Elastic Transactions☆

Pascal Felber[a], Vincent Gramoli[b,1], Rarchid Guerraoui[c]

[a] *University of Neuchâtel, Rue Emile-Argand 11, B-114, CH-2000 Neuchâtel, Switzerland.*
[b] *School of Information Technologies, Bldg J12, 1, Cleveland St, University of Sydney, NSW 2006, Sydney, Australia.*
[c] *EPFL Station 14, CH-1015 Lausanne, Switzerland.*

Abstract

This paper presents *elastic transactions*, an appealing alternative to traditional transactions, in particular to implement search structures in shared memory multicore architectures. Upon conflict detection, an elastic transaction might drop what it did so far within a separate transaction that immediately commits, and resume its computation within a new transaction which might itself be elastic.

We present the elastic transaction model and an implementation of it, then we illustrate its simplicity and performance on various concurrent data structures, namely *double-ended queue*, *hash table*, *linked list*, and *skip list*. Elastic transactions outperform classical ones on various workloads, with an improvement of 35% on average. They also exhibit competitive performance compared to lock-based techniques and are much simpler to program with than lock-free alternatives.

1. Introduction

Transactions are an appealing synchronization paradigm for they enable average programmers to leverage modern multicore architectures. The power of the paradigm lies in its abstract nature: there is no need to know the internals of shared object implementations, it suffices to delimit every critical sequence of shared object accesses using transactional boundaries. The inherent difficulty of synchronization is hidden from the programmer and encapsulated inside the transactional memory, implemented once and for all by experts in concurrent programming.

Not surprisingly, and precisely because it hides synchronization issues, the transaction abstraction may severely hamper parallelism. This is particularly true for search data structures where transactions do not know a priori where to insert an element unless they possibly explore a big part of the data structure. Search structures implement key abstractions like queues, heaps, key-value stores, or collections but turn out to be the contention hot spots of applications aiming at leveraging modern multicore machines [59]. In an attempt to minimize this contention, transactions are typically chosen to synchronize search structures by redirecting shared accesses at runtime, instead of conservatively protecting extra memory locations ahead of time.

To illustrate the limitation of transactions, consider the bucket hash table depicted in Figure 1 implementing an integer set and exporting operations search, insert, and remove. A bucket, itself implemented with a sorted linked list as in [44], indicates where an integer should be stored. Consider furthermore a situation involving two concurrent transactions: the first seeks to insert an integer m at some position whereas the second searches for an integer n and reads m. In a strict sense, there is a read-write conflict that may cause to block or abort one of the transactions; yet this is a false (search-insert) conflict. Because they are sensitive to these kinds of conflicts, regular transactions hamper concurrency, and this might have a significant impact on performance, should the data structures be large and shared by many concurrent transactions. It is important to notice here that the issue is not related to the way transactions are used, but to the paradigm itself. More specifically, assuming transactions in their traditional sense, i.e., accessing shared objects through read and write primitives, even an expert programmer *has to choose between violating consistency and hampering concurrency.*

Addressing the issue above with locks is simpler. A well-known lock-based technique to access the aforementioned sorted linked list is to parse it starting from its first head element, by acquiring multiple consecutive elements, before releasing the first of these. The technique is called *hand-over-hand locking* [4]: it looks like a *right hand* acquires the $i+1^{st}$ elements, then the *left hand* releases the i^{th} before it acquires the $i+2^{nd}$, and so on. This enables a level

☆A preliminary extended abstract of this work has been published in the proceedings of DISC 2009 [15], the current version extends it by generalizing the model, applying it to additional data structures, and comparing it against existing synchronization alternatives.

Email addresses: pascal.felber@unine.ch (Pascal Felber), vincent.gramoli@sydney.edu.au (Vincent Gramoli), rachid.guerraoui@epfl.ch (Rarchid Guerraoui)

[1]Corresponding author. Address:School of Information Technologies, Bldg J12, 1, Cleveland St, University of Sydney, NSW 2006, Sydney, Australia, Phone: +61 2 9036 9270, E-mail address: vincent.gramoli@sydney.edu.au.

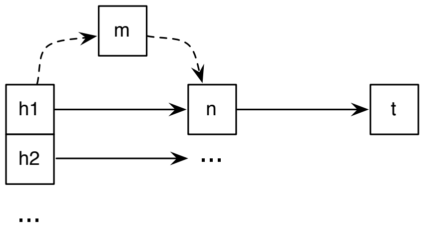


Figure 1: A bucket hash table where a transaction (insert(m)) invalidates an almost complete transaction (search(n)) that accesses the same bucket.

of concurrency that is hard to get with regular transactions for these are open-closed blocks that cannot overlap with one another. Instead, a transaction keeps track of all its accesses during its entire lifespan, hence a concurrent update on the i^{th} element triggers a conflict even at the point where the transaction accesses the $i + 2^{nd}$ element. This lack of concurrency is problematic in numerous data structures in which a big part of the data structure must be parsed in order to find the targeted location.

Several transactional models were proposed to cope with similar problems. The theory of commutativity [38] helps identifying particular transactional operations that can be reordered without affecting the semantics of the execution. This commutativity was key to multiple transactional models. The consistency criterion of multi-level serializability [69] exploits this commutativity to reorder low level operations within operations at a higher level of abstractions. Similar to these models, elastic transactions do not relieve the programmer from the burden of understanding the semantics of the operations, but as far as we know, no existing transactional models can exploit the runtime information in search data structure executions to decide dynamically whether operations commute.

We propose *elastic* transactions, an efficient alternative to traditional transactions for such search data structures. Just like for a regular transaction yet differently from most transaction models as we discuss in Section 2, the programmer must simply delimit the blocks of code that represent elastic transactions. Nevertheless, during its execution, an elastic transaction can be *cut* (by the elastic transactional memory) into multiple regular transactions, depending on the conflict it encountered at runtime. Intuitively, the cut allows to automatically decide at runtime whether operations commute.

More specifically, upon conflict detection an elastic transaction decides whether it can cut itself; if so it commits the past accesses as if they were part of a regular transaction before resuming into a continuation transaction until it encounters a new conflict or commits. A cut is prohibited if, between the times of two of its consecutive accesses on two locations, these two locations get updated by other transactions. Only in this rare case does the elastic transaction abort. In other words, it is not possible for the elastic transaction to abort if, during the interval where the elastic transaction executes a pair of consecutive accesses on two locations, at most one of these locations gets updated. In case the elastic transaction does not abort, a cut could cause the elastic transaction to execute a constant number of additional accesses before committing the past ones. In a sense, these few extra accesses can be viewed as a partial roll-back that is the price to pay to avoid aborting the elastic transaction. We will see later that, as a result, there is no need to undo any write.

At this point, one might ask why we propose a new transactional model instead of using locks. The reasons are twofold: unlike locks, elastic transactions (i) can be combined with other transactions to permit extensibility through code composition and (ii) enable the direct reuse of sequential code. To illustrate code composition, consider again a hash table implementation. Consider however that this implementation now extends the integer set abstraction into a dictionary abstraction aimed at exporting a move operation, which modifies the key of a value. Given a transactional integer set, one has simply to encapsulate a transactional remove and a transactional insert into a single transaction to obtain an atomic move. By contrast, using locks explicitly is known to be a difficult task [50, 55] prone to deadlocks when one process moves from bucket ℓ_1 to bucket ℓ_2 while another moves from ℓ_2 to ℓ_1. Given a lock-based integer set, the programmer must know the granularity of internal locks, like the size of lock stripes, to make sure that the new move and existing updates on common parts are mutually exclusive. Even so, the original implementation tuned to provide an efficient integer set interface may provide an inefficient extension.

Finally, lock-free implementations can neither ensure both extensibility and concurrency. To ensure the atomicity of the move resulting from the composition of lock-free remove and insert, one could modify a copy of the data structure before switching a pointer from one copy to another [27]. This, however, prevents two concurrent updates, which modify disjoint locations, from succeeding. One could also use a multi-word compare-and-swap instruction [21] but this is often considered inefficient and upcoming architectures rather favor general-purpose transactions. A remarkable example of the lack of extensibility of efficient lock-free algorithm is the complete redesign of a hash table structure into a split-ordered linked list to support a lock-free resize operation [58]. Although the resize allows hash table buckets to move among consecutive list nodes, it does not allow nodes to move among hash table buckets.

Elastic Transactions: a Primer

To give an indication of the main idea underlying elastic transactions, consider the integer set abstraction implemented using the linked list data structure. Each of the insert, remove, and search operations consists of lower-level operations: some reads and possibly some writes. Consider an execution in which two transactions, i and j, try to insert keys 3 and 1. Each insert transaction parses the nodes in ascending order up to the node before which it should insert its key. Let $\{2\}$ be the initial state of the integer set and let h, n, t denote respectively the memory locations where the head pointer, the single node (its key and next pointer) and the tail key are stored. Let $\mathcal{H}$ be the following history of operations where transaction j inserts 1 while transaction i is parsing the data structure to insert 3 at its end. (We indicate only operations of non-aborting transactions and omit commit events for simplicity.)

$$\mathcal{H} = r(h)^i, r(n)^i, r(h)^j, r(n)^j, w(h)^j, r(t)^i, w(n)^i.$$

This history is clearly not serializable [51] since there is no sequential history where $r(h)^i$ occurs before $w(h)^j$ and $r(n)^j$ occurs before $w(n)^i$. A traditional transactional scheme would detect two contradicting conflicts between transactions i and j, and the transactions could not both commit. Nonetheless, history $\mathcal{H}$ does not violate the correctness (in this case the linearizability) of the integer set: 1 appears to be inserted before 3 in the linked list and both are present at the end of the execution.

This situation can be efficiently addressed with elastic transactions as we explain now. To make a transaction elastic, the programmer has simply to label this transaction i as being so. History $\mathcal{H}$ can now be viewed as the composition $f(\mathcal{H})$ of several pieces resulting from the application of the cutting function f:

$$f(\mathcal{H}) = \boxed{r(h)^i, r(n)^i}^{s_1}, r(h)^j, r(n)^j, w(h)^j, \boxed{r(t)^i, w(n)^i}^{s_2}.$$

Specifically, elastic transaction i has been cut into two transactions s_1 and s_2. Crucial to the correctness of this cut is the very fact that the value returned by the read of t has been the successor of n at some point in time. More precisely, the specific operations inside elastic transaction i ensure that no two modifications on n and t have occurred between $r(n)^{s_1}$ and $r(t)^{s_2}$. Otherwise the transaction would have to abort. (Note that only one of these two modifications on either n or t is permitted.)

Basically even though a read value has been freshly modified by another transaction, it might not be necessary to abort i. Assume that a transaction i searches for an integer that is not in the linked list while a transaction j is inserting a new integer node after the k^{th} node. Let $h, n_1, ..., n_\ell, t$ denote respectively the memory locations of the linked list: n_k denotes the memory location of the k^{th} node integer and its next pointer. In the following history $\mathcal{H}'$, transaction i reads node n_k and can detect that it has freshly been modified by another transaction j.

$$..., r(n_k)^j, r(n_{k-1})^i, w(n_k)^j, r(n_k)^i, r(n_{k+1})^i, ...$$

In this example, transaction i does not have to abort because (a) i accesses n_k for the first time and (b) the preceding node n_{k-1} has not been overwritten since it has been accessed by i. Hence, if we could check that n_{k-1} has not been modified, transaction i could resume and commit, as if its read of n_k was part of a new transaction s_k, serialized after j. With i being an elastic transaction, we consequently get the following history with the new transaction s_k at the end:

$$..., r(n_k)^j, r(n_{k-1})^i, w(n_k)^j, \boxed{r(n_k)^i, r(n_{k+1})^i}^{s_k}, ...$$

Performance Overview

We developed $\mathcal{E}$-STM, an implementation of the elastic transaction model, in C and Java for x86-64 and SPARC architectures. $\mathcal{E}$-STM uses timestamps, two-phase-locking, and the atomic primitives CAS and fetch-and-increment. $\mathcal{E}$-STM supports both regular transactions and elastic transactions: the latter ones achieve high concurrency but retain the abstraction simplicity of regular transactions.

To evaluate the performance and simplicity of $\mathcal{E}$-STM, we developed with it four data structure applications: (i) double-ended queue, (ii) hash table, (iii) linked list, and (iv) skip list. We compared $\mathcal{E}$-STM with three other synchronization techniques: (i) regular STM transactions, (ii) lock-based (fine-grained locks), and (iii) lock-free techniques. The regular STM used as a reference point relies on TinySTM, one of the most efficient STMs on some of search data structures we use here [14, 23]. The lock-based implementations are based on the algorithms of Heller et al. [26] and Herlihy et al. [31]. The lock-free implementations are based on the algorithms of Harris [24], Michael [44] and Fraser [17].

In short, $\mathcal{E}$-STM improves regular transactions with an average factor of 35%, with an improvement on all workloads except the double-ended queue. Intuitively, this is because no concurrency can be exploited between the very small update transactions of the double-ended queue. The mean improvement reaches 170% on linear access time data structures because transactions can be cut into multiple pieces, 8% on logarithmic time data structures and only 2% on constant-time data structures whose transactions are already small.

Table 1 summarizes the complexity and performance of every tested synchronization technique by giving the amount of extra code that each technique requires and the throughput gain it provides on multicore, when compared to the bare code running sequentially. We compared the final number of lines of code against the length of the bare sequential code (column 2) and we evaluated the ability for such a technique to extend a hash table with more complex operations, move and sum (column 3). Additionally, we computed the improvement of each concurrent execution over the bare code running sequentially, in both the original (column 4) and the extended workload (column 5). The lock-based algorithm optimized for a fixed set of operations (search, insert and remove) can run faster than elastic transactions, however, its performance drops below the performance of the sequential code when used to

Synchronization technique	Programming complexity		Performance	
	Code length	Extensibility	Original	Extended
lock-free	+326%	×	7.2×	⊥
lock-based (fine-grained)	+197%	✓	6.6×	0.1×
regular transaction	+6%	✓	2.3×	1.4×
elastic transaction ($\mathcal{E}$-STM)	+6%	✓	3.1×	1.4×

Table 1: Evaluating programming overhead and performance improvement with respect to sequential implementations of four synchronization techniques on various data structures; the second column compares the lines of code, the third column indicates whether the code can be reused and extended by another programmer, the last two columns give the throughput improvement against bare sequential of the base algorithms and the extended algorithms, respectively

extend the hash table with additional operations (move and sum). Although $\mathcal{E}$-STM is less efficient than ad-hoc lock-free techniques, it does not hamper extensibility and require almost no additional code. By contrast, efficiently extending lock-free algorithms remains an open question and writing a thousand lines of code may be necessary to implement a simple lock-free skip list.

Roadmap

The rest of this paper is organized as follows. Section 2 positions elastic transactions with respect to the related work. Section 3 presents our general model and Section 4 introduces elastic transactions. We propose an STM library implementing elastic and regular transactions and discuss regular STM, lock-based, and lock-free alternative implementations in Section 5. Then, we elaborate on the advantage of using elastic transactions through data structure implementations in Section 6 and we present the performance we obtained in Section 7. We conclude the paper in Section 8. We argue about the correctness of $\mathcal{E}$-STM and our linked list example in Appendices Appendix A and Appendix B, respectively.

2. Related Work

Relaxed transactional models that account for the semantics of applications to boost concurrency were proposed long ago, almost at the same time as transactions themselves [53].

Unlike our elastic transaction model that exploits the semantics of search structures that constitute hot spots in concurrent libraries, the initial relaxed models [53, 8, 49] exploited the semantics of aggregate fields, often considered hot spots in databases. In the initial relaxed models, for example, two increments on an aggregate field within the same transaction could be interleaved by a concurrent decrement of the same field, as the correctness of an aggregate field is not impacted by intermediary values. In our elastic transaction model, two reads on some locations within the same transaction may be interleaved with writes on the same locations, because modifying some subpart of the data structure does not necessarily impact the lookup of an element that is located in a distinct subpart.

2.1. Commutativity

Commutativity [38, 57, 67], a binary relation over operations, was extensively used to increase inter-transaction concurrency. It was noted that when two operations are commutative, their ordering does not matter [38].

A theory of dependencies among transactional operations depending on the abstract type they operate upon was illustrated using dictionaries [57]. Similarly to identifying non-commutative operations, the idea lies in identifying dependent operations as those whose ordering affect the outcome of the transactions and hence the equivalence of the history to a sequential one.

Commutativity [67] of two operations is defined depending on their arguments and return values[2]. More precisely, the forward and backward types of commutativity, distinguished by the states on which an operation is defined, capture the possible ordering within two implementation designs: *undo-log* and *redo-log*. Two operations commute-forward if they commute in the states in which they are defined and the resulting state is also defined, whereas they commute backward if they commute in all states. Hence $r(x)$ returning value v can commute forward with $w(x)$ writing value v but cannot commute backward as $r(x)$ returning v is defined if executing after $w(x)$ but may not be defined if executing before $w(x)$ (typically if x had originally another value $v_0 \neq v$). Interestingly, the redo-log design allows forward commutation only whereas the undo-log design allows backward commutation only.

The programmer can exploit commutativity on search structures using transaction models like open nesting [65, 46, 47] and transactional boosting [29] or parallelization models dedicated to irregular applications, like the Galois system [40]. These models are flexible as they let the programmer define additional abstractions other than set or dictionary types, however, they all require the programmer to specify explicitly the commutativity of operations and abort handlers with appropriate compensate actions.

Elastic transactions also exploit commutativity to relax serializability by letting for example two insert operations executed by distinct transactions commute if they

[2] In contrast with ours, the insert and remove used to illustrate these properties always return true [67].

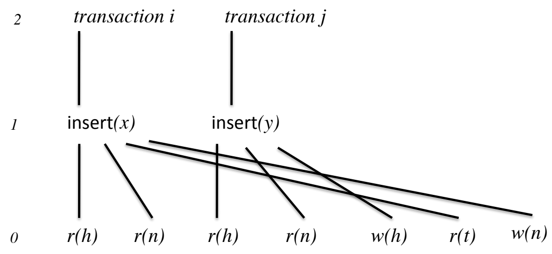


Figure 2: A history where operations are represented at different levels of semantics.

insert different values. As they are used for search structures elastic transactions enable, however, additional concurrency between two insert by allowing for example the linked list history $r(h)^1, r(n)^2, w(h)^2, w(n)^1$ in which neither $r(n)^2$ and $w(n)^1$ nor $r(h)^1$ and $w(h)^2$ commute.

2.2. Multi-level Concurrency Control

The consistency criterion called multi-level serializability is described in [69]. This criterion applies to operations at different levels of abstraction, thus expressing the impact low level reads and writes may have on the higher-level operations that comprise them. More precisely, ℓ-level serializability is defined iteratively assuming that any history is 0-level serializable. A history that is ℓ-level serializable and in which we can find a serialization of level $\ell+1$ operations by commuting level ℓ operations, is also $(\ell+1)$-level serializable. Identifying two operations that commute at level ℓ is thus used to define an $\ell+1$ level serialization.

Consider the simple linked list history $\mathcal{H}$ of Section 1 using Weikum's hierarchy [69] as depicted by Figure 2. The two transactions i and j appear at level 2 of this hierarchy, the integer set operations called by transactions i and j appear at level 1 and the read and write operations they called appear at level 0. The history is neither 2-level nor 1-level serializable because of two dependencies: between $r(h)^i$ and $w(h)^j$ and between $r(n)^j$ and $w(n)^i$, however, both would safely commit if i is elastic.

