory among all the algorithms (OPT-HT-MVOSTM-GC, OPT-HT-MVOSTM, HT-KOSTM, HT-MVOSTM-GC, HT-MVOSTM) by a factor of 1.07, 1.16, 1.15, 1.15, 1.21 for W1 , 1.01, 1.08, 1.06, 1.07, 1.19 for W2, and 1.01, 1.03, 1.02, 1.03, 1.08 for W3, respectively. Similarly, Figure 23 depicts the comparison of memory consumption by all the variants of proposed optimized *MVOSTM* with all variants of *MVOSTM* for list objects. *OPT-list-KOSTM* consumes minimum memory among all the algorithms (OPT-list-MVOSTM-GC, OPT-list-MVOSTM, list-KOSTM, list-MVOSTM-GC, list-MVOSTM) by a factor of 1.01, 1.05, 1.05, 1.04, 1.11 for W1, 1.02, 1.1, 1.1, 1.11 1.19 for W2, and 1.01, 1.03, 1.05, 1.08, 1.13 for W3, respectively.

Finite version *OPT-MVOSTM* (*OPT-KOSTM*): To find the ideal value of K such that performance does not degrade or can be increased as compared to *OPT-MVOSTM-GC*, we perform experiments on all the workloads (W1, W2, and W3) for both (*OPT-HT-KOSTM* and *OPT-list-KOSTM*). Figure 24 (a) and (b) shows the best value of K as 5 for *OPT-HT-KOSTM* and *OPT-list-KOSTM* on all the workloads for both hash-table and list objects.

We have also observed the behavior of our proposed algorithm against state-of-the-art STMs in more realistic settings where we have considered a different methodology for the following set of experiments. In our first experiment, we varied the number of threads from 1 to 64 over 5000 transactions, each transaction performs 10 randomly chosen operations with 10000 keys on 5 buckets hash-table data structure and as well as on list data structure for lookup intensive workload W1. Here, we analyzed the performance of *OPT-HT-KOSTM* and *OPT-list-KOSTM* against the state-of-the-art STMs including sequential implementation without STM (*Serial*). Figure 25 (a) and Figure 26 (a) unveil the power of concurrency and shows that as the number of threads increases the performance of all the STM algorithms improves till 32 threads. As we are performing our experiments on 56 cores machine, so when the thread count increases beyond 56 (i.e., the thread counts becomes 64) the performance degrades. It can be seen that till 2 threads proposed OPT-HT-KOSTM and OPT-list-KOSTM behaves as an overhead over a serial implementation without STMs (Serial). But, as soon as the number of threads increases, the performance achieved by proposed OPT-HT-KOSTM and OPT-list-KOSTM significantly improve as compared to Serial and state-of-the-art STMs.

Our second experiment, Figure 25 (b) and Figure 26 (b) show the increase in throughput (operations per second) while varying the number of threads till 32. In our third experiment, we vary the number of transactions from 5000 to 30000 while fixing the number of threads as 32. We have observed from Figure 25 (c) and Figure 26 (c), as the number of transactions increases the speedup of *OPT-HT-KOSTM* and *OPT-list-KOSTM* increases as compared to Serial and state-of-the-art STMs. We have done these three experiments for other workloads (W2 and W3) as well but due to space constraints we have shown the results for W1 only and similar observations have been found for W2 and W3.

9. Conclusion

With the rise of multi-core systems, concurrent programming becomes popular. 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 [4, 5, 6] which export higher level objects operation such as hash-table insert, delete, and lookup 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 [23, 10].

This paper proposed the notion of the optimized multi-version object based STMs (*OPT-MVOSTM*s) and compares their effectiveness with multi-version object based STMs (MVOSTMs), single-version object based STMs and multi-version read-write STMs. We find that *OPT-MVOSTM* provides a significant benefit over above-mentioned state-of-the-art STMs for different types of workloads. Specifically, we have evaluated the effectiveness of OPT-MVOSTM for the hash-table and list data structure as *OPT-HT-MVOSTM* and *OPT-list-MVOSTM*, respectively.

