A Scalable, Correct Time-Stamped Stack

Mike Dodds
University of York
mike.dodds@york.ac.uk

Andreas Haas
University of Salzburg
ahaas@cs.uni-salzburg.at

Christoph M. Kirsch
University of Salzburg
ck@cs.uni-salzburg.at

Abstract

Concurrent data-structures, such as stacks, queues, and deques, often implicitly enforce a total order over elements in their underlying memory layout. However, much of this order is unnecessary: linearizability only requires that elements are ordered if the insert methods ran in sequence. We propose a new approach which uses timestamping to avoid unnecessary ordering. Pairs of elements can be left unordered (represented by unordered timestamps) if their associated insert operations ran concurrently, and order imposed as necessary by the eventual remove operations.

We realise our approach in a new non-blocking data-structure, the TS (timestamped) stack. In experiments on x86, the TS stack outperforms and outscales all its competitors – for example, it outperforms the elimination-backoff stack by factor of two. In our approach, more concurrency translates into less ordering, giving less-contended removal and thus higher performance and scalability. Despite this, the TS stack is linearizable with respect to stack semantics.

The weak internal ordering in the TS stack presents a challenge when establishing linearizability: standard techniques such as linearization points work well when there exists a total internal order. We present a new stack theorem, mechanised in Isabelle, which characterises the orderings sufficient to establish stack semantics. By applying our stack theorem, we show that the TS stack is indeed correct. Our theorem constitutes a new, generic proof technique for concurrent stacks, and it paves the way for future weakly-ordered data-structure designs.

1. Introduction

This paper presents a new approach to building ordered concurrent data-structures, a realisation of this approach in a high-performance stack, and a new proof technique required to show that this algorithm is linearizable with respect to sequential stack semantics.

Our general approach is aimed at pool-like data-structures, e.g. stacks, queues and deques. The key idea is for insertion to attach timestamps to elements, and for these timestamps to determine the order in which elements should be removed. This idea can be instantiated as a stack by removing the element with the latest timestamp, or as a queue by removing the element with the earliest timestamp. Both kinds of operation can be combined to give a deque. For most of this paper we will focus on the TS (timestamped) stack, given in high-level pseudocode in Figure 1 (the TS queue / deque variants are discussed briefly in §8).

One might assume that generating a timestamp and adding an element to the data-structure has to be done together, atomically. This intuition is wrong: linearizability allows concurrent operations to take effect in any order within method boundaries – only sequential operations have to keep their order [13]. Therefore we need only order inserted elements if the methods inserting them execute sequentially. We exploit this fact by splitting timestamp generation from element insertion, and by allowing unordered timestamps. Two elements may be timestamped in a different order than they were inserted, or they may be unordered, but *only* when the surrounding methods overlap, meaning the elements could legitimately be removed in either order. The only constraint is that elements of sequentially executed insert operations receive ordered timestamps.

By separating timestamp creation from adding the element to the data-structure, our insert method avoids two expensive synchronisation patterns – atomic-write-after-read (AWAR) and read-after-write (RAW). We take these patterns from [2], and refer to them collectively as *strong synchronisation*. Timestamping can be done by a stuttering counter or a hardware instruction like the x86 `RDTSCP` instruction, neither of which require strong synchronization. Timestamped elements can be stored in per-thread single-producer multiple-consumer buffers, implemented as singly-linked lists. Such buffers also do not require strong synchronization in the insert operation. Thus stack insertion avoids strong synchronization, radically reducing its cost.

The lack of synchronization in the insert operation comes at the cost of contention in the remove operation. Indeed, [2] proves that stacks, queues, and deques cannot be implemented without some strong synchronisation. Perhaps surprisingly, this problem can be mitigated by reducing the ordering between timestamps: intuitively, less ordering results in more opportunities for parallel removal, and thus less contention. To weaken the element order, we associate elements with *intervals* represented by pairs of timestamps. Interval timestamps allow our TS stack to achieve performance and scalability better than state-of-the-art concurrent stacks. For example, we believe the elimination-backoff stack is the current world leader; in our experiments on x86, the TS stack outperforms it by a factor of two.

Establishing correctness for the TS stack presents a challenge for existing proof methods. The standard approach would be to locate *linearization points*, syntactic points in the code which fix the order that methods take effect. This simply does not work for timestamped structures, because the order of overlapping `push` operations is fixed by the order of future `pop` operations. In the absence of `pop` operations, elements can remain entirely unordered. We solve this with a new theorem (mechanised in the Isabelle proof assistant) which builds on Henzinger et al.'s aspect oriented technique [11]. Rather than a total order, we need only generate an order from `push` to `pop` operations, and vice versa, which avoids certain violations. This partial order *can* be generated from syntactic points in the TS stack code, allow-

Listing 1: TS stack algorithm. Implementations for TS buffer operations are given in Listing 2 on page 8.

```
1  TS_Stack {
2    TS_Buffer buffer;
3
4    void push(Element element) {
5      item = buffer.ins(element);
6      ts = buffer.newTimestamp();
7      buffer.setTimestamp(item,ts);
8    }
9
10   Element pop() {
11     ts = buffer.getStart();
12     do {
13       item = buffer.tryRem(ts);
14     } while (!item.isValid());
15     if (item.isEmpty())
16       return EMPTY;
17     else
18       return item.element;
19   }
20 }
```

ing us to show that it is correct. By generalising away from linearization points, our theorem paves the way for future correct, weakly-ordered concurrent data-structures.

Contribution. To summarise, our contributions are:

- A new class of data-structure based on timestamping, realised as a stack, queue, and deque.
- A new optimisation strategy, interval timestamping, which exploits the weak ordering permitted by timestamped data-structures.
- A new proof technique for establishing the linearizability of concurrent stacks, and a mechanisation of the core theorem in Isabelle.
- A detailed application of this proof technique to show that the TS stack is linearizable with respect to its sequential specification.
- An experimental evaluation showing our TS stack outperforms the best existing concurrent stack.

Artifacts. We have produced two research artifacts:

- The TS stack itself, implemented in C, along with queue and deque variants, and benchmark code used to test it.
- The Isabelle mechanisation of our stack theorem.

Both artifacts are included with the supplementary material.

Paper structure. §2 describes the key ideas behind our approach. §3 surveys the related work on concurrent data-structures. In §4 we describe our aspect-oriented proof strategy, while in §5 we use it to establish that the TS stack is correct. In §6 we give more details of our algorithm implementation. In §7 we discuss our experiments, both with different implementation choices, and with respect to other concurrent data-structures. §8 discusses TS queue and deque variants. §9 concludes. Selected proofs are given in Appendix A.

Longer proofs and other auxiliary material are included in supplementary appendices: Appendix B discusses the Isabelle proof of our core stack theorem. Appendix C discusses why the core theorem requires an insert-remove order on methods. Appendix D describes our single-producer buffer algorithm. Appendix E gives further details about our TS stack correctness proof. Appendix F gives a proof of correctness for the TS buffer algorithm.

2. Key Ideas

Algorithm structure. Listing 1 shows the high-level structure of our TS stack. To simplify the presentation, we define the TS stack using a lower-level structure called a *timestamped buffer* (TS buffer). Intuitively, a TS buffer is a map from identifiers to values, optionally associated with timestamps. It supports the following operations:

- `ins` – add an element to the buffer without attaching a timestamp, and return a reference to the item.
- `newTimestamp` – generate a fresh timestamp used for labelling an item in the buffer.
- `setTimestamp` – attach a timestamp to a given item.
- `getStart` – generate a timestamp for removing an item.
- `tryRem` – try to remove or eliminate a maximal item – the timestamp argument is used to decide which.

Our TS buffer implementation is sketched below, and discussed in detail in §6.2.

With the TS buffer operations defined, the structure of Listing 1 should be clear. To push an element, the algorithm inserts an un-timestamped element into the buffer (line 5), generates a fresh timestamp (line 6), and sets the new element's timestamp (line 7). To pop an element, the algorithm repeatedly tries to remove a youngest element (line 13) until it succeeds or discovers the buffer is empty. The start time generated on line 11 is used to determine which elements can be eliminated (see 'optimisations' below).

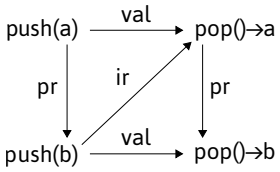
Algorithm correctness. To prove that a stack is linearizable, we need to show that for any execution there exists a total linearization order satisfying the stack specification. Intuitively, the TS stack is linearizable because any two `push` operations that run sequentially receive ordered timestamps, and are therefore removed in order. Elements arising from concurrent `push` operations can be ordered any way, so this ensures that elements are removed in a linearizable order. However, proving the existence of a linearization order directly is challenging, because the order between parallel `push` operations can be fixed by the order on later `pop` operations, while the order between parallel `pop` operations can likewise be fixed by earlier `push`es.

Instead, we present a new theorem for verifying stacks. Rather than generate a total linearization order directly, this theorem only requires an order covering all pairs of a `push` and a `pop` – we call this order ir, for 'insert-remove'. Thus, the theorem relieves us of the need to resolve the problematic order between parallel `push` or parallel `pop` operations.

The theorem also assumes two other orders on methods: *precedence*, pr, which relates methods that run in sequence; and val, which relates a `push` to the `pop` receiving the associated value. Both of these orders can easily be extracted from executions of the algorithm. Informally, the stack theorem says:

> *If for some trace $\mathcal{T}$ of the algorithm there exists an* ir *order which, along with* pr *and* val*, satisfy the stack conditions, then $\mathcal{T}$ is linearizable with respect to sequential stack semantics.*

The stack conditions express particular features of stack behaviour. For example, the following combination of orders is forbidden.



Avoiding this shape enforces LIFO ordering. If push(a) and push(b) are sequentially ordered, and push(b) is related to pop()→a in ir, then the two pops cannot also be sequentially ordered – this would correspond to FIFO behaviour.

Unlike a total linearization order, the insert-remove order ir *can* be constructed directly from a TS stack execution. We use a modified version of the linearization point method: rather than a single linearization point, we identify multiple points, each of which generates an order over some of the method calls. These sub-orders are:

- vis (for 'visibility') – the order from `ins` to `getStart`. Informally, if a `push`(a) and `pop` are ordered by vis, the value a inserted by `push` is necessarily visible to the `pop`.
- rr (for 'remove-remove') – the order between successful `tryRem` operations. If two operations `pop()`→a and `pop()`→b are ordered in rr, elements a and b are removed in order from the underlying TS buffer.

The order ir is built by taking vis as a core, and using rr to correct cases that contradict LIFO order.

Implementation approach. The TS buffer is the data-structure that underlies the TS stack (see Section 6.2, Listing 2 for pseudocode). We implement the TS buffer as a collection of single-producer multiple-consumer buffers (SP buffers). Each thread is associated with a SP buffer into which it inserts, and `tryRem` searches through all these buffers to find a maximal element. Elements are removed using an atomic compare-and-swap (i.e. an AWAR) to write a '`taken`' flag; if the `CAS` fails, the `tryRem` operation fails. This contention means that our algorithm is lock-free but not wait-free; a thread can be forced to wait indefinitely by contending threads.

