Elastic Transactions*

Pascal Felber[1], Vincent Gramoli[1,2], and Rachid Guerraoui[2]

[1] University of Neuchâtel, Switzerland
[2] EPFL, Switzerland

Abstract. This paper presents *elastic transactions*, a variant of the transactional model. Upon conflict detection, an elastic transaction might drop what it did so far within a separate transaction that immediately commits, and initiate a new transaction which might itself be elastic. Elastic transactions are a complementary alternative to traditional transactions, particularly appealing when implementing search structures. Elastic transactions can be safely composed with normal ones, but significantly improve performance if used instead.

1 Introduction

Background. Transactional memory (TM) is an appealing synchronization paradigm for leveraging modern multicore architectures. The power of the paradigm lies in its abstract nature: no need to know the internals of shared object implementations, it suffices to delimit any critical sequence of shared object accesses using transactional boundaries. Not surprisingly, however, this abstraction sometimes severely hampers parallelism. This is particularly true for search data structures where a transaction do not know a priori where to add an element unless it explores a large part of the data structure. Consider for instance an integer set that supports search, insert, and remove operations. Assume furthermore that the set is implemented with a bucket hash table. A bucket, implemented with a sorted linked list, indicates where an integer should be stored. Consider a situation where one transaction searches for an integer whereas another one seeks to insert an integer after a node that has been read by the first transaction: in a strict sense, there is a read-write conflict, yet this is a false (search-insert) conflict.

We propose *elastic* transactions, a new type of transactions that enables to efficiently implement search data structures and use them with regular transactional applications. As for a regular transaction, the programmer must simply delimit the blocks of code that represent elastic transactions. Nevertheless, during its execution, an elastic transaction can be cut into multiple normal transactions, depending on the conflicts detected. We show that this model is very effective whenever operations parse a large part of the structure while their effective update is localized.

* This work was supported in part by the European Velox project (ICT-216852).

I. Keidar (Ed.): DISC 2009, LNCS 5805, pp. 93–107, 2009.

Elastic transactions: a primer. To give an intuition of the idea behind elastic transactions, consider again the integer set abstraction. 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 concurrently in the same linked list. Each insert transaction parses the nodes in ascending order up to the node before which they should insert their 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 resulting history of operations where transaction j inserts 1 while transaction i is parsing the data structure to insert 3 at its end. (In the following history examples we indicate only operations of non-aborting transactions, thus, commit events have been omitted 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 since there is no sequential history that allows $r(h)^i$ to occur before $w(h)^j$ and also $r(n)^j$ to occur before $w(n)^i$. A traditional transactional model would detect a conflict between transactions i and j, and the transactions could not both commit. Nonetheless, history $\mathcal{H}$ does not violate the high-level 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.

To make a transaction elastic, the programmer has simply to label this transaction as being so and use its associated operations to access the shared memory. Assume indeed that transaction i has been labelled as elastic. History $\mathcal{H}$ can now be viewed as a slightly different history, $f(\mathcal{H})$:

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

The elastic transaction i has been cut in two transactions s_1 and s_2, each being atomic. The cut is only possible because 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 the elastic transaction ensure that no modifications on n and t have occurred between $r(n)^{s_1}$ and $r(t)^{s_2}$. Otherwise, i would have to abort.

Even though a read value has been freshly modified by another transaction, it might not be necessary to abort and restart from the beginning. Assume that a transaction i searches for a key that is not in the linked list while a transaction j is inserting a 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 key and its next pointer. In the following history $\mathcal{H}'$, transaction i reads node n_k and detects that it has freshly been modified by another transaction j.

$$\mathcal{H}' = ..., r(n_k)^j, r(n_{k+1})^j, r(n_{k-1})^i, w(n_k)^j, r(n_k)^i, r(n_{k+1})^i, ...$$

In this example, transaction i does not have to abort and restart from the beginning because it is the first time it accesses n_k and because the preceding node accessed by i has not been overwritten since then. Hence, after making sure that

the previously read node n_{k-1} has not been modified, transaction i can resume and commit, as if its read of n_k was part of a new transaction s_k, serialized after j. Hence, we get the following history.

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

$\mathcal{E}$-STM. We propose $\mathcal{E}$-STM, an implementation of our transactional model that uses timestamps, two-phase-locking, and universal operations. It provides both normal transactions and elastic transactions, allowing the latter ones to be cut to achieve high concurrency, but it retains the abstraction simplicity of transactional memory.