OPT-HT-MVOSTM and *OPT-list-MVOSTM* use the unbounded number of versions for each key. To utilize the memory efficiently, we limit the number of versions and develop two variants for both hash-table and list data structures: (1) A garbage collection method in *OPT-MVOSTM* to delete the unwanted versions of a key, denoted as *OPT-MVOSTM-GC*. (2) Placing a limit of K on the number of versions in *OPT-MVOSTM*, resulting in *OPT-KOSTM*. Both these variants (*OPT-MVOSTM-GC* and *OPT-KOSTM*) gave a performance gain of 16% and 24% over *OPT-MVOSTM* in the best case. *OPT-KOSTM* consumes minimum memory among all the variants of it. We represent *OPT-MVOSTM-GC* in hash-table and list as *OPT-HT-MVOSTM-GC* and *OPT-list-MVOSTM-GC*, respectively. Similarly, we represent *OPT-KOSTM* in hash-table and list as *OPT-HT-KOSTM* and *OPT-list-KOSTM*, respectively.

OPT-HT-KOSTM performs best among its variants and outperforms state-of-the-art hash-table based STMs (HT-OSTM, ESTM, RWSTM, HT-MVTO, HT-KSTM) by a factor of 3.62, 3.95, 3.44, 2.75, 1.85 for workload W1, 1.44, 2.36, 4.45, 9.84, 7.42 for workload W2, and 2.11, 4.05, 7.84, 12.94, 10.70 for workload W3, respectively. Similarly, *OPT-list-KOSTM* performs best among its variants and outperforms state-of-the-art list based STMs (list-OSTM, Trans-list, Boosting-list, NOrec-list, list-MVTO, list-KSTM) by a factor of 2.56, 25.38, 23.57, 27.44, 13.34, 5.99 for W1, 1.51, 20.54, 24.27, 29.45, 24.89, 19.78 for W2, and 2.91, 32.88, 28.45, 40.89, 173.92, 124.89 for W3 respectively. We rigorously proved that *OPT-MVOSTM*s satisfy the correctness criteria as opacity. An immediate future direction for OPT-MVOSTM is to extend it for arbitrary data structures such as tree, queue, stack, etc.

References

[1] C. Juyal, S. S. Kulkarni, S. Kumari, S. Peri, A. Somani, 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. doi:10.1007/978-3-030-03232-6_19.
URL https://doi.org/10.1007/978-3-030-03232-6_19

[2] P. Felber, V. Gramoli, R. Guerraoui, Elastic Transactions, J. Parallel Distrib. Comput. 100 (C) (2017) 103–127. doi:10.1016/j.jpdc.2016.10.010.
URL https://doi.org/10.1016/j.jpdc.2016.10.010

[3] L. Dalessandro, M. F. Spear, M. L. Scott, Norec: Streamlining stm by abolishing ownership records, in: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 67–78. doi:10.1145/1693453.1693464.
URL https://doi.org/10.1145/1693453.1693464

[4] M. Herlihy, E. Koskinen, 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. doi:10.1145/1345206.1345237.
URL http://doi.acm.org/10.1145/1345206.1345237

[5] A. Hassan, R. Palmieri, B. Ravindran, Optimistic transactional boosting, in: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, ACM, New York, NY, USA, 2014, pp. 387–388. doi:10.1145/2555243.2555283.
URL http://doi.acm.org/10.1145/2555243.2555283

[6] S. Peri, A. Singh, A. Somani, Efficient means of achieving composability using object based semantics in transactional memory systems, in: Networked Systems - 6th International Conference, NETYS 2018, Essaouira, Morocco, May 9-11, 2018, Revised Selected Papers, 2018, pp. 157–174. doi:10.1007/978-3-030-05529-5_11.
URL https://doi.org/10.1007/978-3-030-05529-5_11

[7] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. S. III, N. Shavit, A Lazy Concurrent List-Based Set Algorithm, Parallel Processing Letters 17 (4) (2007) 411–424.

[8] R. Guerraoui, M. Kapalka, On the correctness of transactional memory, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '08, ACM, New York, NY, USA, 2008, pp. 175–184.

doi:10.1145/1345206.1345233.
URL http://doi.acm.org/10.1145/1345206.1345233

[9] G. Weikum, G. Vossen, Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery, Morgan Kaufmann, 2002.

