Elastic Transactions

Pascal Felber
Univ. of Neuchâtel, Switzerland
pascal.felber@unine.ch

Vincent Gramoli
EPFL and Univ. of Neuchâtel
vincent.gramoli@epfl.ch

Rachid Guerraoui
EPFL, Switzerland
rachid.guerraoui@epfl.ch

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. Both forms of transactions can safely be combined within the same application.

We implemented software support for elastic transactions and evaluated them on four common data structure applications, namely *linked list, skip list, red-black tree* and *hash table*. Our implementation is faster than a state-of-the-art software transactional memory in various workloads and with an improvement of 36% on average. It also presents an improvement over lock-based solutions of 89% on average.

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 transactions 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.

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, $f(\mathcal{H})$:

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

The elastic transaction 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 the transaction 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 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 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.

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

$\mathcal{E}$-STM. We propose $\mathcal{E}$-STM, an implementation of our transactional model that uses timestamps, two-phase-locking, and universal atomic primitives. 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 compared $\mathcal{E}$-STM with three other synchronization techniques: (i) regular STM transactions, (ii) lock-based, and (iii) lock-free. The regular STM technique relies on TinySTM [2], the fastest STM for micro-benchmarks we know of [2, 5]. The lock-based and lock-free implementations are based on the algorithms of Herlihy, Luchangco, Shavit et al. [8, 10], and of Fraser, Harris, and Michael [3, 6, 12], 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 improvement of 36%. $\mathcal{E}$-STM presents competitive performance compared to lock-based techniques and its average improvement is 89%. Although it is less efficient than lock-free techniques, (regression of 47% on average), it is significantly simpler and, as we will argue in Section 6, does not hamper extensibility.

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) that we prove correct (Section 5). Then, we elaborate on the advantage of using $\mathcal{E}$-STM, we present four data structure applications that take advantage of the simplicity of our implementation, and we compare their performance with other synchronization techniques (Section 6). Finally, we present the related work (Section 7) before concluding (Section 8). The optional appendix includes the correctness proof of our $\mathcal{E}$-STM-based linked list implementation (Appendix A).

2. System Model

Our system comprises transactions and objects similarly to [19]. 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 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 $X \times T \times V \times type$.

2.1 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 invocation for t in $\mathcal{H}$;
- q is a commit 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 $committed(\mathcal{H})$ (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.

2.2 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 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 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 $\pi = \pi'$.

Next, we define three binary relations on the read and write operations of transactions. First, π_1 precedes directly π_2, denoted by $\pi_1 \prec_d \pi_2$ if π_1 and π_2 are two consecutive operations of the same transaction t. Second, π_1 precedes π_2 on object x, denoted by $\pi_1 \prec_x \pi_2$ if $\pi_1(x, v)$ is a write operation and $\pi_2(x) : v$ is a read

operation that returns the value written by π_1 (π_2 reads from π_1) or π_1 is a read or a write operation and π_2 is a write operation that overwrites the value accessed by π_1. The transitive closure of the union of these two precedence relations is denoted $\prec$. More precisely, let $\prec_*$ be the union of the two relations $\prec_d$ and $\prec_x$, hence $\pi_1 \prec_* \pi_2$ if and only if either $\pi_1 \prec_d \pi_2$ or $\pi_1 \prec_x \pi_2$. We obtain the following recursive definition for the precedence relation $\prec$. We say that $\pi_1 \prec \pi_2$ if and only if one of the two following properties hold:

- either $\pi_1 \prec_* \pi_2$,
- or there exists π_3 such that $\pi_1 \prec_* \pi_3$ and $\pi_3 \prec \pi_2$.

A sequential history $\mathcal{H}$ is *legal* 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 $\pi'(x, v) = max_{\rightarrow_\mathcal{H}}\{\pi'(x, *) \in \mathcal{H} \text{ s.t. } \pi' \rightarrow_\mathcal{H} \pi\}$ or 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 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 $S_\mathcal{H}$.

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 3.1 (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-histories 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 3.1.