Accessing the heads of multiple per-thread buffers imposes a cost through cache contention. However, our experiments show that this can be less expensive than contention on a single location with an opportunistic compare-and-swap approach. In our experiments, we mitigate contention and thereby improve performance by introducing a small `NOP` delay to the `pop` search loop. However, even without this optimisation, the TS stack outperforms the EB stack by a factor of two.

We have experimented with various implementations for timestamping itself. Most straightforwardly, we can use a strongly-synchronised fetch-and-increment counter. We can avoid strong synchronisation by using a vector of thread-local counters, meaning the counter may stutter (many threads get the same timestamp). We can also use a hardware timestamping operation – for example the `RDTSCP` instruction which is available on all modern x86 hardware. In the past, such instructions have largely been used for analysis and logging. Our benchmarks show that hardware timestamping provides the best `push` performance. However, the picture is more complicated in the presence of optimisation.

Optimisations. Timestamping enables several optimisations of the TS stack, most importantly elimination (a standard strategy in the literature), and interval timestamping (contribution of this paper).

In a stack, a concurrent `push` and `pop` can always soundly eliminate each other, irrespective of the state of the stack [9]. Therefore a thread can remove any concurrently-inserted element, not just the stack top. Unlike [9], our mechanism for detecting elimination exploits the existence of timestamps. We read the current timestamp at the start of a `pop`; any element with a later timestamp has been pushed during the current `pop`, and can be eliminated.

Surprisingly, it is not optimal to insert elements as quickly as possible. The reason is that removal is quicker when there are many unordered maximal elements, reducing contention and avoiding failed CASes. To exploit this, we define timestamps as *intervals*, represented by a pair of start and end times. Overlapping interval timestamps are considered unordered, and thus there can be *many* top elements in the stack. To implement this, `newTimestamp()` pauses for a predetermined interval after generating a start timestamp, then generates an end timestamp.

Pausing allows us to trade off the performance of push and pop: an increasing delay in insertion can reduce the number of retries in pop (for evidence see §7.2). Though pausing may make push slower than a single AWAR instruction, our experiments suggest what is expensive is not individual instructions, but rather contention that causes many instructions to be repeated. Our experiments show that by weakening the order of stored elements, interval timestamping can substantially increase overall throughput and decrease the latency of pops.

Similarly, although interval timestamping increases the non-determinism of removal (i.e. the variance in the order in which pushed elements are popped), this need not translate into greater overall non-determinism compared to other high-performance stacks. A major source of non-determinism in existing concurrent data-structures is in fact contention [7]. While interval timestamping increases the potential for non-determinism in one respect, it decreases it in another.

Performance vs. Elimination-Backoff stack. To the best of our knowledge the Elimination-Backoff (EB) stack [9] is the fastest stack previously proposed. In our experiments (§7.1) the TS stack configured with elimination and interval timestamping outperforms the EB stack by a factor of two. Several design decisions contribute to this success. The lack of insert-contention and mitigation of contention in the remove makes our stack fast even without elimination. Also, timestamping allows us to integrate elimination into normal stack code, rather than in a separate back-off code.

3. Related Work

Timestamping. Our timestamping approach is inspired by Attiya et al.'s Laws of Order paper [2], which proves that any linearizable stack, queue, or deque necessarily uses the RAW or AWAR patterns in its remove operation. While attempting to extend this result to insert operations, we were surprised to discover a counter-example: the TS stack. We believe the Basket Queue [14] was the first algorithm to exploit the fact that enqueues need not take effect in order of their atomic operations, although unlike the TS stack it does not avoid strong synchronisation when inserting.

Gorelik and Hendler use timestamping in their AFC queue [6]. As in our stack, enqueued items are timestamped and stored in single-producer buffers. Aside from the obvious difference in kind, our TS stack differs in several respects. The AFC dequeue uses flat-combining-style consolidation – that is, a combiner thread merges timestamps into a total order. As a result, the AFC queue is blocking. The TS stack avoids enforcing an internal total order, and instead allows non-blocking parallel removal. Removal in the AFC queue depends on the expensive consolidation process, and as a result their producer-consumer benchmark shows remove performance significantly worse than other flat-combining queues. Interval timestamping lets the TS stack trade insertion and removal cost, avoiding this problem. Timestamps in the AFC queue are Lamport clocks [17], not hardware-generated intervals. (We also experiment with Lamport clocks – see TS-stutter in §6.1). Finally, AFC queue elements are timestamped *before* being inserted – in the TS stack, this is reversed. This seemingly trivial difference enables timestamp-based elimination, which is important to the TS stack's performance.

The LCRQ queue [1] and the SP queue [10] both index elements using an atomic counter. However, dequeue operations do not look for *one of* the youngest elements as in our TS stack, but rather for the element with the enqueue index that matches the dequeue index *exactly*. Both approaches fall back to a slow path when the dequeue counter becomes higher than the enqueue counter. In contrast to indices, timestamps in the TS stack need not be unique or even ordered, and the performance of the TS stack does not depend on a fast path and a slow path, but only on the number of elements which share the same timestamp.

Our use of the x86 `RDTSCP` instruction to generate hardware timestamps is inspired by work on testing FIFO queues [7]. There the `RDTSC` instruction is used to determine the order of operation invocations. (Note the distinction between the synchronised `RDTSCP` and unsynchronised `RDTSC`). `RDTSCP` has since been used in the design of an STM by Ruan et al. [23], who investigate the instruction's multi-processor synchronisation behaviour.

Correctness. Our stack theorem lets us prove that the TS stack is linearizable with respect to sequential stack semantics. This theorem builds on Henzinger et al. who have a similar theorem for queues [11]. Their theorem is defined (almost) entirely in terms of the sequential order on methods – what we call precedence, pr. That is, they need not generate a linearization order. In contrast, our stack theorem requires a partial order between inserts and removes. We suspect it is impossible to define such a theorem for stacks without an auxiliary insert-remove order. Intuitively, push-pop pairs contend for the same 'end' of the abstract stack, and thus are more closely dependent than enqueue-dequeue pairs (see supplementary Appendix C for further discussion).

A stack must respect several non-LIFO correctness properties: elements should not be lost or duplicated, and `pop` should correctly report when the stack is empty. Henzinger et al. build these properties into their theorem, making it more complex and arguably harder to use. Furthermore, each `dequeue` that returns EMPTY requires a partition 'before' and 'after' the operation, effectively reintroducing a partial linearization order. However, these correctness properties are orthogonal to LIFO ordering, and so we simply require that the algorithm also respects set semantics.

Implementation features. Our TS stack implementation reuses concepts from several previous data-structures.

Storing elements in multiple partial data-structures is used in the distributed queue [8], where insert and remove operations are distributed between partial queues using a load balancer. One can view the SP buffers of the TS buffer as partial queues and the TS buffer itself as the load balancer. The TS buffer emptiness check also originates from the distributed queues. However, the TS stack leverages the performance of distributed queues while preserving sequential stack semantics.

Elimination originates in the elimination-backoff stack [9]. However, in the TS stack, elimination works by comparing timestamps rather than by accessing a collision array. As a result, in the TS stack a `pop` which eliminates a concurrent `push` is faster than a normal uncontended `pop`. In the elimination-backoff stack such an eliminating `pop` is slower, as synchronization on the collision array requires at least three successful `CAS` operations instead of just one.

4. Correctness Theorem for Stacks

Linearizability [13] is the standard correctness condition for concurrent algorithms.[1] It ensures that every behaviour observed by an algorithm's calling context could also have been produced by a sequential (i.e. atomic) version of the same algorithm. We call the ideal sequential version of the algorithm the *specification*, e.g. below we define a sequential stack specification.

Interactions between the algorithm and calling context in a given execution are expressed as a *history*.

Definition 1. *A* history $\mathcal{H}$ *is a tuple* $\langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ *where* $\mathcal{A}$ *is a finite set of operations (for example,* `push(5)`*), and* $\mathsf{pr}, \mathsf{val} \subseteq \mathcal{A} \times \mathcal{A}$ *are the* precedence *and* value *relations, respectively. A history is* sequential *if* pr *is total.*

A history is extracted from a *trace*, $\mathcal{T}$, the interleaved sequence of events that took place during an execution of the algorithm. To extract the history, we first generate the set $\mathcal{A}$ of executed operations in the trace (as is standard in linearizability, assume that all calls have corresponding returns). A pair (x, y) is in pr if the return event of operation x is ordered before the call event of y in $\mathcal{T}$. A pair (x, y) is in val if x is an insert, y a remove, and the value inserted by x was removed by y. Note that we assume that values are unique.

Linearizability requires that algorithms only interact with their calling context through invocation and response events. Therefore, a history captures all interactions between algorithm and context. We thus define a datastructure specification as just a set of histories (e.g. STACK is the set of histories produced by an ideal sequential stack). Linearizability is defined by relating implementation and specification histories.

Definition 2. *A history* $\langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ *is* linearizable *with respect to some specification* $\mathcal{S}$ *if there exists a* linearization order pr^T *such that* $\mathsf{pr} \subseteq \mathsf{pr}^T$*, and* $\langle \mathcal{A}, \mathsf{pr}^T, \mathsf{val} \rangle \in \mathcal{S}$.

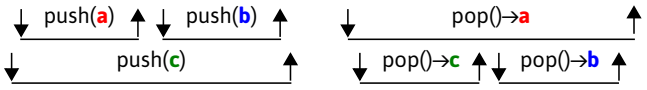
[1] Our formulation of linearizability differs from the classic one [13]. Rather than have a history record the total order on invocations and responses, we convert this information into a partial order pr. Likewise, linearizability between histories is defined by inclusion on partial orders, rather than by reordering invocation and response events. This approach, taken from [4], is convenient for us because our stack theorem is defined by constraints on partial orders. However, the two formulations are equivalent.

An implementation C is linearizable with respect to $\mathcal{S}$ if any history $\mathcal{H}$ arising from the algorithm is linearizable with respect to $\mathcal{S}$.

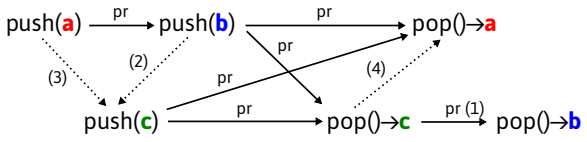
The problem with linearization points. Proving that a concurrent algorithm is linearizable with respect to a sequential specification amounts to showing that, for every possible execution, there exists a total linearization order. The standard strategy is to identify *linearization points* in the algorithm's syntax. Conceptually, when a linearization point is reached, the method 'takes effect' and is appended to the linearization order, pr^T. Thus the implementation and specification histories are constructed in lock-step, allowing the algorithm designer to show they correspond.

It has long been understood that linearization points are a limited approach. Algorithms may have linearization points inside other methods or fixed nondeterministically by future behaviour. The TS stack is a particularly acute example of this problem. Two push methods that run concurrently may insert elements with unordered timestamps, giving no information to choose a linearization order. However, if the items are later popped sequentially, an order is imposed on the earlier pushes. Worse, ordering two pushes can implicitly order other methods, leading to a cascade of linearizations back in time.

Consider the following history. Horizontal lines represent execution time, ↓ represents invocations, and ↑ responses.



This history induces the precedence order pr represented by solid lines in the following graph.



First consider the history immediately before the return of pop()→c (i.e. without order (1) in the graph). As push(b) and push(c) run concurrently, elements b and c may have unordered timestamps. At this point, there are several consistent ways that the history might linearize, even given access to the TS stack's internal state.

