Efficient means of Achieving Composability using Object based Semantics in Transactional Memory Systems⋆

Sathya Peri, Ajay Singh, and Archit Somani⋆⋆

Department of Computer Science & Engineering, IIT Hyderabad, India
(sathya_p, cs15mtech01001, cs15resch01001)@iith.ac.in

Abstract. Composing together the individual atomic methods of concurrent data-structures (cds) pose multiple design and consistency challenges. In this context composition provided by transactions in software transaction memory (STM) can be handy. However, most of the STMs offer read/write primitives to access shared cds. These read/write primitives result in unnecessary aborts. Instead, semantically rich higher-level methods of the underlying cds like lookup, insert or delete (in case of `hash-table` or lists) aid in ignoring unimportant lower level read/write conflicts and allow better concurrency.
In this paper, we adapt transaction tree model in databases to propose OSTM which enables efficient composition in cds. We extend the traditional notion of conflicts and legality to higher level methods of cds using STMs and lay down detailed correctness proof to show that it is co-opaque. We implement OSTM with concurrent closed addressed `hash-table` (*HT-OSTM*) and list (*list-OSTM*) which exports the higher-level operations as transaction interface.
In our experiments with varying workloads and randomly generated transaction operations, *HT-OSTM* shows speedup of 3 to 6 times and w.r.t aborts *HT-OSTM* is 3 to 7 times better than ESTM and read/write based STM, respectively. Where as, *list-OSTM* outperforms state of the art lock-free transactional list, NOrec STM list and boosted list by 30% to 80% across all workloads and scenarios. Further, *list-OSTM* incurred negligible aborts in comparison to other techniques considered in the paper.

Keywords: Concurrent Data Structures, Composability, Software Transactional Memory, Opacity, Co-opacity

1 Introduction

Software Transaction Memory Systems (*STMs*) are a convenient programming interface for a programmer to access shared memory without worrying about concurrency issues [1, 2] and are natural choice for achieving composability [3].

Most of the *STMs* proposed in the literature are specifically based on read/write primitive operations (or methods) on memory buffers (or memory registers). These *STMs* typically export the following methods: *STM_begin* which begins a transaction, *t_read* which reads from a buffer, *t_write* which writes onto a buffer, *tryC* which validates the operations of the transaction and tries to commit. We refer to these as *Read-Write STMs or RWSTMs*. As a part of the validation, the STMs typically check for *conflicts* among

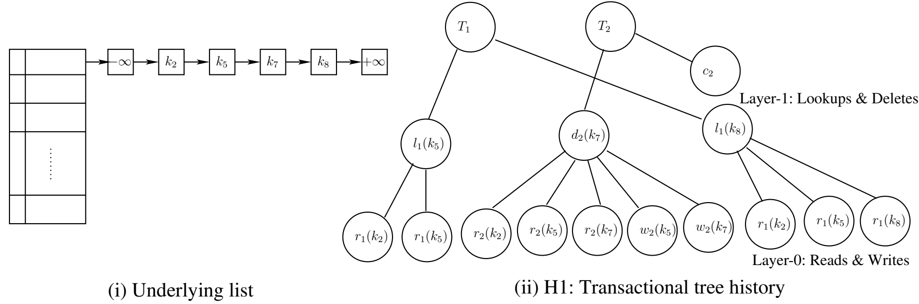
⋆ A preliminary version of this work was accepted in AADDA 2017 as work in progress.

⋆⋆ Author sequence follows lexical order of last names.

the operations. Two operations are said to be conflicting if at least one of them is a write (or update) operation. Normally, the order of two conflicting operations cannot be commutated. On the other hand, *Object STMs* or *OSTMs* operate on higher level objects rather than read & write operations on memory locations. They include more semantically rich operations such as enq/deq on queue objects, push/pop on stack objects and insert/lookup/delete on sets, trees or `hash-table` objects depending upon the underlying data structure used to implement OSTM.

It was shown in databases that object-level systems provide greater concurrency than read/write systems [4, Chap 6]. Along the same lines, we propose a model to achieve composability with greater concurrency for *STMs* by considering higher-level objects which leverage the richer semantics of object level methods. We motivate this with an interesting example.

Consider an *OSTM* operating on the `hash-table` object called as *Hash-table Object STM* or *HT-OSTM* which exports the following methods - *STM_begin*which begins a transaction (same as in *RWSTMs*); *STM_insert* which inserts a value for a given key; *STM_delete* which deletes the value associated with the given key; *STM_lookup* which looks up the value associated with the given key and *STM_tryC* which validates the operations of the transaction.



(i) Underlying list

(ii) H1: Transactional tree history

Fig. 1: Motivational example for OSTMs

A simple way to implement the concurrent *HT-OSTM* is using a list (a single bucket) where each element of the list stores the $\langle$key, value$\rangle$ pair. The elements of the list are sorted by their keys similar to the set implementations discussed in [5, Chap 9]. It can be seen that the underlying list is a concurrent data-structure manipulated by multiple transactions. So, we may use the lazy-list based concurrent set [6] to implement the operations of the list denoted as: *list_insert*, *list_del* and *list_lookup*. Thus, when a transaction invokes *STM_insert*(shortened as i), *STM_delete*(shortened as d) and *STM_-lookup*(shortened as l) methods, the STM internally invokes the *list_insert*, *list_del* and *list_lookup* methods respectively.