DEFINITION 3.2 (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 3.2 and the second sub-history of $\mathcal{C}4'$ starts with a write operation, that is, property (3) of Definition 3.2 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 3.3 (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 3.3 of consistent cut, and the definition of opacity [4]. More precisely, it states that a system is elastic-opaque if, for any history of this system, there exists a consistent cut for each elastic transaction such that the history resulting from the composition of these cuts is opaque.

DEFINITION 3.4 (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:

$$r(x)^e, r(y)^e, r(x)^n, r(y)^n, r(z)^n, w(x)^n, r(t)^e, w(z)^e.$$

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. Elastic Transactions: Implementation

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 operations that

Algorithm 1 $\mathcal{E}$-STM

```
1:  clock ∈ ℕ, initially 0

2:  State of variable x:
3:      val ∈ V
4:      tlk a record with fields: // time-based lock
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 the same location

10: State of transaction t:
11:     type ∈ {elastic, normal}, initially the
12:         type of the ancestor transaction or ⊥
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 it is nested inside a normal, be normal
25:     if type ≠ normal then type ← tx-type

26: try-extend()_t:
27:     // make sure read values have not 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: read(x)_t:
46:     // log normal reads for later extensions
47:     if type = normal ∨ w-set ≠ ∅ then
48:         ⟨ℓx, vx⟩ ← ver-val-ver(x, true)
49:         if ℓx.owner ∉ {t, ⊥} then ctn_mgt()
50:         else if ℓx.owner = t then
51:             vx ← ℓx.w-entry.val
52:         else // ℓx.owner = ⊥
53:             if ℓx.time > ub then try-extend()
54:         r-set ← r-set ∪ {⟨x, vx, ℓx.time⟩}
55:     // ...or log only the most recent elastic read
56:     if type = elastic ∧ w-set = ∅ then
57:         ⟨ℓx, vx⟩ ← ver-val-ver(x, false)
58:         if ℓx.time > ub then
59:             if last-r-entry ≠ ⊥ then
60:                 ⟨y, ∗⟩ ← last-r-entry
61:                 ⟨ℓy, ∗⟩ ← ver-val-ver(y, false)
62:                 if ℓy.time > ub then abort()
63:             ub ← ℓx.time
64:         last-r-entry ← ⟨x, ℓx.time⟩
65:     return vx

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

88: abort()_t:
89:     for all ⟨x, ∗, ∗⟩ ∈ write-set do
90:         x.tlk.owner ← ⊥
91:     begin(type) // restart from the beginning

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

are supported at the hardware level by most common architectures: compare-and-swap (Lines 75), fetch-and-increment (Line 95), and atomic loads and stores.

4.1 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 $r\text{-}set$ to log the reads and a $w\text{-}set$ to log 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 $last\text{-}r\text{-}entry$ (Lines 17 and 18) to log a single address and its version instead of the entire set $r\text{-}set$. 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 [1]. 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) $w\text{-}entry$, a reference to the

corresponding entry in the owner's write set (Lines 4–8). Timestamps are given by a global counter, $clock$ (Line 1), that does not hamper scalability [1, 2, 15].

4.2 Normal Transactions

The algorithm restricted to normal transactions builds upon TinySTM [2] 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, $w\text{-}set$, 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 75) 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 49 and 70). 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 [2, 15]. 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, $r\text{-}set$, 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.

4.3 Elastic Transactions