Now consider the history after pop()→b. Dotted edges represent linearization orders forced by this operation. As c is popped before b, LIFO order requires that push(b) has to be linearized before push(c) – order (2). Transitivity then implies that push(a) has to be ordered before push(c) – order (3). Furthermore, ordering push(a) before push(c) requires that pop()→c is ordered before pop()→a – order (4). Thus a method's linearization order may be fixed long after it returns, frustrating any attempt to choose syntactic linearization points.

Specification-specific conditions (AKA aspects). For a given sequential specification, it may not be necessary to find the entire linearization order to show that an algorithm is linearizable. A degenerate example is the specification which contains all possible sequential histories; in this case, we need not find a linearization order, because any order consistent with pr will do. One alternative to linearization points is thus to invent special-purpose conditions for particular sequential specifications.

Henzinger et al. [11] have just such a set of conditions for queues. (They call this approach *aspect-oriented*). One attractive property of their approach is that their queue conditions are mostly expressed using precedence order, pr. In other words, most features of queue behaviour can be checked without locating linearization points at all.

Stack and set specifications. Our theorem makes use of two sequential specifications: Stack, and a weaker specification Set that does not respect LIFO order. We define the set of permitted histories by defining updates over abstract states. Assume a set of values Val. Abstract states are finite sequences Val^*. Let $\sigma \in \mathsf{Val}^*$ be an arbitrary state. In Stack, push and pop have the following sequential behaviour ('·' means sequence concatenation):

- push(v) – Update the abstract state to $\sigma \cdot [v]$.
- pop() – If $\sigma = []$, return EMPTY. Otherwise, σ must be of the form $\sigma' \cdot [v']$. Update the state to σ', return v'.

In Set, push is the same, but pop behaves as follows:

- pop() – If $\sigma = []$, return EMPTY. Otherwise, σ must be of the form $\sigma' \cdot [v'] \cdot \sigma''$. Update the state to $\sigma' \cdot \sigma''$, return v'.

The stack theorem. We have developed stack conditions sufficient to ensure linearizability with respect to Stack. Unlike [11], our conditions are not expressed using only pr (indeed, we believe this would be impossible – see supplementary Appendix C). Rather we require an auxiliary insert-remove order ir which relates pushes to pops and vice versa, but that does not order pairs of pushes or pairs of pops. In other words, our theorem shows that for stacks it is sufficient to identify just *part* of the linearization order.

We begin by defining the helper orders ins and rem over push operations and pop operations, respectively. Informally, ins and rem are fragments of the linearization order that are imposed by the combination of ir and the precedence order pr. In all the definitions in this section, assume that $\mathcal{H} = \langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ is a history. Below we write $+a, +b, +c$ etc. for push operations, and $-a, -b, -c$ etc. for pop operations.

Definition 3 (derived orders ins and rem)**.** *Assume an insert-remove order* ir*.*

- *For all $+a, +b \in \mathcal{A}$, $+a \xrightarrow{\mathsf{ins}} +b$ if either $+a \xrightarrow{\mathsf{pr}} +b$ or there exists an operation $-c \in \mathcal{A}$ with $+a \xrightarrow{\mathsf{pr}} -c \xrightarrow{\mathsf{ir}} +b$.*
- *For all $-a, -b \in \mathcal{A}$, $-a \xrightarrow{\mathsf{rem}} -b$ if either $-a \xrightarrow{\mathsf{pr}} -b$ or there exists an operation $+c \in \mathcal{A}$ with $-a \xrightarrow{\mathsf{ir}} +c \xrightarrow{\mathsf{ir}} -b$.*

Here ins is weaker than rem – note the pr rather than ir in the final clause. However, our stack theorem also holds if the definitions are inverted, with rem weaker than ins. The version above more convenient in verifying the TS stack.

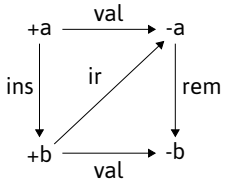
The order ins expresses ordering between pushes imposed either by precedence, or transitively by insert-remove. Likewise rem expresses ordering between pops. Using ins and rem, we can define order-correctness, which expresses the conditions necessary to achieve LIFO ordering in stack.

Definition 4 (alternating)**.** *We call a partial order r on $\mathcal{A}$* alternating *if every pair $+a, -b \in \mathcal{A}$ consisting of one push and one non-empty pop is ordered, and no other pairs are ordered.*

Definition 5 (order-correct)**.** *We call $\mathcal{H}$* order-correct *if there exists an alternating order* ir *on $\mathcal{A}$, and derived orders* ins *and* rem*, such that:*

1. $\mathsf{ir} \cup \mathsf{pr}$ *is cycle-free; and*
2. *Let* $+a, -a, +b \in \mathcal{A}$ *with* $+a \xrightarrow{\mathsf{val}} -a$ *and* $+a \xrightarrow{\mathsf{pr}} -a$. *If* $+a \xrightarrow{\mathsf{ins}} +b \xrightarrow{\mathsf{ir}} -a$, *then there exists* $-b \in \mathcal{A}$ *with* $+b \xrightarrow{\mathsf{val}} -b$ *and* $-a \not\xrightarrow{\mathsf{rem}} -b$;

Condition (2) above is at the heart of our proof approach. The key ordering it forbids is as follows:



As ins, rem and ir are all fragments of the eventual linearization order, this shape corresponds to a non-LIFO ordering that would violate the stack specification.

Order-correctness *only* imposes LIFO ordering; it does not guarantee non-LIFO correctness properties. For a stack these are (1) elements should not be lost; (2) elements should not be duplicated; (3) popped elements should come from a corresponding push; and (4) `pop` should report EMPTY correctly. The last is subtle, as it is a global rather than pairwise property: `pop` should return EMPTY only at a point in the linearization order where the abstract stack is empty. Fortunately, these properties are also orthogonal to LIFO ordering: we just require that the algorithm is linearizable with respect to Set (simple to prove for the TS stack).

Theorem 1 (stack correctness)**.** *Let C be a concurrent algorithm. If every history arising from C is order-correct, and C is linearizable with respect to* Set*, then C is linearizable with respect to* Stack*.*

Proof. Here we only sketch the structure. For full details see supplementary Appendix B and our Isabelle mechanisation, provided in supplementary file `stackthm.tgz`.

The proof has five stages. (1) Order all `pop` operations which do not return empty and which are ordered with their matching `push` operation in the precedence order. (2) Adjust the ir order to deal with the definition of ins discussed above. Again we ignore all `push`-`pop` pairs with overlapping execution times. (3) Order all `push` operations which remain unordered after the first two stages and show that the resulting order is within Stack. (4) Show that `push`-`pop` pairs with overlapping execution times can always be added to a correct linearization order without violating Stack. (5) Show that also `pop` operations which return EMPTY can always be added to a correct linearization order as long as they are correct with respect to Set. □

Our stack theorem is generic, not tied to the TS stack. It characterises the internal ordering sufficient for an algorithm to achieve stack semantics. Intuitively, our theorem seems close to the lower bound for stack ordering: that is, we would expect any concurrent stack to enforce orders as strong as the ones in our theorem. (See supplementary Appendix C for evidence of this). Thus, Theorem 1 points towards fundamental constraints on the structure of concurrent stacks.

For the TS stack, the advantage of Theorem 1 is that problematic orders need not be resolved. In the example discussed above, `push(a)` and `push(c)` can be left unordered in ir, removing the need to decide their eventual linearization order; likewise `pop(a)` and `pop(c)`. As we show in the next section, the ir order *can* be extracted from the TS stack using an adapted version of the linearization point method.

5. TS Stack Correctness

TS buffer specification. The TS buffer is the underlying data-structure we use to implement our TS stack. Informally, a TS buffer is a specialised index which associates a unique identifier with each stored value and timestamp.

The TS buffer is linearizable with respect to the sequential specification TSbuf (proved in §6.2). As with Stack and Set, we define TSbuf by tracking updates to abstract states. Formally, we assume a set of *buffer identifiers*, ID, representing individual buffer elements; and a set of *timestamps*, TS, with partial order $<_{\mathsf{TS}}$ and top element $\top$. Many elements in the buffer can be associated with the same timestamp.

A TSbuf abstract state is a tuple (B, S). $B \in \mathsf{Buf}$ is a partial map from identifiers to value-timestamp tuples, representing the current values stored in the buffer. $S \in \mathsf{Snapshots}$ is a partial map from timestamps to Buf, representing snapshots of the buffer at particular timestamps.

$$\mathsf{Buf} : \mathsf{ID} \rightharpoonup (\mathsf{Val} \times \mathsf{TS}) \qquad \mathsf{Snapshots} : \mathsf{TS} \rightharpoonup \mathsf{Buf}$$

We implicitly assume that all timestamps in the buffer were previously generated by `newTimestamp`. This lets us simplify our abstract specification by restricting the set of use-cases.

Snapshots are needed to support globally-consistent removal. To remove from the buffer, `pop` first calls `getStart` to generate a timestamp `t` – abstractly, $[\mathtt{t} \mapsto B]$ is added to the library of snapshots. When `pop` calls `tryRem(t)`, elements that were present when `t` was generated may be removed normally, while elements added or timestamped more recently than `t` may be eliminated out of order. The stored snapshot $S(\mathtt{t})$ determines which element should be removed or eliminated.

The TS buffer functions have the following specifications, assuming (B, S) is the abstract state before the operation:

- `newTimestamp()` – pick a timestamp $t \neq \top$ such that for all $t' \neq \top$ already in B, $t' <_{\mathsf{TS}} t$. Return t.

  (Note that this means many elements can be issued the same timestamp if the thread is preempted before writing it into the buffer.)
- `ins(v)` – Pick an ID $i \notin \mathrm{dom}(B)$. Update the state to $(B[i \mapsto (v, \top)], S)$ and return i.
- `setTimestamp(i,t)` – assume that $B(\mathtt{i}) = (v, \top)$ and `t` was generated by `newTimestamp()`. Update the abstract state to $(B[\mathtt{i} \mapsto (v, \mathtt{t})], S)$.
- `getStart()` – pick a timestamp $\mathtt{t} \neq \top$ such that $\mathtt{t} \notin \mathrm{dom}(S)$ or $\mathtt{t} \in \mathrm{dom}(S)$ and $S(\mathtt{t}) = B$. If $\mathtt{t} \notin \mathrm{dom}(S)$, update the state to $(B, S[\mathtt{t} \mapsto B])$. Return `t`.
- `tryRem(t)` – Assume $\mathtt{t} \in \mathrm{dom}(S)$. There are four possible behaviours:
  1. *failure.* Nondeterministically fail and return INVALID. This corresponds to a failed `tryRemSP` caused by another thread pre-empting the removal.
  2. *emptiness check.* If the map is empty (i.e. $\mathrm{dom}(B) = \emptyset$) return EMPTY.
  3. *normal removal.* Pick an ID i with $i \in \mathrm{dom}(S(\mathtt{t})) \cap \mathrm{dom}(B)$ and $B(i) \mapsto (v_i, t_i)$ such that t_i is maximal with respect to other timestamps in B, i.e.

$$\nexists i', t' .\, i' \in (\mathrm{dom}(S(\mathtt{t})) \cap \mathrm{dom}(B)) \wedge B(i') = (\_, t') \wedge t_i <_{\mathsf{TS}} t'$$

Update the abstract state to $(B[i \mapsto \bot], S)$ and return v_i. Note that there may be many maximal elements that could be returned.