To evaluate the performance and simplicity of our solution, we implement it on four data structure applications: (i) linked list, (ii) skip list, (iii) red-black tree, and (iv) hash table. We compare $\mathcal{E}$-STM with three other synchronization techniques: (i) regular STM transactions, (ii) lock-based, and (iii) lock-free. The regular STM technique relies on TinySTM [1], the fastest STM for micro-benchmarks we know of [1, 2]. The lock-based and lock-free implementations are based on the algorithms of Herlihy, Luchangco, Shavit et al. [3, 4], and of Fraser, Harris, and Michael [5, 6, 7], respectively. We also implemented complex operations, move and sum, to illustrate how transactions can be combined.

The results we obtained indicate that $\mathcal{E}$-STM speeds up regular transactions on all workloads and with an average speedup factor of 36%. By lack of space, the detailed experiments are deferred to the companion technical report [8].

Roadmap. In the remainder of this paper, we present our system model (Section 2) and our transactional model (Section 3), and we give an implementation of it, called $\mathcal{E}$-STM (Section 4). Then, we elaborate on the advantages of using $\mathcal{E}$-STM, by illustrating its use in some data structure applications (Section 5). Finally, we present the related work (Section 6) before concluding (Section 7).

2 System Model

Our system comprises transactions and objects similarly to [9]. 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, it consists of a sequence of events that are an *operation invocation*, an *operation response*, a *commit invocation*, a *commit response*, and an *abort event*.

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 is denoted by T and we consider two types of transactions: normal and elastic. We assume that the type of all transactions is initially known. The sets of normal transactions and elastic transactions are denoted by $\mathcal{N}$ and $\mathcal{E}$, respectively. The set of possible objects is denoted by X and the set of

possible values is V. An *operation* accessing an object x, 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 $X \times T \times V \times type$.

Histories. We consider only 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 commit and abort, and (iii) a transaction cannot invoke an operation after having completed. We refer to these well-formed sequences as *histories*.

A history $\mathcal{H}$ is complete if all its transactions are completed. 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 request for t in $\mathcal{H}$;
- q is a commit or an abort event if there is a commit request 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 $committed(\mathcal{H})$ (resp. $aborted(\mathcal{H})$). The history of all committed transactions of a given history $\mathcal{H}$ is denoted by $permanent(\mathcal{H}) = \mathcal{H}|committed(\mathcal{H})$. 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 t *precedes* t' in $\mathcal{H}$ (denoted 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 none 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.

Operation sequences. For simplicity, 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 event and response event 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 and operation 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 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 the serial specification of which requires that a read operation on x returns the last value written on x, or its default value v_x (if the value has not been written before). Without loss of generality, 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 refer to a transaction that never writes an object value in the shared memory as an *invisible* transaction. Observe that an invisible transaction may, however, write some metadata (e.g., lock ownership) in the shared memory. An example is a transaction that acquires some locks before aborting.

3 Elastic Transactions: Definition

An *elastic* transaction is a transaction the size of which may vary depending on conflicts. More precisely, such transaction may cut itself upon conflict detection as if the start of the transaction has moved forward, hence the name elastic. Next, we explain how a cut is achieved.

First, note that a sequence of operations is a totally ordered set, hence, 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 $S_{\mathcal{H}}$. More generally we refer to any sequence $\mathcal{S}$ as a totally ordered set denoted by $\langle S_{\mathcal{S}}, \rightarrow_{\mathcal{S}} \rangle$.

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 $\rightarrow_{\mathcal{H}'} \subseteq \rightarrow_{\mathcal{H}}$. Next, we define the notion of cut and its well-formedness.

Definition 1 (Cut). *A* cut *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 cut sub-history contains only consecutive operations of $\mathcal{H}$: 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 $\pi_i \in \mathcal{H}'$;*
2. *if one sub-history precedes another in $\mathcal{C}$ then the operations of the first precede the operations of the second in $\mathcal{H}$: 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 $\pi_1 \rightarrow_{\mathcal{H}} \pi_2$;*
3. *any operation of $\mathcal{H}$ is in exactly one sub-history of the cut: $\bigcup_{\forall \mathcal{H}' \in S_{\mathcal{C}}} S_{\mathcal{H}'} = S_{\mathcal{H}}$ 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 example, there are four cuts of history a, b, c, denoted by $\mathcal{C}1 = \{a, b\ ;\ c\}$, $\mathcal{C}2 = \{a\ ;\ b, c\}$, $\mathcal{C}3 = \{a\ ;\ b\ ;\ c\}$, and $\mathcal{C}4 = \{a, b, c\}$, where semi-colons are used to separate consecutive sub-histories of the cut and braces are used for clarity to enclose a cut. In contrast, neither $\{a, c\ ;\ b\}$ nor $\{a\ ;\ a, b, c\}$ are cuts of $\mathcal{H}$. The reason it that the former violates property (1) while the latter violates property (3) of Definition 1.