An important difference between normal and elastic transactions is that elastic transactions never use the $r\text{-}set$ 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 $last\text{-}r\text{-}entry$ field to log the last read operation. In our implementation all reads following a write in an elastic transaction will use the $r\text{-}set$ like normal transactions (Lines 46–54), 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 $last\text{-}r\text{-}entry$ to be sure that the value read has not been overwritten since then (Lines 58–63). This can be viewed as a partial rollback 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 86). If, however, no other transaction tried to update this address since it has been read, then the write executes as normal (Lines 67–79).

5. Correctness of $\mathcal{E}$-STM

Here, we show that $\mathcal{E}$-STM is elastic-opaque in three steps. First, we give preliminary definitions. Second, we show that for each committed elastic transaction of $\mathcal{E}$-STM, there exists a consistent cut so that $f_{\mathcal{C}}$ 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.

THEOREM 5.1. *$\mathcal{E}$-STM is elastic-opaque.*

DEFINITION 5.2 (Opacity). *A history $\mathcal{H}$ is* opaque *if it satisfies the following properties:*

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

We define $sub\text{-}complete$ as a mapping between history $\mathcal{H}$ and a history $\mathcal{H}' = sub\text{-}complete(\mathcal{H})$ 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 $\mathcal{H}$, $sub\text{-}complete(\mathcal{H}) \in complete(\mathcal{H})$.

First-of-all, we show that there exists a consistent cut for any 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 5.3. *Let $\mathcal{H}$ be any history of $\mathcal{E}$-STM. There exists a consistent cut $\mathcal{C}_t$ of $\mathcal{H}|t$ with respect to $\mathcal{H}$.*

Proof. The proof relies essentially on the definition of a consistent cut (Definition 3.3). 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 $\pi(x)^t \rightarrow_{\mathcal{H}} \pi(x)^{t'} \rightarrow_{\mathcal{H}} \pi(x)^t$ nor be two writes operations $\pi(x)^{t'}\ \pi(y)^{t''}$ from transaction $t' \neq t$ and $t'' \neq t$ such that $\pi(y)^t \rightarrow_{\mathcal{H}} \pi(x)^{t'} \rightarrow_{\mathcal{H}} \pi(x)^t$ and $\pi(y)^t \rightarrow_{\mathcal{H}} \pi(y)^{t''} \rightarrow_{\mathcal{H}} \pi(x)^t$. First, we start by showing that the latter case cannot happen, the impossibility of the former case will follow.

Assume that such writes $\pi(x)^{t'}$ and $\pi(y)^{t''}$ exist, we show that t would abort leading to a consistent cut. 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 $t.ub$. Since t is elastic, this observation leads it to verify that the version of y has not changed (Line 58). Since, we have $y.tlk.time > t.ub$, the transaction aborts as indicated Line 62 and the resulting sequence of operations of t is a consistent cut. 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. Property (1) of Definition 5.2 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 5.4. *Let $\mathcal{H}$ be any history of $\mathcal{E}$-STM and let $\mathcal{H}' = sub\text{-}complete(f_{\mathcal{C}}(\mathcal{H}))$. If $t' \in aborted(\mathcal{H}')$ then t' is an invisible transaction.*

Proof. The proof is divided in 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 $t \in aborted(\mathcal{H})$, 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 memory can only be updated during the for-loop of the $\mathsf{commit}()_t$ function (Line 97). 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 $\mathsf{try\text{-}extend}()$ call returns at Line 96. Since Line 96 is before the beginning of the for-loop, $\tau_2 < \tau_1$ and the result follows.
2. Second, we show that if $t \in aborted(\mathcal{H}') \setminus aborted(\mathcal{H})$ 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 $\mathcal{H}' \in complete(\mathcal{H})$ such that if $t \in aborted(\mathcal{H}')$ then t is invisible. □

To show Property (2) of Definition 5.2, we determine a serialization point for each 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 time the last $\ell_1 \leftarrow x.tlk$ of the loop occurs (Line 39).
2. write operation π: its serialization point $ser(\pi)$ is the time its compare-and-swap occurs (Line 75).

