Composing Relaxed Transactions

Vincent Gramoli
The University of Sydney
vincent.gramoli@sydney.edu.au

Rachid Guerraoui
EPFL
rachid.guerraoui@epfl.ch

Mihai Letia
EPFL
mihai.letia@epfl.ch

Erratum: the published version contains a citation error, corrected here in the 4th paragraph of the introduction.

***Abstract*—As the classic transactional abstraction is sometimes considered too restrictive in leveraging parallelism, a lot of work has been devoted to devising relaxed transactional models with the goal of improving concurrency. Nevertheless, the quest for improving concurrency has somehow led to neglect one of the most appealing aspects of transactions: software composition, namely, the ability to develop pieces of software independently and compose them into applications that behave correctly in the face of concurrency. Indeed, a closer look at relaxed transactional models reveals that they do jeopardize composition, raising the fundamental question whether it is at all possible to devise such models while preserving composition. This paper shows that the answer is positive.**

We present outheritance, a necessary and sufficient condition for a (potentially relaxed) transactional memory to support composition. Basically, outheritance requires child transactions to pass their conflict information to their parent transaction, which in turn maintains this information until commit time. Concrete instantiations of this idea have been used before, classic transactions being the most prevalent example, but we believe to be the first to capture this as a general principle as well as to prove that it is, strictly speaking, equivalent to ensuring composition.

We illustrate the benefits of outheritance using elastic transactions and show how they can satisfy outheritance and provide composition without hampering concurrency. We leverage this to present a new (transactional) Java package, a composable alternative to the concurrency package of the JDK, and evaluate efficiency through an implementation that speeds up state of the art software transactional memory implementations (TL2, LSA, SwissTM) by almost a factor of 3.

***Index Terms*—transactional memory; multicore processing; scalability**

I. Introduction

One of the most desirable properties in software engineering is *composition*. Basically, pieces of software, called *components*, should be developed and tested independently and then later composed to create larger software pieces and ultimately applications. Szyperski [1] argues that in all engineering disciplines, after having matured, composition has come to play a crucial role. It has now come to software engineering to embrace composition, naming reuse, time to market, quality and viability as some of the key benefits.

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 238639, ITN project TRANSFORM.

Composition in the sequential domain has been studied extensively and techniques such as object oriented programming have proved to be very useful in this regard. However, recent technological trends have introduced concurrency into programming, rendering composition significantly more challenging. Key properties such as *atomicity* and *deadlock freedom* are hard to preserve under composition.

Other research [2], [3] considers *parallel composition* of software. In this view, two operations π and π' are said to be composed when they are executed in parallel by different processes. We consider our work to be complementary in the sense that we reason about *concurrent composition* of software. An operation obtained through concurrent composition invokes a set of simpler (child) operations in sequence just like in the case of sequential composition, but multiple instances of the composed operation can be executed in parallel, just like in the case of parallel composition.

The following simple example from Harris et al. [4] illustrates the difficulty of concurrent composition in the case of lock-based programs: remove and put cannot be composed into a move operation because a concurrent execution with two instances of move, one moving a value from key k to k' and another one from key k' to k would be deadlock-prone. In the same vein, lock-free implementations are generally hard to compose. It is for instance impossible to use the remove and put operations of a hash table to obtain a concurrent move operation that can be used to rebalance the table after a resize [5]. Indeed, the difficulty of composing lock-free operations [6], [7] is a major limitation of the java.util.concurrent package [8] of the JDK. For example, the size method of the ConcurrentSkipListMap class is known to not be atomic, forcing the user to explicitly lock existing sequential data structures in a coarse-grained manner, which then severely hampers concurrency.

A memory *transaction* is an appealing concurrent programming abstraction for it makes programs easily composable [4], [9]–[11]. Composing with transactions simply consists of encapsulating operations inside a new transaction [4], without needing to modify the code, in contrast with techniques based on compare-and-swap [12]. The result is a composition that preserves atomicity and deadlock-freedom. One can furthermore use transactions to compose operations that are themselves obtained through composition, and so on. This modular development process has the potential to greatly simplify the

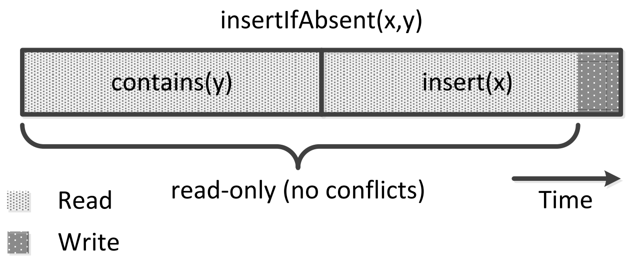


Fig. 1. Composing an insertIfAbsent.

task of the programmer.

Yet, transactions in their classic form are often considered too restrictive [13]. They tend to reduce concurrency by over-conservatively aborting transactions even in executions that would semantically be correct at the application level, should the transactions be actually committed. Several variants of the original transactional abstraction have been proposed, all promoting concurrency by accounting for application semantics when orchestrating transaction interleaving [14]–[19]. These are referred to as *relaxed* (or *weak*) models because they allow more interleavings with the intent of providing better performance. Typically, these models do not blindly reason at the level of memory reads and writes when detecting conflicts between concurrent transactional operations, but rather require from the programmer to somehow encode the way higher-level operations conflict. Zhang et al. [20] show relaxed transactions to significantly outperform classic transactions using realistic workloads such as the STAMP [21] benchmark suite.

However, the attention has mainly been devoted to efficiently implementing these models, forgetting sometimes about one of the most appealing aspects of transactions, namely *composition*. In fact, a closer look reveals that composition can sometimes easily be compromised when relaxing the original transactional model.

To illustrate the composition problem, consider the elastic model [16] designed to take advantage of the semantics of search data structures. Elastic transactions improve concurrency by ignoring conflicts generated by their read-only prefix, as in the case of lists, trees, etc. Now assume a set abstraction having the operations $\mathsf{contains}(\mathsf{x})$ and $\mathsf{insert}(\mathsf{x})$ implemented using elastic transactions, and a programmer willing to compose them to obtain the operation $\mathsf{insertIfAbsent}(\mathsf{x}, \mathsf{y})$. This operation will insert x only if y is not present in the set. The programmer is now required to make a choice about whether to label the transaction as elastic, and he must choose wisely. If he chooses to make it elastic, the insertIfAbsent will not generate any conflicts based on its read-only prefix, including the entire $\mathsf{contains}(\mathsf{y})$ operation, as shown in Figure 1. If such conflicts are ignored, a concurrent transaction could insert y after the insertIfAbsent finds it absent but before inserting x. The result is an execution that violates atomicity. As an alternative, the model allows the programmer to make the insertIfAbsent a regular transaction, thus loosing the performance of having composed elastic insert and contains instead of ones implemented using regular transactions. Whatever the choice, either correctness or performance is sacrificed.

The motivation of our work is to determine whether relaxing the original transaction model inherently hampers composition. We believe the question to be fundamental because, without its ability to facilitate concurrent software composition, the transaction abstraction loses most of its appeal. Relaxed transactions provide a means for experienced programmers to use advanced features of a transactional memory in order to gain performance. It is through composition that novice programmers can reuse these relaxed transactions and improve the efficiency of their programs.

This paper starts by defining a framework to reason about the notion of concurrent composition of software. We believe this framework to be interesting in its own right since, to the best of our knowledge, the problem of concurrent composition has not been studied theoretically. In short, we propose a simple yet precise way to capture the very idea that one should be able to construct a new operation that invokes existing operations in sequence. Given that existing operations behave *correctly* in the face of concurrency, both new and old operations must execute *correctly* in a concurrent setting. In our context, the desired correctness criterion is atomicity.

To the best of our knowledge, the present paper is the first to clearly define a correctness criterion for composing relaxed transactions. We continue by presenting a property we call *outheritance*, which we show to be necessary and sufficient for ensuring composition of relaxed transactions. The property is defined using the notion of a *protected set*. Basically, every transaction t protects certain elements (memory locations, locks, etc.) in order to detect when atomicity might be violated. These elements form what we call the *protected set* of t. In a nutshell, outheritance stipulates that the protected set of the child transactions must be included in the protected set of the parent transaction, to preserve atomicity under composition.

