A TimeStamp Based Multi-version STM Algorithm

Priyanka Kumar[1,⋆], Sathya Peri[1], and K. Vidyasankar[2]

[1] CSE Dept, Indian Institute of Technology Patna, India
{priyanka,sathya}@iitp.ac.in
[2] CS Dept, Memorial University, St John's, Canada
vidya@mun.ca

Abstract. Software Transactional Memory Systems (STM) are a promising alternative for concurrency control in shared memory systems. Multiversion STM systems maintain multiple versions for each t-object. The advantage of storing multiple versions is that it facilitates successful execution of higher number of read operations than otherwise. Multi-Version permissiveness (mv-permissiveness) is a progress condition for multi-version STMs that states that a read-only transaction never aborts. Recently a STM system was proposed that maintains only a single version but is mv-permissive. This raises a natural question: how much concurrency can be achieved by multi-version STM. We show that fewer transactions are aborted in multi-version STMs than single-version systems. We also show that any STM system that is permissive w.r.t opacity must maintain at least as many versions as the number of live transactions. A direct implication of this result is that no single-version STM can be permissive w.r.t opacity.

In this paper we present a time-stamp based multiversion STM system that satisfies opacity and is easy to implement. We formally prove the correctness of the proposed STM system. Although many multi-version STM systems have been proposed in literature that satisfy opacity, to the best of our knowledge none of them has been formally proved to be opaque. We also present garbage collection procedure which deletes unwanted versions of the transaction objects. We show that with garbage collection the number of versions maintained is bounded by number of live transactions.

1 Introduction

In recent years, Software Transactional Memory systems (STMs) [10, 21] have garnered significant interest as an elegant alternative for addressing concurrency issues in memory. STM systems take optimistic approach. Multiple transactions are allowed to execute concurrently. On completion, each transaction is validated and if any inconsistency is observed it is *aborted*. Otherwise it is allowed to *commit*.

An important requirement of STM systems is to precisely identify the criterion as to when a transaction should be aborted/committed. A commonly accepted correctness-criterion for STM systems is *opacity* proposed by Guerraoui, and Kapalka [8]. Opacity requires all the transactions including aborted one to appear to execute sequentially

⋆ The author was awarded most promising researcher at the Ph.D forum of ICDCN-13 where a premiminary version of this work was presented.

in an order that agrees with the order of non-overlapping transactions. Opacity unlike traditional serializability [16] ensures that even aborted transactions read consistent values.

Another important requirement of STM system is to ensure that transactions do not abort unnecessarily. This referred to as the *progress* condition. It would be ideal to abort a transaction only when it does not violate correctness requirement (such as opacity). However it was observed in [1] that many STM systems developed so far spuriously abort transactions even when not required. A *permissive* STM [7] does not abort a transaction unless committing it violates the correctness-criterion.

With increase in concurrency, more transactions may conflict and abort, especially in presence of many long-running transactions. This can have a very bad impact on performance [2]. Perelman et al [18] observe that read-only transactions play a significant role in various types of applications. But long read-only transactions could be aborted multiple times in many of the current STM systems [11, 5]. In fact Perelman et al [18] show that many STM systems waste 80% their time in aborts due to read-only transactions.

It was observed that by storing multiple versions of each object, multi-version STMs can ensure that more read operations do not abort than otherwise. History $H1$ illustrates this idea. $H1 : r_1(x, 0)w_2(x, 10)w_2(y, 10)c_2r_1(y, 0)c_1$. In this history the read on y by T_1 returns 0 but not the previous closest write of 10 by T_2. This is possible by storing multiple versions for y. As a result, this history is opaque with the equivalent correct execution being T_1T_2. Had there not been multiple versions, $r_2(y)$ would have been forced to read the only available version which is 10. This value would make the read $r_2(y)$ to not be consistent (opaque) and hence abort.

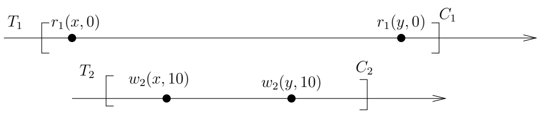

Fig. 1. Pictorial representation of a History $H1$

Maintaining multiple versions was first successfully employed in databases. Since then, many STM systems have been developed that store multiple version of objects [4, 6, 15, 18, 19, 20]. The progress condition relating to multi-version STMs is *multi-version permissiveness* or *mv-permissiveness* [19]. A mv-permissive STM system never aborts a read-only transaction; it aborts an update transaction (i.e transaction that also writes) when it conflicts with other update transactions. Unlike permissiveness, mv-permissiveness is not defined w.r.t any correctness-criterion.

Interestingly, Attiya and Hillel [1] proposed a single-version STM system that is mv-permissive. This raises an interesting question: what is the the advantage of having multiple versions if mv-permissiveness can be achieved by single vesion STMs. In this paper, we address this issue by formally proving that by storing multiple versions greater concurrency can be obtained. We then show that any STM system that is permissive w.r.t opacity must maintain at least L versions where L is the maximum number of live transactions in the system. The number of live transactions represents the concurrency in a STM system and hence can be unbounded. Thus, an implication of this result is that no single-version or for that matter any bounded-version STM can be permissive w.r.t opacity.