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 5.5. *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 $permanent(\mathcal{H}')$. 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 $\pi_2 \not\prec_d \pi_1$.*

Proof. We show by contradiction that we cannot have $ser(\pi_1) < ser(\pi_2)$ and $\pi_2 \prec_d \pi_1$. Assume by absurd that $\pi_2 \prec_d \pi_1$, 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 $\pi_2 \prec_d \pi_1$ implies that $ser(t_1)$ is after t_2 releases its lock on x (Line 43 or 76) otherwise t_1 would have aborted (Line 49 if π_1 is a read or Line 70 if π_1 is a write) contradicting $t_1 \in permanent(\mathcal{H}')$.
- 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 $\pi_1 \prec_d \pi_2$, or π_2 would detect that another transaction owns the lock, so t_2 would abort (Line 49) contradicting that $t_2 \in permanent(\mathcal{H}')$.

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

INVARIANT 5.6. $clock \geq lb$.

Proof. Initially, $clock = x.tlk.time = 0$, for any 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 $\prec_d$.

COROLLARY 5.7. *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 $permanent(\mathcal{H}')$. 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 distant transactional operations. For the proof, let a_π denote the state of a when it is set in operation π for the first time, or when π starts if it is never set by π.

LEMMA 5.8. *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 $permanent(\mathcal{H}')$. 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. 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 $\pi_1(a) \prec_d r(a) \prec \pi'_1$, but $r(a)$ as a read would have aborted (because $x.tlk.owner_r = t_1$, Line 49), or would loop while $x.tlk.owner_r = t_1$ (Line 43), 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 $\pi_1(a) \prec_d w(a) \prec \pi'_1$.

There are two cases to consider whether π'_1 is a write. First, assume that π'_1 is a write. By Invariant 5.6 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 (contradicting the assumption). Second, if π'_1 is a read, there is a write $w'(a')$ such that $\pi_2 \prec w'(a') \prec_d \pi'_1(a')$. Hence $x.tlk.time_{\pi'_1} > clock_w \geq clock_{\pi_1}$, and by Invariant 5.6 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 $a'.tlk.owner \notin \{\bot, t_1\}$ 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 and the result follows. □

COROLLARY 5.9. *Let $\mathcal{H}$ be any history of $\mathcal{E}$-STM and let $\mathcal{H}' = sub\text{-}complete(f_{\mathcal{C}}(\mathcal{H}))$. History $permanent(\mathcal{H}')$ 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 5.8, 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. □

6. Elastic Transactions: Evaluations

We evaluate here the performance of elastic transactions on four data structure applications.

6.1 Qualitative Assessment

$\mathcal{E}$-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.

6.1.1 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 depicts the entire program that uses $\mathcal{E}$-STM plus a lock-free harris-ll-find function for comparison purpose. This function is at the core of the lock-free linked list of Harris [6]. It is pretty 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 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.

6.1.2 Extensibility

$\mathcal{E}$-STM combines elastic transactions with normal transactions. As a result the code that uses $\mathcal{E}$-STM is easily extensible. To illustrate this, we implemented the hash table example presented in Section 1 extended 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 22 and 16) execute in the normal mode. 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 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 im-

Algorithm 2 Linked list implementation built on $\mathcal{E}$-STM (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)

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)∨(t.key < i)
59:     next = t
60:     // 2. check nodes are adjacent
61:     if c_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, c_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
```