Concrete instantiations of the principle we call outheritance have been used before. Probably the most notable one is represented by classic transactions, using flat nesting. A classic transaction protects all the memory locations it accesses and after commit, they are protected by its parent transaction. However, we believe outheritance to be a general principle that can be used for ensuring atomicity of different types of relaxed transactions or even other synchronization mechanisms. Due to its simple formulation, outheritance can easily be used to check the composability of a relaxed transactional model, as well as when designing a new model, to ensure that it provides composition.

We show however that outheritance can be achieved without necessarily hampering concurrency and performance. We describe our new Software Transactional Memory (STM), called O$\mathcal{E}$-STM, which satisfies outheritance while implementing the elastic transaction model [16]. (Outheritance is by no means tied to any specific transactional model and other models could have been considered instead of the elastic one). We compare our STM to three state-of-the-art ones, TL2 [22], LSA [23] and SwissTM [24]. Our STM speeds up these STMs by up to $2.7\times$ on a 64-hardware-thread machine.

The rest of the paper is organized as follows. Section II introduces our system model and Section III defines composition. Section IV presents outheritance and shows it to be necessary and sufficient for composition. Section V describes the design of our new STM, which we used to build the transactional alternative to the concurrency package of the JDK described in Section VI. Section VII shows the performance of this package. Section VIII reviews related work and Section IX concludes the paper.

II. System model

Our transactional model builds upon that of Weihl [25], which we refine by introducing the notion of *protection element* that abstracts away the conflict detection mechanisms employed by relaxed transactional models. Using this we then proceed to define relax-serializability, a correctness condition for relaxed transactions, as well as a correctness criterion for composing these transactions.

For our purposes, a system is composed of processes and objects. *Objects* represent the state of the system; they provide operations through which processes executing transactions examine and change the system state. Objects are the only mean through which processes can pass information among themselves. We denote by $\mathcal{O}$ the set of objects in the system and for an object $o \in \mathcal{O}$, $o.ops$ is the set of operations provided by the object, while $o.vals$ is the set of return values of these operations. *Processes* are sequential threads of control that change the state of the system by executing *transactions* supplied by a transactional memory. We denote by $\mathcal{P}$ the, potentially infinite, set of processes in the system.

We consider that a *transactional memory* exposes an interface allowing processes to start transactions, invoke operations on objects in the system, and finally attempt to commit the transaction. A transaction can either *commit*, making its changes visible to other transactions, or *abort*, in which case none of its changes are visible. Each transaction has a unique transaction identifier $t \in \mathcal{T}$, where $\mathcal{T}$ is the set of all transaction identifiers. The transactional memory guarantees serializability, or relax-serializability, as defined later in this section. Throughout this work we are interested only in transaction instances and, by abuse of notation, we refer to them simply as transactions.

As we are reasoning about the atomicity properties of a transactional memory system, we are interested in events that occur at the interface between the transactional memory and the objects. To make our reasoning simpler, we also consider virtual events representing the beginning, commit or abort of transactions. Thus we identify several types of events:

- process $p \in \mathcal{P}$ begins transaction $t \in \mathcal{T}$, written $\langle begin(t), p\rangle$;
- transaction $t \in \mathcal{T}$ invokes operation $op \in o.ops$ on object o, written $\langle op, o, t\rangle$;
- operation op of object o invoked by transaction $t \in \mathcal{T}$ terminates with result $v \in o.vals$, written $\langle v, o, t\rangle$;
- transaction $t \in \mathcal{T}$ executed by process $p \in \mathcal{P}$ commits, written $\langle commit(t), p\rangle$;
- transaction $t \in \mathcal{T}$ executed by process $p \in \mathcal{P}$ aborts, written $\langle abort(t), p\rangle$.

As we do not reason about progress properties of the transactional memory, we found no need to separate the $\langle begin(t), p\rangle$, $\langle commit(t), p\rangle$ and $\langle abort(t), p\rangle$ events into invocation and response pairs.

We model an observation of the system as a finite sequence of events and we use the operator $\cdot$ to denote sequence concatenation. We consider the virtual event $\langle begin(t), p\rangle$ to precede the first operation invocation performed by transaction t, while the virtual event $\langle commit(t), p\rangle$ follows the last response received by the transaction.

For a sequence of events H and an object $o \in \mathcal{O}$, we denote by $H|o$ the subsequence of H containing events involving object o. For a transaction identifier $t \in \mathcal{T}$ and a process $p \in \mathcal{P}$, we say that transaction t is executed by process p in event sequence H, if H contains the event $\langle begin(t), p\rangle$. We then denote by $H|p$ the subsequence of H containing the events involving transactions executed by p.

A sequence of events is said to be a *transaction* having transaction identifier t if:

- the first event is a $\langle begin(t), p\rangle$ for some process p;
- the next events are pairs of matching invocation and response events involving transaction t;
- the sequence ends with either a commit event $\langle commit(t), p\rangle$ or an abort event $\langle abort(t), p\rangle$.

Not all event sequences make sense as observations and as such we restrict our attention on sequences where for every process p that appears in H, $H|p$ can be extended by possibly appending a response and a commit event to a sequence of transactions. We refer to these "well-formed" sequences as *histories*.

For a history H, we define $transactions(H)$ to be the set of transactions t such that $\langle begin(t), p\rangle \in H$ for some process p. We then define $committed(H)$ to be the subset of $transactions(H)$ containing all transactions t such that $\langle commit(t), p\rangle \in H$ and $aborted(H)$ as the subset containing all transactions t such that $\langle abort(t), p\rangle \in H$. We also define $live(H)$ as the set $live(H) = transactions(H) \setminus (committed(H) \cup aborted(H))$. We denote by $committed\text{-}ops(H)$ the subsequence of H containing all operation invocation and response events that involve transactions $t \in committed(H)$. As we continue to reason about the correctness of committed and live transactions, we remove from histories all events involving aborted transactions.

In the same way as Weihl [25], we consider the *serial specification* $o.seq$ of an object o to model the acceptable behavior of the object in a sequential environment. If ω is a sequence of pairs $[op, v]$, with $op \in o.ops$ and $v \in o.vals$, then $o.seq$ is the set of all sequences ω that are considered acceptable behavior for the object in a sequential environment.

As concurrency control would not be necessary in systems where all operations trivially commute, we state the following non-triviality condition. We use it when arguing that our outheritance condition is indeed necessary for correct composition. For an object o, a sequence ω of pairs $[op, v]$, with $op \in o.ops$ and $v \in o.vals$, is said to be *trivially commutative* if $\forall \omega', \omega''$ sequences of $[op, v]$ pairs of o, $\omega \cdot \omega' \cdot \omega'' \in o.seq$ if and only if $\omega \cdot \omega'' \cdot \omega' \in o.seq$.

For a history H and any two events $e, e' \in H$, we denote the binary and anti-reflexive relation e' *follows* e in H by $e \prec e'$. By abuse of notation, for two transactions $t, t' \in committed(H)$, if $\langle commit(t), p\rangle \prec \langle commit(t'), p'\rangle$ in H we also say that t' *follows* t in H and we denote it by $t \prec t'$. We say that t' *immediately follows* t in H and denote it by $t \prec^i t'$ if $t \prec t'$ and $\nexists t'' \in committed(H) \setminus \{t, t'\}$ such that $t \prec t'' \prec t'$ in H. We say that two transactions, t executed by process p, and t', executed by p', are *concurrent* in history H if $\langle begin(t), p\rangle \prec \langle begin(t'), p'\rangle$ and $\langle begin(t'), p'\rangle \prec \langle commit(t), p\rangle$ in H. History H is said to be *sequential* if no transactions are concurrent in H.

For a history H we denote by $ops(H)$ the subsequence of H containing all the operation events (invocations and responses). For a sequential history H, we define $opseq(H)$ to be the sequence obtained from $ops(H)$ by mapping all the matching pairs of invocation and response events $\langle op, o, t\rangle, \langle v, o, t\rangle$ to operation, value pairs $[op, v]$. To do the opposite, for an object o and transaction identifier t, we denote by $\omega \mapsto t$ the sequence of events obtained by converting every pair $[op, v] \in \omega$ to the pair of events $\langle op, o, t\rangle, \langle v, o, t\rangle$.