As far as we know, there is no transaction model to exploit the runtime information about the interleaving to decide dynamically whether operations commute. Multi-level atomicity [43] and its derived consistency criteria [39] require nested transactions to be explicitly interleaved at predetermined breakpoints.

Some language constructs put additional burden on the programmer to better ignore conflicts. In the database context, a transaction can ignore some conflicts yet ensure *snapshot isolation* [5] as this property allows a transaction to commit provided that the values it has written have not been overwritten [12]. In some snapshot isolated databases, the programmer can statically SelectForUpdate to avoid the *write-skew* problem [5] where two transactions read memory locations before they update distinct locations among the previously read ones: both transactions read the same value while one should not. Such a construct makes the reads visible without distinction at runtime. In the context of transactional memory, the programmer can statically call specific actions within a transaction to unprotect previously read locations [34] or to differentiate protected read calls [6, 3]. Unlike elastic transactions that preserve sequential code, these techniques require careful modifications to the code as often pointed out by their authors.

Elastic transactions commute additional operations by exploiting the interleaving information obtained at *runtime*. Specifically, an elastic transaction checks dynamically whether two consecutive reads accessing two elements in the same transactions are interleaved with two writes from other transactions on these two elements. If so, then the read operations do not commute with the writes and the elastic transaction aborts. If not, a cut indicates that the former read commutes with the write that accessed the same element, as if there existed a serialization of these two reads after the two writes.

3. Model

Before presenting our elastic transaction paradigm, we first give a general model of transactional computation. As in [68], our system comprises transactions and objects, and the states of all objects define the state of the system. A transaction is a sequence of read and write operations that can examine and modify, respectively, the state of the objects. More precisely, a transaction consists of a sequence of events including an *operation invocation*, an *operation response*, a *commit invocation*, a *commit response*, or an *abort event*. These events are used below to distinguish transactions.

An operation whose response event occurred is considered as *terminated* while a transaction whose commit response or abort event occurred is considered as *completed*.

The set of transactions in the system is denoted by T and we consider two types of transactions: *regular* and *elastic*. We assume that the type of all transactions is initially known, being fixed by the programmer. The sets of regular and elastic transactions are denoted by $\mathcal{N}$ and $\mathcal{E}$, respectively. The set of objects is denoted by X and the set of values is V. An *operation* accessing an object x and belonging to a transaction t, can be of two *type*s (read or write), and either takes as an argument or returns a value v. Hence, an operation is denoted by a tuple in $X \times T \times V \times type$.

3.1. Histories

We consider well-formed sequences of events that consist of a set of transactions, each satisfying the following constraints: (i) a transaction must wait until its operation terminates before invoking a new one, (ii) no transaction both commits and aborts, and (iii) a transaction cannot invoke an operation after having completed. Hence, we assume that each operation is part of a transaction and we do not consider non-transactional operations.[3] We refer to these well-formed sequences as *histories*.

A history $\mathcal{H}$ is complete if all its transactions are completed. To take into consideration pending transactions, we define a completing function *complete*() that maps any history $\mathcal{H}$ to a set of complete histories by appending an event q to each non-completed transaction t of $\mathcal{H}$ such that:

– q is an abort event if there is no commit invocation for t in $\mathcal{H}$;

– q is a commit response or an abort event if there is a commit invocation for t in $\mathcal{H}$.

Given a set of transactions T and a history $\mathcal{H}$, we define $\mathcal{H}|T$, the restriction of $\mathcal{H}$ to T, to be the subsequence of $\mathcal{H}$ consisting of all events of any transaction $t \in T$. We refer to the set of transactions that have committed (resp. aborted) in $\mathcal{H}$ as $\mathit{committed}(\mathcal{H})$ (resp. $\mathit{aborted}(\mathcal{H})$). The history of all committed transactions of a given history $\mathcal{H}$ is denoted by $\mathit{permanent}(\mathcal{H}) = \mathcal{H}|\mathit{committed}(\mathcal{H})$. Similarly, for a set of objects X we denote by $\mathcal{H}|X$ the subsequence of $\mathcal{H}$ restricted to X. For the sake of simplicity, to denote $\mathcal{H}|\{x\}$, for $x \in X$ (resp. $\mathcal{H}|\{t\}$, for $t \in T$) we simply write $\mathcal{H}|x$ (resp. $\mathcal{H}|t$).

Let $\rightarrow_{\mathcal{H}}$ be the total order on the events in $\mathcal{H}$. We say that transaction t *precedes* transaction t' in $\mathcal{H}$ (denoted, by extension, by $t \rightarrow_{\mathcal{H}} t'$) if there are no events $q \in \mathcal{H}|t$ and $q' \in \mathcal{H}|t'$ such that $q' \rightarrow_{\mathcal{H}} q$. Two transactions t and t' are called *concurrent* if neither precedes the other, i.e., $t \not\rightarrow_{\mathcal{H}} t'$ and $t' \not\rightarrow_{\mathcal{H}} t$. A history $\mathcal{H}$ is *sequential* if no two transactions of $\mathcal{H}$ are concurrent.

3.2. Operation Sequences

For simplicity, and as in [68], we consider a sequence of operations instead of a sequence of events to describe histories and transactions. An operation π is a pair of invocation and response events such that the invocation and response correspond to the same operation, accessing the same object and being part of the same transaction. A given history $\mathcal{H}$ is thus an operation sequence $\mathcal{S}_{\mathcal{H}} = \pi_1, ..., \pi_n$ resulting from $\mathcal{H}$ where commit invocations, commit responses, and invocations that do not have a matching response have been omitted. Concurrent operations ordering is determined by the object serial specification described below. We say that two histories $\mathcal{H}$ and $\mathcal{H}'$ are *equivalent* if for any transaction t, $\mathcal{H}|t = \mathcal{H}'|t$.

The *distance* between two operations π_i and π_j in a history $\mathcal{H}$, denoted by $\mathit{dist}_{\mathcal{H}}(\pi_i, \pi_j)$, is the difference of their position in the total order $\rightarrow_{\mathcal{H}}$. More precisely, $\mathit{dist}_{\mathcal{H}}(\pi_i, \pi_j) = |\{\pi'_j : \pi_i \rightarrow_{\mathcal{H}} \pi'_j \rightarrow_{\mathcal{H}} \pi_j\}| + 1$ if $\pi_i \rightarrow_{\mathcal{H}} \pi_j$, or $\mathit{dist}_{\mathcal{H}}(\pi_i, \pi_j) = |\{\pi'_j : \pi_j \rightarrow_{\mathcal{H}} \pi'_j \rightarrow_{\mathcal{H}} \pi_i\}| + 1$ otherwise.

The serial specification of an object is the set of acceptable sequences of its operations. Each object x is initialized with a default value v_x and accessed either by a write operation, $\pi(x, v)$, that writes a value v or by a read operation, $\pi(x) : v$, that returns a value v. That is, we only focus on read/write objects, whose serial specification requires that a read operation on x returns the last value written on x, or its default value v_x (if no value has been written before). We assume that each written value is unique, hence: let $\pi(x, v)$ and $\pi'(x', v')$ be two write operations, if $v = v'$ then $x = x'$ and $\pi = \pi'$.

We define an ordering relation on operations similarly to the ordering on message events in a message-passing model [41]. To this end, we define two binary relations on the read and write operations of transactions. We say that an operation π_i *precedes* operation π_j, denoted by $\pi_i \prec \pi_j$ if at least one of the following properties is satisfied (note that three of them are denoted by write-after-read (WAR), read-after-write (RAW) and write-after-write (WAW) for later reuse in Section 4.4):

– π_i and π_j are two consecutive operations of the same transaction t,

– (WAR) π_i is a read operation of transaction t and π_j is a write operation of t' that overwrites the value accessed by π_i,

– (RAW) $\pi_i(x, v)$ is a write operation of transaction t and $\pi_j(x) : v$ is a read operation of t' that returns the value written by π_i (π_j reads from π_i) or

– (WAW) π_i is a write operation of transaction t and π_j is a write operation of t' that overwrites the value accessed by π_i.

The transitive closure of this precedence relation is denoted $\prec^*$. More precisely, we obtain the following recursive definition for the precedence relation $\prec^*$. We say that $\pi_i \prec^* \pi_j$ if one of the two following properties holds:

– either $\pi_i \prec \pi_j$,

– or there exists π_ℓ such that $\pi_i \prec \pi_\ell$ and $\pi_\ell \prec^* \pi_j$.

A sequential history $\mathcal{H}$ is *legal* if the serial specification of all objects accessed in $\mathcal{H}$ is satisfied, i.e., if each read operation π on some object x returns either the value written by the last write operation on x, preceding π, or the default value v_x if no such write operation exists. More precisely, $\mathcal{H}$ is legal if the value v returned by any $\pi(x) : v \in \mathcal{H}$ is either such that $\pi'(x, v) = \mathit{max}_{\rightarrow_{\mathcal{H}}}\{\pi'(x, *) \in \mathcal{H} \text{ s.t. } \pi' \rightarrow_{\mathcal{H}} \pi\}$ or $v = v_x$ if there is no $\pi'(x, *) \in \mathcal{H}$ such that $\pi' \rightarrow_{\mathcal{H}} \pi$.

We refer to a transaction that never writes an object value in the shared memory as an *invisible* transaction. Observe that invisible transactions may write some metadata (e.g., lock ownership) in the shared memory. As an example a transaction that acquires some locks before aborting can be invisible.

[3] For transaction semantics tolerating non-transactional code support, we refer the interested reader to the AME programming model [1] or to extensions providing privatization-safety [63].

4. Elastic Transactions

An *elastic* transaction is a transaction whose size depends on the conflicts it encounters. In short, such a transaction may be automatically cut upon conflict detection as if the start of the transaction had moved forward. We first define the very notion of a *cut*. (When the programmer should decide to use elastic transactions is explained in Section 4.4.)

4.1. Cut

A sequence of operations is a totally ordered set. We refer to a history $\mathcal{H}$ as a tuple $\langle S_{\mathcal{H}}, \rightarrow_{\mathcal{H}}\rangle$ where $S_{\mathcal{H}}$ is the corresponding set of operations and $\rightarrow_{\mathcal{H}}$ a total order defined over $S_{\mathcal{H}}$. A *sub-history* $\mathcal{H}'$ of history $\mathcal{H} = \langle S_{\mathcal{H}}, \rightarrow_{\mathcal{H}}\rangle$ is a history $\mathcal{H}' = \langle S_{\mathcal{H}'}, \rightarrow_{\mathcal{H}'}\rangle$ such that $S_{\mathcal{H}'} \subseteq S_{\mathcal{H}}$ and $\rightarrow_{\mathcal{H}'} \subseteq \rightarrow_{\mathcal{H}}$. We now define the notion of cut and its well-formedness.

Definition 1 (k-sized Cut). A *cut* of size $k > 0$ of a history $\mathcal{H}$ is a sequence $\mathcal{C} = \langle S_{\mathcal{C}}, \rightarrow_{\mathcal{C}}\rangle$ of sub-histories of $\mathcal{H}$ such that:

1. each of the sub-histories of $S_{\mathcal{C}}$, with the exception of the first one and the last one in $\mathcal{C}$, contains at least $k+1$ operations;
2. each of the sub-histories of $S_{\mathcal{C}}$ contains only consecutive operations of $\mathcal{H}$, i.e., for any sub-history $\mathcal{H}' = \pi_1, ..., \pi_n$ in $S_{\mathcal{C}}$, if there exists $\pi_i \in \mathcal{H}$ such that $\pi_1 \rightarrow_{\mathcal{H}} \pi_i \rightarrow_{\mathcal{H}} \pi_n$, then $\pi_i \in \mathcal{H}'$;
3. if one sub-history precedes another in $\mathcal{C}$ then the operations of the first precede the operations of the second in $\mathcal{H}$, i.e., for any sub-histories $\mathcal{H}_1$ and $\mathcal{H}_2$ in $S_{\mathcal{C}}$ and two operations $\pi_1 \in \mathcal{H}_1$ and $\pi_2 \in \mathcal{H}_2$, if $\mathcal{H}_1 \rightarrow_{\mathcal{C}} \mathcal{H}_2$ then $\pi_1 \rightarrow_{\mathcal{H}} \pi_2$;
4. any operation of $\mathcal{H}$ is in exactly one sub-history of the cut, i.e., $\bigcup_{\forall \mathcal{H}' \in S_{\mathcal{C}}} S_{\mathcal{H}'} = S_{\mathcal{H}}$ and for any $\mathcal{H}_1, \mathcal{H}_2 \in S_{\mathcal{C}}$, we have $S_{\mathcal{H}_1} \cap S_{\mathcal{H}_2} = \emptyset$.

For instance history a, b, c has three 1-sized cuts $\mathcal{C}1 = \{a, b \ ; \ c\}$, $\mathcal{C}2 = \{a \ ; \ b, c\}$ and $\mathcal{C}3 = \{a, b, c\}$, where semicolons are used to separate consecutive sub-histories of the cut and braces are used for clarity to enclose a cut. By contrast, neither $\{a, c \ ; \ b\}$ nor $\{a \ ; \ a, b, c\}$ are cuts of $\mathcal{H}$. The reason is that the former violates property (2) while the latter violates property (4) of Definition 1.

The following well-formedness definition states that a write can neither be separated from other writes nor from its k preceding read operations within the same transaction.

Definition 2 (Well-formed cut). A cut $\mathcal{C}_t$ of size $k > 0$ of history $\mathcal{H}|t$, where t is a transaction, is *well-formed* if for any of its sub-histories s, s' the following properties are satisfied:

1. if $\pi_i \in s$ and $\pi_j \in s'$ are any two write operations of t, then $s = s'$;
2. if π_i is the i^{th} operation of s with $i \leq k$, then either π_i is a read operation or π_i is the i^{th} operation of t.

Consider for instance the following history $\mathcal{H}_1|t$ where t is an elastic transaction, and where $r(x)$ and $w(x)$ refer to a read and a write operation on some object x, respectively. (For the following examples, we omit the values returned by the read operations and consider that the object serial specification is satisfied.)

$$\mathcal{H}_1|t = r(u), r(v), w(x), r(y), r(z).$$

On the one hand, there are well-formed 1-sized cuts of history $\mathcal{H}_1|t$: $\mathcal{C}1' = \{r(u), r(v), w(x), r(y), r(z)\}$ and $\mathcal{C}2' = \{r(u), r(v), w(x) \ ; \ r(y), r(z)\}$, for example. On the other hand, $\mathcal{C}3' = \{r(u), r(v) \ ; \ w(x), r(y), r(z)\}$ is not a well-formed 1-sized cut. More precisely, the second sub-history of $\mathcal{C}3'$ starts with a write operation, that is, property (2) of Definition 2 is violated.

In the remainder of this paper we only consider well-formed cuts and all these cuts have size 1, unless specified otherwise.

4.2. Consistent Cut

We define a consistent cut with respect to a history of potentially concurrent transactions. This definition is crucial as it indicates the difference between regular and elastic transactions. The programmer can label a transaction as elastic if *(i)* its series of read/write operations does not need to appear to have executed at a common point in time, making regular transaction unnecessary; *(ii)* but still requires that all k-tuples of consecutive operations or the operations enclosed by write operations in this transaction appear as having been executed at a common point in time, preventing the programmer from hand-crafted cutting transactions. Basically, a cut of $\mathcal{H}$ is consistent if there are no writes separating two accesses each accessing one of the object written by these writes.

Definition 3 (Consistent cut). A cut $\mathcal{C}_t$ of $\mathcal{H}|t$ of size k is *consistent* with respect to history $\mathcal{H}$ if, for any operation π_i and π_j of any two of its sub-histories s_i and s_j respectively ($s_i \neq s_j$) such that $dist_{\mathcal{H}|t}(\pi_i, \pi_j) \leq k$, the two following properties hold:

- there is no write operation $\pi'(x)$ from a transaction $t' \neq t$ such that $\pi_i(x) \rightarrow_{\mathcal{H}} \pi'(x) \rightarrow_{\mathcal{H}} \pi_j(x)$;
- there are no two write operations $\pi'(x)$ and $\pi''(y)$ from transactions $t' \neq t$ and $t'' \neq t$ such that $\pi_i(x) \rightarrow_{\mathcal{H}} \pi'(x) \rightarrow_{\mathcal{H}} \pi_j(y)$ and $\pi_i(x) \rightarrow_{\mathcal{H}} \pi''(y) \rightarrow_{\mathcal{H}} \pi_j(y)$.

For example, consider the following history $\mathcal{H}_2$, depicted in Figure 3, where e is an elastic transaction and n is a regular transaction, and where $r(x)^t$ and $w(x)^t$ refer to a read and a write operation on x in transaction t.

$$\mathcal{H}_2 = r(x)^e, r(y)^e, w(y)^n, r(z)^e, w(u)^e.$$

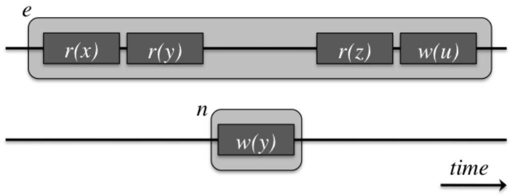


Figure 3: History $\mathcal{H}_2$ has five consistent cuts of elastic transaction e.

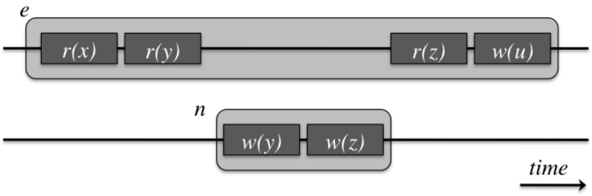


Figure 4: History $\mathcal{H}_3$ with respect to which $\mathcal{C}3 = \{r(x)^e, r(y)^e\ ;\ r(z)^e, w(u)^e\}$ is not a consistent cut of elastic transaction e.

Five consistent cuts of $\mathcal{H}_2|e$ with respect to $\mathcal{H}_2$ are possible. One contains three sub-histories $\mathcal{C}1 = \{r(x)^e;\ r(y)^e, r(z)^e\ ;\ w(u)^e\}$, three contain two sub-histories $\mathcal{C}2 = \{r(x)^e\ ;\ r(y)^e, r(z)^e, w(u)^e\}$, $\mathcal{C}3 = \{r(x)^e, r(y)^e\ ;\ r(z)^e, w(u)^e\}$, $\mathcal{C}4 = \{r(x)^e, r(y)^e, r(z)^e\ ;\ w(u)^e\}$ and the last one contains one sub-history $\mathcal{C}5 = \{r(x)^e, r(y)^e, r(z)^e, w(u)^e\}$. Observe for example that $\mathcal{C}3$ is consistent because there are no two writes from other transactions that occur at objects between the operations of e on these objects, hence $r(y)^e$ and $r(z)^e$ appear to execute atomically at the time $r(y)^e$ occurs. By contrast, consider history $\mathcal{H}_3$ depicted in Figure 4 where e is elastic and n is regular.

$$\mathcal{H}_3 = r(x)^e, r(y)^e, w(y)^n, w(z)^n, r(z)^e, w(u)^e.$$

The cut $\mathcal{C}3 = \{r(x)^e, r(y)^e\ ;\ r(z)^e, w(u)^e\}$ of $\mathcal{H}_3|e$ with respect to $\mathcal{H}_3$ is not a consistent cut because n writes y and z between the times e reads each of them.

4.3. Elastic Opacity

Here we describe *elastic opacity*, a criterion that captures the consistency property that a system supporting elastic and regular transactions must ensure. Intuitively, a system is elastic opaque if there exist some consistent cuts of its elastic transactions such that: (a) the transactions resulting from these cuts and the regular transactions always access a consistent state of the system (even if they are pending or aborted), (b) they look like they were executed sequentially, and (c) this sequential execution satisfies the real-time precedence of non-concurrent transactions and is legal.