4. *elimination.* Pick an ID i such that $i \in \mathrm{dom}(B)$, $i \notin \mathrm{dom}(S(\mathtt{t}))$ and $B(i) \mapsto (v, \_)$, or $i \in \mathrm{dom}(S(\mathtt{t}))$ and $S(\mathtt{t})(i) \mapsto (v, \top)$. Update the abstract state to $(B[i \mapsto \bot], S)$ and return v.

   This corresponds to the case where v was inserted or timestamped after `pop` called `getStart`, and v can therefore be removed using elimination.

TS stack correctness proof. We now prove that the TS stack is correct. We first generate two orders vis ('visibility') and rr ('remove-remove') from syntactic points inside the TS stack implementation. This is analogous to the linearization point method, except that we generate two, possibly conflicting orders. The points we choose are TS buffer operations – as the TS buffer is linearizable, we can treat these operations as atomic.

- vis – generated from the final `ins()` in a `push` to the final `getStart()` in a non-empty `pop`. This order is similar to ir, but may contradict order-correctness.
- rr – generated between final `tryRem()` operations in non-empty `pop` operations.

As with ins / rem in the stack theorem, it is useful to define a helper order ts ('timestamp') on `push` operations. This order is imposed by precedence and insert-remove transitivity. Informally, if two `push` operations are ordered in ts their elements are ordered in $<_{\mathsf{TS}}$.

Definition 6 (derived order ts)**.** *Assume a history $\mathcal{H} = \langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ and order vis on $\mathcal{A}$. Two operations $+a, +b \in \mathcal{A}$ are related $+a \xrightarrow{\mathsf{ts}} +b$ if: $+a \xrightarrow{\mathsf{pr}} +b$; or $+a \xrightarrow{\mathsf{pr}} -c \xrightarrow{\mathsf{vis}} +b$ for some $-c \in \mathcal{A}$; or $+a \xrightarrow{\mathsf{pr}} +d \xrightarrow{\mathsf{vis}} -c \xrightarrow{\mathsf{vis}} -b$ for some $-c, +d \in \mathcal{A}$.*

The following lemma connects vis, rr and ts to the insert-remove order ir used in our stack theorem.

Lemma 2. *Let $\mathcal{H} = \langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ be a history. Assume vis, an alternating order on $\mathcal{A}$, and rr, a total order on non-empty pop operations in $\mathcal{A}$. Assume the derived order ts. If:*

1. *$\mathsf{pr} \cup \mathsf{vis}$ and $\mathsf{pr} \cup \mathsf{rr}$ are cycle-free; and*
2. *for all $+a, -a, +b \in \mathcal{A}$ such that $+a \xrightarrow{\mathsf{val}} -a$, $+a \xrightarrow{\mathsf{pr}} -a$, and $+a \xrightarrow{\mathsf{ts}} +b \xrightarrow{\mathsf{vis}} -a$, there exists $-b \in \mathcal{A}$ such that $+b \xrightarrow{\mathsf{val}} -b$ and $-b \xrightarrow{\mathsf{rr}} -a$;*

then $\mathcal{H}$ is order-correct according to Definition 5.

Proof. Follows from the insight that either vis is a witness that $\mathcal{H}$ is order-correct, or vis can be adjusted locally such that it becomes a witness. Details given in Appendix A. □

Lemma 3. *TS stack is linearizable with respect to* Set.

Proof. Straightforward from the fact that the TS buffer is linearizable with respect to TSbuf. We take the linearization point for `push` as the call to `ins` and the linearization point for `pop` as the call to `tryRem`. Correctness follows from the specification of TSbuf. □

Theorem 4. *TS stack is linearizable with respect to* Stack.

Proof. Lemma 3 deals with the first clause of the stack theorem (Theorem 1). The other clause requires the existence of an ir order that satisfies order-correctness. It suffices to show that vis, rr, and ts satisfy the conditions of Lemma 2. The first requirement, that $\mathsf{pr} \cup \mathsf{vis}$ and $\mathsf{pr} \cup \mathsf{rr}$ are cycle-free, follows from the fact that the TS buffer is linearizable with respect to TSbuf. The second requirement for the lemma follows from the fact that `tryRem` removes elements in an order that respects $<_{\mathsf{TS}}$, and the fact that ordering in ts implies ordering in $<_{\mathsf{TS}}$. More details are given in Appendix A. □

Theorem 5. *The TS stack is lock-free.*

Proof. Straightforward from the fact that the TS buffer is lock-free. This follows from the structure of `tryRem`: an attempt to remove an element can only fail when another thread pre-empts it. □

6. Implementation Details

6.1 Timestamping Algorithms

We have experimented with several different timestamping strategies.

TS-atomic: This algorithm takes a timestamp from a global counter using an atomic fetch-and-increment instruction. Such instructions are available on most modern processors – for example the `LOCK XADD` instruction on x86.

TS-hardware: This algorithm uses the x86 `RDTSCP` instruction [16] to read the current value of the TSC register. The TSC register counts the number of processor cycles since the last processor reset.

An obvious concern is that if TSC is not synchronised across cores, relaxed-memory effects may lead to stack-order violations. Aside from `RDTSCP`, we use C11 sequentially-consistent atomics throughout, forbidding all other relaxed behaviours. Ruan et al. [23] have tested `RDTSCP` on various x86 systems as part of their transactional memory system. Our understanding of [23] and the Intel x86 architecture guide [16] is that `RDTSCP` provides sufficient synchronisation to ensure correctness on multi-core and multi-socket machines. Furthermore, we have observed no violations of stack semantics in experiments across different machines.

However, it has become clear in recent years that the memory-order guarantees offered by multi-processors are often under-specified and inaccurately documented – see e.g. Sewell et. al.'s work on a formalized x86 model [21] (which does not cover `RDTSCP`). One message of the TS stack is that cheap timestamp generation can form the basis of high-performance concurrent datastructures. We hope that our work will motivate further research into TSC, `RDTSCP`, and hardware timestamp generation more generally.

TS-stutter: This algorithm uses thread-local counters which are synchronized by Lamport's algorithm [17]. To generate a new timestamp a thread first reads the values of all thread-local counters. It then takes the maximum value, increments it by by one, stores it in its thread-local counter, and returns the stored value as the new timestamp. Note that the TS-stutter algorithm does not require strong synchronization. TS-stutter timestamping may return the same timestamp multiple times, but it never returns a timestamp that already exists in the buffer.

Listing 2: TS buffer implementation. The SP buffer implementation is described in supplementary Appendix D.

```
1  TS_Buffer{
2   ListOfSPBuffers spBuffers;
3
4   TimestampedItem ins(Element element) {
5     TimestampedItem item = createItem(element);
6     SPBuffer buffer = threadSPBuffer();
7     buffer.insSP(item);
8     return item;
9   }
10
11  int getStart() {
12    return getMaxTimestamp();
13  }
14
15  void setTimestamp(TimestampedItem item, t) {
16    item.timestamp = t;
17  }
18
19  TimestampedItem tryRem(startTime) {
20    TimestampedItem youngest;
21    SPBuffer buf;
22    forall (spBuffer in spBuffers) {
23      TimestampedItem item = spBuffer.getSP();
24      // Eliminate item if possible.
25      if (item.timestamp > startTime) {
26        if (spBuffer.tryRemSP(item))
27          return item;
28      }
29      if (item.timestamp > youngest.timestamp) {
30        youngest = item;
31        buf = spBuffer;
32      }
33    }
34    if (empty(spBuffers)) // Emptiness check.
35      return emptyItem;
36      if (buf.tryRemSP(youngest)) {
37      return youngest;
38    }
39    return invalidItem;
40  }
41 }
```

TS-interval: This algorithm does not return one timestamp value, but rather an interval consisting of a pair of timestamps generated by one of the algorithms above. Let $[a, b]$ and $[c, d]$ be two such interval timestamps. They are ordered $[a, b] <_{\mathsf{TS}} [c, d]$ if and only if $b < c$. That is, if the two intervals overlap, the timestamps are unordered. The TS-interval algorithm is correct because the maximum of any interval timestamp stored in the buffer is less than the lower limit generated by calls to TS-interval.

In our experiments we use the TS-hardware algorithm (i.e. the x86 `RDTSCP` instruction) to generate the start and end of the interval, because it is faster than TS-atomic and TS-stutter. Adding a delay between the generation of the two timestamps increases the size of the interval, allowing more timestamps to overlap and thereby reducing contention during element removal. The effect of adding a delay on overall performance is analyzed in Section 7.2.

6.2 The TS Buffer

SP buffers. The TS buffer is the data-structure underlying the TS stack. We implement the TS buffer using a list of SP (single-producer) buffers. Each thread inserts elements into its own SP buffer but may remove elements from any thread's SP buffer. SP buffers support the following operations:

- `insSP` – insert an element into the buffer.
- `getSP` – return the identifier of the maximal element according to $<_{\mathsf{TS}}$.
- `tryRemSP` – try to remove an identified element.

Our SP buffer implementation is a singly-linked list design (see discussion in supplementary Appendix D). Only a single thread inserts elements, so no strong synchronization is needed in `insSP`. Removal is a two-stage process: in a first stage the thread marks a `taken` flag to indicate the node has been removed. In a second stage marked nodes are unlinked. Unlinking can be done either immediately or by later operations.

To avoid polluting our benchmarks with memory management effects, unlinked nodes are not reclaimed. In a real-world implementation, it would be straightforward to use garbage collection or hazard pointers [18] for reclamation.

TS buffer operations. Listing 2 shows the pseudocode of our TS buffer implementation. `newTimestamp` is not shown because it is already described in Section 6.1.

All the SP buffers are linked from a list of `spBuffers`. Additionally each thread has a thread-local pointer to its SP buffer. Our implementation supports thread registration and removal by adding and removing buffers from the list.

The `ins` operation first retrieves the SP buffer of the executing thread and then inserts the element using its `insSP` operation. The `setTimestamp` operation simply assigns the timestamp to the `timestamp` field of the item. The `getStart` operation returns a timestamp which is at least as late as the latest generated timestamp.

Starting at a random TS buffer, the `tryRem` operation searches all SP buffers for youngest elements (lines 22–33). Randomization helps avoid collisions between concurrent `tryRem` operations. It then tries to remove one of the youngest elements from the identified SP buffer using `tryRemSP`. The `startTime` given as a parameter is used to determine if elimination is possible (line 24).

To simplify the presentation, we have omitted the emptiness check. Our implementation uses the approach from [8]: each thread has an array the size of the number of SP buffers. Whenever a thread encounters an empty buffer, it stores the `top` pointer of the buffer in this array. If in two subsequent executions of `tryRem` no SP buffer returns an element, and the `top` pointers of all SP buffers have not changed, then `tryRem` returns EMPTY. Note that this requires an ABA-counter to avoid the ABA-problem. See [8] for a proof of correctness.