Definition 2 (Well-formed cut). *A cut $\mathcal{C}_t$ of history $\mathcal{H}|t$, where t is a transaction, is* well-formed *if for any of its sub-histories s_i the following properties are satisfied:*

1. *if s_i contains only one operation, then there is no other $s_j \in S_{\mathcal{C}_t}$;*
2. *if $\pi_i \in s_i$ and $\pi_j \in s_j$ are two write operations of t, then $s_i = s_j$;*
3. *if π_i is the first operation of s_i, then either π_i is a read operation or π_i is the first operation of t.*

For example, consider 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 x. (For the following examples, we omit the values returned by the read operations and consider that the object serial specification is satisfied.)

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

There are two well-formed cuts of history $\mathcal{H}_1|t$ that are $\mathcal{C}1' = \{r(u), r(v), w(x), r(y), r(z)\}$ and $\mathcal{C}2' = \{r(u), r(v), w(x)\ ;\ r(y), r(z)\}$, however, neither $\mathcal{C}3' = \{r(u)\ ;\ r(v), w(x)\ ;\ r(y), r(z)\}$ nor $\mathcal{C}4' = \{r(u), r(v)\ ;\ w(x), r(y), r(z)\}$ are well-formed. More precisely, the first sub-history of $\mathcal{C}3'$ contains only one operation violating property (1) of Definition 2 and the second sub-history of $\mathcal{C}4'$ starts with a write operation, that is, property (3) of Definition 2 is violated. In the remainder of this paper, we only consider well-formed cuts.

Next, we define a consistent cut with respect to a history of potentially concurrent transactions. This definition is crucial as it indicates the singularity of elastic transactions. The programmer can label a transaction as elastic if he(she) does not need this transaction to appear as atomic, but still he(she) requires that a set of consecutive operations in this transaction appear as atomic, as formalized below. In a history $\mathcal{H}$, a cut is consistent if there are no writes separating two of its sub-histories each accessing one of the object written by these writes.

Definition 3 (Consistent cut). *A cut $\mathcal{C}_t$ of $\mathcal{H}|t$ 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 ($s_i \neq s_j$), 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 $\mathcal{H}_2$ where e is an elastic transaction and n is a normal transaction, and where $r(x)^t$ and $w(x)^t$ refer to a read and a write operation on x in transaction t.

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

Two consistent cuts of $\mathcal{H}_2|e$ with respect to $\mathcal{H}_2$ are possible. One contains two sub-histories $\mathcal{C}1 = \{r(x)^e, r(y)^e\ ;\ r(z)^e, w(u)^e\}$ while the other contains one sub-history $\mathcal{C}2 = \{r(x)^e, r(y)^e,\ r(z)^e, w(u)^e\}$. Observe that $\mathcal{C}1$ is consistent because there are no two writes from other transactions that occur at objects between the accesses of e to these objects, hence $r(y)^e$ and $r(z)^e$ seem to execute atomically at the time $r(y)^e$ occurs. In contrast, consider history $\mathcal{H}_3$ where e is elastic and n is normal.

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

There is no consistent cut of $\mathcal{H}_3|e$ with respect to $\mathcal{H}_3$ because n writes y and z between the times e reads each of them.

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 $\pi_i = \langle x, t, v, type\rangle \in s_i^t$ then $\pi'_i = \langle x, s_i^t, v, type\rangle$, otherwise $\pi_i = \pi'_i$, and if $t \in committed(\mathcal{H})$ then $s_i^t \in committed(f_{\mathcal{C}_t}(\mathcal{H}))$, otherwise $s_i^t \in aborted(f_{\mathcal{C}_t}(\mathcal{H}))$. We denote the composition of f for a set of cuts $\mathcal{C} = \{\mathcal{C}_1, ..., \mathcal{C}_m\}$ by $f_{\mathcal{C}} = f_{\mathcal{C}_1} \circ ... \circ f_{\mathcal{C}_m}$.