We then propose a multi-version STM system based on timestamp ordering called as *multi-version timestamp ordering* algorithm or *MVTO*. We formally prove that our algorithm satisfies opacity. In order to prove correctness we use a graph characterization of opacity for sequential histories which is based on the characterization developed by Guerraoui and Kapalka [9]. We believe that our algorithm is very intuitive and easy to implement. By storing multiple versions, the algorithms ensures that no read-only transaction aborts.

Although many multi-version STM systems have been proposed in literature that satisfy opacity, none of them have been formally proved. To the best of our knowledge, this is the first work that formally proves a multi-version STM to be opaque.

Another nice feature of MVTO algorithm proposed is that it does not require transactions to be annotated as read-only before the start of their execution unlike for instance in [15]. This can be very useful for transactions that have multiple control paths, where some of the paths are read-only whereas the others are not and it is not known in advance which path might be chosen.

An important issue that arises with multi-version STMs is that over time, some versions will no longer be required. So multi-version STMs must have mechanisms of deleting unwanted versions. It is necessary to regularly delete unused versions which otherwise could use a lot of memory. Some multi-version STMs solve this issue by having only fixed number of versions. Other systems have garbage collection procedures running alongside that delete older versions. In [19], Perelman et al outline principles for garbage collection.

Finally, we give an algorithm for garbage collection to delete unwanted versions in MVTO and prove its correctness. We then show that the number of versions maintained by the garbage collection algorithm is bounded by the total number of live transactions in the system.

Roadmap. The paper is organized as follows. We describe our system model in Section 2. In Section 3, we formally show that higher concurrency can be achieved by storing multiple versions. In Section 4 we formally define the graph characterization for implementing opacity. In Section 5, we describe the working principle of MVTO protocol and its algorithm. In Section 6 we are give the outline of the garbage collection algorithm. Finally we conclude in Section 7. Due to space constraints, we have only outlined the main idea. The full details can be found in [12].

2 System Model and Preliminaries

The notions and definitions described in this section follow the definitions of [13]. We assume a system of n processes, $p_1, \dots, p_n$ that access a collection of *objects* via atomic *transactions*. The processes are provided with four *transactional operations*: the *write*(x, v) operation that updates object x with value v, the *read*(x) operation that returns a value read in x, *tryC*$()$ that tries to commit the transaction and returns *commit*

($\mathcal{C}$ for short) or *abort* ($\mathcal{A}$ for short), and *tryA*() that aborts the transaction and returns $\mathcal{A}$. Some STM systems also provide for a begin transaction function. The objects accessed by the read and write operations are called as t-objects. For the sake of simplicity, we assume that the values written by all the transactions are unique.

Operations *write*, *read* and *tryC*() may return $\mathcal{A}$, in which case we say that the operations *forcefully abort*. Otherwise, we say that the operation has *successfully* executed. Each operation is equipped with a unique transaction identifier. A transaction T_i starts with the first operation and completes when any of its operations returns a or c. Abort and commit operations are called *terminal operations*.

For a transaction T_k, we denote all its read operations as $Rset(T_k)$ and write operations $Wset(T_k)$. Collectively, we denote all the operations of a transaction T_i as $evts(T_k)$.

Histories. A *history* is a sequence of *events*, i.e., a sequence of invocations and responses of transactional operations. The collection of events is denoted as $evts(H)$. For simplicity, we only consider *sequential* histories here: the invocation of each transactional operation is immediately followed by a matching response. Therefore, we treat each transactional operation as one atomic event, and let $<_H$ denote the total order on the transactional operations incurred by H. With this assumption the only relevant events of a transaction T_k are of the types: $r_k(x, v)$, $r_k(x, \mathcal{A})$, $w_k(x, v)$, $w_k(x, v, \mathcal{A})$, $tryC_k(C)$ (or c_k for short), $tryC_k(\mathcal{A})$, $tryA_k(\mathcal{A})$ (or a_k for short). We identify a history H as tuple $\langle evts(H), <_H \rangle$.

Let $H|T$ denote the history consisting of events of T in H, and $H|p_i$ denote the history consisting of events of p_i in H. We only consider *well-formed* histories here, i.e., (1) each $H|T$ consists of a read-only prefix (consisting of read operations only), followed by a write-only part (consisting of write operations only), possibly *completed* with a $tryC$ or $tryA$ operation[1], and (2) each $H|p_i$ consists of a sequence of transactions, where no new transaction begins before the last transaction completes (commits or a aborts).

We assume that every history has an initial committed transaction T_0 that initializes all the data-objects with 0. The set of transactions that appear in H is denoted by *txns*(H). The set of committed (resp., aborted) transactions in H is denoted by *committed*(H) (resp., *aborted*(H)). The set of *live* (or *incomplete*) transactions in H is denoted by *live*(H) (*live*(H) = *txns*(H) − *committed*(H) − *aborted*(H)). For a history H, we construct the *completion* of H, denoted $\overline{H}$, by inserting a_k immediately after the last event of every transaction $T_k \in$ *live*(H).