The formal definition of elastic opacity relies on the Definition 3 of the notion of consistent cut, and the definition of opacity of transactions accessing read/write objects [22]. Opacity is ensured by many existing software transactional memories and builds upon the notion of serializability [51] for active transactions. First, we recall the definition of opacity.

Definition 4 (Opacity). A history $\mathcal{H}$ is *opaque* if there exists a history $\mathcal{H}' \in \mathit{complete}(\mathcal{H})$ that satisfies the following properties:

1. All transactions that abort in $\mathcal{H}'$ are invisible.
2. The history $\mathcal{H}'$ is equivalent to a sequential history (where all non-concurrent transactions are ordered as in $\mathcal{H}$) that is legal.

The definition of elastic opacity is parameterized by a constant k, which indicates the size of its cuts. More precisely, this parameter sizes the tuples of consecutive operations of an elastic transaction that should execute atomically. We first define for some given cuts, the mapping of the elastic transactions to the transactions resulting from these cuts. Given a cut $\mathcal{C}_t = s_1^t, ..., s_n^t$ of $\mathcal{H}|t$ for each elastic transaction $t \in \mathcal{H}|\mathcal{E}$, we define a *cutting function* $f_{\mathcal{C}_t}$ that replaces an elastic transaction t by the transactions s_i^t resulting from its cut. More precisely, $f_{\mathcal{C}_t}$ maps a history $\mathcal{H} = \pi_1, ..., \pi_n$ to a history $f_{\mathcal{C}_t}(\mathcal{H}) = \pi_1', ..., \pi_n'$ where if $\pi_i = \langle x, t, v, \mathit{type}\rangle \in s_j^t$ then $\pi_i' = \langle x, s_j^t, v, \mathit{type}\rangle$, otherwise $\pi_i = \pi_i'$, and if $t \in \mathit{committed}(\mathcal{H})$ then $s_j^t \in \mathit{committed}(f_{\mathcal{C}_t}(\mathcal{H}))$, otherwise $s_j^t \in \mathit{aborted}(f_{\mathcal{C}_t}(\mathcal{H}))$ (for any such i, j). We denote the composition of f for a set of cuts $\mathcal{C} = \{\mathcal{C}_1, ..., \mathcal{C}_m\}$ by $f_\mathcal{C} = f_{\mathcal{C}_1} \circ ... \circ f_{\mathcal{C}_m}$.

Definition 5 (k-Elastic-opacity). A transactional system is *k-elastic-opaque* if, for every history $\mathcal{H}$ of this system and every elastic transaction t of $\mathit{permanent}(\mathcal{H})|\mathcal{E}$ there exists a consistent cut $\mathcal{C}_t$ of size k and with respect to $\mathit{permanent}(\mathcal{H})$, such that $f_{\{\mathcal{C}_t\}}(\mathcal{H})$ is opaque.

As an example, consider the following history $\mathcal{H}_4$, illustrated in Figure 5, and assume e is elastic while n is regular and both transactions commit ($\mathit{permanent}(\mathcal{H}_4) = \mathcal{H}_4$):

$$\mathcal{H}_4 = r(x)^e, r(y)^e, r(x)^n, r(y)^n, r(z)^n, w(x)^n, r(t)^e, w(z)^e.$$

This history would clearly not be serializable in the classical sense [51] (with e and n two regular transactions) since there are no sequential histories that allow not only $r(x)^e$ to occur before $w(x)^n$ but also $r(z)^n$ to occur before $w(z)^e$. (As opacity is strictly stronger than serializability, this history will not be opaque either.) However, there exists one consistent cut $\mathcal{C}_e$ of $\mathcal{H}_4|e$ of size 1 with respect to $\mathit{permanent}(\mathcal{H}_4)$, $\mathcal{C}_e = s_1, s_2$ where $s_1 = r(x)^e, r(y)^e$ and $s_2 = r(t)^e, w(z)^e$ such that, for $\mathcal{C} = \{\mathcal{C}_e\}$, we have:
$f_\mathcal{C}(\mathcal{H}_4) = r(x)^{s_1}, r(y)^{s_1}, r(x)^n, r(y)^n, r(z)^n, w(x)^n, r(t)^{s_2}, w(z)^{s_2}$.
And $\mathcal{H}_4$ is 1-elastic-opaque as $f_\mathcal{C}(\mathcal{H}_4)$ is equivalent to a sequential history: s_1, n, s_2 (and $f_\mathcal{C}(\mathcal{H}_4)$ is opaque).

Another non-serializable history example is $\mathcal{H}_5 = r(x)^e, w(z)^n, r(y)^e, w(x)^n, r(z)^e$. There is a consistent cut

$\mathcal{C}_e$ of $\mathcal{H}_5|e$ of size 1 with respect to *permanent*$(\mathcal{H}_5)$, $\mathcal{C}_e = s_1, s_2$ where $s_1 = r(x)^e$ and $s_2 = r(y)^e, r(z)^e$ because $dist_{\mathcal{H}_5|e}(r(x)^e, r(z)^e) > 1$. For $\mathcal{C} = \{\mathcal{C}_e\}$, we have: $f_{\mathcal{C}}(\mathcal{H}_5) = r(x)^{s_1}, w(z)^n, r(y)^{s_2}, w(x)^n, r(z)^{s_2}$. Therefore, $\mathcal{H}_5$ is 1-elastic-opaque as $f_{\mathcal{C}}(\mathcal{H}_5)$ is equivalent to a sequential history: s_1, n, s_2 (and $f_{\mathcal{C}}(\mathcal{H}_5)$ is opaque).

4.4. How and When to Use Elastic Transactions?

Elastic transactions do not relieve the programmer from the burden of understanding the semantics of the operations she wants to write and in particular the programmer needs to know details of the data structure semantics and implementation. Yet, they help a programmer to build upon or extend existing transactional or sequential code. We discuss below the guidelines a programmer should follow to know whether an operation can be implemented using elastic transactions. To this end, we reuse the theory of dependencies [57] that allows recasting serializability of reads and writes in terms of serializability of shared abstract types. By targeting a specific type implementation, the programmer can disregard a series of dependencies between reads and writes, referred to as *insignificant.* These dependencies are sufficient to detect whether the implementation simply needs elastic transactions, or instead requires regular transactions.

The set of ordered pairs $\{t, t'\}$ for which there exist π_i, π_j and x satisfying WAR, RAW or WAW (cf. Section 3.2) forms a relation denoted $\ll$. If $t \ll t'$ then t' depends on t. Intuitively, $t \ll t'$ if t accesses an object later accessed by t'. A history is orderable iff the transitive closure of $\ll$, denoted by $\ll^*$, is a partial order, i.e., there are no cycles in the graph of transactions linked by dependencies [57]. Insignificant dependencies between reads and writes depend on their role in the transaction. For example, assume t aims at inserting a value $v = 5$ in an ordered linked list by executing at some point a read $\pi_i(x)$ indicating that z whose value is 3 is the next node in the list, while a concurrent transaction t' executes a write $\pi_j(x, y)$ to insert node y right before z in the list. As t must insert v at a node located after z in the list and independently from t' insertion, the dependency $\pi_i^t \ll \pi_j^{t'}$ should be disregarded in this specific case.

To determine whether an operation is a good candidate to be an elastic transaction, the programmer must understand whether each of its inner operations must appear as being executed atomically with all others, or if it must appear as being executed atomically with only a few of its consecutive operations in this transaction. To decide upon the type of a new transaction, the programmer simply needs to identify the subset $\mathcal{I}_t$ of dependencies that are insignificant in this transaction and to compare them with the dependencies $\mathcal{I}_{\mathcal{E}}$ that must be insignificant for the transaction to be elastic.

More specifically, the transaction t used to encapsulate a sequence of operations $\pi_1, ..., \pi_n$ can be denoted k-elastic by the programmer iff for all i and for all j such that $i + k + 1 < j \leq n$, the dependency $r_i^t(x_i) \ll^* r_j^t(x_j)$ is insignificant. This set of insignificant dependencies can be denoted $\mathcal{I}_{\mathcal{E},k}$. For example, if an operation on a data structure whose only consecutive pairs of operations need to be executed by one thread at a time, we have that for all i and for any j such that $i + 2 < j \leq n$, the dependency $r_i^t(x_i) \ll^* r_j^t(x_j)$ is insignificant. Therefore, such an operation can be safely encapsulated into a 1-elastic transaction. (Note that this class of operations comprises those that are implemented using hand-over-hand locking.) As the set $\mathcal{I}_{\mathcal{E},\ell}$ is a superset of $\mathcal{I}_{\mathcal{E},\ell'}$ for all $\ell' > \ell$ and as $\ell = 1$ is the minimal value for elastic transactions, we denote this set of insignificant dependencies, $\mathcal{I}_{\mathcal{E},1}$ by $\mathcal{I}_{\mathcal{E},1} = \mathcal{I}_{\mathcal{E}}$.

To conclude, for a given transaction t if $\mathcal{I}_{\mathcal{E}} \subseteq \mathcal{I}_t$ then t can be elastic. Otherwise, t has to be regular.

5. Elastic Transactions: Implementation

In this section we detail a software transactional memory, $\mathcal{E}$-STM, which implements elastic transactions in addition to regular ones with an easy-to-use interface that simply requires the programmer to delimit transactions in sequential code.

The key concept of our implementation of elastic transactions lies in replacing the traditional read set of a regular transaction by a bounded buffer. This reduces the number of tracked conflicts to a constant k independent of the transaction size and guarantees that consecutive accesses are mutually atomic. For the sake of simplicity in the presentation we describe the algorithm ensuring 1-elastic-opacity ($k = 1$), hence the bounded buffer is simply represented by a unique entry field: *last-r-entry*. While choosing $k = 1$ is enough for the correctness of our data structure implementations, the generalization to $k > 1$ can be easily deduced by extending the buffer size.

5.1. $\mathcal{E}$-STM Description

We depict the key components underlying $\mathcal{E}$-STM in Algorithm 1. Algorithm 2 describes the implementation of the parameterized `begin` and `commit` delimiters that the programmer uses to indicate transactions, and the transactional `read` and `write` operations to which TM compilers (e.g., [2, 13, 48]) can automatically redirect the memory accesses invoked within the transaction delimiters. Recall that our model requires all operations to be part of a transaction, thus we do not specify non-transactional operations here, yet $\mathcal{E}$-STM can be made privatisation-safe using extra validation barriers [63]. Finally, Algorithm 3 provides additional helper functions that are used in Algorithm 2.

Basically, $\mathcal{E}$-STM guarantees elastic opacity by combining timestamps, a lazy update strategy, two-phase locking, and atomic primitives that are supported at the hardware level by most common architectures: CAS (Lines 69), `fetch-and-increment` (Line 87), and atomic `loads` and `stores`.

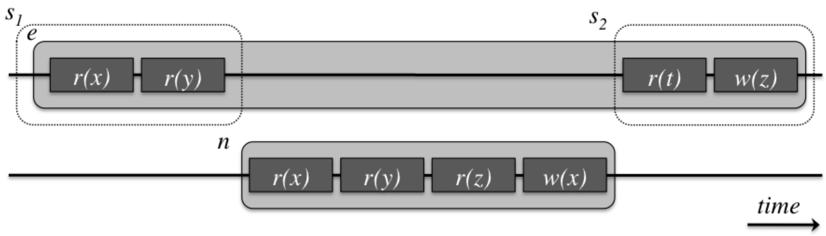


Figure 5: History $\mathcal{H}_4$ is elastic-opaque: elastic transaction e can commit even if transaction n commits as cut s_1, s_2 is consistent.

Algorithm 1 $\mathcal{E}$-STM – State variables

```
1:  clock ∈ ℕ, initially 0

2:  State of variable x:
3:    val ∈ V
4:    tlk a record with fields: // timed lock
5:      owner ∈ T, the lock owner,
6:        initially ⊥
7:      time ∈ ℕ, a version counter,
8:        initally 0
9:      w-entry ∈ X × V × ℕ, an entry
10:       address
11:       initally ⊥
12:     // time/w-entry share same location
13: State of transaction t:
14:   type ∈ {elastic, regular}, initially the
15:     type of the ancestor transaction
16:     or ⊥
17:   r-set, sets of read entries with fields:
18:     addr ∈ X, an address
19:     ts ∈ ℕ, its version timestamp
20:   w-set, sets of write entries with fields:
21:     addr ∈ X, an address
22:     val ∈ V, its value
23:     ts ∈ ℕ, its version timestamp
24:   last-r-entry ∈ X × ℕ, an entry,
25:     initally ⊥
26:   lb ∈ ℕ, initially 0 // time lower bound
27:   ub ∈ ℕ, initially 0 // time upper bound
```

5.1.1. Transaction variables

A transaction t starts with a begin($type$) indicating whether its type is elastic or regular. Then, it accesses the memory locations using read or write operations. Finally, it completes either by a commit call or by an implicit abort that restarts the same transaction. The try-extend and ver-val-ver are helper functions. A transaction t can keep track of the variables it has accessed since it has (re-)started using a read set, *r-set*, to log the reads and a write set, *w-set*, to log the writes. More precisely, the entries of these sets contain the variable address, *addr*, potentially its value *val*, and its version *ts* (Lines 17–23). If t is elastic, it may only need to keep track of the last read operation, so it uses *last-r-entry* (Lines 24 and 25) to log a single address and its version instead of the entire set *r-set*. The two last fields of t indicate a lower-bound *lb* and an upper-bound *ub* on the logical times at which t can be serialized (Lines 26 and 27).

For the sake of clarity in the pseudocode presentation, we consider that each memory location is protected by a distinct lock. We call it the associated memory location of the lock. More precisely, each shared variable x can be represented by a value *val* (Line 3) and a timestamped lock *tlk*, also called versioned write-lock [10]. A timestamped lock has three fields: (i) the *owner* indicating which transaction has acquired the lock, if any, (ii) the *time* the associated memory location of the lock has most recently been written, and (iii) *w-entry*, a reference to the corresponding entry in the owner's write set (Lines 4–11). Timestamps are given by a global counter, *clock* (Line 1), that does not hamper scalability [10, 54, 14].

5.1.2. Regular transactions

The algorithm restricted to regular transactions builds upon TinySTM [14]. Transactions log their operations and use some form of two-phase locking when writing to a memory location: when a transaction performs a write($x, *$), it acquires the lock of x using a CAS (Line 69) and holds it until it commits or aborts. The corresponding update is not reported to the shared memory eagerly but it is lazily buffered into the write-set, *w-set*, until the transaction commit is called. When accessing a locked variable, the transaction detects a conflict and calls the contention manager, through ctn-mgmt, to resolve the conflict (Lines 40 and 63). In our case, we implemented a passive contention manager that simply aborts the current transaction but clever contention management policies could be used instead [62].

When a read request on variable x as part of transaction t is received by $\mathcal{E}$-STM, the value of x is read using the ver-val-ver helper function previously described. The transactions of $\mathcal{E}$-STM use an extension mechanism similar to LSA's [54] and TinySTM's [14]. Each transaction t maintains an interval of time $[lb, ub]$ indicating the time

Algorithm 2 $\mathcal{E}$-STM – Transactional functions

```
28: begin(tx-type)_t:
29:    ub ← clock
30:    lb ← clock
31:    type ← tx-type

32: abort()_t:
33:    for all ⟨x, ∗, ∗⟩ ∈ write-set do
34:       x.tlk.owner ← ⊥

35: read(x)_t:
36:    // log regular reads for later extensions
37:    if type = regular ∨ w-set ≠ ∅ then
38:       ⟨ℓ_x, v_x⟩ ← ver-val-ver(x, true)
39:       if ℓ_x.owner ∉ {t, ⊥} then
40:          ctn-mgmt()
41:       else if ℓ_x.owner = t then
42:          v_x ← ℓ_x.w-entry.val
43:       else // ℓ_x.owner = ⊥
44:          if ℓ_x.time > ub then
45:             try-extend()
46:       r-set ← r-set ∪ {⟨x, ℓ_x.time⟩}
47:    // ...or log only most recent elastic read
48:    if type = elastic ∧ w-set = ∅ then
49:       ⟨ℓ_x, v_x⟩ ← ver-val-ver(x, false)
50:       if ℓ_x.time > ub then
51:          if last-r-entry ≠ ⊥ then
52:             ⟨y, ∗⟩ ← last-r-entry
53:             ⟨ℓ_y, ∗⟩ ← ver-val-ver(y, false)
54:             if ℓ_y.time > ub then abort()
55:             ub ← ℓ_x.time
56:       last-r-entry ← ⟨x, ℓ_x.time⟩
57:    return v_x

58: write(x, v)_t:
59:    // lock and postpone write until commit
60:    repeat:
61:       ℓ ← x.tlk
62:       if ℓ.owner ∉ {⊥, t} then
63:          ctn-mgmt()
64:       else if ℓ.time > ub then
65:          if type = regular then
66:             try-extend()
67:          else abort()
68:       w-entry ← ⟨x, v, ℓ.time⟩
69:       x.tlk ← ⟨t, ∗, w-entry⟩ // cas
70:    until x.tlk.owner = t
71:    lb ← max(lb, ℓ.time)
72:    w-set ← (w-set \ {⟨x, ∗, ∗⟩}) ∪
73:       {w-entry}
74:    // make sure last value read is unchanged
75:    if type = elastic ∧
76:          last-r-entry ≠ ⊥ then
77:       ⟨e, t_e⟩ ← last-r-entry
78:       ⟨ℓ_e, ∗⟩ ← ver-val-ver(e, true)
79:       ow ← ℓ_e.owner
80:       last ← ℓ_e.time
81:       if ow ≠ ⊥ ∨ last ≠ t_e then abort()
82:       r-set ← {last-r-entry}
83:       last-r-entry ← ⊥

84: commit()_t:
85:    // apply writes to mem and release locks
86:    if w-set ≠ ∅ then
87:       ts ← clock++ // fetch&increment
88:       if lb ≠ ts − 1 then try-extend()
89:       for all ⟨x, v, ts⟩ ∈ write-set do
90:          x.val ← v
91:          x.tlk.time ← ts
92:          x.tlk.owner ← ⊥
```

during which t can be serialized. More precisely, for a given transaction t, lb and ub represent respectively the lower and upper bounds on the versions of values accessed by t during its execution. When t reads x, it records the last time x has been modified in its read-set, *r-set*, for future potential checks. Later on, if t accesses a variable y that has been recently updated ($y.tlk.time > ub$), t first tries to extend its interval of time by calling try-extend(). Transaction t detects a conflict only if this extension is impossible (Lines 111), meaning that at least one variable, among the ones t has read, has been updated by another transaction since then.

5.1.3. Elastic transactions

Unlike regular transactions, elastic transactions do not use the *r-set* unless they have previously performed a write, they rather keep track of the most recent read operation. Hence, elastic transactions use the *last-r-entry* field to log this last read operation. In our implementation all reads following a write in an elastic transaction will use the *r-set* like regular transactions (Lines 36–46), however, the implementation could be improved using static analysis to require this only for reads that might be both preceded and succeeded by write operations in the same transaction.

Upon reading x (without having written it before) an elastic transaction must make sure that the value v_x it reads was present at the time the immediately preceding read occurred. This typically ensures that a thread does not return an inconsistent value v_x, for example after having been pre-empted. If the version v_x of the value is too recent, $\ell_x.time > ub$, then the read operation must recheck the value logged in *last-r-entry* to be sure that the value read has not been overwritten since then (Lines 50–55).[4]

[4]This can be viewed as a partial roll-back similar to the one provided by nested models [25], except that no *on-abort* definition is necessary and only a single operation would have to be re-executed here.