Consider an instance of list in which the nodes with keys $\langle k_2\ k_5\ k_7\ k_8\rangle$ are present in the `hash-table` as shown in Figure 1(i) and transactions T_1 and T_2 are concurrently executing *STM_lookup*$_1(k_5)$, *STM_delete*$_2(k_7)$ and *STM_lookup*$_1(k_8)$ as shown in Figure 1(ii). In this setting, suppose a transaction T_1 of *HT-OSTM* invokes methods *STM_lookup* on the keys k_5, k_8. This would internally cause the *HT-OSTM* to invoke *list_lookup* method on keys $\langle k_2, k_5\rangle$ and $\langle k_2, k_5, k_7, k_8\rangle$ respectively.

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

In this execution, we denote the read/write operations (leaves) as layer-0 and *STM_-lookup*, *STM_delete* methods as layer-1. Consider the history (execution) at layer-0 (while ignoring higher-level operations), denoted as $H0$. It can be verified this history is not opaque [7]. This is because between the two reads of k_5 by T_1, T_2 writes to k_5. It can be seen that if history $H0$ is input to a *RWSTMs* one of the transactions among T_1 and T_2 would be aborted to ensure correctness (in this case opacity [7]). On the other hand consider the history $H1$ at layer-1 consisting of *STM_lookup*, *STM_delete* methods while ignoring the underlying read/write operations. We ignore the underlying read & write operations since they do not overlap (referred to as pruning in [4, Chap 6]). Since these methods operate on different keys, they are not conflicting and can be re-ordered either way. Thus, we get that $H1$ is opaque [7] with T_1T_2 (or T_2T_1) being an equivalent serial history.

The important idea in the above argument is that some conflicts at lower-level operations do not matter at higher level operations. Thus, such lower level conflicting operations may be ignored [a]. Harris et al. referred to it as *benign-conflicts* [9]. With object level modeling of histories, we get a higher number of acceptable schedules than read/write model. The history, $H1$ in Figure 1(ii) clearly shows the advantage of considering STMs with higher level *STM_insert*, *STM_delete* and *STM_lookup* operations.

The atomic property of transactions helps to correctly compose together several different individual operations. The above examples demonstrate that the concurrency in such STM can be enhanced by considering the object level semantics. To achieve this, in this paper:**(a)** We propose a generic framework for composing higher level objects based on the notion of conflicts for objects in databases [4, Chap 6]. **(b)** For correctness our framework considers, opacity [7] a popular correctness-criterion for STMs which is different from serializability commonly used in databases. It can be proved that verifying the membership of opacity similar to view-serializability is NP-Complete [10]. Hence, using conflicts we develop a subclass of opacity which is *conflict opacity* or *co-opacity* for objects. We then develop polynomial time graph characterization for co-opacity based on conflict-graph acyclicity. The proposed correctness-criterion, co-opacity is similar to the notion of conflict-opacity developed for *RWSTMs* by Kuznetsov & Peri [11]. **(c)** To show the efficacy of this framework, we develop *HT-OSTM* based on the idea of *basic timestamp order (BTO)* scheduler developed in databases [4, Chap 4]. For showing correctness of *HT-OSTM*, we show that all the methods are linearizabale while the transactions are *co-opaque* by showing that the corresponding conflict graph is acyclic.

[a] While some conflicts of lower level do not matter at higher level, some other conflicts do. An example illustrating this is shown in the technical report [8]

Although we have considered *HT-OSTM* here, we believe that this notion of conflicts can be extended to other high-level objects such as Stacks, Queues, Tries etc.

A simple modification of *HT-OSTM* gives us a concurrent list based STM or *list-OSTM*. Finally, we compared the performance of *HT-OSTM* against a `hash-table` application built ESTM [12] and BTO [4] based RWSTM. The *list-OSTM* is compared with lock-free transactional list [13], NOrec based RSTM list [14] and boosting list [15]. The results in Section 5 represent *HT-OSTM* and *list-OSTM* reduces the number of aborts to minimal and show significant performance gain in comparison to other techniques.

Related Work: Our work differs from databases model in with regard to correctness-criterion used for safety. While databases consider CSR. We consider linearizability to prove the correctness of the methods of the transactions and opacity to show the correctness of the transactions. Earliest work of using the semantics of concurrent data structures for object level granularity include that of open nested transactions [16] and transaction boosting of Herlihy et al. [15] which is based on serializability(strict or commit order serializability) of generated schedules as correctness criteria. Herlihy's model is pessimistic and uses undo logs for rollback. Our model is more optimistic in that sense and the underlying data structure is updated only after there is a guarantee that there is no inconsistency due to concurrency. Thus, we do not need to do rollbacks which keeps the log overhead minimal. This also solves the problem of irrevocable operations being executed during a transaction which might abort later otherwise.

Hassan et al. [17] have proposed optimistic transactional boosting (OTB) that extends original transactional boosting methodology by optimizing and making it more adaptable to STMs. Although there seem similarities between their work and our implementation, we differ w.r.t the correctness-criterion which is co-opacity a subclass of opacity [11] in our case. They did not prove opacity for OTB however, their work extensively talks of linearizability. Furthermore, we also differ in the development of the conflict-based theoretical framework which can be adapted to build other object based STMs. Spiegelman et al. [18] try to build a transactional data structure library from existing concurrent data structure library. Their work is much of a mechanism than a methodology.

Zhang et al. [13] recently propose a method to transform lockfree cds to transactional lockfree linked cds and base the correctness on *strict serializability*. Fraser et. al. [19] proposed OSTM based on shadow copy mechanism, which involves a level of indirection to access the shared objects through *OSTMOpenForReading* and *OSTMOpenForWriting* as exported methods.The exported methods in Fraser et.al's OSTM may allow *OSTMOpenForReading* to see the inconsistent state of the shared objects but our OSTM model precludes this possibility by validating the access during execution of rv_method (i.e. the methods which do not modify the underlying objects and only return some value by performing a search on them).Thus, we can say our motivation and implementation is different from Fraser OSTM [19] and only the name happens to coincide.