Correctness argument. The concrete state of the TS buffer is a partial mapping from a SP buffer ID and buffer identifier (i.e. the memory address of the node which stores an element in an SP buffer) to a value-timestamp tuple:

$$\mathsf{LBuf} : (\mathsf{Thr} \times \mathsf{ID}) \rightharpoonup (\mathsf{Val} \times \mathsf{TS})$$

For the abstract state of the TS buffer snapshots are also needed. We can easily reconstruct them by examining the preceding trace. Snapshots are generated by the state of the buffer at the point `getMaxTimestamp()` is called in `getStart`. Thread identifiers are erased when mapping to the abstract state.

Theorem 6. *The TS buffer implementation is linearizable with respect to the specification* TSBUF *given in §5.*

Proof. Most TS buffer methods consist of a single atomic SP buffer operation, which is its linearization point. The exception is `tryRem`, where the linearization point is the call to `tryRemSP`. The operation `tryRem` always removes a valid element because any element in the snapshot is guaranteed to be contained in one of the SP buffers before `tryRem` starts its search for the youngest element. More details are given in the supplementary Appendix F. □

7. Performance Analysis

Our experiments compare the performance and scalability of the TS stack with two high-performance concurrent stacks: the Treiber stack [22] because it is the de-facto standard lock-free stack implementation; and the elimination-backoff (EB) stack [9] because it is the fastest concurrent stack we are aware of.[2]

We ran our experiments on two x86 machines:

- an Intel-based server with four 10-core 2GHz Intel Xeon processors (40 cores, 2 hyperthreads per core), 24MB shared L3-cache, and 128GB of UMA memory running Linux 3.8.0-36; and
- an AMD-based server with four 16-core 2.3GHz AMD Opteron processors (64 cores), 16MB shared L3-cache, and 512GB of cc-NUMA memory running Linux 3.5.0-49.

Measurements were done in the Scal Benchmarking Framework [5]. To avoid measurement artifacts the framework uses a custom memory allocator which performs cyclic allocation [20] in preallocated thread-local buffers for objects smaller than 4096 bytes. Larger objects are allocated with the standard allocator of glibc. All memory is allocated cache-aligned when it is beneficial to avoid cache artifacts. The framework is written in C/C++ and compiled with gcc 4.8.1 and -O3 optimizations.

Scal provides implementations of the Treiber stack and of the EB stack. Unlike the description of the EB stack in [9] we access the elimination array *before* the stack – this improves scalability in our experiments. We configured the EB stack such that the performance is optimal in our benchmarks when exercised with 80 threads on the 40-core machine, or with 64 threads on the 64-core machine. These configurations may be suboptimal for lower numbers of threads. Similarly, the TS stack configurations we discuss later are selected to be optimal for 80 and 64 threads on the 40-core and 64-core machine, respectively. On the 40-core machine the elimination array is of size 16 with a delay of $18\mu s$ in the high-contention benchmark, and of size 12 with a delay of $18\mu s$ in the low contention benchmark. On the 64-core machine the elimination array is of size 32 with a delay of $21\mu s$ in the high-contention benchmark, and of size 16 with a delay of $18\mu s$ in the low contention benchmark.

On the 64-core machine the Treiber stack benefits from a backoff strategy which delays the retry of a failed `CAS`. On this machine, we configured the Treiber stack with a constant delay which is optimal for the benchmark when exercised with 64 threads. On the 40-core machine performance decreases when a backoff delay is added, so we disable it.

We compare the data-structures in producer-consumer microbenchmarks where threads are split between dedicated producers which insert 1,000,000 elements into the data-structure, and dedicated consumers which remove 1,000,000 elements from the data-structure. We measure performance as total execution time of the benchmark. Figures show the total execution time in successful operations per millisecond to make scalability more visible. All numbers are averaged over 5 executions. To avoid measuring empty removal, operations that do not return an element are not counted.

The contention on the data-structure is controlled by a computational load which is calculated between two operations of a thread. In the high-contention scenario the computational load is a π-calculation in 250 iterations, in the low-contention scenario π is calculated in 2000 iterations. On average a computational load of 1000 iterations corresponds to a delay of $2.3\mu s$ on the 40-core machine.

7.1 Performance and Scalability Results

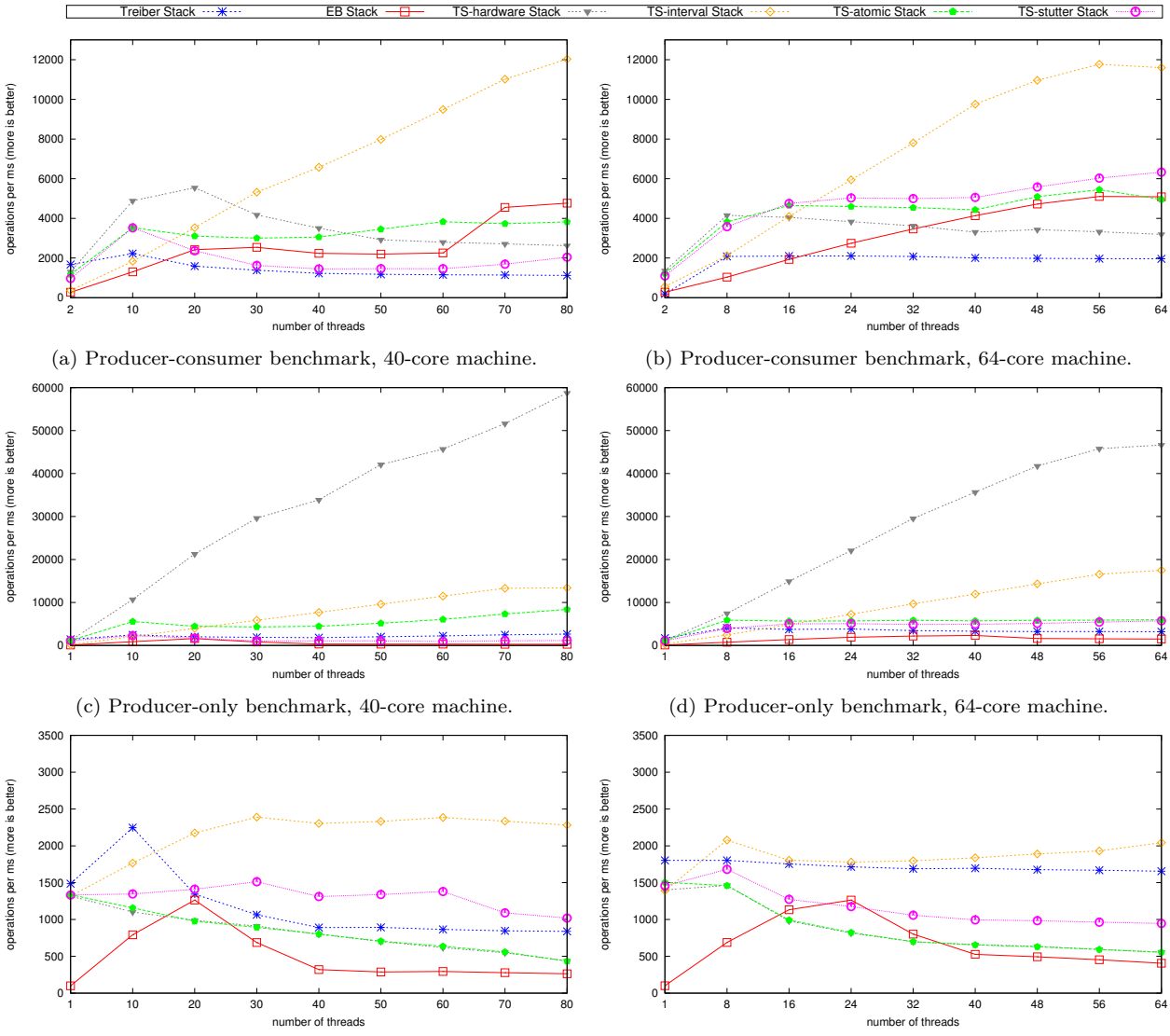
Figures 1a and 1b show performance and scalability in a producer-consumer benchmark where half of the threads are producers and half of the threads are consumers. These figures show results for the high-contention scenario. Results for the low-contention scenario are similar, but less pronounced – see Figure 3 in the supplementary material.

For TS-interval timestamping we use the optimal delay when exercised with 80 threads on the 40-core machine, and with 64 threads on the 64-core machine, derived from the experiments in Section 7.2. The delay thus depends on the machine and benchmark. On the 40-core machine the delay is $7.5\mu s$ and $4.5\mu s$ in the high and low contention benchmark, respectively. On the 64-core machine the delay is $4.5\mu s$ for both the high-contention benchmark and the low contention benchmark. The impact of different delays on performance is discussed in Section 7.2.

Comparison between implementations. The TS-interval stack is faster than the other timestamping algorithms in the producer-consumer benchmarks with an increasing number of threads. Interestingly the TS-atomic stack is faster than the TS-hardware stack in the high-contention producer-consumer benchmark. The reason is that since the `push` operations of the TS-hardware stack are so much faster than the `push` operations of the TS-atomic stack, elimination is possible for more `pop` operations of the TS-atomic stack (e.g. 41% more elimination on the 64-core machine, see Table 1 in the supplementary appendix), which results in a factor of 3 less retries of `tryRem` operations than in the TS-hardware stack. On the 40-core machine the TS-stutter stack is significantly slower than the TS-atomic stack, while on the 64-core machine the TS-stutter stack is faster. The reason is that on the 40-core machine TS-stutter timestamping is significantly slower than TS-atomic timestamping.

Comparison with other data-structures. With more than 16 threads all TS stacks are faster than the Treiber stack. On both machines the TS-interval stack out-performs the EB stack by a factor of 2 in the high-contention producer consumer benchmark with a maximum number of threads, on the 64-core machine also the TS-stutter stack out-performs the EB stack, and the TS-atomic stack is close to the performance of the EB stack.

[2] Of course, other high-performance stacks exist. We decided against benchmarking the DECS stack [3] because (1) no implementation is available for our platform and (2) according to their experiments, in peak performance it is no better than a Flat Combining stack. We decided against benchmarking the Flat Combining stack because the EB stack outperforms it when configured to access the backoff array before the stack itself.



(a) Producer-consumer benchmark, 40-core machine.

(b) Producer-consumer benchmark, 64-core machine.

(c) Producer-only benchmark, 40-core machine.

(d) Producer-only benchmark, 64-core machine.

(e) Consumer-only benchmark, 40-core machine.

(f) Consumer-only benchmark, 64-core machine.

Figure 1: TS stack performance in the high-contention scenario on 40-core machine (left) and 64-core machine (right).

We believe TS-interval's performance increase with respect to the EB stack comes from three sources: (a) more elimination; (b) faster elimination; (c) higher performance without elimination. As shown in producer-only and consumer-only experiments, the lack of push-contention and mitigation of contention in pop makes our stack fast even without elimination. Additional experiments show that the TS-interval stack eliminates 7% and 23% more elements than the EB stack in high-contention scenarios on the 40-core and on the 64-core machine, respectively. Thus we improve on EB in both (a) and (c). (b) is difficult to measure, but we suspect integrating elimination into the normal code path introduces less overhead than an elimination array, and is thus faster.

Push performance. We measure the performance of `push` operations of all data-structures in a producer-only benchmark where each thread pushes 1,000,000 element into the stack. The TS-interval stack uses the same delay as in the high-contention producer-consumer benchmark: 7.5μs on the 40-core machine and 4.5μs on the 64-core machine.