Algorithm 3 $\mathcal{E}$-STM – Helper functions

93: **ver-val-ver**(x, *evenlocked*)$_t$**:**
94:     // *load versioned value from memory*
95:     **repeat:**
96:         $\ell_1 \leftarrow x.tlk$
97:         $v \leftarrow x.val$
98:         $\ell_2 \leftarrow x.tlk$
99:     **until** $(\ell_1 = \ell_2 \wedge$
100:         $(\ell_1.owner = \perp \vee evenlocked))$
101:     return $\langle \ell_1, v \rangle$

102: **try-extend**()$_t$**:**
103:     **if** $type =$ elastic **then** abort()
104:     // *make sure read values have not changed*
105:     $now \leftarrow clock$
106:     **for** all $\langle y, ts \rangle \in r\text{-}set$ **do**
107:         $ow \leftarrow y.tlk.owner$
108:         $last \leftarrow y.tlk.time$
109:         **if** $ow \notin \{t, \perp\} \vee$
110:             $(ow = \perp \wedge last \neq ts)$ **then**
111:             abort()
112:     $ub \leftarrow now$

Upon writing x, a similar verification regarding the last value read is made. If the lock corresponding to this address has been acquired, $ow \neq \perp$, or if the version has changed since then, $last \neq time_e$, then the transaction aborts (Line 81). If, however, no other transaction tried to update this address since it has been read, then the write executes as regular (Lines 59–73).

All transactions commit in the same manner by applying buffered writes in memory and by associating a unique higher version to the written location (Line 87).

5.1.4. Helper functions

The ver-val-ver and try-extend are two helper functions. The former function, ver-val-ver, is a three-step process to read a location x. It consists of loading its timestamped lock $x.tlk$, loading its value $x.val$, and re-loading its lock $x.tlk$. This read-version-value-version is repeated until the two versions read are identical (Line 99) indicating that the value corresponds to that version. Note that the counter used here makes the ABA problem (where threads may change a value A to B and then change it back to A while another is executing this ver-val-ver function) irrelevant in practice similarly to [36]. The value needs to be returned unlocked only in some cases (typically not when called to revalidate the *last-r-entry*) hence the use of the boolean *evenlocked*.

The latter function, try-extend, indicates to regular transactions whether their accesses can be part of the same atomic snapshot. More precisely, it aims to advance the time recorded when the current transaction (re-)started by setting its *ub* field at *now*, the clock value at the time try-extend is called. Before doing so, the *r-set* is checked: neither must all read values be currently modified (i.e., locked) by some other transaction (Line 109) nor must they have been modified since the time *last* at which it was lastly read (Line 110); otherwise the transaction aborts. Observe that elastic transactions immediately abort instead of extending as Line 103 indicates. Our current $\mathcal{E}$-STM proposal does not support extension of elastic transactions by the use of try-extend because the hypothetical gain in concurrency would be counterbalanced by the overhead of parsing the read-set. Nevertheless, such extension could be safely enabled for elastic transactions as well to obtain potentially higher performance in some cases.

5.2. Optimizations

$\mathcal{E}$-STM provides a simple interface such that the programmer has simply to delimit transactions using begin and commit and existing compilers (e.g., Intel's C++ STM compiler[5] and the GNU Compiler Collection[6] supporting `__transaction{}` blocks to delimit transactions) can automatically instrument transactional accesses to call appropriately read and write.

This simplicity induces some overhead in the validation mechanism when executing transactions that are *dynamic* as their set of memory accesses is unknown prior to execution. More precisely when writing for the first time, an elastic transaction must keep track of the version of the location it has just read, however, the written location cannot be determined statically, hence, the version of each memory location that is read must be recorded.

This overhead can be reduced at the cost of breaking the TM interface by explicitly indicating which read operation must be recorded and allowing the other reads of the transactions not to be recorded. We discuss alternative language constructs that could be used to implement elastic transactions.

5.2.1. Early release

There exists an explicit release function that is optionally part of DSTM [34] and allows forgetting about the metadata of any preceding read operation. This mechanism enhances concurrency by decreasing the number of low-level conflicts for some pointer structures. $\mathcal{E}$-STM forgets automatically past reads but checks memory location versions to maximize concurrency without hampering consistency. Early release could be used to implement the elastic transactional model by placing release calls at the right places within a transaction.

[5] http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition

[6] http://www.velox-project.eu/software/gcc-tm

5.2.2. Unit reads

Another potential optimization relies on unit-reads that do not record any metadata as opposed to the usual reads. Prior to this work, we have extended the TinySTM [14] interface to differentiate unit-reads from usual reads that writes timestamps into the transaction read-set. They are part of the current distribution at `http://tmware.org/tinystm`. More precisely, the unit-read on x checks whether x is acquired and spins loading the version of x until the version indicates that x gets released. If not acquired, x is loaded and then its version is reloaded to make sure that x was not modified concurrently as in the ver-val-ver function of $\mathcal{E}$-STM. Upon termination, unit-read returns the pair of value-version loaded.

We have implemented the elastic transactional model with unit-read before developing $\mathcal{E}$-STM. We obtained better performance than with $\mathcal{E}$-STM but we had to change the sequential code of the application as follows. First, the transaction must start getting the current version of the global counter. Second, each time the transaction unit-reads, it has to compare the returned version to its current version. Third, the application must use two distinct variables for storing the previous read address and the last read address. Finally, the application can try to revalidate explicitly the previous read location by executing an additional unit-read, in case the last version loaded is larger than the current transaction version. If the version of the previous read location has been updated in the meantime as well, then the transaction aborts.

View transactions [3] propose light-read as a language construct that provides a lightweight version of read operation similar to a unit-read. It comes in complement to view-pointers used to keep track of read locations forming a critical view the transaction has to revalidate. We believe that light-reads could be used similarly to unit-reads to implement the elastic transaction model, although we did not verify this theory.

6. Putting Elastic Transactions to Work

Here we give the implementation of an integer set abstraction on four data structures using elastic transactions: *double-ended queue*, *hash table*, *linked list*, and *skip list*. We also extend the set abstraction into a dictionary abstraction with operations, move and sum, to illustrate how to compose transactions and to extend concurrent programs, and to show the performance results in more complex workloads. The code of the benchmarks and algorithms we implemented is available in Synchrobench [18] at `http://github.com/gramoli/synchrobench`.

These implementations indicate that elastic transactions are faster than regular transactions and simpler to use than lock-based and lock-free alternatives. In fact, our $\mathcal{E}$-STM is simple to program with for three reasons: (i) it provides a high-level abstraction that does not expose synchronization mechanisms to the programmer, (ii) it preserves sequential code as transaction delimiters are added without changing the sequential code, and (iii) it enables code composition as transactions can be composed into another transaction.

6.1. Program Simplicity

As with a regular transactional model, the programmer can use $\mathcal{E}$-STM to write a concurrent program almost as a sequential program by simply labeling regions of sequential code as transactions. Below we compare $\mathcal{E}$-STM linked list with the lock-free find function from Harris [24] (Algorithm 4) and we compare $\mathcal{E}$-STM double-ended queue with its corresponding non-thread safe sequential code (Algorithm 5).

6.1.1. Linked list

A linked list is a data structure appealing for its simplicity and flexibility, it provides insert and remove operations that only affect a localized part of the data structure, making it a more concurrency-friendly data structure than balanced data structures [64]. In contrast with a linked list, an array needs to be reorganized to keep the mapping of index to values consistent despite modifications. As noted in the Introduction, linked-lists are at the core of more complex data structures, like bucket hash tables.

Algorithm 4 depicts a sorted linked list implementation of an integer set, where integers (node *keys*) can be searched, removed, and inserted. The implementation is based on $\mathcal{E}$-STM but we also provide the pseudocode of a lock-free harris-ll-find function for comparison purpose. This function is at the core of the lock-free linked list of Harris [24]. It is clear that the harris-ll-find function is more complex than its ll-find counterpart based on $\mathcal{E}$-STM. In fact, harris-ll-find relies on the use of a mark bit to indicate that a node is logically deleted, and must physically delete the nodes that have been logically deleted to ensure that the size of the list does not grow monotonically.

Unlike the Harris lock-free implementation, $\mathcal{E}$-STM-based functions are very simple, as all synchronizations are handled transparently underneath. The pseudocode on the left is the same as the non-thread-safe sequential version on the right, except that begin(*elastic*), and commit have been placed to delimit the transaction that must appear as atomic. Note that accesses to *key* do not need to be instrumented as *key*s are immutable (Lines 18, 24, 33, 39), this detection could be reasonably automated by the compiler.

6.1.2. Double-ended queue

The double-ended queue data structure (also referred to as *deque*) generalizes the traditional first-in-first-out queue and the last-in-first-out heap by providing pop and push operations at both ends. This data structure is well known for its intricateness [33] when supporting concurrent accesses but is not strictly speaking a search data structure as its elements are not necessarily searched. Here we show

Algorithm 4 Linked list implementation with elastic transactions (the lock-free harris-ll-find is given for comparison)

```
1:  State of process p:
2:    node a record with fields:
3:      key, an integer
4:      next, a node
5:    set a linked-list of nodes with:
6:      head at the beginning,
7:      tail at the end.
8:    Initially, the set contains head and
9:      tail nodes, and head.key = min,
10:     and tail.key = max.

11: free(x)_t:
12:   // memory disposal is postponed
13:   write(x, 0) // write all words of x

14: ll-find(i)_p:
15:   curr ← set.head
16:   while true do
17:     next ← read(curr.next)
18:     if next.key ≥ i then break
19:     curr ← next
20:   return ⟨curr, next⟩

21: ll-insert(i)_p:
22:   begin(elastic)
23:   ⟨curr, next⟩ ← ll-find(i)
24:   in ← (next.key = i)
25:   if !in then
26:     new-node ← ⟨i, next⟩
27:     write(curr.next, new-node)
28:   commit()
29:   return !in

30: ll-search(i)_p:
31:   begin(elastic)
32:   ⟨curr, next⟩ ← ll-find(i)
33:   in ← (next.key = i)
34:   commit()
35:   return in

36: ll-remove(i)_p:
37:   begin(elastic)
38:   ⟨curr, next⟩ ← ll-find(i)
39:   in ← (next.key = i)
40:   if in then
41:     n ← read(next.next)
42:     write(curr.next, n)
43:     free(next)
44:   commit()
45:   return in

46: harris-ll-find(i)_p:
47:   loop
48:     t ← set.head
49:     t-next ← read(curr.next)
50:     // 1. find left and right nodes
51:     repeat:
52:       if !is-marked(t-next) then
53:         curr ← t
54:         c-next ← t-next
55:       curr ← unmarked(next)
56:       if !t-next then break
57:       t-next ← t.next
58:     until is-marked(t-next)∨
59:       (t.key < i)
60:     next = t
61:     // 2. check nodes are adjacent
62:     if c-next = next then
63:       if (next.next∧
64:           is-marked(next.next) then
65:         goto line 48
66:       else
67:         return ⟨curr, next⟩
68:     // 3. remove one or more marked node
69:     if cas(curr.next, c-next,
70:         next) then
71:       if (next.next∧
72:           is-marked(next.next)) then
73:         goto line 48
74:       else
75:         return ⟨curr, next⟩
76:   end loop
```

the simplicity of programming with $\mathcal{E}$-STM by describing how a programmer can simply modify the sequential (non-thread-safe) double-ended queue code to obtain the concurrent (thread-safe) counterpart.

Our $\mathcal{E}$-STM-based implementation of the deque relies on a circular array and uses an oracle function (Lines 13 and 23) similar to the implementation of [33]. This oracle function, whose pseudocode is omitted here, returns the index of the rightmost (resp. leftmost) null value when called with argument right (resp. left). In contrast with the default oracle proposed in [33] that is eventually accurate, ours uses two values that indicate (always accurately) the indices in the array of the leftmost and rightmost null markers.

The code for the dq-rightpush and dq-rightpop is given in Algorithm 5, the left counterpart (dq-leftpush and dq-leftpop) is symmetric and can be deduced from this code. The non-thread-safe sequential code is given on the right side of the figure to show the few changes one has to apply to the sequential code to obtain the concurrent version, when using $\mathcal{E}$-STM. The circular array contains $\mathsf{max}+1$ values: there are at most max non null values in the queue and at least one null value, which serves as a marker. More precisely, the values represent at any time a sequence of non-null values followed by several null values. Note that there is always a single sequence of null values

Algorithm 5 Double-ended queue implementation with elastic transactions (the sequential seq-dq-rightpush and seq-dq-rightpop are given for comparison)

```
1:  State of process p:
2:    node a value
3:    q an array of nodes with:
4:      null the null value
5:    max the capacity of the double-ended
6:      queue
7:    Initially, the queue contains a
8:      sequence of null followed by 256
9:      nodes with arbitrary values.

10: dq-rightpush(i)_p:
11:   begin(elastic)
12:   result ← false
13:   i ← oracle(right)
14:   next ← read(q[(i + 1)mod(max + 1)])
15:   if next = null then // queue not full
16:     write(q[i], val)
17:     result ← true
18:   commit()
19:   return result

20: dq-rightpop(i)_p:
21:   begin(elastic)
22:   result ← false
23:   i ← oracle(right)
24:   prev ← read(q[(max + i)mod
25:     (max + 1)])
26:   if prev ≠ null then // q is not empty
27:     write(q[(max + i)mod(max + 1)],
28:           null)
29:     result ← true
30:   commit()
31:   return result

32: seq-dq-rightpush(i)_p:
33:
34:   result ← false
35:   i ← oracle(right)
36:   next ← q[(i + 1)mod(max + 1)]
37:   if next = null then // queue not full
38:     q[i] ← val
39:     result ← true
40:
41:   return result

42: seq-dq-rightpop(i)_p:
43:
44:   result ← false
45:   i ← oracle(right)
46:   prev ← q[(max + i)mod(max + 1)]
47:
48:   if prev ≠ null then // q is not empty
49:     q[(max + i)mod(max + 1)] ← null
50:
51:     result ← true
52:
53:   return result
```

as the array is circular.

When a value is pushed to the right end of the deque, the oracle gives the index i of the leftmost null value. If the value at index $(i+1)mod(\mathsf{max}+1)$ is also null then we know that the deque is not full and that we can push a new value at index i (Lines 15 and 37). Similarly if the value at index $(\mathsf{max}+i)mod(\mathsf{max}+1)$, i.e., the index that immediately precedes i in the circular array, is not null then we know that the deque is not empty so that we can pop the value at index $(\mathsf{max}+i)mod(\mathsf{max}+1)$ (Lines 26 and 48).

6.1.3. Skip list

Skip lists [52] are known to provide logarithmic search time complexity in expectation. They are generally simpler to program than balanced trees. For instance, deletion and insertion in a red-black tree induce complex rebalancing operations. It is however not straightforward to write a lock-free skip list as it has been the main contribution of some research papers [16].

The skip list can be viewed as a sort of linked list where each node maintains several next pointers, one for each level of the skip list. Each node has a random level, so that its neighbor at level l has at least a level as large as l. (For further details on the data structure, please refer to [52].) Similarly to the linked list implementation, sl-insert, sl-search, and sl-remove start an elastic transaction and use sl-find to traverse the list. Upon update, sl-insert and sl-remove modify the localized part of the data structure returned by the sl-find.

6.2. Program Extensibility

Here we discuss the extensibility of a program taking hash table as an example. As we show, the extensibility depends on its implementation: whether it relies on transactions, locks, or CAS. A hash table data structure provides constant access time. It uses a hash function to map a key to a sorted linked list in which the associated value must be stored.

$\mathcal{E}$-STM allows combining elastic transactions with regular transactions. As a result the code that uses $\mathcal{E}$-STM is easily extensible. Transactions are *extensible* as they allow any existing transaction-based data structure with a fixed set of operations to be extended with other transaction-based operations without changing the pre-existing operations. As we show in Section 7, trying to extend a lock-based structure with another operation may seriously hamper performance while it is unclear how one could extend certain lock-free structures. To illustrate this, we extended the integer abstraction mentioned in Section 1 into a dictionary abstraction with operations move and sum. The pseudocode is presented in Algorithms 6 and 7.

6.2.1. Hash table

We extended the set abstraction into a dictionary abstraction, mapping a key to a value, by adding operations move and sum, depicted in Algorithm 7, to the insert, search, and remove operations of Algorithm 6. Since each bucket of the hash table is implemented with a linked list (Lines 14, 20, and 26 of Algorithm 6), we slightly modified the program of the linked list written in Subsection 6.1.1 to assign a key-value pair to each node of the list. More precisely, a search and a remove return the value associated with the searched key and insert takes a key-value pair as a parameter. Both move and sum operations in Algorithm 7 are simply implemented using regular transactions to show that these transactions combine safely with the concurrent elastic transactions of the search. While not specified in the pseudocode of $\mathcal{E}$-STM, the elastic transaction (Lines 18 and 24) nested inside the regular transaction of move executes as a regular transaction whose begin and commit would simply be ignored. Although it is also possible to implement an elastic version of move, sum cannot be elastic as it requires an atomic snapshot of all elements of the data structure. This example illustrates the way elastic and regular transactions can be combined.

Observe that, although moving a value from one node to one of its predecessors in the same linked list may lead an elastic search to ignore the moved value, the two operations remain correct. Indeed, the search looks for a key associated with a value while the move changes the key of a value v. Hence, if the search looks for the initial key k of v and fails to find it, then the search will be serialized after the move, if the search looks for the targeted key k' of v and does not find it, then the search will be serialized before the move. By contrast, a less usual search-value operation looking for the associated value rather than the key of an element would have to be implemented using regular transactions, otherwise, a concurrent move may lead to an inconsistent state. Another issue, pointed out in [61] and similar to the write-skew problem in databases, may arise when one transaction inserts x if y is absent and another inserts y if x is absent. If executed concurrently, these two transactions may lead to an inconsistent state where both x and y are present. Again, our model copes with this issue as the regular transaction model is provided to encapsulate each conditional insertion. All these regular and elastic transactions are safely combined.

6.2.2. Summing up a lock-based or lock-free hash table

A problem, similar to the *inconsistent analysis* problem [5] in databases, arises with the lock-based hash table implementation we have used when willing to extend it with a sum operation that takes an atomic snapshot. As insert and remove lock only the elements that need to be modified and the sum must detect this modification to ensure that its returned value is correct, hence the sum must lock all elements. One may think that hand-over-hand locking would be sufficient if, starting by locking the first element of the hash table, sum locks the $(i+1)^{st}$ element before releasing the i^{th} until the last element. Unfortunately, a concurrent move operation may not be detected by the sum if it removes y before sum reads it and inserts x after sum could read it. In this example, sum would miss one element because it should have read either y (if linearized before the move) or x (if linearized after the move).

Conversely, $\mathcal{E}$-STM transactions let the sum and move execute concurrently, and if such inconsistency happens then the sum detects it at commit time thanks to the timestamped locks of the modified elements.

A correct sum extension of the lock-based hash table, denoted by lock-ht-sum and depicted in Algorithm 7 (right), takes a snapshot of the elements after having locked all of them and before unlocking any of them. It uses a special validation technique to check whether a node has been deleted (we omit here the calls to unlock deleted elements). Note that an alternative code using a coarser-grained locks to sum up the elements would certainly be more efficient but would require to change the original functions, thus violating extensibility. In Subection 7.4.2, we show the performance slowdown of using function lock-ht-sum to illustrate the limitations reached when extending the lock-based code.