Transaction orders. For two transactions $T_k, T_m \in$ *txns*(H), we say that T_k *precedes* T_m in the *real-time order* of H, denote $T_k \prec_H^{RT} T_m$, if T_k is complete in H and the last event of T_k precedes the first event of T_m in H. If neither $T_k \prec_H^{RT} T_m$ nor $T_m \prec_H^{RT} T_k$, then T_k and T_m *overlap* in H. A history H is *t-sequential* if there are no overlapping transactions in H, i.e., every two transactions are related by the real-time order.

Valid and legal Histories. Let H be a history and $r_k(x, v)$ be a successful read operation (i.e $v \neq \mathcal{A}$) in H. Then $r_k(x, v)$, is said to be *valid* if there is a transaction T_j in H that commits before r_K and $w_j(x, v)$ is in $evts(T_j)$ (T_j can also be T_0). Formally, $\langle r_k(x, v)$

[1] This restriction brings no loss of generality [14].

is valid $\Rightarrow \exists T_j : (c_j <_H r_k(x,v)) \wedge (w_j(x,v) \in evts(T_j)) \wedge (v \neq \mathcal{A})\rangle$. Since we assume that all the writes are unique there exists only one transaction T_j that writes of v to x. We say that the commit operation c_j is r_k's *valWrite* and formally denote it as $H.valWrite(r_k)$. The history H is valid if all its successful read operations are valid.

We define $r_k(x,v)$'s *lastWrite* as the latest commit event c_j such that c_j precedes $r_k(x,v)$ in H and $x \in Wset(T_i)$. Formally, we denote it as $H.lastWrite(r_k)$. A successful read operation $r_k(x,v)$ (i.e $v \neq \mathcal{A}$), is said to be *legal* if transaction T_j (which contains r_k's lastWrite) also writes v onto x. Formally, $\langle r_k(x,v)$ is legal $\Rightarrow (v \neq \mathcal{A}) \wedge (H.lastWrite(r_k(x,v)) = c_i) \wedge (w_i(x,v) \in evts(T_i))\rangle$. The history H is legal if all its successful read operations are legal. Thus from the definitions we get that if H is legal then it is also valid.

It can be seen that in $H1, c_0 = H1.valWrite(r_1(x,0)) = H1.lastWrite(r_1(x,0))$. Hence, $r_1(x,0)$ is legal. But $c_0 = H1.valWrite(r_1(y,0)) \neq c_1 = H1.lastWrite(r_1(y,0))$. Thus, $r_1(y,0)$ is valid but not legal.

Correctness Criteria and Opacity. A correctness-criterion is a set of histories. We say that two histories H and H' are *equivalent* if they have the same set of events.

A history H is said to be *opaque* [8, 9] if H is valid and there exists a t-sequential legal history S such that (1) S is equivalent to $\overline{H}$ and (2) S respects $\prec_H^{RT}$, i.e $\prec_H^{RT} \subseteq \prec_S^{RT}$. We denote the set of all opaque histories as *opacity*. Thus, opacity is a correctness-criterion. By requiring S being equivalent to $\overline{H}$, opacity treats all the live transactions as aborted.

Implementations and Linearizations. A STM *implementation* provides the processes with functions for implementing read, write, tryC (and possibly tryA) functions. We denote the set of histories *generated* by a STM implementation I as $gen(I)$. We say that an implementation I is correct w.r.t to a correctness-criterion C if all the histories generated by I are in P i.e. $gen(I) \subseteq P$.

The histories generated by an STM implementations are normally not sequential, i.e., they may have overlapping transactional operations. Since our correctness definitions are proposed for sequential histories, to reason about correctness of an implementation, we order the events in a non-sequential history to obtain an equivalent sequential history. The implementation that we propose has enough information about ordering of the overlapping operations. We denote this total ordering on the events as *linearization*.

Progress Conditions. Let C be a correctness-criterion with H in it. Let T_a be an aborted transaction in H. We say that a history H is permissive w.r.t C if committing T_a, by replacing the abort value returned by an operation in T_a with some non-abort value, would cause H to violate C. In other words, if T_a is committed then H will no longer be in C. We denote the set of histories permissive w.r.t C as $Perm(C)$. We say that STM implementation I is permissive [7] w.r.t some correctness-criterion C (such as opacity) if every history H generated by I is permissive w.r.t C, i.e., $gen(I) \subseteq Perm(C)$.

A STM implementation is *mv-permissive* if it forcibly aborts an update transaction that conflicts with another update transaction. A mv-permissive STM implementation does not abort read-only transactions.

3 Concurrency Provided by Multi-version Systems

It has been observed that by storing multiple versions, more concurrency can be gained. History $H1$ is an example of this. Normally, multi-version STM systems, help read operations to read consistent values. By allowing the read operation to read the appropriate version (if one exists), they ensure that read operations do not read inconsistent values and abort. To capture this notion precisely, Perelman et al [19] defined the notion of mv-permissiveness which (among other things) says that a read operations always succeed.