A sequential history H is said to be *legal* if for every object o that appears in H, $opseq(H|o) \in o.seq$. Two histories H and H' are said to be *equivalent* if for every process p, $H|p = H'|p$.

Any history H induces an irreflexive partial order $<_H$ on transactions in H: $t <_H t'$ if $\langle commit(t), p\rangle \prec \langle begin(t'), p'\rangle$ in H. Note that $<_H$ is stricter than $\prec$; indeed $t <_H t'$ implies $t \prec t'$ but not the opposite.

A history H is said to be *serializable* if there exists a legal sequential history S such that:

- *committed-ops*(H) is equivalent to $ops(S)$, and
- $<_H \subseteq <_S$.

Note that we will be considering the strict form of serializability [26] for the course of this work.

A. The protection element abstraction

As classic transactional semantics proved too restrictive for concurrent data structures such as lists and trees [16], *relaxed transactions* have been designed to take advantage of the extra concurrency by ignoring some conflicts. To illustrate, consider a linked list along with an add operation that goes from the head towards the tail of the list and inserts an element at some position. Now an add operation, implemented using a classic transaction, is inserting an element somewhere in the middle of the list, while in the meantime another add is modifying the head of the list. In this situation, most implementations would cause one of the transactions to unnecessarily abort in order to avoid the complex detection of cycles in the conflict graph [13] and still guarantee serializability. However, an elastic transaction [16] would consider this to be a false conflict and hence allow both transactions to commit since semantically the execution does not violate atomicity. This type of transaction ignores conflicts caused by its read-only prefix.

In order to reason about relaxed transactions we introduce the notion of a *protection element*, an abstract entity used to model different existing conflict detection schemes. To each object $o \in \mathcal{O}$ we associate a protection element $\epsilon(o)$ that is *acquired* by transaction t before invoking an operation $op \in o.ops$. Transaction t will then *release* $\epsilon(o)$ when the conflict becomes benign. Between the acquisition of $\epsilon(o)$ by t and its release, we say that $\epsilon(o)$ belongs to the protected set of t, denoted by $P(t)$. Informally, a transaction maintains a protected set in order to detect conflicts between operations it has already applied and operations applied by other concurrent transactions. In Section IV we show that passing the protected set to the parent at commit time is a necessary and sufficient condition for ensuring composition.

An important note is that we chose not to include commutativity in our model. Indeed it would be a simple extension to associate a protection element to each operation of the object instead of the object itself. Then a transaction executing an operation must acquire the protection element associated to the operation as well as those associated to other operations of the object that do not commute with that operation. We chose not to include this extension as it would make expressing our idea more complicated while bringing no extra insight.

We therefore extend histories by adding two types of events: the acquisition of protection element $\epsilon(o)$ by process p, denoted by $\langle a(\epsilon(o)), p\rangle$, and the matching release event $\langle r(\epsilon(o)), p\rangle$. In order to be as general as possible, we only require that in any history H, the invocation and response of any operation $op \in o.ops$, invoked by transaction t executed by process p, be between a pair of acquire and next matching release event of protection element $\epsilon(o)$ by process p, $\langle a(\epsilon(o)), p\rangle \prec \langle op, o, t\rangle \prec \langle v, o, t\rangle \prec \langle r(\epsilon(o)), p\rangle$. We do not allow an acquire or release event between the last response event of a transaction t and the commit event of t. For a history H and a protection element $\epsilon(o)$, we denote by $H|\epsilon(o)$ the subsequence of H containing the acquire and release events involving $\epsilon(o)$.

Using the mechanism of protection elements, we model transactions that do not require all their operations to appear to execute as a single atomic unit. Before invoking an operation on an object, the transaction must acquire the corresponding protection element, which is then released when the conflict becomes benign. This could be anywhere between just after the response of the operation and the commit of the transaction.

In the case of classic transactions, the protection element associated with a memory location is acquired right before reading or writing the location and released after the commit of the transaction. In order to model transactional memories using deferred updates, we consider the protection element to be acquired at the point where the invocation was received by the transactional memory, even though the actual invocation on the object is performed at commit time. For modeling the early release mechanism of DSTM [14], the protection element is released when the release operation of the transactional memory is called, while for elastic transactions, it is released after a new protection element is acquired.

For a history H and a transaction $t \in committed(H)$ executed by process p, the *minimal protected set* of t, denoted by $P^{min}(t)$ is the set of protection elements $\epsilon(o)$ for which $\langle begin(t), p\rangle \prec \langle a(\epsilon(o)), p\rangle \prec \langle commit(t), p\rangle \prec \langle r(\epsilon(o)), p\rangle$. In other words, the minimal protected set contains the protection elements that are acquired by process p during the execution of transaction t and are not released at the time when t commits. We also define the *kernel* of transaction t as the set $ker(t) = \{o \in \mathcal{O} | \epsilon(o) \in P^{min}(t)\}$.

The notion of protection element is more general than a lock and it can model any way in which a transactional memory detects conflicts between transactions. For example, an invisible read also needs to acquire a protection element corresponding to the respective location, meaning that the transaction will recheck the location for consistency before committing, or until it releases the protection element.

The minimal protected set of a transaction t ensures that the abstract postcondition of t is not violated until the elements of the set are released. The content of this set does not depend only on the data structure and semantics of t but also on other transactions that can be executed concurrently. To illustrate, consider a set abstraction S, implemented with a linked list, where transaction t is performing an $\mathsf{insert}(x)$. If the only other possible concurrent transactions are some $\mathsf{remove}(x')$ and $\mathsf{contains}(x'')$, it is safe to consider $P^{min}(t)$ to be the element preceding x. The postcondition $x \in S$ cannot be violated without modifying the element preceding x. However, if we consider that a concurrent transaction can perform an $\mathsf{empty}()$ operation that removes all elements from the list by setting the $head$ pointer to null, $P^{min}(t)$ must be reevaluated because the $\mathsf{empty}()$ operation can remove x without changing the element preceding it.

The reason for which operations sometimes need to acquire elements outside of their minimal protected set is that for many search structures such as lists, trees, graphs, etc., the minimal protected set is not known from the start and the operation needs to find the data and the elements from its minimal protected set.

B. Relax-serializability

As serializability proved too restrictive for efficiently implementing concurrent data structures such as lists and trees, we continue to define relax-serializability, the correctness criterion that we use to reason about relaxed transactions. This condition is weaker than classical serializability and allows us to obtain executions that, although not serializable, are correct at the application level. In a nutshell, relax-serializability allows transactions to be interleaved at a finer granularity. Transactions can be interleaved as long as the acquire and release events involving the same protection element are not.

A history H is said to be *relax-serial* if for every protection element $\epsilon(o)$ that is acquired or released in H, the sequence $H|\epsilon(o)$ is a sequence of pairs of matching acquire and release events, starting with an acquire event. History H is said to be *relax-serializable* if there exists a legal relax-serial history S such that:

- $committed\text{-}ops(H)$ is equivalent to $ops(S)$, and
- $<_H \subseteq <_S$.

We say that a history H contains *relaxed transactions* if H is relax-serializable but not serializable. Note that since relaxation is a property of a history, we cannot say if a single transaction is relaxed or which transaction from a history is relaxed.

To show that relax-serializability allows for more correct histories compared to classic serializability, consider the following history from which we have omitted events showing transaction begin and commit as well as operation response events.

$$\langle a(\epsilon(o_1)), p_1\rangle, \langle read, o_1, t_1\rangle, \langle a(\epsilon(o_2)), p_1\rangle, \langle read, o_2, t_1\rangle,$$
$$\langle r(\epsilon(o_1)), p_1\rangle, \langle a(\epsilon(o_1)), p_2\rangle, \langle write, o_1, t_2\rangle, \langle a(\epsilon(o_3)), p_2\rangle,$$
$$\langle read, o_3, t_2\rangle, \langle r(\epsilon(o_1)), p_2\rangle, \langle r(\epsilon(o_3)), p_2\rangle, \langle a(\epsilon(o_3)), p_1\rangle,$$
$$\langle write, o_3, t_1\rangle, \langle r(\epsilon(o_2)), p_1\rangle, \langle r(\epsilon(o_3)), p_1\rangle$$

This history is relax-serial since every protection element is released before being acquired. However, there is no serial history equivalent to it since this would imply both $\langle read, o_1, t_1\rangle \prec \langle write, o_1, t_2\rangle$, leading to $t_1 < t_2$, as well as $\langle read, o_3, t_2\rangle \prec \langle r(\epsilon(o_3)), p_1\rangle$, leading to $t_2 < t_1$. This history is therefore not serializable.