Figure 1c and Figure 1d show the performance and scalability of the data-structures in the high-contention producer-only benchmark. The `push` performance of the TS-hardware stack is significantly better than the `push` of the other stack implementations. With an increasing number of threads the `push` operation of the TS-interval stack is faster than the `push` operations of the TS-atomic stack and the TS-stutter stack, which means that the delay in the TS-interval timestamping is actually shorter than the execution time of the TS-atomic timestamping and the TS-stutter timestamping. Perhaps surprisingly, TS-stutter, which does not require strong synchronisation, is slower than TS-atomic, which is based on an atomic fetch-and-increment instruction.

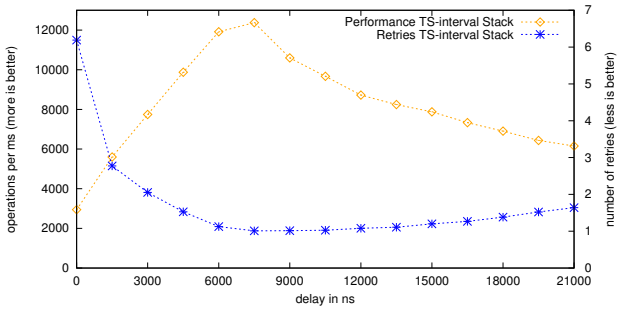


Figure 2: High-contention producer-consumer benchmark using TS-interval timestamping with increasing delay on the 40-core machine, exercising 40 producers and 40 consumers.

Pop performance. We measure the performance of `pop` operations of all data-structures in a consumer-only benchmark where each thread pops 1,000,000 from a pre-filled stack. Note that no elimination is possible in this benchmark. The stack is pre-filled concurrently, which means in case of the TS-interval stack and TS-stutter stack that some elements may have unordered timestamps. Again the TS-interval stack uses the same delay as in the high-contention producer-consumer benchmark.

Figure 1e and Figure 1f show the performance and scalability of the data-structures in the high-contention consumer-only benchmark. The performance of the TS-interval stack is significantly higher than the performance of the other stack implementations, except for low numbers of threads. The TS-stutter stack is faster than the other stack implementations due to the fact that some elements share timestamps and therefore can be removed in parallel. The TS-atomic stack and TS-hardware stack show the same performance because all elements have unique timestamps and therefore have to be removed sequentially. Also in the Treiber stack and the EB stack elements have to be removed sequentially. Depending on the machine, removing elements sequentially from a single list (Treiber stack) is sometimes more and sometimes less expensive than removing elements sequentially from multiple lists (TS stack).

7.2 Analysis of TS-Interval Timestamping

Figure 2 shows the performance of the TS-interval stack along with the average number of `tryRem` calls needed in each `pop` (one call is optimal, but contention may cause retries). These figures were collected with an increasing interval length in the high contention producer-consumer benchmark on the 40-core machine. We used these results to determine the delays for the benchmarks in Section 7.1.

Initially the performance of the TS data-structures increases with an increasing delay, but beyond 7.5μs the performance decreases again. After that point an average `push` operation is slower than an average `pop` operation, the number of `tryRem` operations increases again and also the number of `pop` operations which return `EMPTY` increases.

The figure also shows that high performance correlates strongly with a drop in `tryRem` retries. We conclude from this that the impressive performance we achieve with interval timestamping arises from reduced contention in `remove`. For the optimal delay we have 1.009 calls to `tryRem` per `pop`, i.e. less than 1% of `pop` calls need to scan the SP buffer array more than once. In contrast, without a delay the average number of retries per `pop` call is more than 6.

8. TS Queue and TS Deque Variants

In this paper, we have focussed on the stack variant of our algorithm. However, stored timestamps can be removed in any order, meaning it is simple to change our TS stack into a queue / deque. Doing this requires three main changes:

1. Change the timestamp comparison operator in `tryRem`.
2. Change the SP buffer such that `getSP` returns the oldest / right-most / left-most element.
3. For the TS queue, remove elimination in `tryRem`. For the TS deque, enable it only for stack-like removal.

The TS queue performs well, but the lack of elimination means it does not outperform all competitor algorithms. In our experiments the TS-interval queue out-performs the Michael-Scott queue [19] and the flat-combining queue [15] but is not as fast as the LCRQ [1].

The TS-interval deque is in general slower than the corresponding stack / queue. However, it also out-performs the Michael-Scott queue and the flat-combining queue when used as a queue, and it out-performs the Treiber stack and the EB stack when used as a stack.

9. Conclusions and Future Work

We present a novel approach to implementing ordered concurrent data-structures like queues, stacks, and deques; a high-performance concurrent algorithm, the TS stack; and a new proof technique required to show the TS stack is correct. The broad messages that we draw from our work are:

- In concurrent data-structures, total ordering on internal data imposes a performance cost and is unnecessary for linearizability.
- However, weakened internal ordering makes establishing correctness more challenging. Specification-specific theorems such as our stack theorem can solve this problem.

Our work represents an initial step in designing and verifying timestamped data-structures. In future work, we plan to experiment with relaxing other internal ordering constraints; with dynamically adjusting the level of order in response to contention; with correctness conditions weaker than linearizability; and with relaxing the underlying memory model.

References

[1] Y. Afek and A. Morrison. Fast concurrent queues for x86 processors. In *PPoPP*. ACM, 2013.

[2] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. Michael, and M. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*, 2011.

[3] G. Bar-Nissan, D. Hendler, and A. Suissa. A dynamic elimination-combining stack algorithm. In *OPODIS*, 2011.

[4] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, 2013.

[5] Computational Systems Group, University of Salzburg. Scal framework. URL `http://scal.cs.uni-salzburg.at`.

[6] M. Gorelik and D. Hendler. Brief announcement: an asymmetric flat-combining based queue algorithm. In *PODC*, 2013.

[7] A. Haas, C. Kirsch, M. Lippautz, and H. Payer. How FIFO is your concurrent FIFO queue? In *RACES*. ACM, 2012.

[8] A. Haas, T. Henzinger, C. Kirsch, M. Lippautz, H. Payer, A. Sezgin, and A. Sokolova. Distributed queues in shared

memory—multicore performance and scalability through quantitative relaxation. In *CF*. ACM, 2013.

[9] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*. ACM, 2004.

[10] T. Henzinger, H. Payer, and A. Sezgin. Replacing competition with cooperation to achieve scalable lock-free FIFO queues. Technical Report IST-2013-124-v1+1, IST Austria, 2013.

[11] T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In *CONCUR*, 2013.

[12] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.

[13] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3), 1990.

[14] M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. In *OPODIS*. Springer, 2007.

[15] D. H. I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, 2010.

[16] Intel. Intel 64 and ia-32 architectures software developer's manual, volume 3b: System programming guide, part 2, 2013. URL `http://download.intel.com/products/processor/manual/253669.pdf`.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications ACM*, 21, July 1978.

[18] M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *TPDS*, 15, 2004.

[19] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*. ACM, 1996.

[20] H. H. Nguyen and M. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *ISMM*. ACM, 2007.

[21] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7), 2010.

[22] R. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, April 1986.

[23] Y. L. Wenjia Ruan and M. Spear. Boosting timestamp-based transactional memory by exploiting hardware cycle counters. In *TRANSACT*, 2013.

A. Selected proofs

Lemma 2. *Let $\mathcal{H} = \langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ be a history. Assume* vis*, an alternating order on $\mathcal{A}$, and* rr*, a total order on non-empty pop operations in $\mathcal{A}$. Assume the derived order* ts*. If:*

1. $\mathsf{pr} \cup \mathsf{vis}$ *and* $\mathsf{pr} \cup \mathsf{rr}$ *are cycle-free; and*
2. *for all $+a, -a, +b \in \mathcal{A}$ such that $+a \xrightarrow{\mathsf{val}} -a$, $+a \xrightarrow{\mathsf{pr}} -a$, and $+a \xrightarrow{\mathsf{ts}} +b \xrightarrow{\mathsf{vis}} -a$, there exists $-b \in \mathcal{A}$ such that $+b \xrightarrow{\mathsf{val}} -b$ and $-b \xrightarrow{\mathsf{rr}} -a$;*

then $\mathcal{H}$ is order-correct according to Definition 5.

Proof. (Several steps below are proved in more detail in supplementary appendix E)

We prove the lemma by showing that if vis is not already a witness for $\mathcal{H}$ being order-correct, then it can be adjusted to become a witness.

By assumption $\mathsf{pr} \cup \mathsf{vis}$ is cycle-free and therefore satisfies condition 1 of Definition 5. Thus, if vis is not a witness it must violate condition 2, meaning there exist $+a, -a, +b \in \mathcal{A}$ such that $+a \xrightarrow{\mathsf{val}} -a$, $+a \xrightarrow{\mathsf{ins}} +b \xrightarrow{\mathsf{vis}} -a$; and either no $-b \in \mathcal{A}$ exists such that $+b \xrightarrow{\mathsf{val}} -b$, or it exists and $-a \xrightarrow{\mathsf{rem}} -b$. Whenever $+a \xrightarrow{\mathsf{ins}} +b$, also $+a \xrightarrow{\mathsf{ts}} +b$. Therefore by premise 2 of the current lemma there must exist a $-b$ with $-b \xrightarrow{\mathsf{rr}} -a$. As $\mathsf{pr} \cup \mathsf{rr}$ is cycle-free, $-a \xrightarrow{\mathsf{pr}} -b$ cannot hold. Thus $-a \xrightarrow{\mathsf{rem}} -b$ implies there exists a $+c \in \mathcal{A}$ with $-a \xrightarrow{\mathsf{vis}} +c \xrightarrow{\mathsf{vis}} -b$. We call $(-a, +c, -b)$ a *violating triple*.

As a first step to eliminating violations entirely, we modify vis so that no violating triple $(-a, +c, -b)$ has a corresponding $-c \in \mathcal{A}$ where $+c \xrightarrow{\mathsf{val}} -c$ and $-c \xrightarrow{\mathsf{rr}} -a$. For each such triple we modify vis such that $+c \xrightarrow{\mathsf{vis}'} -a$. This adjusted vis' relation satisfies the same assumptions as vis, but $(-a, +c, -b)$ is no longer violating. Adjustment is guaranteed to complete because histories are finite and each step moves a `push` forward, while no `push` operation is ever moved backward.

After modifying vis, violations of Condition 2 can still exist. However, for any remaining violating triples $(-a, +c, -b)$ either there does not exist a $-c \in \mathcal{A}$ with $+c \xrightarrow{\mathsf{val}} -c$; or $-c \in \mathcal{A}$ exists and $-a \xrightarrow{\mathsf{rr}} -c$. For any such violating triple, we modify vis such that $+b$ is ordered the same as $+c$. That is, for all pop operations $-d$, $-d \xrightarrow{\mathsf{vis}'} +b$ iff $-d \xrightarrow{\mathsf{vis}'} +c$. This modification moves $+b$ backwards in vis, and no new violating triples $(-e, +b, -f)$ are created where the violating triple $(-e, +c, -f)$ did not exist already before the modification. Adjustment is guaranteed to complete as it cannot increase the number of violations in vis and always moves operations backwards. The result is a vis relation without violations, which is a witness that $\mathcal{H}$ is order-correct. □

Theorem 4. *TS stack is linearizable with respect to* Stack.