As far as we know, there is no efficient implementation of a sum in a CAS-based lock-free manner. We further discuss the issue of lock-based and lock-free move extensibility in Subsections 7.4.2 and 7.5.2, respectively.

7. Performance Results

We evaluate the performance of elastic transactions implemented in C and Java on the aforementioned data structures: double-ended queue, hash table, linked list and skip list. We compare our results with the performance of: (i) non-thread-safe sequential code, (ii) regular transactions, (iii) fine-grained locking techniques, and (iv) CAS-based lock-free techniques.

7.1. Experimental Settings

We measure the *throughput* of each implementation depending on the level of parallelism and the update ratio as it is generally the case in other STM evaluations on micro-benchmarks [60, 34, 10, 54, 56]. The throughput represents the number of operations per millisecond averaged over 5 runs of at least 2 seconds each. The level

Algorithm 6 Hash table implementation with elastic transactions

```
State of process p:
  node a record with fields:
    key, an integer
    val, an integer
    next, a node
  map a mapping from an integer to
    a linked list representing a bucket
    of key-value pairs.
  Initially, all buckets of the map
    are empty lists.

ht-search(k)_p:
  begin(elastic)
  a ← hash(k)
  v ← map[a].ll-search(k)
  commit()
  return v

ht-insert(k, v)_p:
  begin(elastic)
  a ← hash(k)
  result ← map[a].ll-insert(k → v)
  commit()
  return result

ht-remove(k)_p:
  begin(elastic)
  a ← hash(k)
  v ← map[a].ll-remove(k)
  commit()
  return v
```

Algorithm 7 Extension of the hash table implementation based on elastic transactions (the lock-based lock-ht-sum, given for comparison, extends the efficient lock-based hash table)

```
ht-sum()_p:
  begin(regular)
  for each bucket in map do
    next ← read(bucket.head.next)
    while next.next ≠ ⊥ do
      sum ← sum + read(next.val)
      next ← read(next.next)
  commit()
  return sum

ht-move(k, k')_p:
  result ← false
  begin(regular)
  v ← ht-remove(k)
  if v ≠ ⊥ then
    result ← ht-insert(k', v)
  commit()
  return result

lock-ht-sum()_p:
  for each bucket in map do
    repeat:
      lock(bucket.head)
      lock(bucket.head.next)
      curr ← bucket.head
      next ← bucket.head.next
    until validate(curr, next)
    // Until they are non logically deleted
    while next.next do
      while true do
        lock(next.next)
        curr ← next
        next ← curr.next
        if validate(curr, next) then
          if !is_marked(next) then
            sum ← sum + curr.val
          break()
  for each bucket in map do
    curr ← bucket.head
    next ← bucket.head.next
    unlock(curr)
    unlock(next)
    while next.next do
      curr ← next
      next ← curr.next
      unlock(next.lock)
  return sum
```

of parallelism ranges from 1 thread to 16 threads as we experimented on a 4 quad-core AMD Opteron machine, i.e., including 16 cores in total. (Additional results on an 8-core Niagara 2 machine uses up to 64 threads in Subsection 7.6.) In a given workload, each thread executes a series of randomly chosen operations whose proportion depends on the update ratio. If not specified otherwise, the type and arguments of operations are chosen uniformly at random.

To compare the throughput obtained using elastic transactions against the throughput from other synchronization techniques, we computed the *speedup* as the throughput of $\mathcal{E}$-STM over the throughput of the other synchronization techniques minus 1. We subtracted 1 so that a positive speedup indicates a performance improvement and a negative speedup indicates a performance slowdown, similarly to what we did in [11]. For the sake of clarity, when the speedup is always negative we refer to its opposite as the *slowdown* of $\mathcal{E}$-STM, i.e., $slowdown = -speedup$. It is worth mentioning that the sequential throughput $s(u)$ for update ratio u is taken from a single-threaded execution (non instrumented and without any synchronization primitive), whereas we computed the throughput $e(u, t)$ of $\mathcal{E}$-STM depending on update ratio u and thread count t ranging from 1 to 16.

The *update ratio* is the percentage of update operations over the total number of operations and can be of two types. First, we executed workloads containing from 0 to 100% of update operations (e.g., insert operation) and the rest were read-only operations (e.g., search). This update ratio, however, does not imply that the memory is written by each of these "update" operations as, for instance, an insert operation might fail by finding that the element to insert is already present. We thus refer to this ratio as the *update attempt* ratio.[7] Second, we executed adaptive workloads to obtain the desired amount of effectively updating operations. These workloads call a read-only, or an update operation, depending on the amount of successful update operations that have been executed in order to obtain the desired amount of update ratio. It was thus impossible to test high update ratios in these workloads while keeping the benchmarks randomized. We tested four different workloads including 0%, 5%, 10% and 15% of update operations that effectively modify the memory and we made sure that the error margin was lower than 1‰ (1 over a thousand).[8]

7.2. Comparing against Sequential

Figure 6 conveys the overview of the speedup of an $\mathcal{E}$-STM linked list over a sequential (non-thread-safe) linked list. (A more detailed comparison will be given in Figures 9 and 13.) The linked list is initially filled with 2^{16} elements, then search, insert and remove are executed with proportions 90-5-5 and the targeted values are randomly chosen among a range of 2^{32} integers so that one operation over two succeeds on average.

As depicted in Figure 6, the $\mathcal{E}$-STM version is slightly slower than the sequential one on a single thread as the

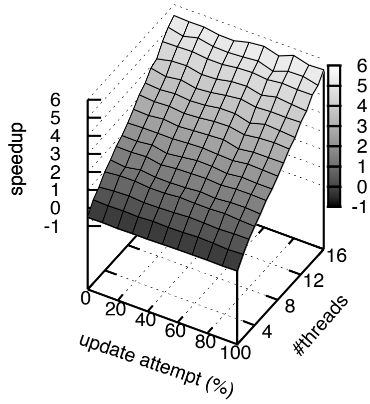


Figure 6: Speedup of $\mathcal{E}$-STM linked list over sequential linked list (2^{16} elements).

speedup is negative when running one thread for all update ratios. This is due to the overhead of $\mathcal{E}$-STM handling metadata for synchronizing threads accessing shared memory locations. Unlike the sequential operations, $\mathcal{E}$-STM transactions lock some locations upon update and log each access to always keep track of the last one. Moreover, the higher the thread count, the higher the speedup of the $\mathcal{E}$-STM linked list over the sequential linked list. The reason is because $\mathcal{E}$-STM performance scales well with the level of parallelism. Finally, we can see that the impact of the update ratio is negligible compared to the impact of the thread count. The linked list is sufficiently large so that two transactions are likely to modify distinct elements of the linked list (even with 100% update attempts). Updating an element that has been read by a concurrent elastic transaction does not prevent this elastic transaction from successfully committing. The conjunction of these two causes leads to the efficiency observed in Figure 6.

Figures 11 and 12 (deferred to the end of this section, p.24–23) present the performance of all our implementations of the 4 data structures initially set up with 2^{12} elements or 2^{16} elements and for update ratios from 0 to 15%. There is an exception with the double-ended queue results as this data structure provides only update operations and the number of elements in the data structure does not impact performance: we only presented the obtained results for 2^{12} elements. These graphs represent the throughput as the thread count grows from 1 to 16. In Figures 11 and 12, we represent sequential performance using a solid line.

Our general observation is that $\mathcal{E}$-STM scales well and outperforms sequential performance in most of the cases. $\mathcal{E}$-STM is slower only for benchmarks with 15% effectively updating transactions. Interestingly, the $\mathcal{E}$-STM linked list suffers less from contention than $\mathcal{E}$-STM hash table when compared to sequential. Indeed, the hash table provides constant access time and the duration of sequential operations does not depend on the data structure size. $\mathcal{E}$-STM transactions induce some overhead independently of their

[7] To keep the expectation of the data structure size constant over the execution, we choose to execute either one of insert and remove operations uniformly at random and their arguments are taken among a range of integer values that is twice as big as the targeted size.

[8] For these workloads, the element to insert is chosen in a range of 2^{32} integer values and the value to remove was the last inserted value. We made sure that removing the lastly inserted value was not impacting the performance of the red-black tree benchmark as this could bias the frequency with which the tree is rebalanced.

length by incrementing a centralized counter at commit time. The contention induced on this counter and the resulting overhead of this increment become significant due to the shortness of the hash table transactions. On the $\mathcal{E}$-STM linked list, as the operation cost is linear in the data structure size, operations are slower and transactions are longer as more elements have to be accessed. Hence, the inherent overhead of $\mathcal{E}$-STM transactions becomes negligible in the face of the length of these transactions.

Finally and as illustrated in Figure 9, the extended $\mathcal{E}$-STM hash table scales well with up to at least 2^{12} elements in it. In contrast with standard search, insert, remove operations accessing a single (often independent) bucket, the move and sum operations access several buckets. This is the reason why the performance of the extended $\mathcal{E}$-STM hash table decreases when the data structure enlarges (to 2^{14} elements), but this is also the reason why the sequential implementation remains less efficient than the $\mathcal{E}$-STM version in all cases.

7.3. Comparing against Regular Transactions

We compare our linked list implementation using elastic transactions against the same implementation using regular transactions. To this end, we used a state-of-the-art STM library, TinySTM.[9] Note that we could have compared against other regular transaction libraries, like NOrecSTM [7], but $\mathcal{E}$-STM regular transactions are more similar to TinySTM's. Since then, elastic transactions has been implemented in other libraries, like PSTM [19, 20].

Figure 7 conveys the throughput of the TinySTM linked list (left) and the speedup gained by using $\mathcal{E}$-STM (right) depending on the number of threads and the update ratio. Clearly, $\mathcal{E}$-STM outperforms TinySTM: (i) the speedup is always positive and, more importantly, (ii) the speedup increases with the contention: not only when the update ratio grows but also when the level of parallelism grows. More specifically, $\mathcal{E}$-STM is up to 15 times faster than TinySTM on this benchmark—the maximum speedup is reached with a 100% update ratio for the maximum amount of threads so that we can expect a higher speedup for higher levels of parallelism. (See Section 7.6 for experiments on 64 hardware threads.) This is essentially due to the fact that regular transactions require all the values read to be present in the data structure at a common point in time (as if an atomic snapshot was required) whatever the operation is, whereas elastic transactions at the core of $\mathcal{E}$-STM relaxes this requirement whenever the read operations are simply used to parse the data structure.

Figure 11 and 12 (deferred to the end of this section, p.24–23) depict, among others, the throughput of $\mathcal{E}$-STM data structures and the throughput of TinySTM data structures. All $\mathcal{E}$-STM data structures run faster than TinySTM data structures except the $\mathcal{E}$-STM double-ended queue. The linked list speedup is 170% on average and up to 514%, while the skip list speedup is 6% on average and up to 12%, and the hash table speedup is 2% on average and up to 8%. We can clearly see that the improvement of elastic transactions increases with the access complexity. This is unsurprising as elastic transactions enhance concurrency by cutting themselves, and the longer transactions we have, the higher concurrency enhancement we get.

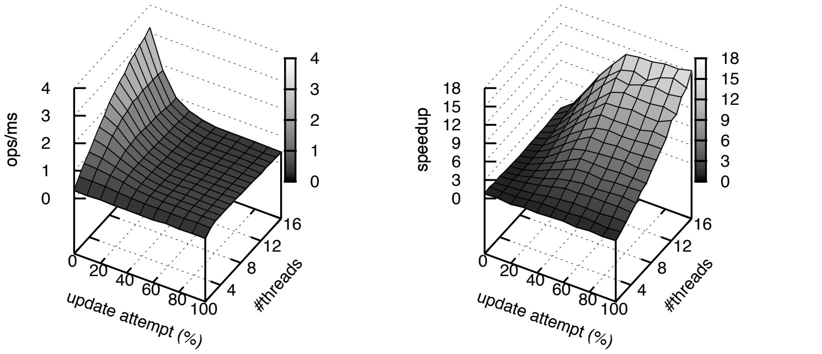
The double-ended queue is an interesting benchmark capturing the overhead of elastic transactions when negligible concurrency can be exploited because transactions are too short. We have tested $\mathcal{E}$-STM on this benchmark using exclusively elastic transactions, their slowdown with respect to TinySTM regular transactions is 3%. The double-ended queue operations always update the data structure by popping or pushing some element (as long as the data structure is neither empty nor full). Moreover, all operations have constant complexity (independent from the deque size). Consequently, deque is a highly contended benchmark where concurrency is hardly exploitable. The performance for 16 threads is surprisingly high for both solutions, a closer look at the performance for each thread count revealed high variation depending on the number of threads running; we believe this is more affected by cache miss effect than other benchmarks due to its high contention. The slowdown of elastic over the regular transactions of TinySTM is due to the cost of trying to exploit concurrency: when writing for the first time, an elastic transaction has to re-read the last read element using its three step ver-val-ver helper function: this consists in three loads, loading the version, loading the value, and re-loading the version a second time to ensure that the value-version mapping is consistent. By contrast, none of the writes inside a regular transaction require this extra re-read. Since all transactions read then write and are very short, the $\mathcal{E}$-STM overhead due to the first write operations represents a significant portion of the execution time.

It is noteworthy, however, that $\mathcal{E}$-STM also provides regular transactions. Hence, a programmer knowing that regular transactions are better suited for such short transactions could implement the $\mathcal{E}$-STM deque by either combining elastic with regular transactions or using exclusively regular transactions to obtain the observed TinySTM performance.

7.4. Comparing against Lock-based Techniques

To compare locks against our $\mathcal{E}$-STM linked list, we implemented a fine-grained locking linked list based on the lazy algorithm [26]. This algorithm is not only faster than coarse-grained alternatives due to finer critical sections but also faster than other existing fine-grained locking algo-

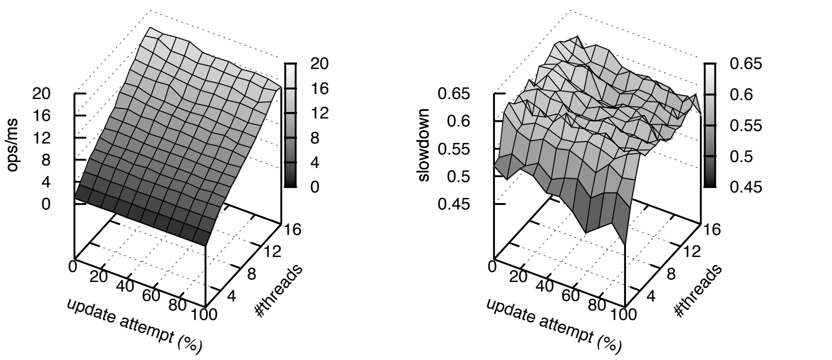
[9] More precisely, we used the most up-to-date version of TinySTM available at the time of the writing (v.0.9.9) with encounter-time locking (i.e., eager acquirement) strategy. TinySTM provides other strategies but encounter-time locking is the one used by $\mathcal{E}$-STM, hence facilitating comparison.



(a) The TinySTM linked list

(b) Speedup of the $\mathcal{E}$-STM linked list over TinySTM linked list

Figure 7: The $\mathcal{E}$-STM linked list compared to the TinySTM linked list (2^{16} elements).



(a) The lock-based linked list

(b) Slowdown of the $\mathcal{E}$-STM linked list compared to the lock-based linked list

Figure 8: The $\mathcal{E}$-STM linked list compared to the lock-based linked lists (2^{16} elements).

rithms [35], including the hand-over-hand locking one[10]. In addition, we re-implemented the Java skip list of [31] in C. This is the most recent lock-based skip list algorithm we know of and it outperforms its CAS-based lock-free counterparts in some circumstances [30].[11] Finally, our fine-grained locking bucket hash table maps a value to the bucket that indicates its location in the data structure. As all modifications apply to the internal buckets, our fine-grained locking hash table simply uses one lazy linked list for each of its buckets.

7.4.1. Advantages of fine-grained locking

The $\mathcal{E}$-STM linked list and hash table execute generally slower than the fine-grained locking versions as conveyed on Figures 11 and 12 (deferred to the end of this section). More precisely, the hash table slowdown is 79% on average and the linked list slowdown is 60% on average. This is due to the cost of logging and the potential cost of updating the timestamps that are used even in the absence of contention in $\mathcal{E}$-STM. The lazy linked list and hash table, and the optimistic skip list, all lock a small number of elements in each operation. This makes the locking complexity negligible as the data structure size grows. As illustrated on Figure 8, the slowdown of $\mathcal{E}$-STM when compared to the lazy linked list is clearly visible as the length of the data structure, 2^{16}, is particularly high. Interestingly, this slowdown does not increase significantly as parallelism grows beyond 4 threads, and even decreases with additional updates. Indeed, these fine-grained locking algorithms additionally check that an element is still reachable to avoid ending up in a deleted part of the data structure. While the cost of this check depends on the data structure size, it remains lower than the cost of using the $\mathcal{E}$-STM linked list.

7.4.2. Advantages of $\mathcal{E}$-STM

Figure 9 presents the performance of the extended hash table implementation for the dictionary abstraction. As the number of elements inserted in a hash table depends on the workload while the number of buckets depend on the frequency with which the hash table gets resized the load factor, which is the ratio of elements per buckets, is not necessarily 1. Here we used a load factor of 10 to explore the effect of a little bit of contention. Operations search, sum, and move are executed with ratio 89-10-1. The first drawback is related to the use of locks: our lock-based move implementation had to lock elements in ascending keys order to avoid deadlocks. The average speedup for all data structure sizes of $\mathcal{E}$-STM over the corresponding fine-grained locking version is 4.67×. In fact, the lazy algorithm, highly optimized for the basic search, insert and remove operations, cannot support efficiently a sum operation.

As explained in Subsection 6.2, the sum operation of the lazy algorithm has to lock all elements to ensure that some element is not concurrently being modified while the sum executes. This locking technique prevents concurrency between the sum and any update operation. While a move changes the key of a value, the sum is thus guaranteed to count either the value once located at its source key or once located at its destination key, but not both. Even though the search is wait-free and ignores all the locks [26], the lazy algorithm is not affordable as its performance is worse than sequential in all cases. An alternative solution would be to re-write the fine-grained locking implementation of search, insert and remove from scratch keeping in mind that sum, and move have to run concurrently. This problem outlines the extensibility limitations of lock-based programs.

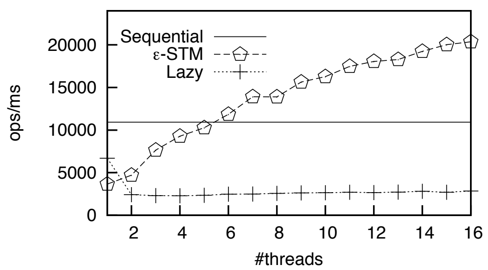
In contrast with the lazy technique, the $\mathcal{E}$-STM sum operation tolerates concurrency with other update operations. This is due to the use of timestamped locks instead of usual locks to determine whether a memory location can be accessed. More precisely, the sum parses the data structure checking the elements and its respective timestamp and a concurrent update can modify the elements that have just been read without violating the linearizability of the operations.

Figure 12, at the end of this section, conveys the performance obtained with our skip list implementations. The $\mathcal{E}$-STM skip list scales better than the lock-based skip list and runs 46% faster. A potential reason is that the use of timestamped locks in the implementation of $\mathcal{E}$-STM allows more concurrency than what usual locks enable. More precisely, the lock-based version requires locking all the next pointers of an element before modifying this element, hence it is like a three phase process: one phase for locking next pointers, a second one to update them, and a third to unlock them. Conversely, the timestamped lock simply checks the timestamp associated with these next pointers to decide whether to update these next pointers and their associated timestamps or to roll-back.