Although Perelman et al defined mv-permissiveness in the context of multi-version STMs, recently Attiya and Hillel [1] proposed a single-version STM system that is mv-permissive. In their implementation, Attiya and Hillel maintain only a single-version for each t-object, but ensure that no read-only transaction aborts. As a result, their implementation achieves mv-permissiveness. Thus, if single-version STMs can achieve mv-permissiveness, a natural question that arises is how much extra concurrency can be achieved by multi-version STMs?

To address this question, we were inspired by the theory developed for multi-versions in databases by Kanellakis and Papadimitriou [17]. They showed that the concurrency achievable by multi-version databases schedulers is greater than single-version schedulers. More specifically they showed that the greater the number of versions stored, the higher the concurrency. For showing this, they used *view-serializability (vsr)*, the correctness-criterion for databases. They defined a classes of histories: 1-vsr, 2-vsr, ... n-vsr, etc., where k-vsr is the class of histories that maintain k versions and are serializable. They showed that 1-vsr $\subset$ 2-vsr $\subset$ ... $\subset$ n-vsr.

We extend their idea to STMs by generalizing the concept of legality. Consider a read $r_i(x, v)$ in a history H. Let r_i's valWrite be c_j. Let $n - 1$ other transactions commit between c_j and r_i that have also written to x in H, i.e., $c_j <_H c_{k1} <_H c_{k2} <_H \dots <_H c_{(n-1)} <_H r_i$. Thus, n versions have been created before r_i. Suppose r_i reads from the version created by c_j. Then, we say that r_i is *n-legal*. Kanellakis and Papadimitriou [17] denote n as *width* of x. Extending this idea further, we say that a history H is m-*legal* if each of its read operation is n-*legal* for some $n \leq m$. Thus by this definition, if a history is n-*legal* then it is also m-*legal* for $n \leq m$ but not he vice-versa. If the history H is 1-*legal*, we denote it as *single-versioned.*

Extending this idea of legality to opacity, we say that a history H is *m-opaque* if H is n-*legal* and opaque, where $n \leq m$. Thus, if a history is n-*opaque* then it is also m-*opaque* for $n \leq m$. We denote the set of histories that are m-*opaque* as m-*opacity*. Clearly, *opacity* $= \bigcup_{m \geq 1} m$-*opacity*. From this, we get the following theorem.

Theorem 1. *For any* $k > 0$, k-*opacity* $\subset (k + 1)$-*opacity.*

This theorem shows that there exists an infinite hierarchy of classes of opacity. This further shows that even though single-version STMs can achieve mv-permissiveness, these systems by definition can only satisfy 1-*opacity*. On the other hand, multi-version STMs storing k version (for $k > 1$) are a subset of k-*opacity*. Hence, higher concurrency can be achieved by these systems.

Next, we show that any multi-version STM must store as many versions as maximum possible live transactions for each t-object for achieving mv-permissiveness. For showing this, we define the notion of *maxLiveSet* for a history H as is the maximum number of live transactions present in any prefix of H. Formally, $maxLiveSet(H) = (\max_{H' \in prefixSet(H)} \{|live(H')|\})$ where $prefixSet(H)$ is the set of all prefixes of H. We say that an implementation I *maintains* k versions if during its execution, it creates k versions. We now have the following theorem,

Theorem 2. *Consider a multi-version implementation I that is permissive w.r.t opacity and let H be a history generated by it. Then, I must maintain at least* $maxLiveSet(H)$ *versions.*

Although, we have assumed sequential histories in our model, these results also generalize to non-sequential histories as well since sequential histories can be viewed as a restriction over non-sequential histories. The number of live transactions represents the concurrency in a STM system and hence can be unbounded. Thus, an implication of Theorem 2 is that no single-version or any fixed-version STM can be permissive w.r.t opacity.

Having proved a few theoretical properties of multi-version STMs, in the following sections we give an implementation of a multi-version STM and we prove it to be opaque. Motivated by Theorem 1 and Theorem 2, the proposed multi-version STM does not keep any bound on the number of versions to maximize concurrency but uses garbage collection to delete unwanted versions. In the following sections, we only consider opacity and not k-opacity any more.

4 Graph Characterization of Opacity

To prove that a STM system satisfies opacity, it is useful to consider graph characterization of histories. In this section, we describe the graph characterisation of Guerraoui and Kapalka [9] modified for sequential histories. It is similar to the characterization by Bernstein and Goodman [3] which is also for sequential histories but developed for databases transactions.

Consider a history H which consists of multiple version for each t-object. The graph characterisation uses the notion of *version order*. Given H and a t-object x, we define a version order for x as any (non-reflexive) total order on all the versions of x 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 version of x 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 $H4 : r_1(x, 0) r_2(x, 0) r_1(y, 0) r_3(z, 0) w_1(x, 5) w_3(y, 15)$ $w_2(y, 10) w_1(z, 10) c_1 c_2 r_4(x, 5) r_4(y, 10) w_3(z, 15) c_3 r_4(z, 10)$. Using the notation that a committed transaction T_i writing to x creates a version x_i, a possible version order for $H4$ $\ll_{H4}$ is: $\langle x_0 \ll x_1 \rangle, \langle y_0 \ll y_2 \ll y_3 \rangle, \langle z_0 \ll z_1 \ll z_3 \rangle$.