plemented using normal transactions, otherwise, a concurrent move may lead to an inconsistent state. Another issue, pointed out in [17], 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 programmer can use a normal transaction to encapsulate each conditional insertion. All 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. Extending a lock-free hash table further to provide a resize operation 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 [16].

6.2 Quantitative Assessment

In this section, we present the performance results obtained when running $\mathcal{E}$-STM, locks, regular STM transactions, and lock-free algorithms on our data structure applications. We also compare the obtained results to the non-thread-safe code executed sequentially.

6.2.1 Testbed data structures

A linked list is an appealing data structure: it is simple yet flexible, and it provides insert and remove operations that only affect a localized part of the data structure, as opposed to arrays. The pseudocode of our $\mathcal{E}$-STM-based linked list have been given in Algorithm 2 and its proof of consistency has been deferred to Appendix A.

Skip lists are known to provide logarithmic search time complexity while being simpler to program than balanced trees. For instance, insert and remove in an

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 empty
8:      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
```

AVL tree induce complex re-balancing operations. Red-black trees are binary trees where each node is colored (red or black) and where leaves have no keys. The coloring is such that the root and the leaves are black, all paths from the root to leaves contain the same number of black nodes, and all children of a red node are black. As a result, a red-black tree remains balanced enough to provide logarithmic operation complexity and relaxes the strict balancing of AVLs. Red-black trees have been largely used as micro-benchmarks in concurrent programming and especially for the evaluation of numerous STMs [1, 2, 11]. Hash tables provide constant access complexity. Our corresponding pseudocode is given in Algorithm 3. Due to lack of space, the pseudocode of the skip list and the red-black tree has been omitted here. Our implementations consist simply of a non-thread-safe code simply enriched with elastic transaction calls (begin(elastic), begin(normal), read, write, commit) as for the linked list.

6.2.2 Lock-based, lock-free, and STM alternatives

Our lock-based linked list implements the lazy algorithm which is more efficient than fine-grained alternatives [8] (e.g., lock-coupling or hand-over-hand locking); our lock-based version of the hash table builds upon it. Using a similar technique, our lock-based skip list is a C version of the optimistic algorithm of [10], the most recent lock-based skip list algorithm we know of.

We consider the Harris-Michael implementation of the lock-free linked list as presented in [6] and the hash table version that is based on it [12]. Unfortunately, this hash table algorithm cannot support complex operations like sum and move, so we ran two distinct sets of experiment on hash tables. For our needs (x86-64), we re-implemented the Fraser lock-free skip list that uses the low-order bit marking technique of Harris-Michael's linked list [3]. Only miss the lock-free and lock-based versions of the red-black tree.

We chose TinySTM [2] v0.9.8 (in its default mode) to compare regular transactions with $\mathcal{E}$-STM because TinySTM is, as far as we know, the most efficient STM on micro-benchmarks such as linked lists and red-black trees [2, 5].

6.2.3 Experimental results

We tested the performance of these data structure implementations on a 4 quad-core AMD Opteron machine, i.e., including 16 cores in total. We ran the aforementioned algorithms using 16 threads and their non-thread-safe counterpart using a single thread, all written in C. Each point on the graphs represents the average value obtained from 5 runs of 10 seconds each.