Roadmap. We explain the system model in Section 2. In Section 3, we build the notion of legality, conflicts to describe opacity, co-opacity and the graph characterization. Based on the model we demonstrate the *HT-OSTM* design in Section 4. In Section 5 we show the evaluation results. Finally, we conclude in Section 6.

2 Building System Model

In this paper, we assume that our system consists finite number of n threads that run in a completely asynchronous manner and communicate using shared objects. The threads communicate with each other by invoking higher-level methods on the shared objects and getting corresponding responses. Consequently, we make no assumption about the relative speeds of the threads. We also assume that none of these processors and threads fail or crash abruptly.

Events & Methods: We assume that the threads execute atomic *events*. We assume that these 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` transaction object. A `hash-table`(*ht*) consists of multiple key-value pairs of the form $\langle k, v\rangle$. The keys and values are respectively from sets $\mathscr{K}$ and $\mathscr{V}$. The methods that a transaction T_i can invoke are: (1) *STM_begin$_i$()*: Begins a transaction and returns an unique id to the invoking thread (2) *STM_insert$_i$*(ht, k, v): Inserts a value v onto key k in `hash-table` *ht* (3) *STM_delete$_i$*(ht, k, v): Deletes the key k from the `hash-table` *ht* & returns the current value v (4) *STM_lookup$_i$*(ht, k, v): returns the current value v for key k in ht (5) *tryC$_i$()* which tries to commit all the operations of T_i and (6) *tryA$_i$()* aborts T_i. We assume that each method consists of an inv and rsp event.

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

Transactions: Following the notations used in database multi-level transactions [4], 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 transaction.

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

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

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

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

We denote two histories $H1, H2$ as *equivalent* if their events are the same, i.e., $evts(H1) = evts(H2)$. 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 ***HT-OSTM***.

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

Sequential Histories: A history H is said to be *sequential* (term used in [11, 20]) if all the methods in it are complete and isolated. From now on wards, most of our discussion would relate to sequential histories.

Since in sequential histories all the methods are isolated, we treat each method as 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. Consider a sequential history H. Let $m_{ij}(ht, k, v/NULL)$ be the first method of T_i in H operating on the key k as $H.firstKeyMth(\langle ht, k\rangle, T_i)$. 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 & 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* [10] 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, $\langle(H \text{ is serial}) \implies (\forall T_i \in txns(H) : (T_i \in term(H)) \land (\forall T_i, T_p \in txns(H) : (T_i \prec_H^{TR} T_p) \lor (T_p \prec_H^{TR} T_i))\rangle$. Since all the methods within a transaction are ordered, a serial history is also sequential.

3 Correctness of *HT-OSTM*: Opacity & Conflict Opacity

In this section, we define the correctness of *HT-OSTM* by extending opacity [7]. We then define a tractable subclass of opacity, co-opacity which is defined using conflict like CSR [4] in databases. We start with legality and opacity.

3.1 Legal Histories & Opacity

In this subsection, we start with defining legal histories. To simplify our analysis, we assume that there exists an initial transaction T_0 that invokes *STM_delete* method on all the keys of all the hash-tables used by any transaction.

We define *legality* of rv_methods (*STM_delete* & *STM_lookup*) on sequential histories which we later use to define correctness criterion. Consider a sequential history H having a rv_method *rvm*$_{ij}(ht, k, v)$ (with $v \neq NULL$) belonging to transaction T_i. We define this *rvm* method to be *legal* if:

LR1 If the rvm_{ij} is not 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 *STM_insert* method then $v = v'$.
  (b) If $m_{ix}(ht, k, v')$ is a *STM_lookup* method then $v = v'$.
  (c) If $m_{ix}(ht, k, v')$ is a *STM_delete* method then $v = NULL$.
  In this case, we denote m_{ix} as the last update method of rvm_{ij}, i.e., $m_{ix}(ht, k, v') = H.lastUpdt(rvm_{ij}(ht, k, v))$.

LR2 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 *STM_insert* method $STM_insert_{pq}(ht, k, v)$ in $methods(H)$ such that T_p committed before rvm_{ij}. Formally, $\langle \exists STM_insert_{pq}(ht, k, v) \in methods(H) : tryC_p \prec_H^{MR} rvm_{ij}\rangle$.
  (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., $tryC_p(ht, k, v)$= $H.lastUpdt(rvm_{ij}(ht, k, v))$.

LR3 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 *STM_delete* method $STM_delete_{pq}(ht, k, v')$ in $methods(H)$ such that T_p (which could be T_0 as well) committed before rvm_{ij}. Formally, $\langle \exists STM_delete_{pq}(ht, k, v') \in methods(H) : tryC_p \prec_H^{MR} rvm_{ij}\rangle$. 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 similar to step 3.1, 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 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*. This idea is utilized in LR1. With reference to LR2 and LR3, it is possible that T_x could have aborted before rvm_{ij}. For LR3, since we are assuming that transaction T_0 has invoked a *STM_delete* method on all the keys used of all `hash-table` objects, there exists at least one *STM_delete* method for every rv_method on k of ht. We formally prove legality in technical report [8] and then we finally show that *HT-OSTM* histories are co-opaque [11].

Coming to *STM_insert* methods, since a *STM_insert* method always returns ok as they overwrite the node if already present therefore they always take effect on the ht. Thus, we denote all *STM_insert* methods as legal and only give legality definition for rv_method. We denote a sequential history H as *legal* or *linearized* [21] if all its *rvm* methods are legal.