Next, we define an elastic-opaque transactional system, which combines normal and elastic transactions; this definition relies on Definition 3 of consistent cut, and the definition of opacity [10].

Definition 4 (Elastic-opacity). *A transactional system is* elastic-opaque *if, for any history $\mathcal{H}$ of this system, there exists a consistent cut $\mathcal{C}_t$ for each elastic transaction t of $\mathcal{H}|\mathcal{E}$ with $\mathcal{C} = \{\mathcal{C}_t\}$, such that $f_{\mathcal{C}}(\mathcal{H})$ is opaque.*

As an example, consider the following history $\mathcal{H}_4$ and assume e is elastic while n is normal and both transactions commit:

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

This history would clearly not be serializable in a traditional model (with e and n two normal transactions) since there is no sequential histories that allow not only $r(x)^e$ to occur before $w(x)^n$ but also $r(z)^n$ to occur before $w(z)^e$. However, there exists one consistent cut $\mathcal{C}_e$ of $\mathcal{H}_4|e$ with respect to $\mathcal{H}_4$, $\mathcal{C}_e = s_1, s_2$ 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 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).

4 Implementation of Elastic Transactions

This section introduces $\mathcal{E}$-STM, a software transactional memory system that implements elastic transactions. The corresponding pseudocode appears in Algorithm 1. $\mathcal{E}$-STM combines two-phase locking, timestamp mechanism, and atomic primitives: compare-and-swap (Lines 79), fetch-and-increment (Line 94), and atomic loads and stores.

Transaction and variable state. A transaction t starts with a begin(*type*) indicating whether its type is elastic or normal. Then, it accesses the memory locations using read or write operations. Finally, it completes either by a commit call or by an abort that restarts the same transaction. The try-extend and ver-val-ver are helper functions. A transaction t may keep track of the variable it has accessed since it has lastly started using a *r-set* to log the reads and a *w-set* to log