Figure 1 depicts the results obtained when running 90% of search operations, 5% of insert, and 5% of remove, except in the 4th row. We varied the initial size of the sets, and we insert and remove the same integers to keep the set size roughly constant in each experiment. As we do not know of any lock-free implementation of a move, we ran two sets of hash table experiments: (i) experiments of row 4 include 10% of move, 10% of sum with 80% of search while the load factor is fixed to 5 (the load factor corresponds to the mean ratio of the number of nodes over the number

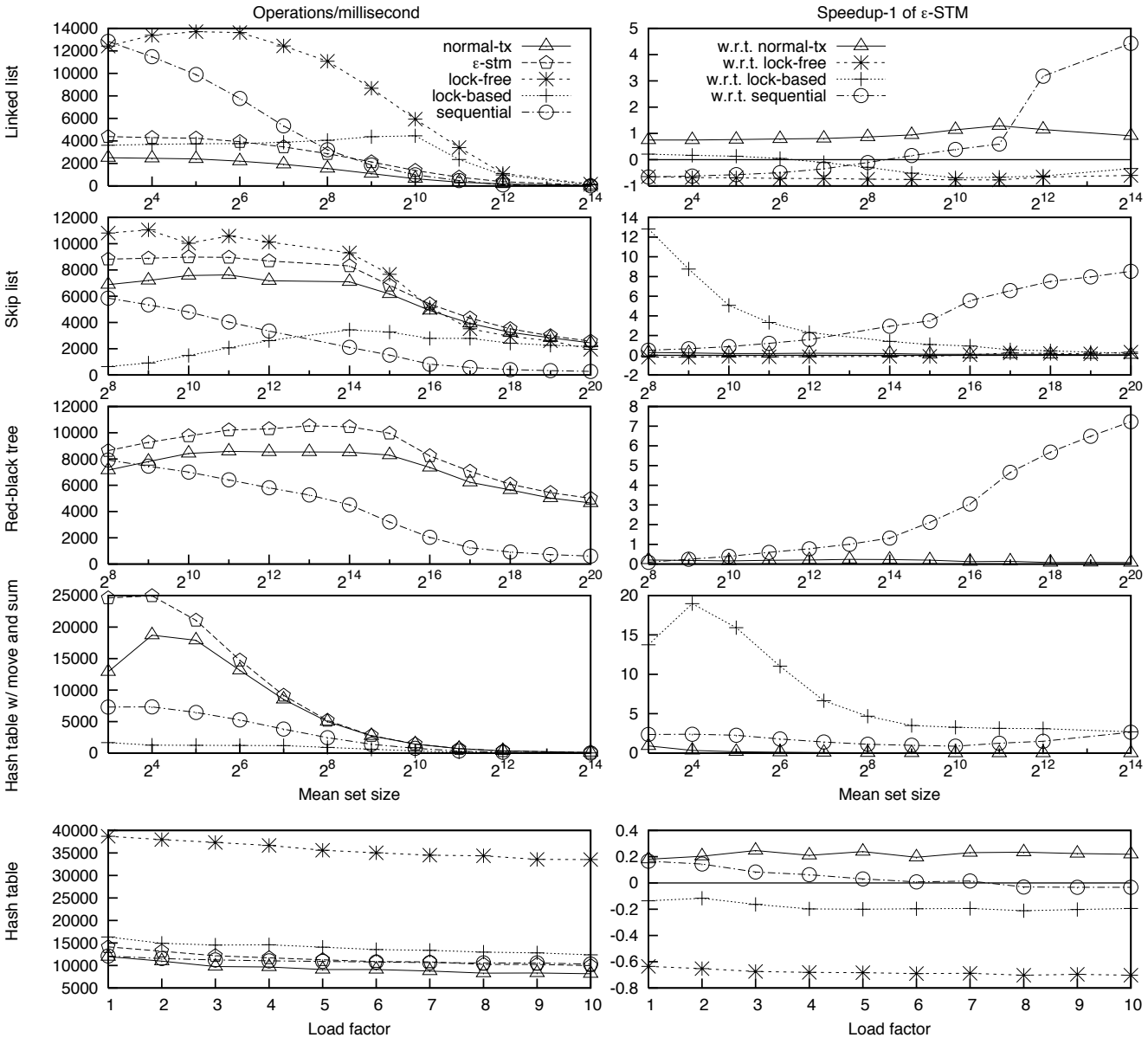

Figure 1. Performance results when running integer set operations. *Left side:* operation throughput of all synchronization techniques (with 16 threads) compared to non-thread-safe throughput (with a single thread) on linked list (1st row), skip list (2nd row), red-black tree (3rd row), and hash table (5th row) with 5% insert, 5% remove, and 90% search operations; and hash table with 10% move, 10% sum, and 80% search (4th row). *Right side:* resulting improvement i (or speedup-1) of throughput u of $\mathcal{E}$-STM over the throughput v of each other techniques (where $i = \frac{u}{v} - 1$), the solid lines representing $f(x) = 0$ is the limit above which $\mathcal{E}$-STM presents better performance than other techniques.

of buckets); (ii) experiments of row 5 include only the three basic integer set operations in the same proportion as above (5-5-90) while the load factor varies and the hash table contains 256 buckets. We chose a reasonably low amount of updating operations (10%) in these experiments because transactions are already known to perform well in highly-contended environment.

The results show that $\mathcal{E}$-STM always improves the performance of the regular STM and can speed up the regular STM by more than 2.3x in some circumstances, with an averaged speedup of 1.36x.[1] We averaged the speedups over linked list, skip list, and hash table using only insert, remove, and search operations for which we had values regarding each implementation. We observe that $\mathcal{E}$-STM performance competes with lock-based performance: the improvement factor of $\mathcal{E}$-STM over lock-based implementations is 1.89x on average. Our implementation is on average slower than lock-free implementations (averaged slowdown of 1.47x) yet significantly simpler. It is noteworthy that, unlike all other implementations, the Harris lock-free algorithms have been implemented "as is" without any memory re-allocation, which may impact the performance. In addition, the $\mathcal{E}$-STM average speedup over the sequential performance is of 2.8x. Finally, $\mathcal{E}$-STM is on average 1.2x faster than regular STM transactions and up to 8.2x faster than sequential, on red-black trees.

More particularly, the fourth row indicates a speedup of $\mathcal{E}$-STM over regular STM transactions that goes up to 1.9x, over the sequential version that goes up to 3.6x, and over lock-based that goes up to 20x, outlining the inherent efficiency obtained when combining normal and elastic transactions like $\mathcal{E}$-STM does.

7. 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 [13] 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 [14].

Transactional boosting [9] 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 [7] 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 [11] 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 [17]: 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.

[1] The improvement (or speedup-1) represented in the graphs is $i = \frac{u}{v} - 1$ and corresponds to the improvement of u over v so that positive values indicate gains and negative values indicate losses, while in the text we refer to the speedup $s = \frac{u}{v}$ as the multiplying factor between u and v.

8. 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 four well-known data structures have confirmed that elastic transactions perform significantly better than traditional transactions and lock-based alternatives, and is much simpler to program with than existing lock-free solutions.

We have evaluated the performance of our approach with data structures where operations parsing the structure are loosely dependent on the subsequent operations modifying a region of it. 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.

Acknowledgments

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

References

[1] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing*, September 2006.