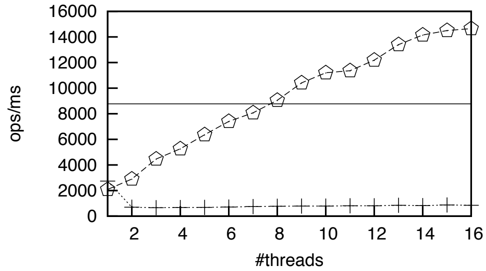
An additional reason is the backoff mechanism we had to add to the lock-based optimistic skip list. While experimenting, we discovered live-locks preventing the lock-based skip list algorithm from terminating. As pointed out in [31], under high contention two concurrent remove and insert operations can repeatedly fail as they both expect validation ensuring that the predecessor in the skip list has not been updated. In fact, we noticed in our experiment a long loop between at least two threads that were repeatedly re-reading concurrently modified elements and attempting to validate over and over. We added an exponential backoff strategy to the fine-grained locking skip

[10] The hand-over-hand locking technique is also known as *lock-coupling* [4].

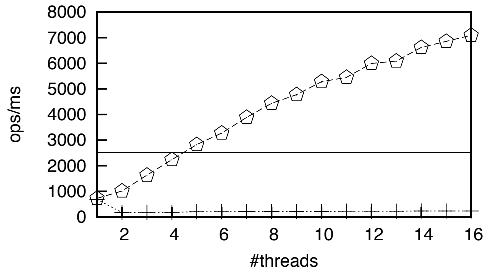
[11] We implemented both `mutex` and `spinlock` versions of these algorithms and observed that all `spinlock` versions run faster than the `mutex` versions and up to 50% faster on the hash table data structure. This performance difference comes from the frequency at which the locks are acquired and the period during which a lock is held: in all these algorithms the acquirement frequency is high and the holding period is rather low so that the context switches of the `pthread_mutex` are too costly.



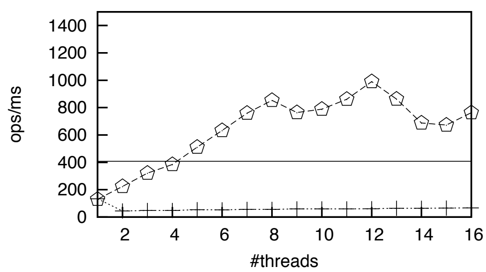
(a) Hash table with 2^8 elements



(b) Hash table with 2^{10} elements

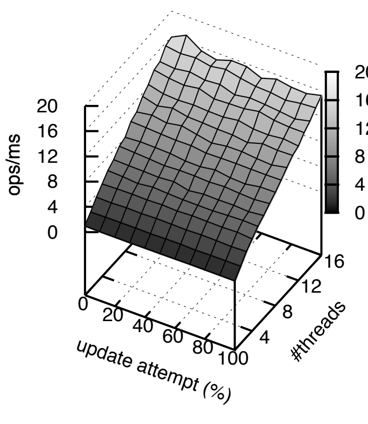


(c) Hash table with 2^{12} elements

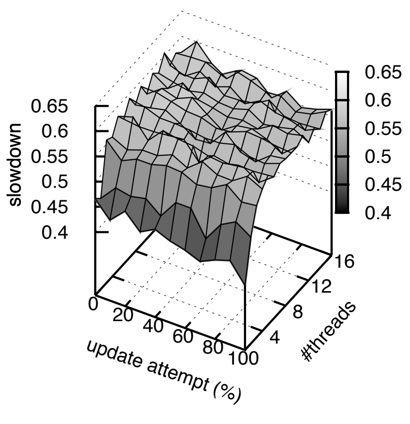


(d) Hash table with 2^{14} elements

Figure 9: Extended $\mathcal{E}$-STM hash table compared to the extended lock-based hash table.



(a) Harris-Michael lock-free linked list



(b) Slowdown of $\mathcal{E}$-STM linked list compared to Harris-Michael linked list

Figure 10: $\mathcal{E}$-STM linked list compared to CAS-based lock-free linked list (2^{16} elements).

list to force one of the two operations to stop contending and let the other terminate as suggested in [31].

$\mathcal{E}$-STM skip list did not suffer from the livelock issue. All writes are protected by a lock and if a transaction notices some changes upon writing, it restarts from the beginning. Additionally, $\mathcal{E}$-STM benefits from the contention management policies that arbitrate between conflicting transactions. There exists a series of related work that discusses the problem of ensuring progress by choosing the right policy. Although discussing these contention management policies [62] is out of the scope of this paper, any of these can be applied to the transactions of $\mathcal{E}$-STM.

7.5. Comparing against CAS-based Lock-free Techniques

Lock-free synchronizations do not expose locks to the programmer. As opposed to STMs, we consider here implementations that use universal primitives as their sole synchronization mean. The `compare-and-swap` (CAS) is such a hardware primitive that stores a value at a memory location depending on the value to be overwritten. It is supported by common architectures, including the one used for our experiments. This kind of implementation is very appealing as it is inherently *non-blocking*: no actions taken by one thread forces some other thread to wait. Although transactional memories do not expose locks, locks are often used internally to prevent one transaction from repeatedly aborting another.

For comparison purposes, we consider the lock-free Harris-Michael linked list algorithm as presented in [24]. This algorithm uses CAS low-level primitives for synchronization and avoids the use of locks. For the sake of compliance with our x86-64 architecture, we re-implemented the Fraser lock-free skip list that uses the low-order bit marking technique of Harris-Michael's linked list [17]. This algorithm has proved efficient and has been adopted for Doug Lea's ConcurrentSkipListMap implementation of the java.util.concurrent package in JDK 7. Our CAS-based lock-free hash table is a bucket hash table as described in [44]. More precisely, the hash table maps a given key to a value whose key indicates a bucket in which this value should be stored. Each bucket is implemented as a sorted linked list that reuses the lock-free Harris-Michael linked list [24] mentioned above.

7.5.1. Advantages of CAS-based implementations

Figure 10 presents the slowdown of $\mathcal{E}$-STM relatively to the CAS-based linked list we have implemented: the Harris-Michael version. The slowdown stems from the costly metadata management common to STMs synchronization that is avoided by hardware CAS.

We observe on Figures 11 and 12 that the lock-free performance is higher than $\mathcal{E}$-STM performance on almost all data structures. This is due to the same reason given above: $\mathcal{E}$-STM accesses metadata each time it accesses an element of the data structure, which incurs a significant overhead.

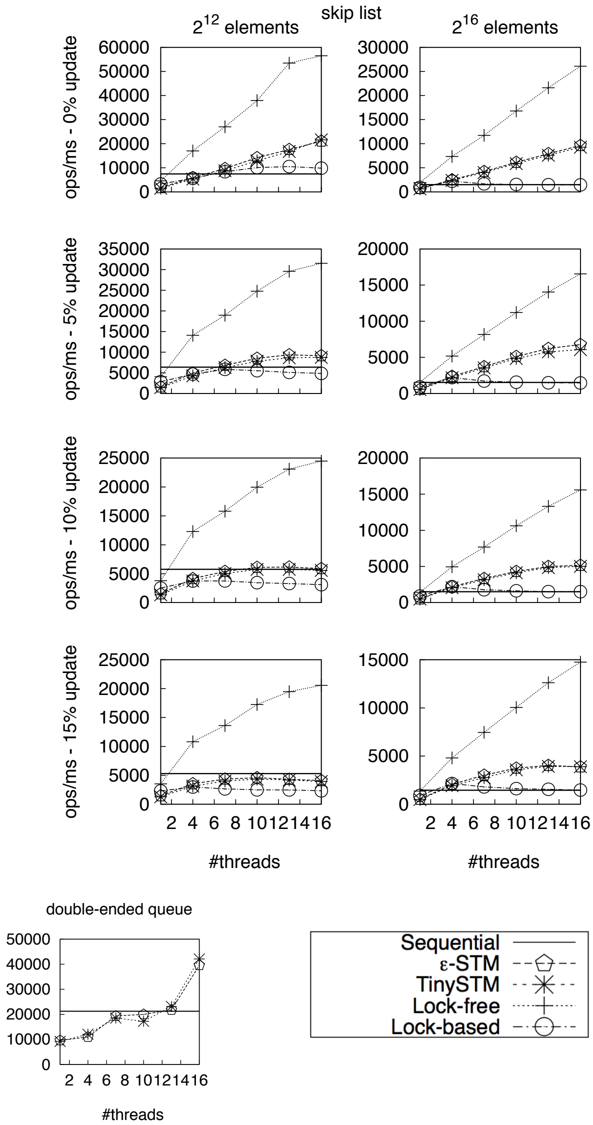


Figure 12: Skip list, and double-ended queue throughput.

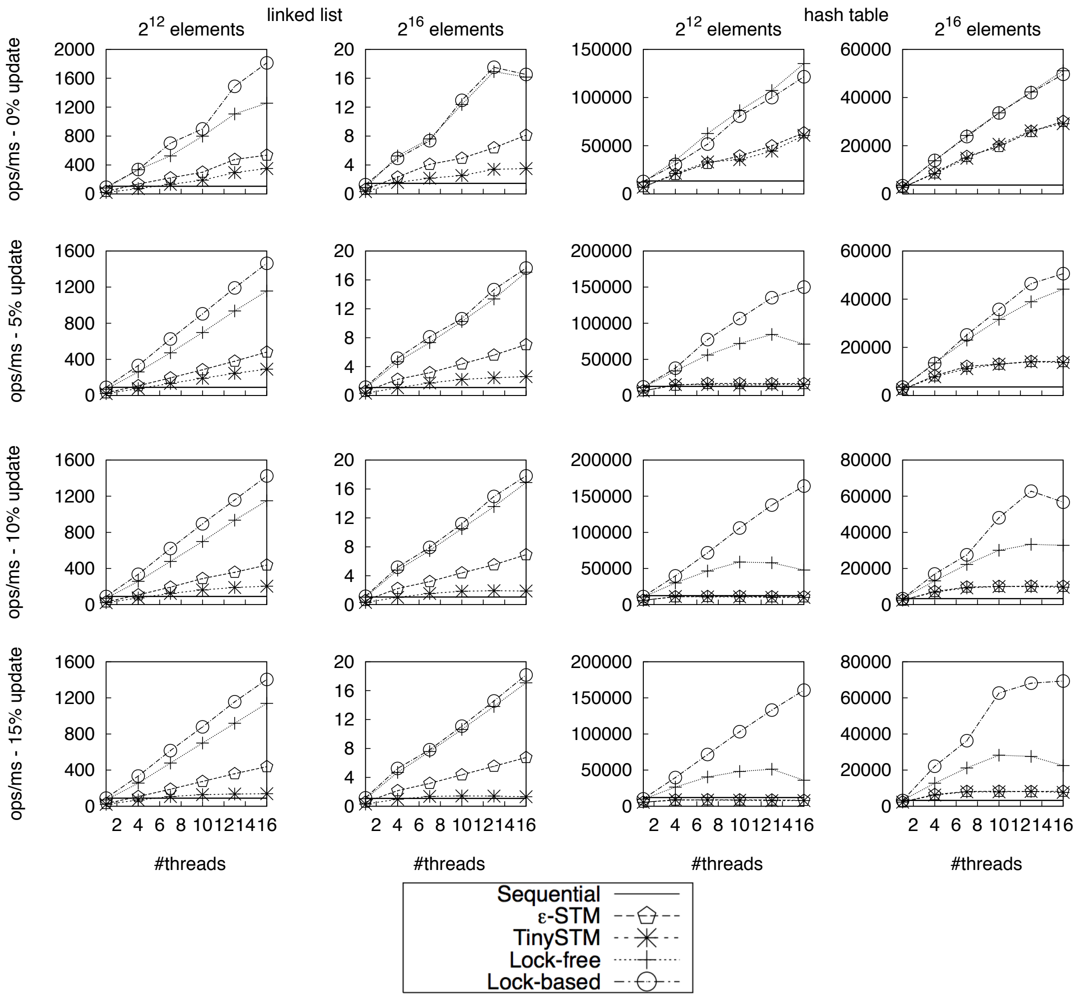


Figure 11: Linked list and hash table throughputs.

7.5.2. Advantages of $\mathcal{E}$-STM

We now present the advantages of using $\mathcal{E}$-STM when compared to using CAS-based lock-free alternatives. These advantages lie essentially in the extensibility of the concurrent code and memory re-allocation.

Extensibility. The code extensibility of $\mathcal{E}$-STM enables straightforward implementations of more complex operations like sum and move operations that are not supported in lock-free alternatives. One can think, for instance, about moving an element from one bucket to another in the same hash table data structure. This hash table move could help rebalance the load, i.e., the ratio of the size over the number of buckets, so that the access time of each operation remains constant. This could also be useful to decrease the load of the hash table by adding new buckets and scattering the stored elements among the new buckets.

The move operation requires the programmer to first remove an element from the hash table and then to insert another element like for modifying the key associated to a value. A sequential move results from the composition of a sequential remove followed by a sequential insert. As $\mathcal{E}$-STM provides code extensibility, it is almost as simple as composing sequential code: a transactional move results simply from the composition of the transactional remove and the transactional insert; one has simply to encapsulate these two transactions inside an enclosing one to delimit the transaction as illustrated in Algorithm 7, p.17.

By contrast, CAS-based lock-free algorithms cannot extend code efficiently. During the execution of two CAS-based remove and insert there might be a concurrent sum operation that fails to find any of these targeted elements—one because it has not been inserted yet, the other because it has already been removed. The same troublesome scenario has been reported by Doug Lea for the lock-free java.util.concurrent.ConcurrentSkipListMap whose size() is "not very useful in concurrent applications" [42] because it does not implement an atomic snapshot. A first solution to ensure atomicity of lock-free move that could remedy the sum problem, is to copy the whole data structure, apply the move to the copy before switching a pointer that indicates the current copy from one copy to another only if no modification occurred in the meantime. An alternative would be to use a multi-word compare-and-swap instruction to insert and remove the elements in a single atomic step. The former solution requires all modifications to copy the data structure as well, thus preventing two updates, even when trying to modify disjoint parts of the data structure, from succeeding when executing concurrently. The latter solution requires a rarely supported instruction that is also considered inefficient.

As an illustration of the difficulty of designing such lock-free composite operations, a hash table had to be structurally redesigned into a split-ordered linked list to support a CAS-based lock-free resize [58]. Although the resize operation allows buckets to move among consecutive linked list nodes, it does not allow nodes to move among the buckets. Implementing an efficient CAS-based lock-free move remains an open question.

Performance. The performance of the lock-free skip list is substantially faster than any other implementation. In particular, on average the lock-free skip list is 3.6× faster than the $\mathcal{E}$-STM one whereas the $\mathcal{E}$-STM skip list is only 10% faster than TinySTM one. Although it is well-known that the Fraser algorithm was appropriately optimised for x86 architectures, its performance gain may look surprising. Lock-free data structures often necessitate a customised memory management mechanism. The memory reclamation used in Fraser's algorithm is epoch-based and its allocator is optimized for this particular skip list algorithm: it allocates contiguous locations in memory in batches to minimize the number of allocations, which tends to reduce significantly the overhead. By contrast, our STM implementations hide a generic memory reclamation mechanism that is common to any data structures we have presented. Although choosing a more efficient library for allocating memory in a concurrent environment would certainly boosts the performance of our algorithm, we leave the performance comparison of memory allocators to future work. Below we discuss the way memory is allocated and reclaimed in our implementations.

Memory re-allocation. Memory management and more precisely memory re-allocation is a tricky problem in lock-free algorithms [28]. If a thread gets pre-empted while about to access a memory location, then no other thread can re-allocate this memory location. Techniques that only rely on the use of CAS or `load-linked/store-conditional` do not use metadata indicating whether a memory location is being accessed by some (possibly pre-empted) node. Unlike CAS-based algorithms, an STM can provide a quite straightforward way to re-allocate the memory. As an STM wraps all shared memory accesses, it can identify the scope in which the value of each shared memory location is used—this is clearly indicated by the transaction delimiters begin and commit.

While our new $\mathcal{E}$-STM integrates a garbage collector, we did not use any garbage collector in the lock-free linked list. we have re-implemented for two reasons because various techniques [32] have been discussed in the literature and some of them require the original algorithm to be adapted [24, 44].

Typically, the reference counting strategy [66, 9] was used in [24] to complement our lock-free linked list, whose haris-ll-find pseudocode is depicted in Algorithm 4. This memory management technique consists of adding a reference counter to each node of the data structure indicating the maximum number of references and thread-local variables operating on the node. A node can be de-allocated, for being re-used afterwards, only after its reference counter reaches zero, ensuring that the de-allocation cannot make a concurrent thread observe an inconsistent state of the node. This technique is however unaffordable as it slows down the algorithm by a factor of 10 to 15 times [24, 44]. Such limitations motivated further research [44] on alternative lock-free algorithms that comply with more efficient memory management algorithms like IBM freelist [36] and safe memory reclamation method [45].

In $\mathcal{E}$-STM, memory is managed with the reference counting strategy. More technically, it wraps `free` calls to simply postpone the appropriate de-allocation to a safe point in time. This point is defined at the moment where all transactions that accessed the memory location have either committed or aborted. When the `free` wrapper is called, the counter of active transactions that access the corresponding locations must reach 0 before de-allocating the appropriate location. Note that the same reference counting strategy applied to the lock-free linked list could make the lock-free solution less efficient than $\mathcal{E}$-STM linked list as $\mathcal{E}$-STM is never 10 times slower than its lock-free counterpart (see Figures 11 and 12 at the end of this section). However, evaluation of the impact of memory management strategies is out of the scope of this paper.

7.6. Using another Programming Language and a Different Architecture

We report here on the experimentation of a variant of $\mathcal{E}$-STM written in a different programming language (Java) and evaluated on an architecture (SPARC) and an instruction set (RISC) different from what we have considered so far (x86-64 for CISC). This shows that the concurrency gains obtained by the elastic transaction model has almost the same positive impact for both programming languages and architectures.

Upon porting our STM in Java, we had to choose the granularity of conflict detection between object-based and field-based. In our implementation we have chosen the finer granularity of the two. More precisely, our implementation detects conflicts between concurrent transactions accessing common objects at the granularity of object fields, hence two transactions accessing distinct fields of the same object do not conflict. This choice was made to obtain results more similar to our word-based implementation in C.

As all experiments showed comparable results, we only report (Figure 13) the results obtained with the variant of $\mathcal{E}$-STM written in Java and run on an UltraSPARC II (Niagara 2) machine including 8 cores, running up to 8 hardware threads each (64 hardware threads in total). We have compared using the Deuce bytecode instrumentation framework [37], the performance of our Java version of $\mathcal{E}$-STM against the performance obtained with a Java field-based version of TinySTM [14] on a Java variant of the aforementioned linked list benchmark. The throughput of Java $\mathcal{E}$-STM and TinySTM obtained for 1 to 64 threads are depicted in Figure 13. Independently from the update ratios or the size of the data structure, $\mathcal{E}$-STM outperforms TinySTM. Typically, $\mathcal{E}$-STM speeds up TinySTM by 3.2$\times$ on 64 threads and by 2.1$\times$ on average. Interestingly, as opposed to previous experiments, these curves indicate that the throughput of TinySTM gets significantly impacted by the contention at high number of threads whereas $\mathcal{E}$-STM scales well up to 64 threads, i.e., the maximum hardware threads we had at our disposal. The difficulty for TinySTM to scale above 32 threads is due to the higher probability for regular transactions to contend than for elastic transactions and the growing number of potentially concurrent transactions at high level of parallelism. This interpretation is confirmed by the fact that TinySTM performs as well as $\mathcal{E}$-STM with 0% updates, i.e., without contention, but performs as bad as sequential on 15% update, i.e., under high contention.