Algorithm 1. $\mathcal{E}$-STM, a transactional memory providing elastic transactions

```
1:  clock ∈ ℕ, initially 0

2:  State of variable x:
3:    val ∈ V
4:    tlk a timestamped lock with fields:
5:      owner ∈ T, the lock owner, initially ⊥
6:      time ∈ ℕ, a version counter, initally 0
7:      w-entry ∈ X × V × ℕ, an entry address
8:        initally ⊥
9:      // time/w-entry share same location

10: State of transaction t:
11:   type ∈ {elastic, normal}, initially the
12:     ancestor transaction type or ⊥ (if none)
13:   r-set and w-set, sets of entries with fields:
14:     addr ∈ X, an address
15:     val ∈ V, its value
16:     ts ∈ ℕ, its version timestamp
17:   last-r-entry ∈ X × ℕ, an entry,
18:     initally ⊥
19:   lb ∈ ℕ, initially 0 // time lower bound
20:   ub ∈ ℕ, initially 0 // time upper bound

21: begin(tx-type)_t:
22:   ub ← clock
23:   lb ← clock
24:   // if nested inside a normal, be normal
25:   if type ≠ normal then type ← tx-type

26: try-extend()_t:
27:   // make sure read values haven't changed
28:   now ← clock
29:   for all ⟨y, ∗, ts⟩ ∈ r-set do
30:     ow ← y.tlk.owner
31:     last ← y.tlk.time
32:     if ow ∉ {t, ⊥}∨
33:          (ow = ⊥ ∧ last ≠ ts) then
34:       abort()
35:   ub ← now

36: ver-val-ver(x, evenlocked)_t:
37:   // load a versioned value from memory
38:   repeat:
39:     ℓ1 ← x.tlk
40:     v ← x.val
41:     ℓ2 ← x.tlk
42:   until (ℓ1 = ℓ2∧
43:     (ℓ1.owner = ⊥ ∨ evenlocked))
44:   return ⟨ℓ1, v⟩

45: abort()_t:
46:   for all ⟨x, ∗, ∗⟩ ∈ write-set do
47:     x.tlk.owner ← ⊥
48:   begin(type) // restart from the beginning

49: read(x)_t:
50:   // log normal reads for later extensions
51:   if type = normal ∨ w-set ≠ ∅ then
52:     ⟨ℓx, vx⟩ ← ver-val-ver(x, true)
53:     if ℓx.owner ∉ {t, ⊥} then ctn_mgt()
54:     else if ℓx.owner = t then
55:       vx ← ℓx.w-entry.val
56:     else // ℓx.owner = ⊥
57:       if ℓx.time > ub then try-extend()
58:     r-set ← r-set ∪ {⟨x, vx, ℓx.time⟩}
59:   // ...or log only the most recent elastic read
60:   if type = elastic ∧ w-set = ∅ then
61:     ⟨ℓx, vx⟩ ← ver-val-ver(x, false)
62:     if ℓx.time > ub then
63:       if last-r-entry ≠ ⊥ then
64:         ⟨y, ∗⟩ ← last-r-entry
65:         ⟨ℓy, ∗⟩ ← ver-val-ver(y, false)
66:         if ℓy.time > ub then abort()
67:         ub ← ℓx.time
68:     last-r-entry ← ⟨x, ℓx.time⟩
69:   return vx

70: write(x, v)_t:
71:   // lock & postpone the write until commit
72:   repeat:
73:     ℓ ← x.tlk
74:     if ℓ.owner ∉ {⊥, t} then ctn_mgt()
75:     else if ℓ.time > ub then
76:       if type = normal then try-extend()
77:       else abort()
78:     w-entry ← ⟨x, v, ℓ.time⟩
79:     x.tlk ← ⟨t, ∗, w-entry⟩ // compare&swap
80:   until (x.tlk.owner = t)
81:   lb ← max(lb, ℓ.time)
82:   w-set ← (w-set \ {⟨x, ∗, ∗⟩}) ∪ {w-entry}
83:   // make sure last value read is unchanged
84:   if type = elastic ∧ last-r-entry ≠ ⊥ then
85:     ⟨e, time_e⟩ ← last-r-entry
86:     ⟨ℓe, ∗⟩ ← ver-val-ver(e, true)
87:     ow ← ℓe.owner
88:     last ← ℓe.time
89:     if ow ≠ t ∨ last ≠ time_e then abort()
90:     last-r-entry ← ⊥

91: commit()_t:
92:   // apply writes to memory and release locks
93:   if w-set ≠ ∅ then
94:     ts ← clock++ // fetch&increment
95:     if lb ≠ ts − 1 then try-extend()
96:     for all ⟨x, v, ts⟩ ∈ write-set do
97:       x.val ← v
98:       x.tlk.time ← ts
99:       x.tlk.owner ← ⊥
```

the writes. More precisely, the entries of these sets contain the variable address, *addr*, its value *val*, and its version *ts* (Lines 13–16). If t is elastic, it may only need to keep track of the last read operation, so it uses *last-r-entry* (Lines 17 and 18) 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 19 and 20).

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 (Line 3) *val* and a timestamped lock *tlk*, also called versioned write-lock [11]. 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 the most recently been written, and (iii) *w-entry*, a reference to the corresponding entry in the owner's write set (Lines 4–9). Timestamps are given by a global counter, *clock* (Line 1), that does not hamper scalability [11, 12, 1].

Normal transactions. The algorithm restricted to normal transactions builds upon TinySTM [1] logging all operations. All transactions use two-phase locking when writing to a memory location. While the location is locked by the transaction at the time it executes the write, all updates are buffered into a write-set, *w-set*, until the commit time at which these updates are applied to the memory. When a transaction performs a write$(x, *)$, it acquires the lock of x using a compare-and-swap (Line 79) and holds it until it commits or aborts. When accessing a locked variable, the transaction detects a conflict and calls the contention manager, which typically aborts the current transaction (Lines 53 and 74). Various contention management policies could be used instead to handle conflicts between normal transactions.

When a read request on variable x as part of transaction t is received by $\mathcal{E}$-STM, the value of x is read in a three-step process called ver-val-ver, which consists in 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 42) indicating that the value corresponds to that version. Only in some cases needs the value be returned unlocked, hence the use of the boolean *evenlocked*. The transactions of $\mathcal{E}$-STM use the extension mechanism of LSA [12, 1]. Each transaction t maintains an interval of time $[lb, ub]$ indicating the time during which t can be serialized. More precisely, for a given transaction t, lb and ub represent respectively 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 check. 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 34), meaning that some variables, among the ones t has read, have been updated by another transaction since then.