Correctness-Criteria & Opacity: A *correctness-criterion* is a set of histories. A history H satisfying a correctness-criterion has some desirable properties. A popular correctness-criterion is *opacity* [7]. A sequential history H is opaque if there exists a serial history

S such that: (1) S is equivalent to $\overline{H}$, i.e. , $evts(\overline{H}) = evts(S)$ (2) S is legal and (3) S respects the transactional real-time order of H, i.e., $\prec_H^{TR} \subseteq \prec_S^{TR}$.

3.2 Conflict Notion & Conflict-Opacity

Opacity is a popular correctness-criterion for STMs. But, as observed in Section 1, it can be proved that verifying the membership of opacity similar to view-serializability (VSR) in databases is NP-Complete [10]. To circumvent this issue, researchers in databases have identified an efficient sub-class of VSR, called conflict-serializability or CSR, based on the notion of conflicts. The membership of CSR can be verified in polynomial time using conflict graph characterization. Along the same lines, we develop the notion of conflicts for ***HT-OSTM*** and identify a sub-class of opacity, co-opacity. The proposed correctness-criterion is extension of the notion of conflict-opacity developed for ***RWSTMs*** by Kuznetsov & Peri [11].

We say two transactions T_i, T_j of a sequential history H for ***HT-OSTM*** are in *conflict* if atleast one of the following conflicts holds:

- **tryC-tryC** conflict:(1) T_i & T_j are committed and (2) T_i & T_j update the same key k of the `hash-table`, *ht*, i.e., $(\langle ht, k\rangle \in updtSet(T_i)) \wedge (\langle ht, k\rangle \in updtSet(T_j))$, where $updtSet(T_i)$ is update set of T_i. (3) T_i's *tryC* completed before T_j's *tryC*, i.e., $tryC_i \prec_H^{MR} tryC_j$.
- **tryC-rv** conflict:(1) T_i is committed (2) T_i updates the key k of `hash-table`, ht. T_j invokes a rv_method rvm_{jy} on the key same k of `hash-table` ht which is the first method on $\langle ht, k\rangle$. Thus, $(\langle ht, k\rangle \in updtSet(T_i)) \wedge (rvm_{jy}(ht, k, v) \in rvSet(T_j)) \wedge (rvm_{jy}(ht, k, v) = H.firstKeyMth(\langle ht, k\rangle, T_j))$, where $rvSet(T_j)$ is return value set of T_j. (3) T_i's *tryC* completed before T_j's *rvm*, i.e., $tryC_i \prec_H^{MR} rvm_{jy}$.
- **rv-tryC** conflict:(1) T_j is committed (2) T_i invokes a rv_method on the key same k of `hash-table` *ht* which is the first method on $\langle ht, k\rangle$. T_j updates the key k of the `hash-table`, *ht*. Thus, $(rvm_{ix}(ht, k, v) \in rvSet(T_i)) \wedge (rvm_{ix}(ht, k, v) = H.firstKeyMth(\langle ht, k\rangle, T_i)) \wedge (\langle ht, k\rangle \in updtSet(T_j))$ (3) T_i's *rvm* completed before T_j's *tryC*, i.e., $rvm_{ix} \prec_H^{MR} tryC_j$.

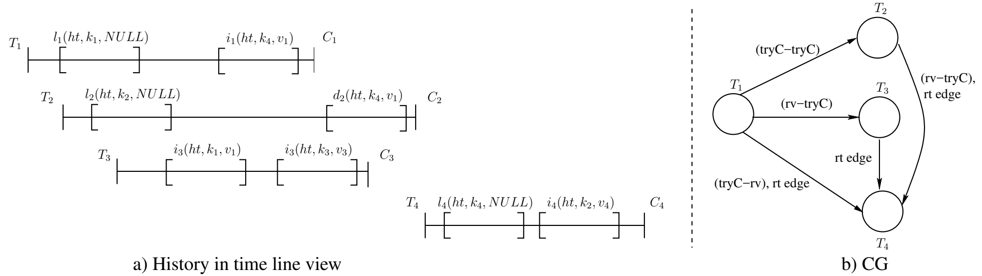


Fig. 2: Graph Characterization of history $H5$

A rv_method rvm_{ij} conflicts with a *tryC* method only if rvm_{ij} is the first method of T_i that operates on `hash-table` with a given key. Thus the conflict notion is defined only by the methods that access the shared memory. $(tryC_i, tryC_j)$, $(tryC_i, STM_lookup_j)$,

($STM_lookup_i, tryC_j$), ($tryC_i, STM_delete_j$) and ($STM_delete_i, tryC_j$) can be the possible conflicting methods. For example, consider the history $H5 : l_1(ht, k_1, NULL)l_2(ht, k_2, NULL)i_2(ht, k_1, v_1)i_1(ht, k_4, v_1)c_1i_3(ht, k_3, v_3)c_3d_2(ht, k_4, v_1)c_2l_4(ht, k_4, NULL) i_4(ht, k_2, v_4)c_4$ in Figure 2. $(l_1(ht, k_1, NULL), i_3(ht, k_1, v_1))$ and $(l_2(ht, k_2, NULL), i_4(ht, k_2, v_4))$ are a conflict of type *rv-tryC*. Conflict type of $(i_1(ht, k_4, v_1), d_2(ht, k_4, v_1))$ and $(i_1(ht, k_4, v_1), l_4(ht, k_4, NULL))$ are *tryC-tryC* and *tryC-rv* respectively.