8. Summary

We present a new transactional model for concurrent search structures. The core idea relies on the combination of traditional transactions with a new type of transactions that are elastic in the sense that their size evolves dynamically depending on conflict detection. As opposed to other relaxed transaction models, the elastic transaction model detects at runtime whether two operations do actually commute in the current interleaving.

We implemented our new model in an STM, called $\mathcal{E}$-STM, that only needs to differentiate elastic from traditional transactions, making it simple to program with as it preserves sequential code. Comparisons on different architectures, with different programming languages and on four popular data structures have confirmed that on the one hand, elastic transactions perform significantly better than traditional transactions and in several circumstances better than fine-grained locking alternatives. Additionally, it is much simpler to program with than existing lock-free solutions and the resulting programs can be extended.

The performance benefit of elastic transactions over regular transactions is proportional to the data structure access time complexity. This gain in performance is negligible for constant access time search structures like non-loaded hash tables whose transactions are already short, but it increases on logarithmic access time data structures and is significant on linear access time data structures like linked lists.

We illustrated the elastic transaction model using a single implementation, $\mathcal{E}$-STM (implementing 1-elastic-opacity), suitable for various data structures, however, one could derive k-elastic-opaque implementations for specific needs, with $k > 1$. Interesting questions that were not addressed here include whether concurrency or safety should be promoted when combining such specific transactions: should a parent k-elastic-opaque transaction enforce its nested transactions to be ℓ-elastic-opaque, where $k \leq \ell$?

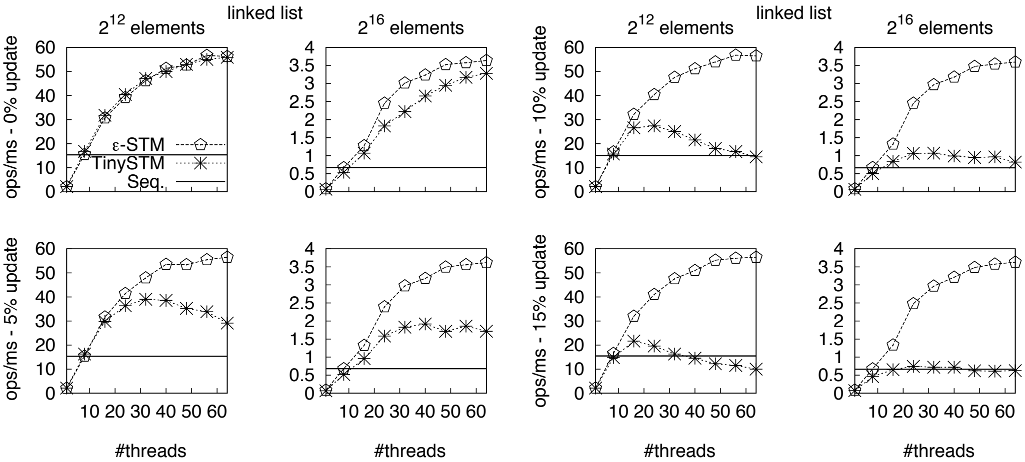


Figure 13: Linked list throughput of the Java version of $\mathcal{E}$-STM and TinySTM running on SPARC Niagara 2 (8 cores, 64 hardware threads).

Acknowledgments

We are grateful to Hagit Attiya for careful reading of earlier versions of this paper, to Stephan Diestelhorst for fruitful discussions on clarifying the proofs, and to Marc Shapiro for pointing out results on the topic of relative atomicity. This research was supported under Australian Research Council's Discovery Projects funding scheme (project number 160104801) entitled "Data Structures for Multi-Core". Vincent Gramoli is the recipient of the Australian Research Council Discovery International Award.

[1] Abadi, M., Birrell, A., Harris, T., Isard, M., January 2011. Semantics of transactional memory and automatic mutual exclusion. ACM Trans. Program. Lang. Syst. 33, 2:1–2:50.

[2] Adl-Tabatabai, A.-R., Lewis, B. T., Menon, V., Murphy, B. R., Saha, B., Shpeisman, T., 2006. Compiler and runtime support for efficient software transactional memory. In: PLDI. ACM, New York, NY, USA.

[3] Afek, Y., Morrison, A., Tzafrir, M., 2010. Brief announcement: view transactions: transactional model with relaxed consistency checks. In: PODC. ACM, New York, NY, USA, pp. 65–66.

[4] Bayer, R., Schkolnick, M., 1988. Concurrency of operations on B-trees. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 129–139.

[5] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P., 1995. A critique of ANSI SQL isolation levels. In: SIGMOD. ACM, New York, NY, USA, pp. 1–10.

[6] Cole, C., Herlihy, M., December 2005. Snapshots and software transactional memory. Sci. Comput. Program. 58, 310–324.

[7] Dalessandro, L., Spear, M. F., Scott, M. L., 2010. Norec: Streamlining stm by abolishing ownership records. In: PPoPP. ACM, pp. 67–78.
URL `http://doi.acm.org/10.1145/1693453.1693464`

[8] Date, C. J., 1983. An introduction to database systems: vol. 2 (2nd ed.). Addison-Wesley.

[9] Detlefs, D. L., Martin, P. A., Moir, M., Steele, Jr., G. L., 2001. Lock-free reference counting. In: PODC. ACM, New York, NY, USA, pp. 190–199.

[10] Dice, D., Shalev, O., Shavit, N., September 2006. Transactional locking II. In: DISC. Vol. 4167. Springer-Verlag, London, UK.

[11] Dragojević, A., Felber, P., Gramoli, V., Guerraoui, R., Apr 2011. Why STM can be more than a research toy. Commun. ACM 54 (4), 70–77.

[12] Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., Shasha, D., June 2005. Making snapshot isolation serializable. ACM Trans. Database Syst. 30 (2), 492–528.

[13] Felber, P., Fetzer, C., Mueller, U., Riegel, T., Suesskraut, M., Sturzrehm, H., 2007. Transactifying applications using an open compiler framework. In: Transact.

[14] Felber, P., Fetzer, C., Riegel, T., 2008. Dynamic performance tuning of word-based software transactional memory. In: PPoPP. ACM, New York, NY, USA.

[15] Felber, P., Gramoli, V., Guerraoui, R., Sep 2009. Elastic transactions. In: DISC. Vol. 5805 of LNCS. Springer-Verlag, London, UK, pp. 93–107.

[16] Fomitchev, M., Ruppert, E., 2004. Lock-free linked lists and skip lists. In: PODC. ACM, New York, NY, USA, pp. 50–59.

[17] Fraser, K., September 2003. Practical lock freedom. Ph.D. thesis, University of Cambridge.

[18] Gramoli, V., Feb 2015. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In: PPoPP. ACM, pp. 1–10.

[19] Gramoli, V., Guerraoui, R., Jan 2014. Democratizing transactional programming. Commun. ACM 57 (1), 86–93.

[20] Gramoli, V., Guerraoui, R., Jul 2014. Reusable concurrent data types. In: ECOOP. Vol. 8586 of LNCS. Springer, pp. 182–206.

[21] Greenwald, M., 2002. Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. In: PODC. ACM, New York, NY, USA, pp. 260–269.
URL `http://doi.acm.org/10.1145/571825.571874`

[22] Guerraoui, R., Kapałka, M., February 2008. On the correctness of transactional memory. In: PPoPP. ACM, New York, NY, USA, pp. 175–184.

[23] Harmanci, D., Gramoli, V., Felber, P., Fetzer, C., 2010. Extensible transactional memory testbed. J. Parallel Distrib. Comput. - Special Issue on Transactional Memory 70 (10), 1053–1067.

[24] Harris, T., 2001. A pragmatic implementation of non-blocking linked-lists. In: DISC. Vol. 2180. Springer-Verlag, London, UK, pp. 300–314.

[25] Harris, T., Stipić, S., 2007. Abstract nested transactions. In: The 2nd SIGPLAN Workshop on Transactional Computing.

[26] Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer III, W. N., Shavit, N., December 2005. A lazy concurrent list-based set algorithm. In: OPODIS. Vol. 3974 of LNCS. Springer-Verlag, London, UK, pp. 3–16.

[27] Herlihy, M., 1993. A methodology for implementing highy concurrent data objects. ACM Trans. Program. Lang. Syst. 15 (5), 745–770.

[28] Herlihy, M., 2006. The art of multiprocessor programming. In: PODC. ACM, New York, NY, USA, pp. 1–2.
[29] Herlihy, M., Koskinen, E., 2008. Transactional boosting: A methodology for highly-concurrent transactional objects. In: PPoPP. ACM, New York, NY, USA.
[30] Herlihy, M., Lev, Y., Luchangco, V., Shavit, N., 2006. A provably correct scalable concurrent skip list. In: OPODIS. Vol. 4305 of LNCS. Springer-Verlag, London, UK.
[31] Herlihy, M., Lev, Y., Luchangco, V., Shavit, N., 2007. A simple optimistic skiplist algorithm. In: SIROCCO. Vol. 4474 of LNCS. Springer-Verlag, London, UK, pp. 124–138.
[32] Herlihy, M., Luchangco, V., Martin, P., Moir, M., 2005. Nonblocking memory management support for dynamic-sized data structures. ACM Trans. Comput. Syst. 23 (2), 146–196.
[33] Herlihy, M., Luchangco, V., Moir, M., 2003. Obstruction-free synchronization: Double-ended queues as an example. In: ICDCS. IEEE Computer Society, Los Alamitos, CA, USA, p. 522.
[34] Herlihy, M., Luchangco, V., Moir, M., Scherer III, W. N., 2003. Software transactional memory for dynamic-sized data structures. In: PODC. ACM, New York, NY, USA, pp. 92–101.
[35] Herlihy, M., Shavit, N., February 2008. The Art of Multiprocessor Programming. Morgan Kauffman.
[36] IBM, 1983. IBM system/370 extended architecture, principles of operation. Tech. Rep. SA22-7085.
[37] Korland, G., Shavit, N., Felber, P., 2010. Deuce: Noninvasive software transactional memory. Transactions on HiPEAC 5 (2), 21.
[38] Korth, H. F., 1983. Locking primitives in a database system. J. ACM 30 (1), 55–79.
[39] Krishnaswamy, V., Agrawal, D., Bruno, J. L., Abbadi, A. E., 1997. Relative serializability: An approach for relaxing the atomicity of transactions,. Journal of Computer and System Sciences 55 (2), 344 – 354.
URL `http://www.sciencedirect.com/science/article/B6WJ0-45KV09M-8/2/c581930999d8b76a459be3ca0c52cb97`
[40] Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L. P., 2007. Optimistic parallelism requires abstractions. In: PLDI. ACM, pp. 211–222.
URL `http://doi.acm.org/10.1145/1250734.1250759`
[41] Lamport, L., 1978. Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21 (7), 558–565.
[42] Lea, D., 2009. Java platform, standard edition 6, API specification. `http://java.sun.com/javase/6/docs/api`.
[43] Lynch, N. A., December 1983. Multilevel atomicity a new correctness criterion for database concurrency control. ACM Trans. Database Syst. 8, 484–502.
[44] Michael, M. M., 2002. High performance dynamic lock-free hash tables and list-based sets. In: SPAA. ACM, New York, NY, USA, pp. 73–82.
[45] Michael, M. M., 2002. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In: PODC. ACM, New York, NY, USA, pp. 21–30.
[46] Moss, J. E. B., February 2006. Open nested transactions: Semantics and support. In: WMPI.
[47] Ni, Y., Menon, V., Abd-Tabatabai, A.-R., Hosking, A. L., Hudson, R. L., Moss, J. E. B., Saha, B., Shpeisman, T., 2007. Open nesting in software transactional memory. In: PPoPP. ACM, New York, NY, USA.
[48] Ni, Y., Welc, A., Adl-Tabatabai, A.-R., Bach, M., Berkowits, S., Cownie, J., Geva, R., Kozhukow, S., Narayanaswamy, R., Olivier, J., Preis, S., Saha, B., Tal, A., Tian, X., 2008. Design and implementation of transactional constructs for C/C++. In: OOPSLA. ACM, pp. 195–212.
URL `http://doi.acm.org/10.1145/1449764.1449780`
[49] O'Neil, P. E., December 1986. The escrow transactional method. ACM Trans. Database Syst. 11, 405–430.
URL `http://doi.acm.org/10.1145/7239.7265`
[50] Pankratius, V., Adl-Tabatabai, A.-R., Otto, F., September 2009. Does transactional memory keep its promises? results from an empirical study. Tech. Rep. 2009-12, University of Karlsruhe, Germany.
[51] Papadimitriou, C. H., October 1979. The serializability of concurrent database updates. J. ACM 26, 631–653.
URL `http://doi.acm.org/10.1145/322154.322158`
[52] Pugh, W., 1990. Skip lists: a probabilistic alternative to balanced trees. Commun. ACM 33 (6), 668–676.
[53] Reuter, A., 1982. Concurrency on high-traffic data elements. In: PODS. ACM, New York, NY, USA, pp. 83–92.
URL `http://doi.acm.org/10.1145/588111.588126`
[54] Riegel, T., Felber, P., Fetzer, C., September 2006. A Lazy Snapshot Algorithm with Eager Validation. In: DISC. Vol. 4167. pp. 284–298.
[55] Rossbach, C. J., Hofmann, O. S., Witchel, E., 2010. Is transactional programming actually easier? In: PPoPP. ACM, New York, NY, USA, pp. 47–56.
URL `http://doi.acm.org/10.1145/1693453.1693462`
[56] Saha, B., Adl-Tabatabai, A.-R., Hudson, R. L., Minh, C. C., Hertzberg, B., 2006. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: PPoPP. ACM, pp. 187–197.
URL `http://doi.acm.org/10.1145/1122971.1123001`
[57] Schwarz, P. M., Spector, A. Z., 1984. Synchronizing shared abstract types. ACM Trans. Comput. Syst. 2 (3), 223–250.
[58] Shalev, O., Shavit, N., 2006. Split-ordered lists: Lock-free extensible hash tables. J. ACM 53 (3), 379–405.
[59] Shavit, N., 2011. Data structures in the multicore age. Commun. ACM 54 (3), 7684.
[60] Shavit, N., Touitou, D., 1995. Software transactional memory. In: PODC. ACM, New York, NY, USA, pp. 204–213.
[61] Skare, T., Kozyrakis, C., June 2006. Early release: Friend or foe? In: Workshop on Transactional Memory Workloads.
[62] Spear, M. F., Dalessandro, L., Marathe, V. J., Scott, M. L., 2009. A comprehensive strategy for contention management in software transactional memory. SIGPLAN Not. 44 (4), 141–150.
[63] Spear, M. F., Marathe, V. J., Dalessandro, L., Scott, M. L., 2007. Privatization techniques for software transactional memory. In: PODC '07. pp. 338–339.
[64] Sutter, H., June 2008. Dr. Dobb's Journal.
[65] Traiger, I. L., 1983. Trends in system aspects of database management. In: ICOD. pp. 1–21.
[66] Valois, J. D., 1995. Lock-free linked lists using compare-and-swap. In: PODC. ACM, New York, NY, USA, pp. 214–222.
[67] Weihl, W. E., 1988. Commutativity-based concurrency control for abstract data types. IEEE Trans. Comput. 37 (12), 1488–1505.
[68] Weihl, W. E., 1989. Local atomicity properties: Modular concurrency control for abstract data types. ACM Trans. Program. Lang. Syst. 11 (2), 249–283.
[69] Weikum, G., 1986. A theoretical foundation of multi-level concurrency control. In: PODS. ACM, New York, NY, USA, pp. 31–43.

Appendix A. Correctness of $\mathcal{E}$-STM

Here, we prove that $\mathcal{E}$-STM is (1-)elastic-opaque. We do so in three steps. First, we give a few preliminary definitions. Second, we show that for each committed elastic transaction of $\mathcal{E}$-STM, there exists a consistent cut C so that the cutting function f_C is well-defined. Third, we show that $\mathcal{E}$-STM is elastic-opaque by differentiating histories restricted to aborting transactions from histories restricted to committed transactions. To avoid the ABA problem we assume that the timed-based lock *tlk* embeds a version counter with enough bits to make full wraparound between the two version loads of ver-val-ver impossible. As already mentioned in Subsection 5.1.4, this ABA problem is very rare in practice as it occurs only if ver-val-ver executes so slowly that as many updates as the counter capacity occurs in the meantime.

Theorem 1. *$\mathcal{E}$-STM is elastic-opaque.*

We define *sub-complete* as a mapping between history $\mathcal{H}$ and a history $\mathcal{H}' = \textit{sub-complete}(\mathcal{H})$ in which (i) all non-completed transactions of $\mathcal{H}$ that have a commit invocation and that do not write any object value into the memory are aborted in $\mathcal{H}'$, (ii) all the transactions of $\mathcal{H}$ that have written an object value into the memory are committed in $\mathcal{H}'$, and (iii) all non-completed transactions of $\mathcal{H}$ with no commit-request are aborted in $\mathcal{H}'$. Observe that for any $\mathcal{H}$, $\textit{sub-complete}(\mathcal{H}) \in \textit{complete}(\mathcal{H})$.

First-of-all, we show that there exists a consistent cut C_t for every elastic transaction t in $\mathcal{H}|\mathcal{E}$. This ensures that the $f_{\mathcal{C}_t}(\mathcal{H}|t)$ is well defined and more generally, that $f_{\mathcal{C}}(\mathcal{H})$ is well-defined.

Lemma 2. *Let $\mathcal{H}$ be any history of $\mathcal{E}$-STM and $\mathcal{H}' = \textit{permanent}(\mathcal{H})$. There exists a consistent cut $\mathcal{C}_t$ with respect to $\mathcal{H}'$ for every elastic transaction t of $\mathcal{H}'|t$.*

Proof. The proof relies essentially on the definition of a consistent cut (Definition 3). First, if t contains a single operation, then $\mathcal{C}_t$ contains only t and the definition is straightforwardly satisfied.

Now consider the case where t has more than one operation. We show that there can neither be a write operation $\pi(x)^{t'}$ from transaction $t' \neq t$ such that $\pi(x)^t \to_{\mathcal{H}'} \pi(x)^{t'} \to_{\mathcal{H}'} \pi(x)^t$ nor be two writes operations $\pi(x)^{t'}$ $\pi(y)^{t''}$ from transaction $t' \neq t$ and $t'' \neq t$ such that $\pi(y)^t \to_{\mathcal{H}'} \pi(x)^{t'} \to_{\mathcal{H}'} \pi(x)^t$ and $\pi(y)^t \to_{\mathcal{H}'} \pi(y)^{t''} \to_{\mathcal{H}'} \pi(x)^t$ provided that $dist_{\mathcal{H}'|t}(\pi(y)^t, \pi(x)^t) \leq 1$. First, we start by showing that the latter case cannot happen, the impossibility of the former case will follow.

Assume by contradiction that such writes $\pi(x)^{t'}$ and $\pi(y)^{t''}$ exist, we show that t would abort. When $\pi(x)^{t'}$ executes, it locks x by setting $x.tlk.owner$ to t' until it commits and sets the associated timestamp $x.tlk.time$ to a new strictly higher clock value than $t.ub$. For the same reason, $y.tlk.time > t.ub$ just after the execution of $\pi(y)^{t''}$. Hence, t reads x after t' and t'' have committed, and t observes that $x.tlk.time$ is larger than its $t.ub$. As t is elastic, this observation leads it to verify that the version of y has not changed (Line 50). Since we have $y.tlk.time > t.ub$, the transaction aborts as indicated Line 54. The same proof holds also for the case where $x = y$.