Elastic transactions. An important difference between normal and elastic transactions is that elastic transactions never use the *r-set* until they read after

a write, as there is at most one read operation the transaction has to keep track of: the most recent one. Hence, elastic transactions use the *last-r-entry* field to log the last read operation. In our implementation all reads following a write in an elastic transaction will use the *r-set* like normal transactions (Lines 50–58), however, the implementation could be improved using static analysis to require this only for reads that are both preceded and succeeded by write operations in the same transaction.

Upon reading x (without having written before) an elastic transaction must make sure that the value v_x it reads was present at the time the immediately preceding read occurs. This typically ensures that a thread does not return an inconsistent value v_x after having been pre-empted, for example. 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 62–67). This can be viewed as a partial roll-back similar to the one provided by nested models, except that no on-abort definition is necessary and only a single operation would have to be re-executed here. 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 89). If, however, no other transaction tried to update this address since it has been read, then the write executes as normal (Lines 71–82).

Next, we state the theorem on the correctness of our implementation. The complete proof has been deferred to the companion technical report [8].

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

5 Evaluation

$\mathcal{E}$-STM is simple to program with for two reasons: (i) it indeed provides a high-level abstraction that do not expose synchronization mechanisms to the programmer, and (ii) it enables code composition.

5.1 Abstraction

As with a classical transactional model, the programmer can use $\mathcal{E}$-STM to write a concurrent program almost as if he (she) was writing a sequential program. Like all TMs, $\mathcal{E}$-STM provides the programmer with labels *begin* and *commit* that can delimit the transactions. Hence, all calls to reading and writing the shared memory are redirected to the wrappers read and write of $\mathcal{E}$-STM, but this redirection does not incur efforts from the programmer and can be made automatic: some compilers already detect transaction labels and redirect memory accesses of these transactions automatically even though it is known that over instrumentation of accesses may unnecessarily impact performance.

To illustrate this, consider the sorted linked list implementation of an integer set, where integers (node *keys*) can be searched, removed, and inserted.

Algorithm 2. Linked list implementation built on $\mathcal{E}$-STM (the lock-free harris-ll-find function 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)

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:         curr_next ← t_next
55:       curr ← unmarked(next)
56:       if !t_next then break
57:       t_next ← t.next
58:     until is_marked(t_next) ∨ (t.key < i)
59:     next = t
60:     // 2. check nodes are adjacent
61:     if curr_next = next then
62:       if (next.next∧
63:           is_marked(next.next) then
64:         goto line 48
65:       else return ⟨curr, next⟩
66:     // 3. remove one or more marked node
67:     if cas(curr.next, curr_next, next) then
68:       if (next.next∧
69:           is_marked(next.next)) then
70:         goto line 48
71:       else return ⟨curr, next⟩
72:   end loop
```

Algorithm 2 depicts the entire program that uses $\mathcal{E}$-STM plus a core function of the lock-free Harris [5] implementation, for comparison purpose. It is pretty clear that this harris-ll-find function is more complex than its ll-find counterparts 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 with each operation. Unlike the Harris lock-free function, $\mathcal{E}$-STM-based functions are very simple, as all synchronizations are handled transparently underneath by $\mathcal{E}$-STM. The pseudocode is the same as the non-thread-safe version, except that begin(*elastic*), and commit have been added at the right places in the code.

5.2 Extensibility

$\mathcal{E}$-STM combines elastic transactions with normal transactions which makes it easily extensible. To illustrate this, we extended the hash table example with operations move and sum. The pseudocode is presented in Algorithm 3.

More specifically, we implemented the insert, search, and remove operations using elastic transactions. Since each bucket of the hash table is implemented with a linked list, we re-used (Lines 12, 18, and 24) the program of the linked list written above. More complex operations like move and sum have been implemented using normal transactions. The elastic transactions nested inside the normal transactions of move (Lines 16 and 22) execute in the normal mode. Although an elastic implementation of move is possible, sum cannot be elastic as it requires an atomic snapshot of all elements of the data structure. This example illustrates the way elastic and normal 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 not to see 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 in finding it, then search will be serialized after move, if search looks for the targeted key k' of v and does not find it, then search will be serialized before move. In 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 normal transactions, otherwise, a concurrent move may lead to an inconsistent state. Another issue, pointed out in [13], 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

Algorithm 3. Hash table implementation built on $\mathcal{E}$-STM and linked list

```
1:  State of process p:
2:    node a record with fields:
3:      key, an integer
4:      next, a node
5:    set a mapping from an integer to a
6:      linkedlist representing a bucket.
7:    Initially, all buckets of the set are
8:      empty lists.