Conflict Opacity: Using this conflict notion, we can now define co-opacity. A sequential history H is conflict-opaque (or co-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., $\prec_H^{TR} \subseteq \prec_S^{TR}$ and (4) S preserves conflicts (i.e. $\prec_H^{CO} \subseteq \prec_S^{CO}$).
Thus from the above definition, it can be seen that any history that is co-opaque is also opaque.
Graph Characterization: We now develop a graph characterization of co-opacity. For a sequential history H, we define *conflict-graph* of H, $CG(H)$ as the pair (V, E) where V is the set of $txns(H)$ and E can be of following types: (a) *conflict edges:* $\{(T_i, T_j) : (T_i, T_j) \in$ conflict(H)$\}$ where, conflict(H) is an ordered pair of transactions such that the transactions have one of the above pair of conflicts. (b) *real-time edge(or rt edge):* $\{(T_i, T_j)$: Transaction T_i precedes T_j in real-time, i.e., $T_i \prec_H^{TR} T_j\}$. Now, we have the following theorem which explains how graph characterization is useful.
Theorem 1. *A legal HT-OSTM history H is co-opaque iff CG(H) is acyclic.*

Using this framework, we next develop *HT-OSTM* using the notion of BTO. We show the transactional level correctness of the proposed algorithm by showing that all conflict graph of the histories generated by it are acyclic in the accompanying report [8].

4 *HT-OSTM*

We design *HT-OSTM* a concurrent closed addressed `hash-table` using above explained legality and conflict notion. The *HT-OSTM* exports *STM_begin()*, *STM_insert()*, *STM_delete()*, *STM_lookup()* and *STM_tryC()* and has m number of buckets, which we refer to as size of the `hash-table`. The main part of interest from concurrency perspective is each bucket of the `hash-table` implemented as lazyrb-list (lazy red-blue list), the shared memory data structure.

Lazyrb-list: It is a linked structure with immutable $head$ and $tail$ sentinel nodes of the form of a tuple ⟨ *key, value, lock, marked, max_ts, rl, bl* ⟩ representing a node. The key represents unique id of the node so that a transaction could differentiate between two nodes. The key values may range from $-\infty$ (key of head node) to $+\infty$ (key of tail node). The $value$ field may accommodate any type ranging from a basic integer to a complex class type. The $marked$ field is to have lazy deletion as popular in lazylists [5, 6] and $lock$ to implement exclusive access to the node.

Lazyrb-list node have two links - *bl* (blue links) and *rl* (red links). First, the nodes which are not marked (not deleted) are reachable by bl from the head. Second, the nodes which are marked (i.e. logically deleted) and are only reached by rl. Thus, the name lazyrb-list. All marked nodes are reachable via *rl* and all the unmarked nodes are reachable via *bl* & *rl* from the head. Thus nodes reachable by *bl* are the subset of the nodes reachable by *rl*. Every node of lazyrb-list is in increasing order of its key.

Furthermore, every lazyrb-list node also has a *time-stamp* field (max_ts) to record the ids of the transaction which most recently executed some method. Augmenting the underlying shared data structure with time-stamps help in identifying conflicts which can cause a cycle in the execution and hence violate co-opacity [11]. This is captured by the graph characterization of a generated history as discussed in Figure 2 which implies that cyclic conflicts leads to non co-opaque execution.

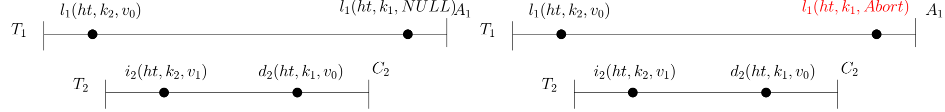
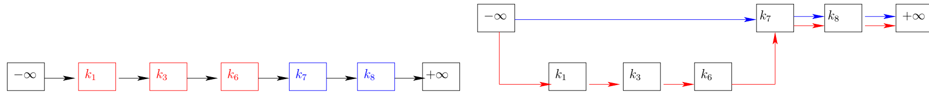


Fig. 3: History H is not co-opaque Fig. 4: co-opaque History H1

Now, we explain why we need to maintain deleted nodes through Figure 3 and 4. History H shown in Figure 3 is not co-opaque because there is no serial execution of T1 & T2 that can be shown co-opaque. In order to make it co-opaque $l_1(ht, k_1, NULL)$ needs to be aborted. And $l_1(ht, k_1, NULL)$ can only be aborted if ***HT-OSTM*** scheduler knows that a conflicting operation $d_2(ht, k_1, v_0)$ has already been scheduled and thus violating co-opacity. One way to have this information is that if the node represented by k_1 records the time-stamp of the delete method so that the scheduler realizes the violation of the time-order [4] and aborts $l_1(ht, k_1, NULL)$ to ensure co-opacity.

Thus, to ensure correctness, we need to maintain information about the nodes deleted from the `hash-table`. This can be achieved by only marking node deleted from the list of `hash-table`. But do not unlink it such that the marked node is still part of the list. This way, the information from deleted nodes can be used for ensuring co-opacity. In this case, after aborting $l_1(ht, k_1)$, we get that the history is co-opaque with $T1$ & $T2$ being the equivalent serial history as shown in Figure 4. The deleted keys (nodes with marked field set) can be reused if another transaction comes & inserts the same key back.



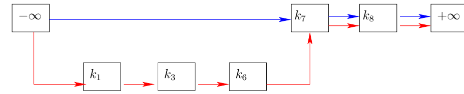


Fig. 5: Searching k_8 over lazylist Fig. 6: Searching k_8 over lazyrb-list

But, the major hindrance in maintaining the deleted nodes as part of the ordinary lazy-list is that it would reduce search efficiency of the data structure. For example, in Figure 5 searching k_8 would unnecessary cause traversal over marked (marked for lazy deletion) nodes represented by k_1, k_3 and k_6. We solve this problem in lazyrb-list by using two pointers. 1) bl(blue link): used to traverse over the actual inserted nodes and 2) rl(red link) used to traverse over the deleted nodes. Hence, in Figure 6 to search for k_8 we can directly use bl saving significant search computations. A question may arise that how would we maintain the time-stamp of a node which has not yet been inserted? Such a case arises when *STM_lookup()* or *STM_delete()* is invoked from rv_method, and node corresponding to the key, say k is not present in *bl* and rl. Then the rv_method will create a node for key k and insert it into underlying data structure as deleted (marked field set) node.