As a result, there always exists a consistent cut $\mathcal{C}_t$ of an elastic transaction t of $\mathcal{H}'$.

In the remainder of the proof, we refer to the cut history $f_{\mathcal{C}}(\mathcal{H})$ as the history $\mathcal{H}$ where each committed elastic transaction has been replaced by its sub-transactions in one of its consistent cut. It remains to show that $f_{\mathcal{C}}(\mathcal{H})$ is opaque. Property (1) of Definition 4 comes from the fact that no value is written in memory unless the transaction is ensured to commit. Hence, the first lemma shows that an aborting transaction is invisible to other transactions.

Lemma 3. *Let $\mathcal{H}$ be any history of $\mathcal{E}$-STM and let $\mathcal{H}' = \textit{sub-complete}(f_{\mathcal{C}}(\mathcal{H}))$. If $t' \in \textit{aborted}(\mathcal{H}')$ then t' is an invisible transaction.*

Proof. The proof is divided into two parts whether we consider the completed transactions of $\mathcal{H}$ or the transactions that were not completed in $\mathcal{H}$ but that are completed in $\mathcal{H}'$.

1. First, we show that if $t \in \textit{aborted}(\mathcal{H})$, then t is invisible. By transaction well-formedness, no $\mathsf{abort}()_t$ can occur after a $\mathsf{commit}()_t$ completes. By examination of the code, we know that the memory can only be updated during the for-loop of the $\mathsf{commit}()_t$ function (Line 89). With no loss of generality let τ_1 be the starting time of the for-loop. A transaction that issued a commit invocation can only abort at time τ_2, before the try-extend() call returns at Line 88. Since Line 88 is before the beginning of the for-loop, $\tau_2 < \tau_1$ and the result follows.
2. Second, we show that if $t \in \textit{aborted}(\mathcal{H}') \setminus \textit{aborted}(\mathcal{H})$ then t is invisible. Since a transaction can only write after a commit invocation, all abort events that are not in $\mathcal{H}$ but that are in $\mathcal{H}'$ are appended only to invisible transactions.

The conjunction of the two parts of the proof states that there exists $\mathcal{H}' \in \textit{complete}(\mathcal{H})$ such that if $t \in \textit{aborted}(\mathcal{H}')$ then t is invisible.

To show Property (2) of Definition 4, we first determine a serialization point for each read/write operation and show that each transaction appears "as if" it was executed atomically at this point in time.

1. read operation π: its serialization point $ser(\pi)$ is the point in the execution where the last $\ell_1 \leftarrow x.tlk$ of the loop occurs (Line 96).
2. write operation π: its serialization point $ser(\pi)$ is the point in the execution where the last CAS of the loop occurs (Line 69).

Observe that serialization points are defined at the time an atomic operation occurs. Hence, two distinct operations on the same object cannot have the same serialization point.

Lemma 4. *Let $\mathcal{H}$ be any history of $\mathcal{E}$-STM and let $\mathcal{H}' = \textit{sub-complete}(f_{\mathcal{C}}(\mathcal{H}))$. Let t_1 and t_2 be two transactions in $\mathcal{H}'$. For any two distinct operations π_1 and π_2 executed respectively in t_1 and t_2 and accessing location x: if $ser(\pi_1) < ser(\pi_2)$, then $\pi_2 \not\prec \pi_1$.*

Proof. We show by contradiction that we cannot have $ser(\pi_1) < ser(\pi_2)$ and $\pi_2 \prec \pi_1$. Assume by absurd that $\pi_2 \prec \pi_1$, there are three cases to consider.

- If π_1 and π_2 are executed in order by the same transaction, then the result follows directly by the well-formedness assumption.

– If π_1 reads or overwrites the value written by π_2, then $\pi_2 \prec \pi_1$ implies that $ser(t_1)$ is after t_2 releases its lock on x (Line 70 or 100) otherwise t_1 would have aborted (Line 40 if π_1 is a read or Line 63 if π_1 is a write) prior to completing π_1.

– If π_1 overwrites the value read by π_2, then either π_2 would detect that its transaction owns the lock and so it would return the value written by π_1 contradicting that $\pi_1 \prec \pi_2$, or π_2 would detect that another transaction owns the lock, so t_2 would abort (Line 40) prior to completing π_2.

Hence, all cases assuming that $\pi_2 \prec \pi_1$ lead to a contradiction, implying that $ser(\pi_2) \leq ser(\pi_1)$. Hence, the equivalent contrapositive $ser(\pi_1) < ser(\pi_2) \Rightarrow \pi_2 \not\prec \pi_1$ gives the result.

Invariant 5. *$clock \geq lb$.*

PROOF. Initially, $clock = x.tlk.time = 0$, for every variable x. Since $clock$ is monotonically increasing, $x.tlk.time$ can only be set to $clock$, and lb can only be set to $clock$ or $x.tlk.time$, the result follows.

Next, we generalize the previous lemma to $\prec^*$, the transitive closure of $\prec$.

Corollary 6. *Let $\mathcal{H}$ be any history of $\mathcal{E}$-STM and let $\mathcal{H}' = sub\text{-}complete(f_{\mathcal{C}}(\mathcal{H}))$. Let t_1 and t_2 be two transactions in $\mathcal{H}'$. For any two operations π_1 and π_2 executed respectively in t_1 and t_2: if $ser(\pi_1) < ser(\pi_2)$, then $\pi_2 \not\prec^* \pi_1$.*

Next lemma indicates that any history is equivalent to some sequential history. More precisely, it shows that all operations of a single transaction are ordered in the same manner with respect to other transactional operations. For the proof, let a_π denote the state of a field a when it is set in operation π for the first time, or when π starts (if it is never set by π).

Lemma 7. *Let $\mathcal{H}$ be any history of $\mathcal{E}$-STM and let $\mathcal{H}' = sub\text{-}complete(f_{\mathcal{C}}(\mathcal{H}))$. Let t_1 and t_2 be two transactions in $\mathcal{H}'$. Let π_1 and π_2 be some operation of transaction t_1 and t_2, respectively. If $\pi_1 \prec^* \pi_2$, then for any $\pi_1' \in t_1 : \pi_2 \not\prec^* \pi_1'$.*

PROOF. By contradition we assume that $\pi_1 \prec^* \pi_2 \prec^* \pi_1'$ and we show that t_1' aborts prior to completing π_1'. With no loss of generality, let π_1 and π_1' access a and a' respectively, we first show that π_1 can only be a read and then consider the two cases whether π_1' is a write or a read.

π_1 cannot be a write operation, otherwise there should be a read operation $r(a)$ such that $\pi_1(a) \prec r(a) \prec^* \pi_1'$, but $r(a)$ as a read would have aborted (because $x.tlk.owner_r = t_1$, Line 40), or would loop while $x.tlk.owner_r = t_1$ (Line 100), leading in both cases to the contradiction $r(a) \not\prec^* \pi_1'$. Since π_1 is a read $\pi_1 \prec^* \pi_2$ implies that there is a write operation w such that $\pi_1(a) \prec w(a) \prec^* \pi_1'$.

There are two cases to consider, whether π_1' is a write. First, assume that π_1' is a write. By Invariant 5 we know that $lb_{\pi_1} \leq clock_{\pi_1}$ and by the write w: $clock_{\pi_1} < clock_{\pi_1'}$ so that $lb_{\pi_1} < clock_{\pi_1'}$ and try-extend occurs in t_1 or t_1 would have aborted during π_1 (contradicting the assumption). Second, if π_1' is a read, there is a write $w'(a')$ such that $\pi_2 \prec^* w'(a') \prec \pi_1'(a')$. Hence $x.tlk.time_{\pi_1'} > clock_w \geq clock_{\pi_1}$, and by Invariant 5 we have $clock_{\pi_1} \geq ub_{\pi_1} \geq ub_{\pi_1'}$, whose conjunction leads to $x.tlk.time_{\pi_1'} > ub_{\pi_1'}$. Now observe that either $a'.tlk.owner \notin \{\perp, t_1\}$ and t_1 aborts, $a'.tlk.owner = t_1$, and w' would have aborted, or try-extend occurs in t_1. Hence, whether π_1' is a read or a write, try-extend occurs at t_1.

Finally, because π_1 is a read and w is a write that occurs between π_1 and π_1', t_1 aborts during the try-extend occurrence of π_1'. This contradicts the assumption that π_1 completes and the result follows.

By Lemma 7, we can generalize the definition of $\prec^*$ to transactions of $sub\text{-}complete(f_{\mathcal{C}}(\mathcal{H}))$ where $\mathcal{H}$ is a history of $\mathcal{E}$-STM, such that $t_1 \prec t_2$ if for two operations π_1 and π_2 of t_1 and t_2 respectively, $\pi_1 \prec \pi_2$.

Corollary 8. *Let $\mathcal{H}$ be any history of $\mathcal{E}$-STM and let $\mathcal{H}' = sub\text{-}complete(f_{\mathcal{C}}(\mathcal{H}))$. History $\mathcal{H}'$ is equivalent to a sequential history that is legal.*

PROOF. Let $\prec_t$ be an ordering on the set of transactions of $\mathcal{H}'$ such that $\forall t_1 \neq t_2$, $t_1 \prec_t t_2$ if there exist operations π_1 in t_1 and π_2 in t_2 such that $\pi_1 \prec^* \pi_2$. This ordering $\prec_t$ is an irreflexive partial order because (i) it is antisymmetric by Lemma 7, and (ii) it is irreflexive and transitive by definition of $\prec^*$. This ordering $\prec_t$ defines a set S of histories that are equivalent to $\mathcal{H}'$. This set is non-empty because $\prec_t$ is a partial order. It is easy to see that for any $s \in S$, s is sequential by the antisymmetry property of $\prec_t$. Finally and because $\prec_t \subseteq \prec^*$, s is legal as well.

Appendix B. Implementation Correctness: the Linked List Example

Here, we prove that our linked list presented in Algorithm 4 implements a linearizable integer set. The proof relies on the elastic opacity of $\mathcal{E}$-STM (cf. Appendix Appendix A). First-of-all, we recall the semantics of the integer set. Given a set S:

– search(i) operation returns true if the node i is present in S, false otherwise;

– insert(i) operation augments the set S with the node i if i is not in S, S is unchanged otherwise;

– remove(i) operation removes the node i from S if i is in S, S is unchanged otherwise.

In the following, we refer to such an integer set operation as Π to distinguish it from a read/write operation π. Next, we state preliminary definitions.

Definition 6. A node n is *reachable* if one of the two following properties holds:

– $set.head.next = n$ or

– there exists a reachable node m such that $m.next = n$.

Definition 7. Integer i is *in the set* if there is a reachable node n such that $n.key = i$.

The first lemma gives an important result for proving the correctness of operation search($*$) and relies on the definition of consistent cut (Definition 3).

Lemma 9. *Operation* find(i) *returns a pair of nodes* $\langle curr, next\rangle$ *such that*

1. $curr.key < i \leq next.key$ *and*
2. *curr and next are consecutive nodes of the list at some point of the corresponding execution of* find.

Proof. We show the two points separately.

1. First, we show that $curr.key < i \leq next.key$. At the beginning $curr$ is initialized at the head of the linked list, while $next$ is initialized to its successor in the linked list. As the loop iterates, the $curr$ and $next$ parse the linked list in ascending order, unless the transaction aborts. Observe that the function exits the main loop (and returns) if $next.key \geq i$ at Line 18. If so, $curr$ and $next$ are returned as is. By absurd, if $curr.key \geq i$ then the loop would have ended at least one iteration before, so $curr.key < i \leq next.key$. Finally, observe that $next \leftarrow$ read($curr.next$). Hence, during the last iteration of the main loop of read, $curr.next = next$.
2. Second, we show that $curr$ and $next$ are consecutive nodes of the list at some point of the corresponding execution of find. Observe that a find operation is always called as part of an elastic transaction. As a consequence of Definition 3, there are no two write operations on $curr$ and $next$ between read($curr$) and read($next$) executed by find. Hence, there are two cases to consider whether one of these two writes is missing. If there is no write on $curr$, then at the time the second read occurs on $next$, a read of $curr$ would return the same value as the one that has been returned before by read($curr$). If there is no write on $next$, then at the time the first read on $curr$ occurs, the second would have returned the same value as the one it will return with read($next$). Note that both cases are true for the case $next = curr$, because none of these writes can occur between the reads by Definition 3. It follows that each of the two nodes are consecutive at some point during the execution of this find.

The result follows.

We know by elastic-opacity (Definition 5) that no aborting transactions modify the state or observe an inconsistent state of the system; hence an aborting transaction does not violate safety.

Invariant 10. *For any node curr in the linked list, if* $curr.next = next$ *then* $curr.key < next.key$.

Proof. By Lemma 9, the $curr$ and $next$ returned by find(i) are such that $curr.key < i \leq next.key$. By examination of the code of insert(i) node i can only be inserted between a and b such that $a.key < i < b.key$. The result follows.

We focus on the search operation and on the end of operations insert and remove. The major difficulty stems from the concurrent updates problem [66] as read-only operations like search do not affect the structure, hence we consider the accesses executed by remove and insert after their search part is complete, and we denote them as suffix-remove and suffix-insert. A suffix-remove accesses two mutable shared data items, $curr.next$ and $curr.next.next$, whereas the suffix-insert access only one mutable shared data item, $curr.next$. With no loss of generality we fix the number of operations involved in a single conflict to two and we observe the situations in which these two operations can conflict. There are two ways two suffix-remove operations can conflict depicted in Figures 14(a) and 14(b), a single way two suffix-insert conflict (depicted in Figures 14(c)) and two ways a suffix-remove conflict with a suffix-insert (Figures 14(d) and 14(e)).

Next, we show that the three scenarios of Figures 14(a), 14(c) and 14(e) cannot occur because $curr.next = next$ until the remove or the insert completes.

Lemma 11. *Let* π *be the* write($curr.next, *$) *of an* insert *or, a* read($curr.next.next$) *or a* write($curr.next, *$) *of a* remove *operation. While* π *executes:*

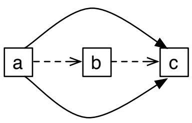
– *either* $curr.next = next$,

– *or* π *aborts its transaction.*

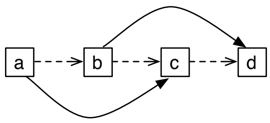
Proof. Assume that $curr.next \neq next$. The proof relies on the fact that π detects that $curr$ and $next$ are not consecutive and aborts its transaction.

By functions find() and read(), at some time t during the last iteration of its main loop we have $curr.next = next$. At this time t, either $curr.tlk.time \leq ub$ and $next.tlk.time \leq ub$ or ub is updated to $next.tlk.time$ such that $next.tlk.time > curr.tlk.time$ before find() returns. Hence $curr.tlk.time \leq ub$ and $next.tlk.time \leq ub$ when the regular transaction of π starts.

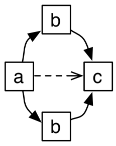
Since any modification uses a two-phase locking mechanism to modify the value and the timestamp of a location, either π detects that $curr$ is locked, i.e., $curr.tlk.owner \neq \bot$ or it detects that its version is too recent, i.e., $curr.tlk.time > ub$. In the former case, since no previous write operation occurs in the same transaction,



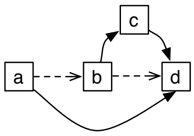
(a) remove(b) // remove(b)



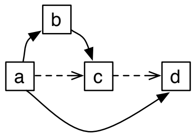
(b) remove(b) // remove(c)



(c) insert(b) // insert(b)



(d) remove(b) // insert(c)



(e) remove(c) // insert(b)

Figure B.14: The five scenarios in which accesses at the end of remove and insert conflict on mutable variables.

curr.tlk.owner $\neq t$ and this transaction aborts as indicated Lines 40 and 63 of Algorithm 2. In the latter case, it cannot extend and aborts because the transaction type is elastic (Line 67).

Consequently, either *curr.next* = *next* or π aborts its enclosing transaction (Line 63 or Line 67 of Algorithm 2).

Given Lemma 11, the remaining scenarios represented in Figures 14(b) and 14(d) can neither occur because *next.next* is not updated while remove completes, as shown below.

Lemma 12. *Let π be a* read(*curr.next.next*) *of a* remove *operation. While π executes:*

- *either curr.next.next = n,*
- *or π aborts its transaction.*

Proof. First, by Lemma 9 n is set to *next.next* when find returns. The search part of the remove, and the find it invokes, execute both within the scope of the remove transaction so that when the search returns, we have *curr.next* = *next*. Therefore n = *curr.next.next* at Line 41 of Algorithm 4.

Second, we have to show that n = *curr.next.next* remains true for the commit step of *remove* to occur. When write(*curr.next*, n) occurs (Line 42 of Algorithm 4) we have *last-r-entry* $\neq \emptyset$, hence any modification to *last-r-entry* = *curr.next.next* is checked. If the timestamp of *curr.next.next* has changed or its lock is acquired, then the remove transaction aborts as shown Line 81 of Algorithm 2, otherwise the *curr.next.next* get transferred to the *r-set* of the transaction. When free is invoked (Line 43 of Algorithm 4), another write executes but now with *last-r-entry* $\neq \emptyset$. The *curr.next.next* belonging now to the *r-set* is checked, and if a change has occurred indicated by its lock being taken or its timestamp being increased, then the remove transaction aborts. Consequently, for the commit to be invoked, no changes should have occurred to *curr.next.next* before. In case no such abort occurred at the time of the commit invocation, the remove has already acquired a lock on *curr.next.next* as its free previously wrote the *curr.next* node including its *curr.next.next* field (Line 13 of Algorithm 4), so that no concurrent transactions can update *curr.next.next* before the lock gets released at the end of the commit.

Theorem 13. *The linked list set implemented by Algorithm 4 is linearizable.*

Proof. Observe that by Invariant 10 the linked list is a well-formed sorted list with no duplicates. First, we fix a serialization point for each operation occurring between its invocation and response times that defines a partial order on all operations. Consider, Π_1 and Π_2, two update operations that conflict as indicated in Figure 14(a), 14(c) and 14(e) (resp. Figures 14(b) and 14(d)). Let $\Pi_1 \rightarrow_A \Pi_2$ where Π_1 is the first to acquire the lock within its write(*a.next*) (resp. write(*b.next*)) operation. Consider now a find operation Π_1 and an update operation Π_2. If Π_1 executes a read(*next*) on a variable after it was locked by the write of Π_2, then $\Pi_1 \rightarrow_A \Pi_2$, otherwise $\Pi_2 \rightarrow_A \Pi_1$. By definition, the relation $\rightarrow_A$ defines a partial order on the set of all operations.

Second, we indicate that any linked list execution is equivalent to some sequential execution where each operation executes correctly at its serialization point. If a read occurs immediately after an update acquires a lock on *curr*, then find will spin over the lock until *curr.tlk.owner* = $\perp$. Hence the read will observe the result of the update and will appear to be ordered after and by Lemma 9, *curr.next.key* $\geq i$ at the time of read. If the read is part of a search operation, then the search will also be ordered after, as it simply checks whether the immutable *key* field of the node *next* returned by the find is actually i. If the read is part of an update operations, then the update will not execute before the concurrent write, because it will increments the clock to a higher value than the written version. Now if two writes conflict, then the operation writing first will not be ordered after the other because of Lemmas 11 and 12. The serial specification of the set follows from the serial specification of read/write data items.