[2] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[3] Keir Fraser. *Practical lock freedom*. PhD thesis, University of Cambridge, September 2003.

[4] Rachid Guerraoui and Michał Kapałka. On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–184, February 2008.

[5] Derin Harmanci, Pascal Felber, Vincent Gramoli, and Christof Fetzer. TMunit: Testing software transactional memories. In *TRANSACT '09: 4th ACM SIGPLAN Workshop on Transactional Computing*, February 2009.

[6] Tim Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314, London, UK, 2001. Springer-Verlag.

[7] Tim Harris and Srdjan Stipić. Abstract nested transactions. In *The 2nd ACM SIGPLAN Workshop on Transactional Computing*, 2007.

[8] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. In *OPODIS '05: Proceedings of the 9th International Conference on Principles of Distributed Systems*, volume 3974 of *LNCS*, pages 3–16. Springer, December 2005.

[9] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[10] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *SIROCCO '07: Proceedings of the 14th Colloquium on Structural Information and Communication Complexity*, volume 4474 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2007.

[11] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[12] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82, New York, NY, USA, 2002. ACM.

[13] J. Eliot B. Moss. Open nested transactions: Semantics and support. In *Workshop on Memory Performance Issues (WMPI 2006)*, February 2006.

[14] Yang Ni, Vijay Menon, Ali-Reza Abd-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2007.

[15] Torvald Riegel, Pascal Felber, and Christof Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing*, pages 284–

298, September 2006.

[16] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, 2006.

[17] Travis Skare and Christos Kozyrakis. Early release: Friend or foe? In *Workshop on Transactional Memory Workloads*, June 2006.

[18] John D. Valois. Lock-free linked lists using compare-and-swap. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222, New York, NY, USA, 1995. ACM.

[19] William E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Trans. Program. Lang. Syst.*, 11(2):249–283, 1989.

A. Proof of Correctness of the Linked list

Here, we prove that our linked list algorithm implements an integer set. The proof relies on elastic opacity (Definition 3.4) and the pseudocode of Algorithm 2. First-of-all, we recall the semantics of the integer set. Given a set S:

- $\mathsf{search}(i)$ operation returns true if the node i is present in S, false otherwise;
- $\mathsf{insert}(i)$ operation augments the set S with the node i if i is not in S, S is unchanged otherwise;
- $\mathsf{remove}(i)$ 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. Next, we state preliminary definitions.

DEFINITION A.1. *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 A.2. *Integer i is* in the set *if and only if there is a reachable node n such that $n.key = i$.*

First lemma gives an important result for proving the correctness of operation $\mathsf{search}(*)$ and relies on the definition of consistent cut (Definition 3.3).

LEMMA A.3. *Operation* $\mathsf{find}(i)$ *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 in time during 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 $next$ parses 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 $next \leftarrow \mathsf{read}(curr.next)$. 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 in time during the corresponding execution of find. Observe that a find operation is always called as part of an elastic transaction. As a consequence of Definition 3.3, there is no two write operations on $curr$ and $next$ between $\mathsf{read}(curr)$ and $\mathsf{read}(next)$ 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 $\mathsf{read}(curr)$. 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 $\mathsf{read}(next)$. Note that both cases are true for the case $next = curr$, because none of these writes can occur between the reads by Definition 3.3. It follows that each of the two nodes are consecutive at some time during the execution of this find.

The result follows. □