III. Composition

To better capture the intuition behind composition, we use as an example the Collection interface from the JDK, used to represent a group of objects. This interface has methods to add or remove elements from the group, check if an element belongs to the group, and so on, and is implemented by a number of classes such as HashSet, TreeSet, LinkedList, etc. Assume that a programmer, say Alice, starts writing a class implementing this interface, but she only implements the methods that she needs in her program, leaving the others as stubs. As such, Alice correctly implements add, remove and contains, but she does not implement others methods, such as addAll, that adds several elements to the group in a single atomic step.

Now a second programmer, Bob, gets the code written by Alice, wants to reuse it, but Bob also needs the addAll method. Bob quickly thinks that he can implement the addAll method by calling the add method for each of the elements that he wants to add and decides to use a loop to accomplish his task. Bob is relying on the correctness of the methods implemented by Alice but also on the fact that when he puts together these correct building blocks, the result is also correct. An important observation is that the newly created addAll method must behave as intended when other operations such as add and remove are executed concurrently. This issue is specific to concurrent programming.

As another example, Bob could use the add and remove methods written by Alice to implement a move operation that moves an element between two collections. Bob would again be relying on the fact that the whole, here the move, is equal to the sum of the parts, the add and remove. Bob would of course like his move operation to be able to run concurrently with instances of add and remove.

Our definition of composition captures the concept of creating a new operation by using existing operations as building blocks, and having the new operation invoke the existing ones. Starting with a set of atomic operations, the programmer would be able to compose them to obtain new atomic operations.

For a history H, a set of transactions $C \subseteq committed(H)$ is said to be a *composition* of process p if:

- all $t \in C$ are executed by p, and
- $\forall t \in C$, either $\exists t' \in C$ such that $t \prec^i t'$ in history $H|p$ or $\forall t' \in C \setminus \{t\}$, $t' \prec t$ in $H|p$; in the latter case, t is the supremum of C, denoted by $Sup(C)$.

We impose the additional requirement that $|C| \geq 2$, where $|C|$ denotes the cardinal of C. We do this because it does not make sense to have an empty composition or one containing a single transaction.

Definition 3.1 (Strongly composable): Let H be a history and C a composition over H. History H is said to be strongly composable with respect to C if there exists a relax-serial history S such that all of the following hold:

- $ops(S)$ is equivalent to $committed\text{-}ops(H)$,
- $<_H \subseteq <_S$,
- $\forall t_i, t_j \in C$ with $t_i \prec t_j$, $\nexists t_k \in transactions(S) \setminus C$, such that $t_i \prec t_k \prec t_j$ in S.

Informally, the above definition requires all transactions t_i and t_j from composition C to appear to execute one immediately after the other when observed from any object in the system. One might consider this to be a reasonable correctness condition for composing relaxed transactions. However, in the next section we show that this condition is too strong and therefore we relax it by only requiring t_i and t_j to appear one immediately after the other when observed from objects $o \in ker(t_i)$, a condition we call weak composability.

Definition 3.2 (Weakly composable): Let H be a history and C a composition over H. History H is said to be weakly composable with respect to C if there exists a relax-serial history S such that all of the following hold:

- $ops(S)$ is equivalent to $committed\text{-}ops(H)$,
- $<_H \subseteq <_S$,
- $\forall t \in C$ and $\forall o \in ker(t)$, $\nexists t' \in transactions(S) \setminus C$ such that $t \prec t' \prec sup(C)$ in history $S|o$.

If $\mathcal{C}$ is a set of compositions, a relaxed-sequential history H is said to be strongly (weakly) *composition-consistent* with respect to $\mathcal{C}$ if $\forall C \in \mathcal{C}$, H is strongly (weakly) composable with respect to C.

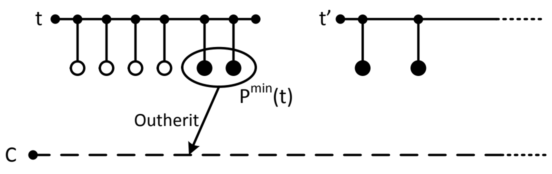


Fig. 2. Outheritance: minimal protected set is passed to the composed transaction.

IV. Outheritance

We now define *outheritance* and, although it is not sufficient for ensuring strong composition (Theorem 4.2), we show it to be both necessary (Theorem 4.3) and sufficient (Theorem 4.4) for ensuring that a (potentially relaxed) transactional memory provides weak composition.

Definition 4.1 (Outheritance): A history H is said to satisfy outheritance with respect to composition C executed by process p if, $\forall t \in C$ and $\forall \epsilon(o) \in P^{min}(t)$, $\nexists e = \langle r(\epsilon(o)), p\rangle$ such that $\langle commit(t), p\rangle \prec e \prec \langle commit(Sup(C)), p\rangle$ in H.

Informally, outheritance prevents each protection element from the minimal protected set of each transaction in C from being released before the commit of the last transaction in C. Figure 2 shows transaction t passing its minimal protected set $P^{min}(t)$ containing two elements to composition C. These protection elements will not be released until the last transaction in C commits. For a transactional memory to satisfy outheritance, it needs to ensure that all the produced histories satisfy outheritance. This is done by making the transactions being composed pass their protection elements (be they locks or something else) to their parent transaction, which in turn will hold them until it commits.

Some relaxed transactional models such as that of Felber et al. [16] do not satisfy outheritance and therefore can break composition. Indeed, one can compose two elastic transactions inside another elastic transaction, causing the protection elements of the first composed transaction to be released as soon as it commits instead of passing them to the resulting transaction, situation depicted in Figure 1. Since this practice can produce executions that are not atomic, the authors provide a workaround, namely by advising the programmer to predict these situations and use regular mode when composing.

A. Outheritance and strong composition

In this section we prove that outheritance is not a sufficient condition for ensuring that a relaxed transactional memory ensures strong composition.

Theorem 4.2: There exists a history H and a composition C over H such that H satisfies outheritance with respect to C but does not satisfy strong composition with respect to C.

Proof: We perform this proof by construction. History H contains three transactions, t_1, t_2 and t_3, with t_1 and t_3

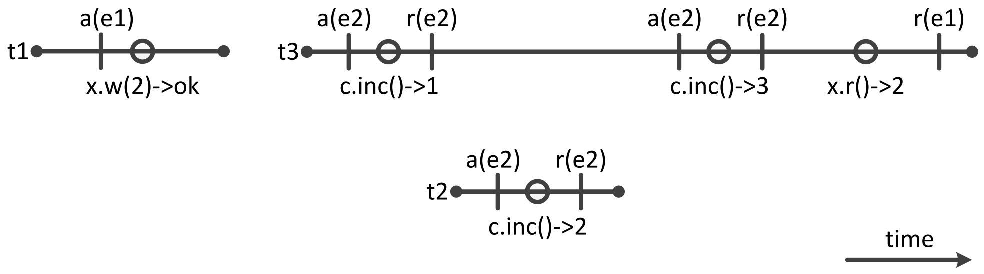


Fig. 3. Execution that satisfies outheritance but does not satisfy strong composition.

executed by process p_1 and t_2 executed by p_2 such that $t_1 <_H t_2$ and $t_1 <_H t_3$. We also consider composition $C = \{t_1, t_3\}$ over H. Let H be the following history:

$$H = \langle begin(t_1), p_1\rangle, \langle a(\epsilon_1), p_1\rangle, \langle w(2), x, t_1\rangle, \langle ok, x, t_1\rangle,$$

$$\langle commit(t_1), p_1\rangle, \langle begin(t_3), p_1\rangle, \langle a(\epsilon_2), p_1\rangle, \langle inc(), c, t_3\rangle,$$

$$\langle 1, c, t_3\rangle, \langle r(\epsilon_2), p_1\rangle, \langle begin(t_2), p_2\rangle, \langle a(\epsilon_2), p_2\rangle, \langle inc(), c, t_2\rangle,$$

$$\langle 2, c, t_2\rangle, \langle commit(t_2), p_2\rangle, \langle r(\epsilon_2), p_2\rangle, \langle a(\epsilon_2), p_1\rangle,$$

$$\langle inc(), c, t_3\rangle, \langle 3, c, t_3\rangle, \langle r(\epsilon_2), p_1\rangle, \langle r(), x, t_3\rangle, \langle 2, x, t_3\rangle,$$

$$\langle commit(t_3), p_1\rangle, \langle r(\epsilon_1), p_1\rangle.$$

This situation is depicted in Figure 3.