For example, lookup wants to search key k_{10} in Figure 6 which is not present in the *bl* as well as rl. Therefore, lookup method will create a new node corresponding to the

key k_{10} and insert it into rl (refer the Figure 7). So, we discuss in detail the invariants and properties of the lazyrb-list and ensure that no duplicate nodes are inserted while proving the method level correctness in technical report [8].

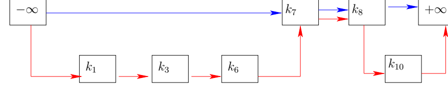


Fig. 7: Execution under lazyrb-list. k_{10} is added in lazyrb-list if not present.

Transaction log. Each transaction maintains local log called $txlog$. It stores transaction id and status: live, commit or abort signifying that transaction is executing, has committed or has aborted due to some method failing the validation, respectively.

Each entry of the $txlog$ is called log_entry (shortened as le) stores the meta information of each method a transaction encounters as $updtSet()$ and $rvSet()$ as formalized in Section 3.2. The le is a tuple of type $\langle key, value, status, preds, currs \rangle$. A method may have OK and $FAIL$ as it's status. The $preds$ and $currs$ are the array of nodes in rl and bl identified during the traversal over the lazyrb-list by each method. It depicts the location over the lazyrb-list where the method would take effect.

***HT-OSTM* Methodology:**
In this section, we provide the working idea of the methods exported to transactions of the *HT-OSTM* and detailed algorithms are provided in the accompanying report [8]. Execution of every transaction T_i can be categorized into *rv_method execution* phase and *upd_method execution* phase.

rv_method execution phase:

1. $\forall\, m_{ij}(k) \in \{$*STM_lookup(), STM_delete()*$\}$
   (a) If legality rule 3.1 is applicable.
      i. update the $txlog$ and return.
   (b) If legality rule 3.2 & 3.3 is applicable.
      i. Traverse the cds to identify pred and curr nodes for both the rl and bl as done in lazy-lists or skip lists. Then, acquire ordered locks on the nodes.
      ii. *Validate*. If the *Validate()* returns $\mathscr{A}$, the $m_{ij}(k)$ aborts and subsequently T_i is aborted. Otherwise, if *Validate()* returns *retry* then $m_{ij}(k)$ is retried from step 1.(b).
      iii. If validation succeeds, create a new le in $txlog$ & update the le. And, insert a node in rl if the node is not present in lazyrb-list as explained in Figure 7.
      iv. Release locks and return.
2. If $m_{ij}(k) \in \{$*STM_insert()*$\}$
   (a) Update the $txlog$ and return.

We validate *STM_lookup()* immediately and do not validate again in *STM_tryC()* unlike the implementation of OTB by Hassan et. al [17]. This is required to ensure that the execution is opaque.

Validate():

1. First the current operation validates for any possible interference due to concurrent transactions through method validation.
   methodValidation rule: If the preds are marked and the next node of pred is not curr, implies a conflicting concurrent operation has also made changes.

Thus, the current operation has to *retry*. Otherwise method validation is said to succeed.

2. Time order validation is performed when method validation succeeds.
   time orderValidation rule [4, Chap 4]: If a transaction T_i with time-stamp i want to access a node n. Also, Let T_j be a conflicting transaction with time-stamp j which accessed n previously. Now, If $i < j$ then T_i is aborted. Else this method returns ok.
3. Return *abort* or *retry* or *ok*.

STM_delete() in *rv_method execution* phase behaves as *STM_lookup()* but it is validated twice. First, in *rv_method execution* similar to *STM_lookup()* and secondly in *upd_method execution* to ensure co-opacity. We adopt lazy delete approach for *STM_-delete()*. Thus, nodes are marked for deletion and not physically deleted for *STM_delete()* method. In the current work we assume that a garbage collection mechanism is present and we do not worry about it.

upd_method execution phase. During this phase a transaction executes *STM_tryC()*. It begins by ordering the $txlog$ in increasing order of the keys. This way locks can be acquired in increasing order of keys to avoid deadlock. We re-validate upd_method in $txlog$ to ensure that the *pred & curr* for the methods has not changed since the point they were logged during *rv_method execution* phase. Please note that $txlog$ only contains the log entry (le) for upd_method. Because we do not validate the lookup and failed delete again in *STM_tryC()*.

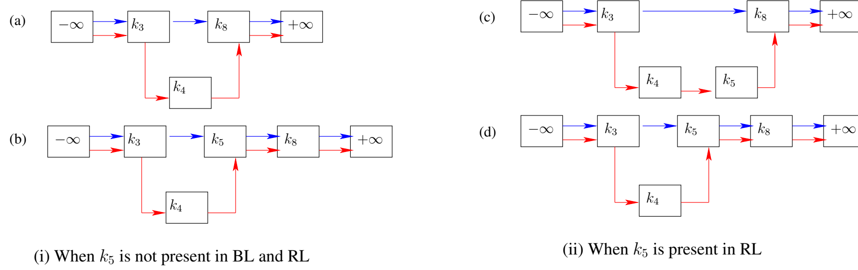


Fig. 8: Insert of k_5 in *STM_tryC()*. (i) bl & rl of k_5 is set to K_8 then bl of k_3 linked to K_5 & rl of k_4 is linked to k_5. (ii) Only bl of k_5 is set to K_8 then bl of k_3 linked to K_5.