We know by elastic-opacity (Definition 3.4) that no aborting transactions modify the state of the system hence if a transaction aborts, we are ensured that safety is preserved (only committing transactions can violate safety). The major difficulties when implementing integer set stem from the concurrent updates problem [18]. For instance, when attempting to insert two distinct values concurrently between nodes a and b, particular ef-

forts are required to prevent one of these insert from hiding the result of the other. Another critical example is when one inserts a value between a and b while another is concurrently deleting a, special care is needed to ensure that the insert will be effective. Finally deleting two successive nodes a and b concurrently may easily result in the sole deletion of node a. Next, we show that no update can annihilate the effect of another update on our linked list implementation.

LEMMA A.4. *Let π be the* write$(curr.next, *)$ *of an* insert *or a* read$(curr.next)$ *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 normal 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., $curr.tlk.owner \neq \perp$ 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, $curr.tlk.owner \neq t$ and this transaction aborts as indicated Lines 49 and 70 of Algorithm 1. In the latter case, it cannot extend and aborts because the transaction type is elastic (Line 73).

Consequently, either $curr.next = next$ or π aborts its enclosing transaction (Line 70 or Line 73 of Algorithm 1). □

Next, we show that the time at which insert or remove operation acquires its lock during its write operation serves as a serialization point for the operation.

LEMMA A.5. *Let Π_1 and Π_2 be two update operations on the linked list. Then among them, the first to acquire the lock in its* write *appears to have executed before the other.*

Proof. There are only two types of operations to consider here: insert$(*)$ and remove$(*)$. It is easy to see that if these operations do not conflict then they can be arbitrarily ordered and if these operations are not concurrent then the first to execute can not see the result of the second one.

Consider the cases where they conflict, hence either (i) insert$(*)$ inserts a node between nodes a and b and remove(a) deletes a or deletes b, or (ii) two insert(a) insert at node a, or (iii) two remove(a) delete node a, or finally, (iv) remove(a) deletes node a and remove(b) deletes node b.

If insert acquires the lock first before remove acquires it then insert acquires the lock on a either before the read of remove occurs, or between the read and write of remove. In the former case, the read detects it and aborts as indicated in Lemma A.4. In the latter case, by the elastic-opacity of transactions (Theorem 5.1), one must abort (i.e, when the transaction of the remove commits, it sees that $clock$ has been increased leading to an abort). Hence insert appears as occurring before the remove if it acquires the lock first.

In the other cases, the operations conflict on write$(a.next, *)$ so that one acquires the lock before the other, modifying $a.next$ and the other aborts as indicated by Lemma A.4. Again the first to acquire the lock executes while the other aborts and restarts later. □

LEMMA A.6. *A* search(i) *operation appears as if it is executed atomically at the time the* read$(next)$ *of its* find *occurs such that $curr.next.key \geq i$.*

Proof. If a search occurs immediately after an update acquires a lock on $curr$, then search 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, so as the enclosing search. If however, it consults $curr$ before any operation acquires a lock on it, then it will be ordered before. Finally, by Lemma A.3, $curr.next.key \geq i$. □

INVARIANT A.7. *For any node $curr$ in the linked list, if $curr.next = next$ then $curr.key < next.key$.*

Proof. By Lemma A.3, 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. □

THEOREM A.8. *The linked list implementation presented in Algorithm 2 is correct.*

Proof. By Invariant A.7 the linked list is a sorted linked list with no duplicates. By Lemma A.5, an update operation admits a serialization point at which it appears to be atomic. Hence the semantics of remove and insert are satisfied. Finally, by Lemma A.3 the search operation respects its semantics and by Lemma A.6, search is linearizable. □