We define the graph characterisation 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. *rt*(real-time) edges: If T_i commits before T_j starts in H, then there is an edge from v_i to v_j. This set of edges are referred to as $rt(H)$.
2. *rf*(reads-from) edges: If T_j reads x from T_i in H, then there is an edge from v_i to v_j. Note that in order for this to happen, T_i must have committed before T_j and $c_i <_H r_j(x)$. This set of edges are referred to as $rf(H)$.
3. *mv*(multiversion) edges: The mv edges capture the multiversion relations and is based on the version order. Consider a successful read operation $r_k(x, v)$ and the write operation $w_j(x, v)$ belonging to transaction T_j such that $r_k(x, v)$ reads x from $w_j(x, v)$ (it must be noted T_j is a committed transaction and $c_j <_H r_k$). Consider a committed transaction T_i which writes to x, $w_i(x, u)$ where $u \neq v$. Thus the versions created x_i, x_j are related by $\ll$. Then, if $x_i \ll x_j$ we add an edge from v_i to v_j. Otherwise ($x_j \ll x_i$), we add an edge from v_k to v_i. This set of edges are referred to 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.

Theorem 3. *A valid history H is opaque iff there exists a version order $\ll_H$ such that $OPG(H, \ll_H)$ is acyclic.*

5 Multiversion Timestamp Ordering (MVTO) Algorithm

We describe a timestamp based algorithm for multi-version STM systems, multiversion timestamp ordering (MVTO) algorithm. We then prove that our algorithm satisfies opacity [9, 8] using the graph characterization developed in the previous section.

5.1 The Working Principle

In our algorithm, each transaction, T_i is assigned a unique timestamp, i, when it is initially invoked by a thread. We denote i to be the id as well as the timestamp of the transaction T_i. Intuitively, the timestamp tells the "time" at which the transaction began. It is a monotonically increasing number assigned to each transaction and is numerically greater than the timestamps of all the transactions invoked so far. All read and write operations carry the timestamp of the transaction that issued it. When an update transaction T_i commits, the algorithm creates new version of all the t-objects it writes to. All these versions have the timestamp i.

Now we describe the main idea behind read, write and tryC operations executed by a transaction T_i. These ideas are based on the read and write steps for timestamp algorithm developed for databases by Bernstein and Goodman [3]:

1. **read rule:** T_i on invoking $r_i(x)$ reads the value v, where v is the value written by a transaction T_j that commits before $r_i(x)$ and j is the largest timestamp $\leq i$.
2. **write rule:** T_i writes into local memory.

3. **commit rule:** T_i on invoking tryC operation checks for each t-object x, in its *Wset*:
   (a) If a transaction T_k has read x from T_j, i.e. $r_k(x, v) \in evts(T_k)$ and $w_j(x, v) \in evts(T_j)$ and $j < i < k$, then $tryC_i$ returns abort,
   (b) otherwise, the transaction is allowed to commit.

5.2 Data Structures and Pseudocode

The algorithm maintains the following data structures. For each transaction T_i:

– $T_i.RS$(read set): It is a list of data tuples (d_tuples) of the form $\langle x, v\rangle$, where x is the t-object and v is the value read from the transaction T_i.
– $T_i.WS$(write set): It is a list of (d_tuples) of the form $\langle x, v\rangle$, where x is the t-object to which transaction T_i writes the value v.

For each transaction object (t_object) x:

– $x.vl$(version list): It is a list consisting of version tuples (v_tuple) of the form $\langle ts, v, rl\rangle$ where ts is the timestamp of the committed transaction that writes the value v to x. The list rl is the read list consisting of a set of transactions that have read the value v (described below). Informally the version list consists of all the committed transaction that have ever written to this t-object and the set of corresponding transactions that have read each of those values.
– rl(read list): This list contains all the read transaction tuples (rt_tuples) of the form $\langle j\rangle$, where j is the timestamp of the reading transaction. The read list rl is stored in each tuple of the version list described above.

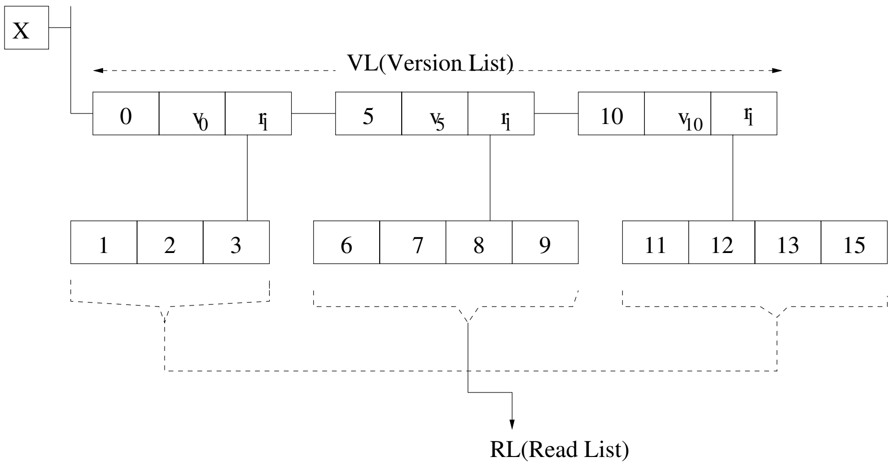

Fig. 2. Data Structures for Maintaining Versions

Figure 2 illustrates how the version list and read list are managed. In addition, the algorithm maintains two global data-structures:

- tCounter: This counter is used to generate the ids/timestamp for a newly invoked transaction. This is incremented everytime a new transaction is invoked.
- liveList: This list keeps track of all the transactions that are currently incomplete or live. When a transaction begins, its id is added to this list. When it terminates (by abort or commit), the id is deleted from this list.

The STM system consists of the following operations/functions. These are executed whenever a transaction begins, reads, write or tries to commit:

$initialize()$: This operation initializes the STM system. It is assumed that the STM system knows all the t-objects ever accessed. All these t-objects are initialized with value 0 by the initial transaction T_0 in this operation. A version tuple $\langle 0, 0, nil \rangle$ is inserted into all the version list of all the t-objects.

$begin\_tran()$: A thread invokes a transaction by executing this operation. It returns an unique transaction identifier which is also its timestamp. The id is used in all other operations exported by the STM system. The id is further stored in the $liveList$.

$read_i(x)$: This operation is invoked when transaction T_i wants to read a t-object x. First, the t-object x is locked. Then the version list of x is searched to identify the correct $version\_tuple$ (i.e the version created by a writing transaction). The tuple with the largest timestamp less than i, say $\langle j, v \rangle$ is identified from the version-list. Then, i is added to the read list of j's version tuple. Finally, the value v written by transaction j, is returned.

$find\_lts(i, x)$: This function is invoked by $read_i(x)$ and finds the tuple $\langle j, v, rl \rangle$ having the <u>l</u>argest <u>t</u>imestamp value j <u>s</u>maller than i (denoted as lts).

$write_i(x, v)$: Here write is performed onto the local memory by transaction T_i. This operation appends the data tuple $\langle x, v \rangle$ into the WS of transaction T_i.

$tryC_i()$: This operation is invoked when a transaction T_i has completed all its operations and wants to commit. This operation first checks whether T_i is read-only transaction or not. If it is read-only then it returns commit. Otherwise, for each t-object x (accessed in a predefined order) in T_i's write set, the following check is performed: if timestamp of T_i, i, lies between the timestamps of the T_j and T_k, where transaction T_k reads x from transaction T_j, i.e $j < i < k$, then the transaction T_i is aborted.

If this check succeeds for all the t-objects written by T_i, then the version tuples are appended to the version lists and the transaction T_i is committed. Before returning either commit or abort, the transaction id i is removed from liveList.

The system orders all the t-objects ever accessed as $x_1, x_2, ..., x_n$ by any transaction (assuming that the system accesses a total of n t-objects). In this operation, each transaction locks and access t-objects in this increasing order which ensures that the system does not deadlock.

$check\_versions(i, x)$: This function checks the version list of x. For all version tuples $\langle j, v, rl \rangle$ in $x.vl$ and for all transactions T_k in rl, it checks if the timestamp of T_i is between the timestamp of the T_j and T_k, i.e $j < i < k$. If so, it returns false else true.

Theorem 4. *The history generated by MVTO algorithm is opaque.*

Theorem 5. *Assuming that no transaction fails and all the locks are starvation-free, every operation of MVTO algorithm eventually returns.*

Algorithm 1. STM $initialize()$: Invoked at the start of the STM system. Initializes all the t-objects used by the STM System

```
1: for all $x$ used by the STM System do
2:     /* $T_0$ is initializing $x$ */
3:     add $\langle 0, 0, nil\rangle$ to $x.vl$;
4: end for;
```

Algorithm 2. STM $begin\_tran()$: Invoked by a thread to being a new transaction T_i

```
1: lock $liveList$;
2: // Store the latest value of $tCounter$ in $i$.
3: $i = tCounter$;
4: $tCounter = tCounter + 1$;
5: add $i$ to $liveList$;
6: unlock $liveList$;
7: return $i$;
```

Algorithm 3. STM $read_i(x)$: A Transaction T_i reads t-object x

```
1: lock $x$;
2: // From $x.vls$, identify the right $version\_tuple$.
3: $\langle j, v, rl\rangle = find\_lts(i, x)$;
4: Append $i$ into $rl$; // Add $i$ into $j$'s $rl$.
5: *unlock* $x$;
6: return $(v)$; // v is the value returned
```