If we combine $t_1 <_H t_3$ with $t_1 <_H t_2$ we obtain two possible orderings, $t1 \prec t_2 \prec t_3$ and $t_1 \prec t_3 \prec t_2$. Since $t_1 \prec t_2 \prec t_3$ does not satisfy composition C, the only possibility we are left with is $t_1 \prec t_3 \prec t_2$.

As such, history H is equivalent to history S, where $t_1 \prec t_3 \prec t_2$. This makes $\langle inc(), c, t_2\rangle \prec \langle 2, c, t_2\rangle \prec \langle inc(), c, t_3\rangle \prec \langle 3, c, t_3\rangle$ in S and $opseq(S|c) = [inc, 1]$, $[inc, 3], [inc, 2]$. However $opseq(S|c) \notin c.seq$ and therefore S is not a legal history.

We therefore conclude that H is not equivalent to any relax-serial history S such that $<_H \subseteq <_S$ and $t_1 \prec^i t_3$, i.e. H is not strongly composable with respect to C and our proof is complete. ∎

B. Ensuring weak composition

In this section we prove that outheritance is both necessary (Theorem 4.3) and sufficient (Theorem 4.4) for ensuring that a (potentially relaxed) transactional memory ensures weak composition.

Theorem 4.3: For any history H and any composition C over H such that $\exists t \in (C \cap live(H))$, with t executed by process p, and H satisfies outheritance with respect to C, if we extend H to history $H' = H \cdot \langle r(\epsilon(o)), p\rangle$ such that H' does not satisfy outheritance with respect to C and $opseq(H|o)$ is not trivially commutative, then H' can be extended to history H'' that is not weakly composable with respect to C.

Proof: If $H' = H \cdot \langle r(\epsilon(o)), p\rangle$ does not satisfy outheritance with respect to C while H does satisfy it, we deduce that $\epsilon(o)$ is part of the minimal protected set $P^{min}(t')$ of some $t' \in (C \cap committed(H))$.

Since $opseq(H|o)$ is not trivially commutative, there exist two sequences of operations of o, ω_o and ω'_o such that $opseq(H|o) \cdot \omega_o \cdot \omega'_o \in o.seq$ but $opseq(H|o) \cdot \omega'_o \cdot \omega_o \notin o.seq$. Hence we can append to H' a new transaction t'' executed by some process p', $\langle begin(t''), p'\rangle, \langle a(\epsilon(o)), p'\rangle \cdot \omega_o \mapsto t'' \cdot \langle commit(t''), p'\rangle, \langle r(\epsilon(o)), p'\rangle$. We can now complete transaction t by appending $\langle a(\epsilon(o)), p\rangle \cdot \omega'_o \mapsto t \cdot \langle commit(t), p\rangle, \langle r(\epsilon(o)), p\rangle$ to the resulting sequence and we obtain H''.

If S is a relax-serial history such that *committed-ops*(H) is equivalent to $ops(S)$ and $<_H \subseteq <_S$, then *committed-ops*(H'') is equivalent to $ops(S')$ for

$$S' = S \cdot \langle r(\epsilon(o)), p\rangle, \langle begin(t''), p'\rangle, \langle a(\epsilon(o)), p'\rangle \cdot \omega_o \mapsto t'' \cdot$$

$$\langle commit(t''), p'\rangle, \langle r(\epsilon(o)), p'\rangle, \langle a(\epsilon(o)), p\rangle \cdot \omega'_o \mapsto t \cdot$$

$$\langle commit(t), p\rangle, \langle r(\epsilon(o)), p\rangle$$

that is relax-serial but does not satisfy composition with respect to C since $t' \prec t'' \prec t$ in history $S'|o$.

History *committed-ops*(H'') is also equivalent to $ops(S'')$ for

$$S'' = S \cdot \langle r(\epsilon(o)), p\rangle, \langle a(\epsilon(o)), p\rangle \cdot \omega'_o \mapsto t \cdot \langle commit(t), p\rangle,$$

$$\langle r(\epsilon(o)), p\rangle, \langle begin(t''), p'\rangle, \langle a(\epsilon(o)), p'\rangle \cdot \omega_o \mapsto t'' \cdot$$

$$\langle commit(t''), p'\rangle, \langle r(\epsilon(o)), p'\rangle$$

and $t' \prec t \prec t''$ in history $S''|o$, but S'' is not legal since $opseq(S''|o) = opseq(H|o) \cdot \omega'_o \cdot \omega_o \notin o.seq$. ∎

According to Theorem 4.3, it is necessary for all histories produced by a transactional memory to satisfy outheritance in order to ensure composition. Informally, the proof argues that releasing a single protection element $\epsilon(o)$ that belongs to $P^{min}(t')$ such that the history obtained violates outheritance, is enough to produce a history that is not weakly composable with respect to C and thus violates correctness. The condition of not having operations trivially commute is necessary for excluding cases where all executions are correct even without concurrency control. Systems where all operations trivially commute are not particularly interesting for concurrency control.

Theorem 4.4: Any relax-serializable history H that satisfies outheritance with respect to composition C is weakly composable with respect to C.

Proof: We perform this proof by contradiction. Assume there exists a relax-serializable history H that satisfies outheritance with respect to C but is not weakly composable with respect to C.

Let p be the process that executes composition C. Since H is relax-serializable, then there exists a relax-serial history S such that $ops(S)$ is equivalent to $committed\text{-}ops(H)$ and $<_H \subseteq <_S$. Since S is equivalent to $committed\text{-}ops(H)$, it follows that $S|p = H|p$. And since H satisfies outheritance with respect to C, we have that $\forall t \in C$ and $\forall \epsilon(o) \in P^{min}(t), \nexists e = \langle r(\epsilon(o)) \rangle$ such that $\langle commit(t), p \rangle \prec e \prec \langle commit(Sup(C), p) \rangle$ in H. And because $\langle commit(t), p \rangle, e$ and $\langle commit(Sup(C), p) \rangle$ all involve process p, it follows that $\langle commit(t), p \rangle \prec e \prec \langle commit(Sup(C), p) \rangle$ in history $H|p$ and in $S|p$, since $S|p = H|p$ and finally in history S. We therefore have that relax-serial history S satisfies outheritance with respect to composition C.

Since H is not weakly composable with respect to C, but S is relax-serial, $ops(S)$ is equivalent to $committed\text{-}ops(H)$ and $<_H \subseteq <_S$, it follows that $\exists t, t' \in C, t'' \in transactions(S) \setminus C$, and $\epsilon(o) \in P^{min}(t)$ such that $t \prec t'' \prec t'$ in history $S|o$. Since S is a relax-serial history and $\forall o \in ker(t), \langle commit(t'), p \rangle \prec \langle r(\epsilon(o)), p \rangle$, then $\nexists e = \langle a(\epsilon(o)), p' \rangle$ such that $\langle commit(t), p \rangle \prec e \prec \langle commit(t'), p \rangle$. Therefore $\nexists e' = \langle op, o, t''' \rangle$ in S such that $\langle commit(t), p \rangle \prec e' \prec \langle commit(t'), p \rangle$. It follows that in history $S|o$, $\nexists e'' = \langle op, o, t''' \rangle$ such that $commit(t) \prec e'' \prec commit(t')$. We conclude that history $S|o$ is equivalent to history S' where $commit(t''') \prec commit(t) \prec commit(t')$ and we have reached a contradiction. ■

According to Theorem 4.4, it is sufficient for all histories produced by a transactional memory to satisfy outheritance in order for composition to be ensured.