[10] P. Kumar, S. Peri, K. Vidyasankar, A timestamp based multi-version stm algorithm, in: Proceedings of the 15th International Conference on Distributed Computing and Networking - Volume 8314, ICDCN 2014, Springer-Verlag New York, Inc., New York, NY, USA, 2014, pp. 212–226. doi:10.1007/978-3-642-45249-9_14.
URL http://dx.doi.org/10.1007/978-3-642-45249-9_14

[11] T. Harris, J. Larus, R. Rajwar, Transactional Memory, 2nd Edition, Morgan and Claypool Publishers, 2010.

[12] J. a. Cachopo, A. Rito-Silva, Versioned boxes as the basis for memory transactions, Sci. Comput. Program. 63 (2) (2006) 172–185. doi:10.1016/j.scico.2006.05.009.
URL https://doi.org/10.1016/j.scico.2006.05.009

[13] L. Lu, M. L. Scott, Generic multiversion STM, in: Y. Afek (Ed.), Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings, Vol. 8205 of Lecture Notes in Computer Science, Springer, 2013, pp. 134–148. doi:10.1007/978-3-642-41527-2_10.
URL https://doi.org/10.1007/978-3-642-41527-2_10

[14] I. Keidar, D. Perelman, Multi-versioning in transactional memory, in: R. Guerraoui, P. Romano (Eds.), Transactional Memory. Foundations, Algorithms, Tools, and Applications - COST Action Euro-TM IC1001, Vol. 8913 of Lecture Notes in Computer Science, Springer, 2015, pp. 150–165. doi:10.1007/978-3-319-14720-8_7.
URL https://doi.org/10.1007/978-3-319-14720-8_7

[15] D. Perelman, A. Byshevsky, O. Litmanovich, I. Keidar, SMV: selective multi-versioning STM, in: D. Peleg (Ed.), Distributed Computing - 25th International Symposium, DISC 2011, Rome, Italy, September 20-22, 2011. Proceedings, Vol. 6950 of Lecture Notes in Computer Science, Springer, 2011, pp. 125–140. doi:10.1007/978-3-642-24100-0_9.
URL https://doi.org/10.1007/978-3-642-24100-0_9

[16] S. M. Fernandes, J. a. Cachopo, Lock-free and scalable multi-version software transactional memory, in: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 179–188. doi:10.1145/1941553.1941579.
URL https://doi.org/10.1145/1941553.1941579

[17] D. Zhang, D. Dechev, Lock-free transactions without rollbacks for linked data structures, in: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '16, ACM, New York, NY, USA, 2016, pp. 325–336. doi:10.1145/2935764.2935780.
URL http://doi.acm.org/10.1145/2935764.2935780

[18] R. Guerraoui, M. Kapalka, Principles of Transactional Memory, Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers, 2010. doi:10.2200/S00253ED1V01Y201009DCT004.
URL https://doi.org/10.2200/S00253ED1V01Y201009DCT004

[19] P. Kuznetsov, S. Peri, Non-interference and local correctness in transactional memory, Theor. Comput. Sci. 688 (2017) 103–116. doi:10.1016/j.tcs.2016.06.021.
URL https://doi.org/10.1016/j.tcs.2016.06.021

[20] P. Kuznetsov, S. Ravi, On the cost of concurrency in transactional memory, in: Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings, 2011, pp. 112–127. doi:

10.1007/978-3-642-25873-2_9.
URL https://doi.org/10.1007/978-3-642-25873-2_9

[21] C. H. Papadimitriou, The serializability of concurrent database updates, J. ACM 26 (4) (1979) 631–653. doi:10.1145/322154.322158.
URL http://doi.acm.org/10.1145/322154.322158

[22] T. L. Harris, A pragmatic implementation of non-blocking linked-lists, in: Proceedings of the 15th International Conference on Distributed Computing, DISC '01, Springer-Verlag, London, UK, UK, 2001, pp. 300–314.
URL http://dl.acm.org/citation.cfm?id=645958.676105

[23] D. Perelman, R. Fan, I. Keidar, On maintaining multiple versions in stm, in: Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC '10, ACM, New York, NY, USA, 2010, pp. 16–25. doi:10.1145/1835698.1835704.
URL http://doi.acm.org/10.1145/1835698.1835704