Algorithm 4. $find\_lts(i, x)$: Finds the tuple $\langle j, v, rl\rangle$ created by the transaction T_j with the largest timestamp smaller than i

```
1: // Initialize $closest\_tuple$
2: $closest\_tuple = \langle 0, 0, nil\rangle$;
3: for all $\langle k, v, rl\rangle \in x.vl$ do
4:     if $(k < i)$ and $(closest\_tuple.ts < k)$ then
5:         $closest\_tuple = \langle k, v, rl\rangle$;
6:     end if;
7: end for;
8: return $(closest\_tuple)$;
```

Algorithm 5. STM $write_i(x, v)$: A Transaction T_i writes into local memory

```
1: Append the $d\_tuple\langle x, v\rangle$ to $T_i.WS$.
2: return $ok$;
```

Algorithm 6. STM $tryC()$: Returns ok on commit else return Abort

```
1: if $(T_i.WS == NULL)$ then
2:     $removeId(i)$;
3:     return $ok$; // A read-only transaction.
4: end if;
5: for all $d\_tuple(x, v)$ in $T_i \cdot WS$ do
6:     /* Lock the t-objects in a predefined order to avoid deadlocks */
7:     Lock $x$;
8:     if $(check\_versions(i, x) == false)$ then
9:         $removeId(i)$;
10:        unlock all the variables locked so far;
11:        return $Abort$;
12:    end if;
13: end for;
14: /* Successfully checked for all the write variables and not yet aborted. So the new write
    versions can be inserted. */
15: for all $d\_tuples\langle x, v\rangle$ in $T_i.WS$ do
16:     insert $v\_tuple\langle i, v, nil\rangle$ into $x.vl$ in the increasing order;
17: end for;
18: $removeId(i)$;
19: unlock all the variables;
20: return $ok$;
```

Algorithm 7. $check\_versions(i, x)$:Checks the version list; it returns True or false

```
1: for all $v\_tuples\langle j, v, rl\rangle$ in $x \cdot vl$ do
2:     for all $T_k$ in $rl$ do
3:         /* $T_k$ has already read the version created by $T_j$ */
4:         if $(j < i < k)$ then
5:             return $false$;
6:         end if;
7:     end for;
8: end for;
9: return $true$;
```

Algorithm 8. $removeId(i)$:Removes transaction id i from the $liveList$

```
1: lock $liveList$;
2: remove $i$ from $liveList$;
3: unlock $liveList$;
```

On mv-permissiveness of MVTO: It can be seen that MVTO algorithm does not abort read-only transactions. It also has the nice property that it does not require transactions to be annotated as read-only before execution. Interestingly this implementation satisfies the definition of mv-permissiveness as defined by Perelman et al [19]: an update transaction aborts only when it conflicts with another update transaction.

But intuitively by mv-permissiveness, Perelman et al meant "read-only transactions do not cause update transactions to abort" as stated by them in [19, section 4]. But this property is not true for MVTO. For instance, consider the following history generated by the MVTO algorithm: $H5 : w_1(x,1)w_1(y,1)c_1r_4(x,1)w_2(x,2)w_1(y,2)a_2$. Here, when T_2 tries to commit after writing to x, the MVTO algorithm will abort it. Thus, it can be argued that the read-only transaction T_4 has caused the update transaction T_2 to abort.

6 Garbage Collection

As one can see with multi-version STMs, multiple versions are created. But storing multiple versions can unnecessarily waste memory. Hence, it is important to perform garbage collection by deleting unwanted versions of t-objects. Some of the earlier STM systems solve this problem by maintaining a fixed number of versions for each t-object. We on the other hand, do not restrict the number of versions. The STM system will detect versions that will never again be used (i.e. have become garbage) and delete them. The garbage collection routine will be invoked from the tryC function whenever the number of versions of a t-object has become greater than a predefined threshold. If required, the threshold can be decided dynamically by the application invoking the STM system based on the current memory requirements. If the threshold is set to 0, then the garbage collection routine will be invoked every time an update transaction invokes tryC and commits.

The STM system will delete a version of a t-object x created by transaction T_i when the following conditions are satisfied: (1) At least one another version of x has been created by T_k and $i < k$; (2) Let T_k be the transaction that has the smallest timestamp larger than i and has created a version of x. Then for every j such that $i < j < k$, T_j has terminated (either committed or aborted).

The reason for having condition 1 is to ensure that there exists at least one version in every state. This rule also ensures that the version created by the transaction with the largest timestamp is never deleted. Now in condition 2, if all the transactions between T_i and T_k have committed then no other transaction will ever read from T_i. Hence, the version of T_i can be deleted.

The complete details of the algorithm is mentioned in [12]. The garbage collection algorithm maintains the read and write rules (described in SubSection 5.1). Hence, its correctness is preserved and the histories generated are opaque. Further, it can be shown that the garbage collection algorithm has the nice property that the number of versions maintained by MVTO is bounded by the total number of live transactions.

7 Discussion and Conclusion

There are many applications that require long running read-only transactions. Many STM systems can cause such transactions to abort. Multi-version STM system ensure that a read-only transactions does not need to abort by maintaining sufficient versions. To capture this notion precisely, Perelman et al [19] defined the notion of mv-permissiveness which (among other things) says that read operations always succeed.