V. O$\mathcal{E}$-STM

We briefly present here our software transactional memory, O$\mathcal{E}$-STM (Outheritance-Elastic STM) that allows programmers to compose relaxed (elastic) transactions while preserving both atomicity and performance. It is largely based on $\mathcal{E}$-STM of Felber et al. [16], which we modified in order to satisfy outheritance and thus correctly compose.

The elastic transaction model allows the programmer to use either an elastic or a regular transaction for implementing an operation, depending on the semantics of that operation. An *elastic transaction* ignores all conflicts induced by its read-only prefix, i.e., all conflicts involving its reads that precede its first write access. In the implementation, an elastic operation is executed optimistically and during its execution it temporarily keeps track of the immediate past read access while ensuring that each read returns a consistent value. Upon writing, the transaction starts keeping track permanently of all accesses including the immediate past read. At commit-time, the transaction checks that the access sequence it kept track of appears to be atomic and decides to commit or abort accordingly. More precisely, if a transaction π invokes only read operations, then the minimal protected set of π is $P^{min}(\pi) = \{r_n\}$, where r_n is the last memory location read by π. Otherwise, if r_k is the first memory location written by transaction π, its minimal protected set is $P^{min}(\pi) = \{r_k, \dots, r_n\}$, where r_n is the last memory location accessed by π.

```
1: outherit()$_t$:
2:   if $t_{parent} \neq \perp$ then
3:       $t_{parent}$.add-to-protected-set($read\text{-}set$, $last\text{-}read\text{-}entry$, $write\text{-}set$)

4: add-to-protected-set($r\text{-}s$, $l\text{-}r$, $w\text{-}s$)$_t$:
5:   $r\text{-}set \leftarrow r\text{-}set \cup r\text{-}s \cup l\text{-}r$
6:   $w\text{-}set \leftarrow w\text{-}set \cup w\text{-}s$
```

Fig. 4: Changes to elastic transactions to satisfy outheritance.

In order to satisfy outheritance, elastic transactions must pass their protected set to their parent transaction when committing. More concretely, they need to add the read set as well as the last read memory location into the read set of the parent transaction and also add the write set to that of the parent. Figure 4 presents the pseudocode that needs to be added to $\mathcal{E}$-STM in order to satisfy outheritance. The outherit() function must be invoked by every transaction before invoking the usual commit function of $\mathcal{E}$-STM. This function first checks if the transaction has a parent, and if so, invokes the add-to-protected-set function of the parent, to which it passes its read set, last read location and write set. The parent transaction then proceeds to add them to its own read and write set.

VI. Illustration: a Java Transactional Package

We illustrate the importance of composition (and thus outheritance) for a relaxed transactional model by building a highly-concurrent composable Java package, called e.e.c (edu.epfl.compositional). Our solution provides composition, unlike the similar j.u.c (java.util.concurrent) package [8]. Our implementation performs very well compared to other composable alternatives, as shown in Section VII.

a) The java.util.concurrent package.: Although this package provides invaluable low level atomic primitives for concurrent programming, it is not composable and several of its methods violate atomicity. For instance, the JDK6 documentation describes the ConcurrentSkipListSet by saying that "the bulk operations addAll, removeAll [...] are not guaranteed to be performed atomically" while the ConcurrentLinkedQueue iterator is said to be "weakly consistent". This lack of atomicity makes it hard to reason about the semantics of these methods. For example, the concurrent execution of removeAll and addAll that both take as argument a Collection containing integers 1 and 2 may lead to an inconsistent state where only one of the two integers is present.

To compose the methods of these lock-free algorithms while preserving atomicity, the modifications could apply speculatively on some copy of the whole data structure before a current copy pointer could be compared-and-swapped from the former copy to the new one, provided that no concurrent accesses were executed. The time and space complexity of such a solution justified the implementation of non-atomic operations in java.util.concurrent.

```
SkipListSet
add(Value val)
  begin[relaxed]
      Node[] preds = new Node[topLevel+1];
      Node curr = head;
      Node next = curr.getNext(topLevel);
      for (int l = topLevel; l>0; l--) {
        next = curr.getNext(l);
        while (next.getVal() < val) {
          curr = next;
          next = curr.getNext(l);
        }
        preds[l] = cur;
      }
      if (next.getVal() != val) {
        node = new Node(getVal(), getRndLevel());
        for (int j=0; j<topLevel; j++) {
          node.getNext(j) = preds[j].getNext(j);
          preds[j].setNext(j, node);
        }
      }
      return (next.getVal() != val);
  end

addAll(Collection c)
  begin
        boolean result = false;
        for (Value x : c) result |= this.add(x);
        return result;
  end
```

Fig. 5: The pseudocode for operations add and addAll of SkipListSet of the e.e.c.

b) The edu.epfl.compositional package.: Figure 5 depicts the atomic operations add and addAll of SkipListSet, the skip list implementation of a set abstraction in e.e.c.

One can see that the add implementation is similar to its sequential counterpart: no locks or synchronization primitives are exposed to the programmer. What makes the code concurrent are the region delimiters begin[relaxed] and end indicating that the region should execute as a relaxed transaction. All reads and writes are then instrumented automatically as long as they appear in the delimited region. Section VII further details automatic transactional instrumentation in Java.

The addAll operation performs a series of invocations of add, delimited by begin and end. This code is almost identical to the one used when composing sequential programs. No modification to the add operation are needed and the addAll executes correctly and efficiently.

VII. Evaluation

We evaluate our transactional memory implementation, O$\mathcal{E}$-STM, using as benchmark the new Java package, edu.epfl.compositional (e.e.c), that we introduced in Section VI. In short, our e.e.c package is a composable alternative to the JDK concurrency package. This package provides classes having set operations contains, add and remove, as well as operations resulting from their composition such as removeAll and addAll that are all part of the Java Collection interface. Although these composed operations tend to limit concurrency, due to their results depending on elements located at different places in the data structure, we show that our solution scales well with the level of parallelism and performs better than other state-of-the art STMs.

A. Experimental setting

We compare O$\mathcal{E}$-STM against bare sequential code as well as state-of-the-art STMs using an UltraSPARC T2 with 8 cores, each running up to 8 hardware threads. For each run we average the number of executed operations per millisecond and aborts (for STMs) over 10 runs of 10 seconds each. We use Java SE 1.6.0_12-ea in server mode and HotSpot JVM 11.2-b01. All our workloads comprise 20% attempted updates on a data structure containing 2^{12} elements. More precisely, each add/remove picks a random value among a range of 2^{13} for a success rate of $1/2$, while each addAll/removeAll takes one value v in the same range and a second value as the closest integer to $v/2$. The rest is made up 80% of contains operations.

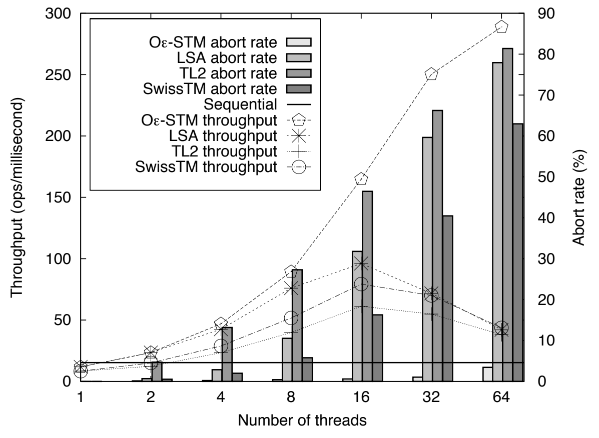
B. Performance comparison

TL2 [22] is an efficient STM featuring writes that are not visible before commit-time and timestamp intervals for validating transactions at commit-time; LSA [23] relies on a lazy snapshot algorithm that uses eager lock acquirement and extends the validity interval of the transaction as much as possible in order to increase concurrency; SwissTM [24] builds upon LSA while adding mixed eager and lazy conflict resolution to abort as soon as possible while trying to maximize throughput.