Now after successful validation, we update the shared lazyrb-list using the log entries (le) of the $txlog$ one by one. There may be two cases when a node is inserted into lazyrb-list by the *STM_insert()*. First, the node is not reachable by both rl and bl (not present in cds). Figure 8(i) represents this case when k_5 is neither reachable by bl and nor in rl. It adds k_5 to lazyrb-list at location $preds\langle k_3, k_4\rangle$ and $currs\langle k_8, k_8\rangle$ (in the notation, first and second index is the key reachable by bl & rl, respectively). Figure 8(i)(a) is lazyrb-list before addition of k_5 and Figure 8(i)(b) is lazyrb-list state post addition. Second, if the node is reached only by rl. Figure 8(ii) represents this case where k_5 is reached only by rl. It adds k_5 to lazyrb-list at location $pred\langle k_3, k_4\rangle$ and $curr\langle k_5, k_8\rangle$. Figure 8(i)(c) is lazyrb-list before addition of k_5 with bl and Figure 8(i)(d) is lazyrb-list state post addition.

During *STM_delete()* if a node to be removed is reachable with bl then its marked field is set and the links are modified such that it is not reachable by bl. Figure 9 shows a

case where a node k_5 needs to be deleted from the lazyrb-list in Figure 9(i). So, here the node k_5 sets its marked field and then is detached from the bl(Figure 9(ii)).

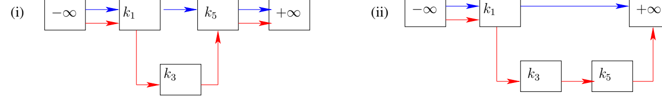


Fig. 9: Delete of k_5 in *STM_tryC()*. k_5 is unlinked from bl by linking bl of k_1 to ∞.

Correctness: In object based STM techniques like *HT-OSTM* where methods are intervals, proving that its methods can be partially ordered or linearized is complex. But, proving the correctness of cds requires taking into account the semantics and implementation details as asserted by work of Hassan et al. [17]. We establish that all methods can be linearized at *method level* before arguing about the co-opacity of *HT-OSTM* history at *transaction level* using graph characterization. The accompanying technical report [8] provides detailed proof, here we only state the major theorem which contributes to proving that *HT-OSTM* is co-opaque.

Theorem 2. *Consider a history H generated by HT-OSTM. Then there exists a sequential & legal history H' equivalent to H such that the conflict-graph of H' is acyclic.*

5 Evaluation

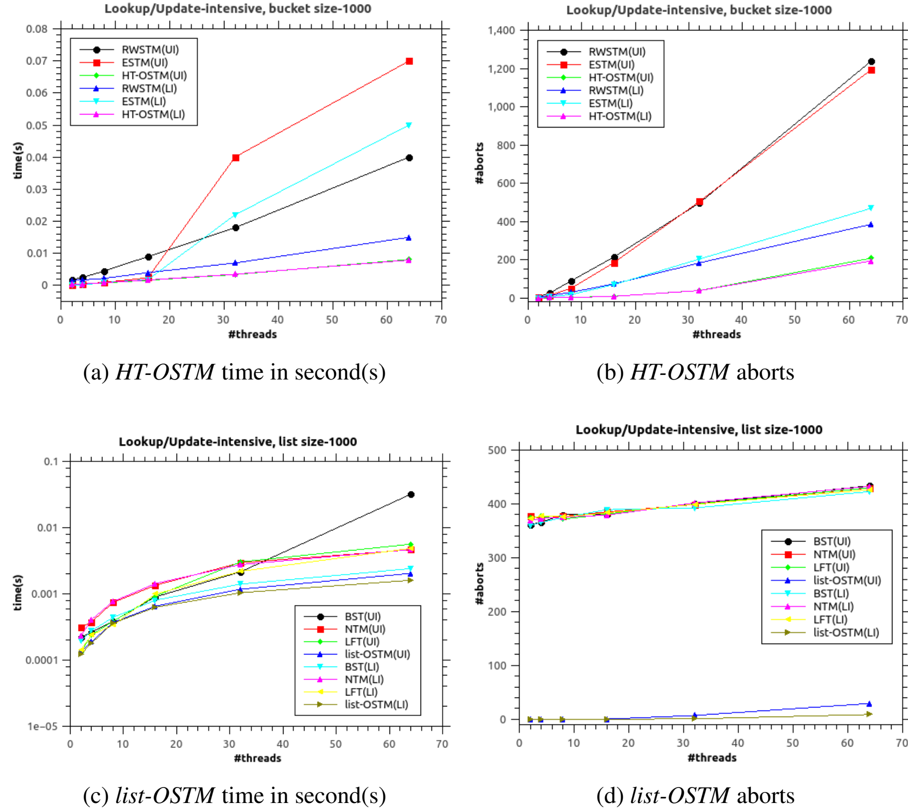
We performed all the experiments on Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz machine with 56 CPUs and 32K L1 data cache and 32 GB memory. Each thread spawns 10 transactions each of which randomly generate up to 5 methods of *HT-OSTM*. We assume that the `hash-table` of *HT-OSTM* has 5 buckets and each of the bucket (or list in case of *list-OSTM*) can have maximum size of 1K keys. We ran the experiments to calculate two parameters: (1) time taken for a transaction to commit. Upon abort, a transaction is retried until it commits. (2) Number of aborts incurred until all the transactions commit.

We compare *HT-OSTM* with the ESTM [12] based `hash-table` and the transactional `hash-table` application built using RWSTM which is synchronised by basic time stamp ordering protocol [4, Chap 4]. Further, we evaluate *list-OSTM* with the state of the art lock-free transactional list (LFT) [13], NOrec STM list (NTM) [14] and boosting list (BST) [15]. All these implementations are directly taken from the TLDS framework[b]. The experiments were performed under two kinds of workloads. Update intensive(lookup:50%, insert:25%, delete:25%) and lookup intensive(lookup:70%, insert:10%, delete:20%). Here, upto 70% lookups *HT-OSTM* performs better but with more than 70% of lookups ESTM shows better performance when contention is higher. The evaluation is done by varying threads from 2 to 64 in power of 2. Before each application is run there is a initialization phase where the data structure is populated randomly with nodes of half its maximum size.