Recently Attiya and Hillel [1] proposed a single-version STM system that is mv-permissive. Thus, if single-version STMs can achieve mv-permissiveness, a natural question that arises is how much extra concurrency can be achieved by multi-version STMs. To address this issue, we have formally shown that with multiple versions, higher concurrency can possibly be achieved. We then show that any STM system that is permissive w.r.t opacity must maintain at least L versions where L is the maximum number of live transactions in the system. The number of live transactions represents the concurrency in a STM system and hence can be unbounded. Thus, an implication of this result is that no single-version or any bounded-version STM can be permissive w.r.t opacity.

We then presented a timestamp based multiversion STM system (MVTO) that satisfies opacity and ensures that no read-only transaction aborts. We believe that the proposed algorithm is very intuitive and easy to implement. Our algorithm is similar in spirit to Lu and Scott's multiversion STM system [15]. Like their algorithm, even our algorithm uses timestamps to decide which version to read from. But unlike their algorithm, it does not need transactions to be annotated as read-only prior to the start of the execution. We also presented an algorithm for garbage collection that deletes version that will never be used. The garbage collection algorithm ensures that on being invoked, reduces the number of versions to less than or equal to the number of live transactions in the history (if number of live transactions is greater than 0).

Although several multi-version STM systems have been proposed that are opaque, to the best of our knowledge none of them have formally been proved to be opaque. In fact, Perelman et al [19] formally prove that their algorithm satisfies strict-serializability. Even though they claim that their algorithm can be easily proved to satisfy opacity, it is not clear how.

As a part of future work, we would like to modify our algorithm to ensure that it satisfies mv-permissiveness. We also plan to implement this algorithm and test its performance on various benchmarks. As a part of the implementation, we would like to compare the performance of our algorithm with Attiya and Hillel's algorithm. We would to see how much benefit do multiple versions offer in practice and the cost required to achieve this.

References

[1] Attiya, H., Hillel, E.: A Single-Version STM that is Multi-Versioned Permissive. Theory Comput. Syst. 51(4), 425–446 (2012)

[2] Aydonat, U., Abdelrahman, T.: Serializability of Transactions in Software Transactional Memory. In: TRANSACT 2008: 3rd Workshop on Transactional Computing (February 2008)

[3] Bernstein, P.A., Goodman, N.: Multiversion Concurrency Control: Theory and Algorithms. ACM Trans. Database Syst. 8(4), 465–483 (1983)
[4] Cachopo, J., Rito-Silva, A.: Versioned Boxes as the basis for Memory Transactions. Science of Computer Programming 63(2), 172–185 (2006)
[5] Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
[6] Fernandes, S.M., Cachopo, J.: Lock-free and Scalable Multi-version Software Transactional Memory. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP 2011, pp. 179–188. ACM, New York (2011)
[7] Guerraoui, R., Henzinger, T.A., Singh, V.: Permissiveness in Transactional Memories. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 305–319. Springer, Heidelberg (2008)
[8] Guerraoui, R., Kapalka, M.: On the Correctness of Transactional Memory. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 175–184. ACM, New York (2008)
[9] Guerraoui, R., Kapalka, M.: Principles of Transactional Memory, Synthesis Lectures on Distributed Computing Theory. Morgan and Claypool (2010)
[10] Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural Support for Lock-Free Data Structures. SIGARCH Comput. Archit. News 21(2), 289–300 (1993)
[11] Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC 2003: Proc. 22nd ACM Symposium on Principles of Distributed Computing, pp. 92–101 (July 2003)
[12] Kumar, P., Peri, S.: A timestamp based multi-version stm protocol that satisfies opacity. CoRR, abs/1305.6624 (2013)
[13] Kuznetsov, P., Peri, S.: On non-interference of transactions. CoRR, abs/1211.6315 (2012)
[14] Kuznetsov, P., Ravi, S.: On the cost of concurrency in transactional memory. In: Fernàndez Anta, A., Lipari, G., Roy, M. (eds.) OPODIS 2011. LNCS, vol. 7109, pp. 112–127. Springer, Heidelberg (2011)
[15] Lu, L., Scott, M.L.: Unmanaged Multiversion STM. Transact (2012)
[16] Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM 26(4), 631–653 (1979)
[17] Papadimitriou, C.H., Kanellakis, P.C.: On Concurrency Control by Multiple Versions. ACM Trans. Database Syst. 9(1), 89–99 (1984)
[18] Perelman, D., Byshevsky, A., Litmanovich, O., Keidar, I.: SMV: Selective Multi-Versioning STM. In: Peleg, D. (ed.) Distributed Computing. LNCS, vol. 6950, pp. 125–140. Springer, Heidelberg (2011)
[19] Perelman, D., Fan, R., Keidar, I.: On Maintaining Multiple Versions in STM. In: PODC, pp. 16–25 (2010)
[20] Riegel, T., Felber, P., Fetzer, C.: A Lazy Snapshot Algorithm with Eager Validation. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 284–298. Springer, Heidelberg (2006)
[21] Shavit, N., Touitou, D.: Software Transactional Memory. In: PODC 1995: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 204–213. ACM, New York (1995)