9:  ht-search(i)_p:
10:   begin(elastic)
11:   a ← hash(i)
12:   result ← set[a].ll-search(i)
13:   commit()
14:   return result

15: ht-insert(i)_p:
16:   begin(elastic)
17:   a ← hash(i)
18:   result ← set[a].ll-insert(i)
19:   commit()
20:   return result

21: ht-remove(i)_p:
22:   begin(elastic)
23:   a ← hash(i)
24:   result ← set[a].ll-remove(i)
25:   commit()
26:   return result

27: ht-sum()_p:
28:   begin(normal)
29:   for each bucket in set do
30:     next ← read(bucket.head.next)
31:     while next.next ≠ ⊥ do
32:       sum ← sum + read(next.val)
33:       next ← read(next.next)
34:   commit()
35:   return sum

36: ht-move(i, j)_p:
37:   begin(normal)
38:   ht-remove(i)
39:   ht-insert(j)
40:   commit()
41:   return result
```

where both x and y are present. Again, our model copes with this issue as the programmer can use a normal transaction to encapsulate each conditional insertion. These normal and elastic transactions are safely combined.

Unlike elastic transactions, existing synchronization techniques (e.g., based on locks or compare-and-swap) cannot be easily combined with normal transactions. They furthermore introduce a significant complexity. Using a coarse-grained lock to make the hash table move operation atomic would prevent concurrent accesses to the data structure. In contrast, using fine-grained locks may lead to a deadlock if one process moves from bucket ℓ_1 to bucket ℓ_2 while another moves from ℓ_2 to ℓ_1. With a lock-free approach (e.g., based on an underlying compare-and-swap), one could either modify a copy of the data structure before switching a pointer from one copy to another, or use a multi-word compare-and-swap instruction. Unfortunately, the former solution is costly in memory usage whereas the latter solution requires a rarely supported instruction that is also considered as inefficient. Implementing a lock-free resize operations reveals even more as this requires to replace its internal bucket linked lists by a single linked list imposing to re-implement the whole data structure [14].

5.3 Experiments

Here we compare $\mathcal{E}$-STM and the default version of TinySTM on a 16 core machine. We chose TinySTM as it is the fastest STM on micro-benchmarks we know of [1, 2]. We ran the linked list integer set implementation of Algorithm 2 on 16 threads with $\mathcal{E}$-STM and we replaced elastic transaction calls by normal transaction calls to run it with TinySTM. In this experiment, $\mathcal{E}$-STM is almost twice as fast as TinySTM (1.9x faster on average and up to 2.3x faster). For the graphs including other testbed data structures and other synchronization techniques, please refer to the technical report [8].

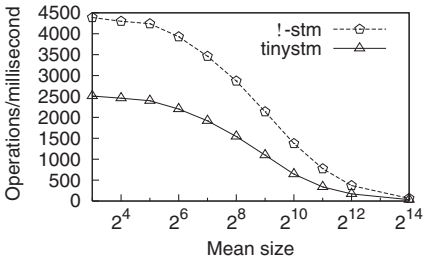

Fig. 1. Performance results when running 5% ll-insert, 5% ll-remove, and 90% ll-search operations as elastic ($\mathcal{E}$-STM) and normal (TinySTM) transactions

6 Discussion and Related Work

One programmer may think of cutting normal transactions himself (herself) instead of using elastic transactions. Nevertheless, hand-crafted cuts must be defined prior to execution which may lead to inconsistencies. As an example, consider that a transaction t searches a linked list. A hand-crafted cut of t between two read operations on x and y may lead to an inconsistent state if another transaction deletes y (by modifying the next pointers of x and y) between those reads: t does not detect that it stops parsing the data structure as soon as

it tries to access y. In contrast, elastic transactions avoid this issue by checking dynamically if a transaction can be safely cut and aborting otherwise.

Besides elastic transactions, there have been several attempts to extend the classical transactional model. Open nesting [15] provides sub-transactions that can commit while the outermost transaction is not completed yet. More precisely, open nesting makes sub-transactions visible before the outermost transaction commits. This requires the programmer to define complex roll-backs [16].

Transactional boosting [17] is a methodology for transforming linearizable objects into transactional objects, which builds upon techniques from the database literature. Although transactional boosting enhances concurrency by relaxing constraints imposed by read/write semantics at low-level, it requires the programmer to identify the commutative operations and to define inverse operations for non-commutative ones.