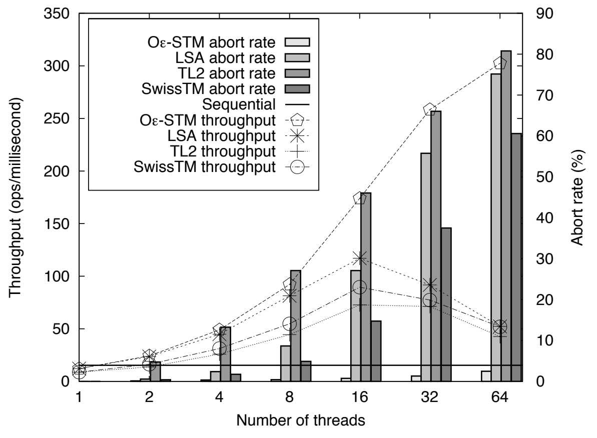
Our STM relies on bytecode instrumentation framework Deuce [27], which instruments delimited Java accesses using the transactional read/write functions defined by the transactional memory. For the sake of comparison, we reused the existing Java version of LSA, TL2 and we implemented SwissTM and O$\mathcal{E}$-STM. To improve concurrency, all STMs protect memory locations at the granularity level of object fields.

We report the throughput as the number of operations performed per millisecond as well as the abort ratio obtained when using three data structures, LinkedListSet (Figure 6), SkipListSet (Figure 7), HashSet (Figure 8). An interesting observation is that the throughput does not drop when increasing the ratio of addAll/removeAll operations from 5% to 10%. O$\mathcal{E}$-STM offers a higher throughput than other STMs at a high level of parallelism, while having similar performance at low parallelism. The cause for the latter effect might be due to the heavy metadata management needed by relaxed transactions.

The abort rate obtained on the linked list benchmark (Figure 6) is significantly higher for classic transactions (LSA, TL2, SwissTM) than it is for relaxed transactions, thus motivating the need for relaxed STMs that provide composition, such as O$\mathcal{E}$-STM. Consequently, O$\mathcal{E}$-STM has a much higher

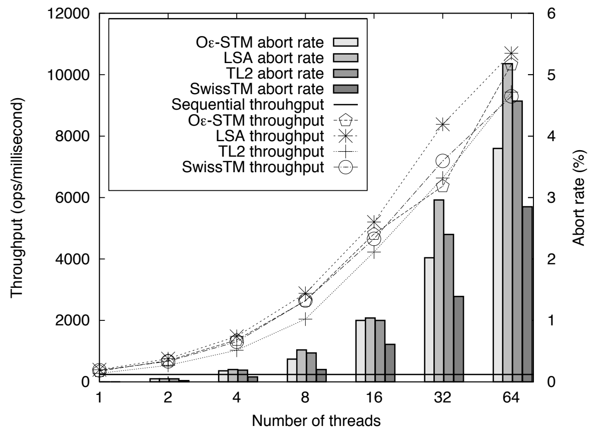


(a) 5 % update

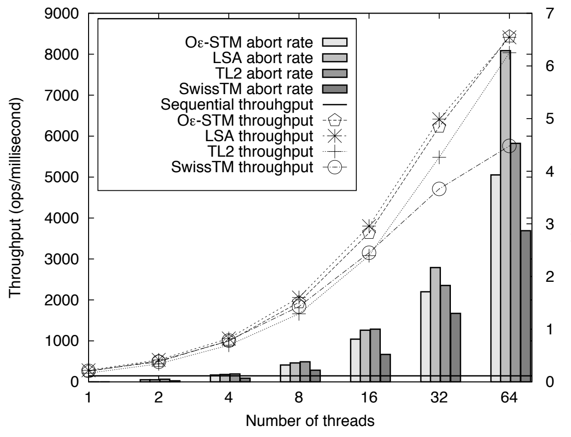


(b) 15 % update

Fig. 6: Throughput and abort ratio of bare sequential code, O$\mathcal{E}$-STM, LSA, TL2 and SwissTM on the LinkedListSet of e.e.c when running 5% (left) and 15% (right) of addAll/removeAll.

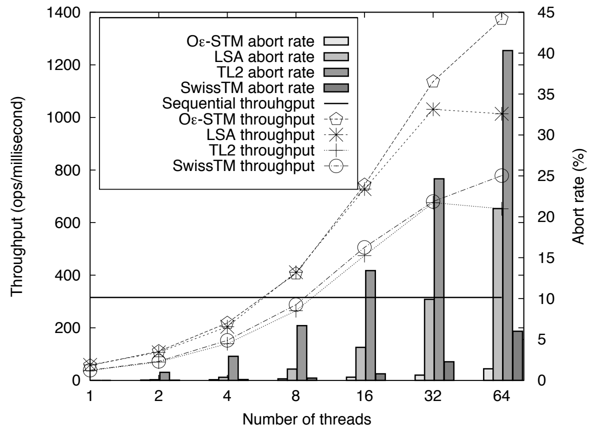


(a) 5 % update

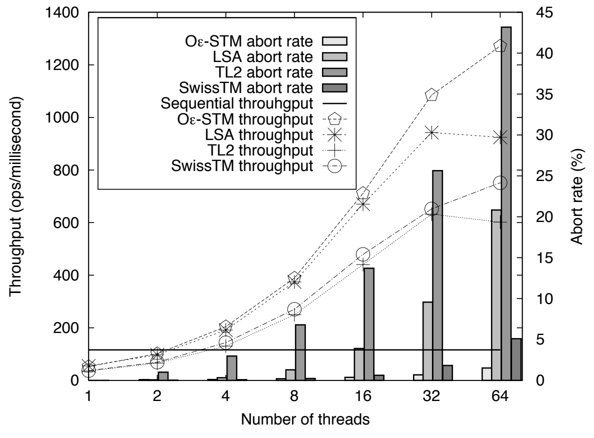


(b) 15 % update

Fig. 7: Throughput and abort ratio of bare sequential code, O$\mathcal{E}$-STM, LSA, TL2 and SwissTM on the SkipListSet of e.e.c when running 5% (left) and 15% (right) of addAll/removeAll.



(a) 5 % update



(b) 15 % update

Fig. 8: Throughput and abort ratio of bare sequential code, O$\mathcal{E}$-STM, LSA, TL2 and SwissTM on the HashSet of e.e.c when running 5% (left) and 15% (right) of addAll/removeAll.

throughput on LinkedListSet than other STMs. Specifically, O$\mathcal{E}$-STM improves on the performance of other STMs by at least $6.6\times$. This is due to the nature of the data structure whose linear time accesses are good candidates for concurrency optimization using relaxed transactions. Moreover, except for O$\mathcal{E}$-STM, other STMs almost never exceed sequential performance (their normalized throughput remains below 1).

Only on the SkipListSet benchmark does another STM perform as well as O$\mathcal{E}$-STM. In this particular workload relaxed transactions do not seem to benefit performance very much. The reason is that each update may modify up to $O(\log n)$ nodes, inducing contention that cannot be avoided through relaxation. Also, the benefit of relaxation is proportional to the number of nodes that transactions need to traverse, which is much lower in the case of a skiplist than for a linear data structure such as a linked list. Finally, O$\mathcal{E}$-STM performs much better than other STMs on the HashSet benchmark (Figure 8), where the load factor of the hash table (i.e., number of nodes / number of buckets) is set to 512 in order to increase contention.

To conclude, O$\mathcal{E}$-STM composes like regular STMs while offering better performance through the use of relaxed transactions that diminishing contention when the application permits.

VIII. Related Work

The problem of composing objects with certain properties to obtain atomic transactions was previously studied by Weihl [25]. Unlike his work, we compose relaxed transactions in order to obtain new transactions that are also relaxed. For this we consider relax-serializability, a condition that is strictly weaker than classic serializability used by Weihl.

Transactional boosting [28] is a transactional model where objects are not only reads/write registers as in classic transactional memories, but also more general objects from a separate thread-safe library. In order to detect conflicts between operations on these objects, abstract locks are used. To represent this behavior in our model, a process would acquire the protection element associated to an object o whenever a transaction executed by the process acquires an abstract lock corresponding to o. Since on rollback it is not sufficient to restore the memory to the state from before starting the transaction, the programmer must define a compensating operation for every operation the transaction executes. Although not described in the paper, passing abstract locks from the child to the parent transaction would make transactional boosting satisfy outheritance and therefore provide composition.