Proof. Lemma 3 deals with one clause of the stack theorem, Theorem 1. To show that there exists an ir order that satisfies the definition of order-correctness it suffices to show that the generated vis, rr, and ts orders satisfy the conditions of Lemma 2.

Assume we have a trace $\mathcal{T}$ consisting of calls and returns to stack methods and atomic calls to the TS buffer methods. We write $a <_{\mathcal{T}} b$ if operations a and b are ordered in $\mathcal{T}$.

Order rr is a subset of $<_{\mathcal{T}}$, and is therefore cycle-free. Moreover, rr is included in pr, which means that if two pop operations are ordered in pr, then they are also ordered in rr. Therefore $\mathsf{pr} \cup \mathsf{rr}$ is cycle-free. A similar argument shows that $\mathsf{pr} \cup \mathsf{vis}$ is cycle-free.

Next we show that if two `push` operations are ordered in ts, then either (1) the inserted elements are ordered by $<_{\mathsf{TS}}$, or (2) the second element gets timestamped after the first element was removed.

Assume two `push` operations $+a$ and $+b$ are ordered in ts. Therefore either $+a \xrightarrow{\mathsf{pr}} +b$, or $+a \xrightarrow{\mathsf{pr}} -c \xrightarrow{\mathsf{vis}} +b$ for some $-c \in \mathcal{A}$, or $+a \xrightarrow{\mathsf{pr}} +d \xrightarrow{\mathsf{vis}} -c \xrightarrow{\mathsf{vis}} +b$ for some $-c, +d \in \mathcal{A}$. Let $\texttt{setTimestamp}_{+a}$ be the `setTimestamp` operation of $+a$, and let $\texttt{ins}_{+b}$ and $\texttt{newTimestamp}_{+b}$ be the `ins` and `newTimestamp` operation of $+b$, respectively. In all three cases it holds that $\texttt{setTimestamp}_{+a} <_{\mathcal{T}} \texttt{ins}_{+b}$. Therefore when $+b$ acquires a new timestamp either element a is in the buffer with a timestamp assigned, or it has already been removed.

If a is removed by a `pop` $-a$ before b is inserted, then condition 2 cannot be violated because it cannot be that that $+b \xrightarrow{\mathsf{vis}} -a$. Next we assume that a is removed after b gets inserted and $a <_{\mathsf{TS}} b$. According to Condition 2 assume that

$+b \xrightarrow{\mathsf{vis}} -a$, with $+a \xrightarrow{\mathsf{val}} -a$ and $+a \xrightarrow{\mathsf{pr}} -a$. This means that b is inserted into the TS buffer before $-a$ calls its `getStart` operation $\texttt{getStart}_{-a}$. Therefore either b is removed before $-a$ calls `getStart` or b is within the snapshot generated by `getStart`.

If b is removed before $\texttt{getStart}_{-a}$ then there must exist a `pop` operation $-b$ such that its `tryRem` operation $\texttt{tryRem}_{-b}$ precedes the `tryRem` operation $\texttt{tryRem}_{-a}$ of $-a$, satisfying Condition 2.

Now assume that b is within the snapshot generated by $\texttt{getStart}_{-a}$. $+a \xrightarrow{\mathsf{pr}} -a$ implies that a got timestamped before the snapshot is generated by $-a$, therefore a is removed by a normal remove. According to the TSBUF specification, `tryRem` removes a maximal element in the TS buffer which is also in the snapshot. As both a and b are in the snapshot generated by $\texttt{getStart}_{-a}$ and $a <_{\mathsf{TS}} b$, this means that b must have been removed before $\texttt{tryRem}_{-a}$. Therefore there has to exist a `pop` operation $-b$ which removes b and its `tryRem` operation precedes $\texttt{tryRem}_{-a}$ in $\mathcal{T}$. This completes the proof. □

A Scalable, Correct Time-Stamped Stack

(supplementary appendices)

The Isabelle mechanization of our core theorem is in the tarball `stackthm.tgz`. The theorem itself is in the file `stack_theorem.thy`. Supplementary Appendix B describes the structure of the rest of the proof.

The TS stack, queue, and deque source-code is included in the tarball `scal.tgz` – this is a snapshot of the Scal benchmarking project [5]. The code for the timestamped algorithms is in the sub-directory:

```
./scal/src/datastructures/
```

Follow the instructions in `README.md` to build the whole Scal benchmark suite, including the TS stack. Note that this code has only been tested on Linux. Once the benchmarks have built, `README_TS-Stack.txt` describes the parameters we used in our experiments.

B. Proving the Stack Theorem

This appendix gives more details about the proof of the stack theorem and its mechanisation in Isabelle. Recall that the theorem is as follows:

Theorem 1. *Let C be a concurrent algorithm implementation. If every history arising from C is order-correct, and C is linearizable with respect to* Set*, then C is linearizable with respect to* Stack*.*

Axiomatised sequential specifications. For convenience in the mechanisation, we use axiomatised definitions of the sequential specifications Stack and Set (these are equivalent to the operational definitions given in the paper body). In these definitions $\mathsf{isPush}(a)$ if a is a `push` operation, $\mathsf{isPop}(b)$ if b is a `pop` operation, and $\mathtt{emp}(e)$ if e is a `pop` operation which returns EMPTY.

Definition 7 (Set)**.** *A sequential history $\langle \mathcal{A}, \mathsf{pr}, \mathsf{val}\rangle$ is in* Set *if and only if the following conditions hold:*

1. *No operation is both a* `push` *operation and a* `pop` *operation:*
$$\forall a, b \in \mathcal{A}.\, a \xrightarrow{\mathsf{val}} b \implies \neg\exists c \in \mathcal{A}.\, c \xrightarrow{\mathsf{val}} a \wedge \neg\exists d \in \mathcal{A}.\, b \xrightarrow{\mathsf{val}} d$$
2. *An element is removed at most once:*
$$\forall a, b, c \in \mathcal{A}.\, a \xrightarrow{\mathsf{val}} b \wedge a \xrightarrow{\mathsf{val}} c \implies b = c$$
3. *Each* `pop` *operation removes at most one element:*
$$\forall a, b, c \in \mathcal{A}.\, a \xrightarrow{\mathsf{val}} b \wedge c \xrightarrow{\mathsf{val}} b \implies a = c$$
4. *Elements are inserted before they are removed:*
$$\forall a, b \in \mathcal{A}.\, a \xrightarrow{\mathsf{val}} b \implies a \xrightarrow{\mathsf{pr}} b$$
5. *A* `push` *operation only returns* EMPTY *if the set is actually empty:*
$$\forall e, a \in \mathcal{A}.\, \mathtt{emp}(e) \wedge \mathsf{isPush}(a) \wedge a \xrightarrow{\mathsf{pr}} e \implies \exists b.\, a \xrightarrow{\mathsf{val}} b \wedge b \xrightarrow{\mathsf{pr}} e$$

Definition 8 (Stack)**.** *A sequential history $\langle \mathcal{A}, \mathsf{pr}, \mathsf{val}\rangle$ is in* Stack *if and only if the following conditions hold:*

1. $\langle \mathcal{A}, \mathsf{pr}, \mathsf{val}\rangle$ *is in* Set

	40-core machine	64-core machine
EB stack	92%	76%
TS-hardware stack	46%	65%
TS-interval stack	99%	97%
TS-atomic stack	87%	92%
TS-stutter stack	94%	98%

Table 1: Percentage of elements removed by elimination in the high-contention producer-consumer benchmark.

2. *Elements are removed in a LIFO fashion:*
$$\forall {+a}, {-a}, {+b} \in \mathcal{A}.\, {+a} \xrightarrow{\mathsf{val}} {-a} \wedge {+a} \xrightarrow{\mathsf{pr}} {+b} \xrightarrow{\mathsf{pr}} {-a} \implies \exists {-b} \in \mathcal{A}.\, {+b} \xrightarrow{\mathsf{val}} {-b} \wedge {-b} \xrightarrow{\mathsf{pr}} {-a}$$

Proof structure. In the first stage of the proof mechanisation we show that we can handle `pop` operations which return EMPTY independent of the order requirements of the stack. This means that if a concurrent history $\langle \mathcal{A}, \mathsf{pr}, \mathsf{val}\rangle$ is linearizable with respect to Set, then we can ignore `pop` operations which return EMPTY for the rest of the proof. This part of the proof is done in the file `empty_stack.thy`.

In the second step we show that we can also ignore all `push`-`pop` pair which have overlapping execution times and are therefore candidates for elimination. This means that after this stage we can assume for all operations ${+a}, {-a} \in \mathcal{A}$, if ${+a} \xrightarrow{\mathsf{val}} {-a}$, then also ${+a} \xrightarrow{\mathsf{pr}} {-a}$. This part of the proof is done in the file `elimination_stack.thy`.

In the third stage we show that if a concurrent history $\langle \mathcal{A}, \mathsf{pr}, \mathsf{val}\rangle$ is order-correct, then we can construct a total order $\mathsf{pr}^{\mathsf{pop}}$ on the `pop` operations which will be part of the linearization order. This total order contains rem and all edges $(-a, -b)$ where $-b \xrightarrow{\mathsf{rem}} -a$ would violate order-correctness, i.e. there exist ${+a}, {+b} \in \mathcal{A}$ with ${+a} \xrightarrow{\mathsf{val}} {-a}$, ${+b} \xrightarrow{\mathsf{val}} {-b}$, and ${+b} \xrightarrow{\mathsf{ins}} {+a} \xrightarrow{\mathsf{ir}} {-b}$. The proof details are in the file `remove_relaxed_stack.thy`.

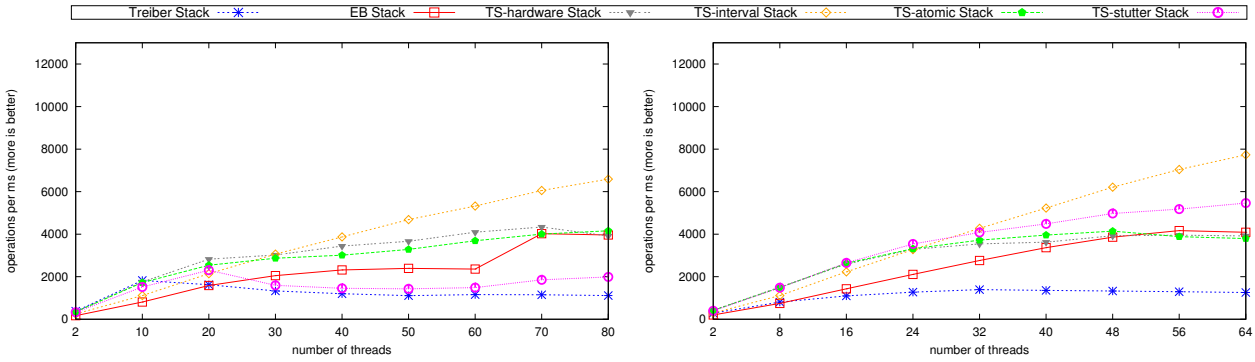
In the forth stage we deal with ins. We adjust the ir order to become part of the final linearization order. Informally we do this by ordering `push` operations as late as possible, even if ir orders them the other way around. Formally a `push` operation ${+a}$ is ordered before a `pop` operation ${-b}$ in the adjusted order ir' if either

1. ${+a} \xrightarrow{\mathsf{pr}} {-b}$ or
2. there exist ${-a}, {+c}, {-c} \in \mathcal{A}$ with ${+a} \xrightarrow{\mathsf{val}} {-a}$, ${+c} \xrightarrow{\mathsf{val}} {-c}$, ${+c} \xrightarrow{\mathsf{ir}'} {-b}$, ${-b} \xrightarrow{\mathsf{pr}^{\mathsf{pop}}} {-c} \xrightarrow{\mathsf{pr}^{\mathsf{pop}}} {-a}$, and ${+a} \xrightarrow{\mathsf{ir}'} {-c}$.