Abstract nesting [18] allows to abort partially in case of low-level conflict. As the authors illustrate, abstract nested transactions can encapsulate independent sub-parts of regular transactions like insert and remove sub-parts of a move transaction. In contrast, abstract nested transactions cannot encapsulate sub-parts of the parsing (as in search/insert/remove) of a data structure. Moreover, abstract nested transactions aim at reducing the roll-back cost due to low-level conflicts, but not at reducing the amount of low-level conflicts.

Early release [19] is the action of forgetting past reads before a transaction ends. This mechanism, presented for DSTM, enhances concurrency by decreasing the number of low-level conflicts for some pointer structures. It requires the programmer to carefully determine when and which objects in every transaction can be safely released [13]: if an object is released too early then the same inconsistency problem as with hand-crafted cuts arises. Finally, early release provides less concurrency than elastic transactions. Consider a transaction t that accesses x and y before releasing x. If y is modified between t accessing x and y then a conflict is always detected. In contrast, if t is an elastic transaction then a conflict is detected only if x and y are consecutively accessed by t and both x and y are modified between those accesses, which is very unlikely in practice.

7 Conclusion

We have proposed a new transactional model that enhances concurrency in a simple fashion. 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. We implemented this model in an STM, called $\mathcal{E}$-STM, that only requires to differentiate elastic from traditional transactions, making it simple to program with. Comparisons on data structures have confirmed that elastic transactions are simpler than lock-based and lock-free techniques and faster than regular transactions. It could be interesting to investigate how much performance other applications could gain from using this model. For example, the counter increment on which the rest of the transaction does not depend.

References

1. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 237–246 (2008)
2. Harmanci, D., Felber, P., Gramoli, V., Fetzer, C.: TMunit: Testing software transactional memories. In: The 4th ACM SIGPLAN Workshop on Transactional Computing (2009)
3. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N., Shavit, N.: A lazy concurrent list-based set algorithm. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 3–16. Springer, Heidelberg (2006)
4. Herlihy, M., Lev, Y., Luchangco, V., Shavit, N.: A simple optimistic skiplist algorithm. In: Prencipe, G., Zaks, S. (eds.) SIROCCO 2007. LNCS, vol. 4474, pp. 124–138. Springer, Heidelberg (2007)
5. Harris, T.: A pragmatic implementation of non-blocking linked-lists. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 300–314. Springer, Heidelberg (2001)
6. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 73–82. ACM, New York (2002)
7. Fraser, K.: Practical lock freedom. PhD thesis, University of Cambridge (2003)
8. Felber, P., Gramoli, V., Guerraoui, R.: Elastic transactions. Technical Report LPD-REPORT-2009-002, EPFL (2009)
9. Weihl, W.E.: Local atomicity properties: Modular concurrency control for abstract data types. ACM Trans. Program. Lang. Syst. 11(2), 249–283 (1989)
10. Guerraoui, R., Kapałka, M.: On the correctness of transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 175–184 (2008)
11. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
12. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 284–298. Springer, Heidelberg (2006)
13. Skare, T., Kozyrakis, C.: Early release: Friend or foe? In: Workshop on Transactional Memory Workloads (2006)
14. Shalev, O., Shavit, N.: Split-ordered lists: Lock-free extensible hash tables. J. ACM 53(3), 379–405 (2006)
15. Moss, J.E.B.: Open nested transactions: Semantics and support. In: Workshop on Memory Performance Issues (2006)
16. Ni, Y., Menon, V., Abd-Tabatabai, A.R., Hosking, A.L., Hudson, R.L., Moss, J.E.B., Saha, B., Shpeisman, T.: Open nesting in software transactional memory. In: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 68–78 (2007)
17. Herlihy, M., Koskinen, E.: Transactional boosting: A methodology for highly-concurrent transactional objects. In: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 207–216 (2008)
18. Harris, T., Stipić, S.: Abstract nested transactions. In: The 2nd ACM SIGPLAN Workshop on Transactional Computing (2007)
19. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing, pp. 92–101. ACM, New York (2003)