***HT-OSTM*.**[c] Figure 10a shows that w.r.t. time taken *HT-OSTM* outperforms ESTM [12] and RWSTM on an average by 3 times for lookup intensive workload. Plus, for update intensive workload *HT-OSTM* on average is 6 times better than ESTM & RWSTM.

[b] https://ucf-cs.github.io/tlds/
[c] lib source code link: https://github.com/PDCRL/ht-ostm

Similarly, in terms of aborts, *HT-OSTM* has 3 & 2 times lesser aborts than ESTM and RWSTM for lookup intensive workload, respectively. Also for update intensive load *HT-OSTM* has 7 and 8 times lesser aborts with ESTM and RWSTM respectively, as can be seen in Figure 10b.



(a) *HT-OSTM* time in second(s)

(b) *HT-OSTM* aborts

(c) *list-OSTM* time in second(s)

(d) *list-OSTM* aborts

Fig. 10: *HT-OSTM* and *list-OSTM* evaluation. Each curve is named as technique name(workload type). LI/UI denotes lookup intensive/ update intensive.

list-OSTM. The average aborts for *list-OSTM* never go beyond 30 in magnitude while that of other techniques (in Figure 10d) are of 388 in the magnitude for both types of workloads. While time taken is 76%, 89% and 33% (with lookup intensive) and 77%, 77% and 154% (with update intensive) better than LFT, NTM and BST respectively (as shown in Figure 10c).

6 Conclusion and Future Work

In this paper, we build a model for building highly concurrent and composable data structures using object based transactional memory. We use the observation that higher concurrency can be obtained by considering OSTMs as compared to traditional *RWSTMs* by leveraging richer object-level semantics. To achieve this, we propose comprehensive theoretical model based on legality semantics and conflict notions for `hash-table` based OSTM. Using these notions we extend the definition of opacity and co-opacity for

*HT-OSTM*s in Section 3. Then, based on this model, we develop a practical implementation of `hash-table` based object STM, *HT-OSTM*. We then perform some extensive experiments to verify the gains achieved as demonstrated in Section 5. As a part of future work, we plan to develop multi-version object STMs similar to multi-version STMs & databases.

Acknowledgment: We extend our thanks to Dr. Roy Friedman and anonymous reviewers for careful reading of the draft and suggestions. This research is partially supported by the grant from Board of Research in Nuclear Sciences (BRNS), India with project number- 36(3)/14/19/2016-BRNS/36019.

References

1. Herlihy, M., B.Moss, J.E.: Transactional memory: Architectural Support for Lock-Free Data Structures. SIGARCH Comput. Archit. News **21**(2) (1993) 289–300
2. Shavit, N., Touitou, D.: Software Transactional Memory. In: PODC. (1995) 204–213
3. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP, New York, NY, USA, ACM (2005) 48–60
4. Weikum, G., Vossen, G.: Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann (2002)
5. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Elsevier Science (2012)
6. Heller, S., Herlihy, M., Luchangco, V., Moir, M., III, W.N.S., Shavit, N.: A lazy concurrent list-based set algorithm. Parallel Processing Letters **17**(4) (2007) 411–424
7. Guerraoui, R., Kapalka, M.: On the Correctness of Transactional Memory. In: PPoPP, ACM (2008) 175–184
8. Peri, S., Singh, A., Somani, A.: Efficient means of achieving composability using transactional memory. CoRR **abs/1709.00681** (2017)
9. Harris, T., et al.: Abstract nested transactions (2007)
10. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4) (1979)
11. Kuznetsov, P., Peri, S.: Non-interference and local correctness in transactional memory. Theor. Comput. Sci. **688** (2017) 103–116
12. Felber, P., Gramoli, V., Guerraoui, R.: Elastic transactions. J. Parallel Distrib. Comput. **100**(C) (February 2017) 103–127
13. Zhang, D., Dechev, D.: Lock-free transactions without rollbacks for linked data structures. SPAA '16, New York, NY, USA, ACM (2016) 325–336
14. Dalessandro, L., Spear, M.F., Scott, M.L.: Norec: streamlining stm by abolishing ownership records. In Govindarajan, R., Padua, D.A., Hall, M.W., eds.: PPOPP, ACM (2010) 67–78
15. Herlihy, M., Koskinen, E.: Transactional boosting: a methodology for highly-concurrent transactional objects. In: PPoPP, ACM (2008) 207–216
16. Ni, Y., Menon, V.S., Adl-Tabatabai, A.R., Hosking, A.L., Hudson, R.L., Moss, J.E.B., Saha, B., Shpeisman, T.: Open nesting in software transactional memory. In: PPoPP, ACM (2007)
17. Hassan, A., Palmieri, R., Ravindran, B.: Optimistic transactional boosting. In Moreira, J.E., Larus, J.R., eds.: PPoPP, ACM (2014) 387–388
18. Spiegelman, A., Golan-Gueta, G., Keidar, I.: Transactional data structure libraries. In: PLDI, ACM (2016) 682–696
19. Fraser, K., Harris, T.: Concurrent programming without locks. ACM Trans. Comput. Syst. **25**(2) (May 2007)
20. Kuznetsov, P., Ravi, S.: On the cost of concurrency in transactional memory. In: OPODIS. (2011) 112–127
21. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3) (1990) 463–492