Open nesting [15] is a relaxed transactional model that allows the programmer to define open transactions that use abstract locks for providing multi-level conflict detection. As in the case of transactional boosting, each open transaction has an abort handler that reverts its effect in case of rollback. As this solution does not satisfy outheritance, no guarantees of atomicity are given and the programmer is responsible for ensuring correctness. The authors do however give guidelines to the programmer for ensuring correctness when using open nesting. In order to model multi-level concurrency control using protection elements, one would associate to each abstract lock a distinct protection element, which would be then acquired and released at the same time as respective abstract lock. In these conditions, outheritance would still guarantee relax-serializability at the lower level, but one could violate outheritance and still obtain the desired correctness of higher level transactions.

View transactions [19] are a type of relaxed transactions that use programmer-specified view pointers to define the critical view of a transaction, which is basically equivalent to our notion of a minimal protected set. When committing, a view transaction must pass its critical view to its parent transaction (if any), thus satisfying outheritance and ensuring composition.

Kulkarni et al. [29] provide yet another concrete instantiation of our principle, outheritance, this time passing the protected set from a child to the parent in the context of automatic parallelization. By satisfying outheritance, their approach ensures correct composition.

The classic way of using a transactional memory to obtain a thread-safe implementation of some abstract operation is to have every access to shared data instrumented by the transactional memory. Bronson [30] proposed solutions where only some accesses to shared data are transactional, while others are performed using synchronization from a separate thread-safe library, as in the case of transactional boosting. When composing such an implementation, the transactional memory passes information about the transactional accesses to the parent transaction as required by outheritance, allowing operations to compose correctly.

Chandy and Sanders [2] reason about parallel composition by extending predicate transformer theory to concurrent programming. They find some properties to be *all-component*, meaning that if all the components have the property, then their composition will have it as well, while other properties are *exists-component*, if at least one component has the property, then their composition will as well.

Gössler and Sifakis [3] describe a parallel composition operator that preserves deadlock-freedom. They distinguish between composability, the property of a component to meet a given property after being composed, and compositionality, which allows one to infer properties of a system from its components' properties. Our work falls into the latter category, namely one can infer the atomicity of composed operations from the atomicity of their sub-operations. This inference is valid when the system satisfies outheritance, which is in essence a concurrent composition operator.

Gava and Garnier [31] present a practical parallel composition operator using a continuation-passing-style transformation. This composition operator, useful for divide-and-conquer style algorithms among others, can be used many times in a single program, making it important for it to have an efficient implementation. In the same vein, an efficient concurrent composition based on our outheritance principle has the potential of being widely used in concurrent programming.

Fei and Lu [32] have studied composition in the context of scientific workflows. They provide a workflow composition framework in which workflows are the only operands for composition, as well as workflow constructs such as Map and Reduce. An easy programming model featuring straightforward composition has the potential of being the go-to solution for scientific computing.

IX. Concluding Remarks

Transactional memory is commonly advertised as an appealing abstraction to bring concurrency to the *masses*. It hides the difficult challenges of synchronization and makes it possible for *inexperienced* programmers to compose concurrent software. This appealing view conveys however a dumbing down of the programmers, for composition, at least in its classic implicit and transparent sense, is possible only if all programmers use transactions. Certain programmers are however skilled enough to seek less transparent concurrency abstractions that boost efficiency by enabling interleavings that would be prevented by the original transactional scheme. Relaxed transactional models are such abstractions. While boosting concurrency, their usage jeopardizes the composition dream.

This paper describes outheritance, a concrete property for ensuring that a transactional memory providing relaxed transactions composes. Using it, one can easily see if a given transactional memory ensures composition or can build a new one that does provide it. In short, outheritance stipulates that each child transaction must pass its conflict information to its parent transaction, which in turn maintains it until commit time. An important note is that outheritance is not tied to any specific type of relaxation and can be used for building transactional memories providing various types of relaxed transactions [33]. As future work we plan to experiment with using outheritance for composing multiple types of relaxed transactions inside the same transactional memory.

The applicability of relaxed transactions spans beyond the scope of the search data structures shown in this paper. Another direction for future work is to use outheritance for the out of order transactions used in the k-means finding problem [34] or the snapshot isolated transactions used in database applications [35].

References

[1] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley, 2002.

[2] K. Chandy and B. A. Sanders, "Predicate transformers for reasoning about concurrent computation," *Sci. Comput. Program.*, 1995.

[3] G. Gössler and J. Sifakis, "Composition for component-based modeling," *Sci. Comput. Program.*, 2005.

[4] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions," in *PPoPP*, 2005.

[5] O. Shalev and N. Shavit, "Split-ordered lists: Lock-free extensible hash tables," *J. ACM*, 2006.

[6] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *PODC*, 1996.

[7] T. Harris, "A pragmatic implementation of non-blocking linked-lists," in *DISC*, 2001.

[8] D. Lea, "The java.util.concurrent synchronizer framework," *Sci. Comput. Program.*, vol. 58, pp. 293–309, December 2005.

[9] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," *SIGARCH Comput. Archit. News*, 1993.

[10] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, 1997.

[11] V. Pankratius, "Transactional memory versus locks - a comparative case study," in *ICSE*, 2009.

[12] D. Cederman and P. Tsigas, "Supporting lock-free composition of concurrent data objects," in *CF*, 2010, pp. 53–62.

[13] V. Gramoli, D. Harmanci, and P. Felber, "On the input acceptance of transactional memory," *Parallel Processing Letters*, vol. 20, no. 1, pp. 31–50, 2010.

[14] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III, "Software transactional memory for dynamic-sized data structures," in *PODC*, 2003.

[15] J. E. B. Moss, "Open nested transactions: Semantics and support," in *Workshop on Memory Performance Issues*, 2006.

[16] P. Felber, V. Gramoli, and R. Guerraoui, "Elastic transactions," in *DISC*, 2009.

[17] E. Koskinen and M. Herlihy, "Concurrent non-commutative boosted transactions," in *PODC*, 2009.

[18] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, "Transactional predication: High performance concurrent sets and maps for STM," in *PODC*, 2010.

[19] Y. Afek, A. Morrison, and M. Tzafrir, "View transactions: Transactional model with relaxed consistency checks," in *PODC*, 2010.

[20] R. Zhang, Z. Budimlic, and W. N. Scherer III, "Composability for application-specific transactional optimizations," in *Transact*, 2010.

[21] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IISWC*, 2008.

[22] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," in *DISC*, 2006.

[23] T. Riegel, P. Felber, and C. Fetzer, "A lazy snapshot algorithm with eager validation," in *DISC*, 2006.

[24] A. Dragojevic, P. Felber, V. Gramoli, and R. Guerraoui, "Why stm can be more than a research toy," *Commun. ACM*, vol. 54, pp. 70–77, April 2011.

[25] W. E. Weihl, "Local atomicity properties: modular concurrency control for abstract data types," *ACM Trans. Program. Lang. Syst.*, 1989.

[26] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. ACM*, 1979.

[27] G. Korland, N. Shavit, and P. Felber, "Noninvasive java concurrency with deuce STM," in *OOPSLA*, 2009, poster session.

[28] M. Herlihy and E. Koskinen, "Transactional boosting: A methodology for highly-concurrent transactional objects," in *PPoPP*, 2008.

[29] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *PLDI*, 2007.

[30] N. Bronson, "Composable operations on high-performance concurrent collections," Ph.D. dissertation, Standford University, 2011.

[31] F. Gava and I. Garnier, "New implementation of a bsp composition primitive with application to the implementation of algorithmic skeletons," in *IPDPS*, 2009.

[32] X. Fei and S. Lu, "A dataflow-based scientific workflow composition framework," *IEEE Transactions on Services Computing*, 2012.

[33] V. Gramoli and R. Guerraoui, "Democratizing transactional programming," in *Middleware*, 2011, pp. 1–19.

[34] A. Udupa, K. Rajan, and W. Thies, "Alter: exploiting breakable dependences for parallelization," in *PLDI*, 2011.

[35] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases," in *SIGMOD*, 2008.