The file `insert_pr_relaxed_stack.thy` contains this stage of the proof.

In the final stage we order the remaining unordered `push` operations according to the order of their matching `pop` operations and show that for the resulting order pr^T it holds that $\langle \mathcal{A}, \mathsf{pr}^T, \mathsf{val}\rangle \in$ Stack. See file `insert_relaxed_stack.thy` for this. The file `stack_theorem.thy` connects the lemmas together into the overall proof, while `concur_history_relations.thy` contains the basic definitions.

To simplify the proof structure, we assume that in every execution, all elements which are pushed also get popped. We justify this by observing that any concurrent history where some elements do not get popped can be extended to one where all elements get popped. If the extended history is linearizable, then also the original history was linearizable.



(a) Producer-consumer benchmark on the 40-core machine.

(b) Producer-consumer benchmark on the 64-core machine.

Figure 3: Performance of the TS stack in the low-contention scenario.

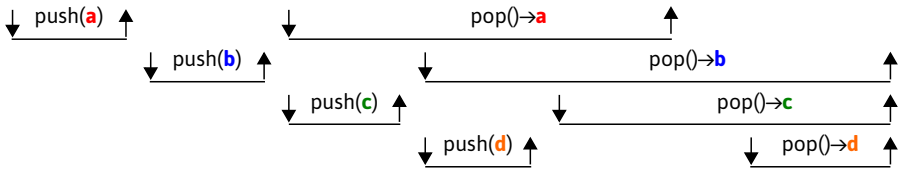


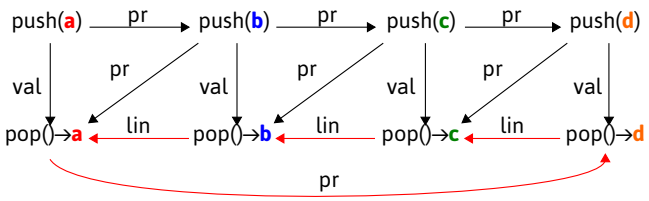
Figure 4: Example bad execution with non-local behaviour.

C. Stacks and Insert-Remove Order

Our stack theorem (Theorem 1) builds on a similar theorem for queues proved by Henzinger et al. [11]. As with our definition of order-correctness (Definition 5), their theorem forbids certain bad orderings between operations. However, their forbidden shapes are defined purely in terms of the precedence order, pr, and value order, val – they do not require the auxiliary insert-remove order ir.

We believe that any stack theorem similar in structure to ours must require some additional information like ir (and as a corollary, that checking linearizability for stacks is fundamentally harder than for queues). By 'similar', we mean a local theorem defined by forbidding a finite number of finite-size bad orderings. It is this locality that makes our theorem and Henzinger's so appealing. It reduces data-structure correctness from global ordering to ruling out a number of specific bad cases.

Our key evidence that the insert-remove order is needed is the execution shown in Figure 4. This execution as a whole is not linearizable – this can be seen more clearly in the following graph, which projects out the pr and val relations. Here lin is the linearization order forced by LIFO ordering. The red edges form a cycle, contradicting the requirement that linearization order includes pr.



However, if for any $i \in \{a, b, c, d\}$ the corresponding `push(`i`)`–`pop(`i`)` pair is deleted, the execution becomes linearizable. Intuitively, doing this breaks the cycle in lin∪pr that appears above. Thus, any forbidden shape based on precedence that is smaller than this whole execution cannot forbid it – otherwise it would forbid legitimate executions. Worse, we can make arbitrarily-large bad executions of this form. Thus no theorem based on finite-size forbidden shapes can define linearizability for stacks. Our insert-remove order introduces just enough extra structure to let us define a local stack theorem.

This kind of execution is not a problem for queues because the insert-remove order does not affect the order of other operations. Ordering an insert-remove pair cannot constrain the insert-insert or remove-remove order of any pair.

D. SP Buffer

An SP buffer consists of a singly-linked list of nodes which is accessed by a `top` pointer. The `top` pointer is annotated with ABA-counters to avoid the ABA-problem [12]. Each node contains a `next` pointer, a `data` field, and a `taken` flag. The `next` pointer points to the next node in the list, the `data` field stores the element, and the `taken` flag indicates if the element of the node has been removed from the SP buffer.

The singly-linked list is closed at its ends by a node which points to itself with the `next` pointer. The list is initialized with a sentinel node. Initially the `top` pointer of the SP buffer and the `next` pointer of the sentinel node point to the sentinel node. The `taken` flag is initialised to `false` indicating that the sentinel node does not contain an element.

An element is contained in the SP buffer if (1) there exists a node in the list that contains the element in its `data` field, if (2) the `taken` flag of that node is not set, if (3) the node

Listing 3: SP buffer algorithm.

```
1 SPBuffer {
2  Node {
3   Node *next,
4   TimestampedItem item,
5   bool taken
6  };
7  <Node*, int> *top; // Pointer with ABA counter.
8  void init() {
9   Node *sentinel =
10    createNode(data=0, taken=true);
11  sentinel.next = sentinel;
12  top = <sentinel, 0>;
13 }
14 void insSP(TimestampedItem item) {
15  Node *newNode = createNode(item=item,
16                             taken=false);
17  <Node*, int> <topMost, topAba> = top;
18  while (topMost->next != topMost
19    && topMost->taken) {
20   topMost = topMost->left;
21  }
22  newNode->next = topMost;
23  top = <newNode, topAba+1>;
24 }
25 <Node*, Node*, int> getSP() {
26  <Node*, int> <oldTop, topAba> = top;
27  Node* result = oldTop;
28  while (true) {
29   if(!result->taken)
30     return <result, oldTop, topAba>;
31   else if (result->next == result)
32    return <NULL, oldTop, topAba>;
33   else
34    result = result->next;
35  }
36 }
37 bool tryRemSP
38   (<Node*, int> <oldTop, aba>, Node *node) {
39  if (CAS(node->taken, false, true)) {
40   CAS (top, <oldTop, aba>, <node, aba>);
41   return true;
42  }
43  return false;
44 }
45 }
```

is reachable from the `top` pointer of the SP buffer following `next` pointers. If one of the three conditions does not hold, the element is not considered as contained in the SP buffer.

The nodes in the list are sorted by their insertion time. The successor of of any node in the list has been inserted earlier than the node itself. By using this order on the list it is guaranteed that the youngest element in the buffer is contained in the top-most node which is not marked as taken.

Listing 3 shows the pseudocode of the SP buffer. To insert an element at the top of the buffer with an `insSP` operation, first a new node is created with the element stored in its `data` field. Initially the `taken` flag is not set. The `insSP` operation then tries to find the top-most node that has not been marked as taken (line 17-21). In line 22-23 the new node is inserted right before that right-most node in the list. If the SP buffer is empty, then the new node is inserted right before the sentinel node.

The `getSP` operation iterates over the list (line 28-35) and returns the first element which has not been marked as taken (line 30). If the iteration reaches the sentinel node `getTl` returns `NULL` (line 32). Additionally `getSP` returns the value of the `top` pointer at the beginning of its execution. The value of the `top` pointer is then used in line 40 in the second `CAS` of the `tryRemSP` operation and in the emptiness check of the TS buffer.

The `tryRemSP` operation tries to set the `taken` flag with a `CAS` (line 39) and returns `true` if it succeeds. Otherwise the `tryRemTl` operation returns `false`. After succeeding in the first `CAS` the operation additionally tries to adjust the `top` pointer of the SP buffer with a `CAS` (line 40. The purpose of that `CAS` is an optimization which is described below.

Correctness. The correctness of the SP buffer is based on the invariant that the list is sorted by the insertion time of its nodes. Thereby the youngest element in the SP buffer can be found simply by finding the top-most node in the list. As the SP buffer allows only a single thread to insert elements, we do not have to care about concurrent `insSP` operations. The atomicity of the `tryRemSP` operations is guaranteed by using the `taken` flag to mark the element of a node logically as removed. Setting the `taken` flag is done atomically with a `CAS` instruction.

The correctness of the `getSP` operation is more subtle. If at the time `getSP` returns an element the returned element is indeed the youngest element in the SP buffer, then the `return` statement is a correct linearization point of `getSP`. If during the iteration through the list a new element has been inserted into the list, then we use the following insight: If `getSP` returns an element which is not youngest element at the time of its `return` statement, then its iteration started before the element got inserted. Therefore these `getSP` can be linearized right before the linearization point of the `insSP` operation which inserted a new element in the meantime. For the same reasons `getSP` is correct when it returns `NULL`.

The second CAS in `tryRemSP` is an optimization. For the performance of `insSP` and `getSP` it is good to have as few taken nodes at the top of the SP buffer as possible. In both line 23 and line 40 the `top` pointer is changed to reduce the number of taken nodes in the SP buffer. To guarantee the correctness of the optimization the `top` pointer is changed in `tryRemSP` only if the top pointer has not been changed since the remove node got returned by `getSP` This is guaranteed by the semantics of `CAS` and the use of an ABA-counter. In `insSP` such a precaution is not necessary. If `top` is changed during the execution of `insSP` after line 17, then this change is undone in line 23. Undoing these changes is allowed as only a single thread (the thread which executes `insSP`) is allowed to insert into the SP buffer and therefore all these changes were done within `tryRemSP` operations. Changes of the `top` pointer in `tryRemSP`, however, could have failed already in the first place.

E. Proof details for Lemma 2

In this section we provide additional details supplementing the proof of Lemma 2 given in Appendix A.

Lemma 7 (abcd)**.** *Let $\langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ be a history derived from a trace $\mathcal{T}$. Given actions $a, b, c, d \in \mathcal{A}$, if $a \xrightarrow{\mathsf{pr}} b$ and $c \xrightarrow{\mathsf{pr}} d$, then either $a \xrightarrow{\mathsf{pr}} d$ or $c \xrightarrow{\mathsf{pr}} b$.*

Proof. Case-split on whether $a_{\mathsf{ret}} \xrightarrow{\mathcal{T}} d_{\mathsf{inv}}$. If so, $a \xrightarrow{\mathsf{pr}} d$. Otherwise, because $\mathcal{T}$ totally orders events, $d_{\mathsf{inv}} \xrightarrow{\mathcal{T}} a_{\mathsf{ret}}$. The premise gives us $c_{\mathsf{ret}} \xrightarrow{\mathcal{T}} d_{\mathsf{inv}} \xrightarrow{\mathcal{T}} a_{\mathsf{ret}} \xrightarrow{\mathcal{T}} b_{\mathsf{inv}}$, which gives us $c \xrightarrow{\mathsf{pr}} b$ by transitivity. □

Lemma 8. *Let $+a, +b \in \mathcal{A}$ be two* `push` *operations. If $+a \xrightarrow{\mathsf{ins}} +b$, then also $+a \xrightarrow{\mathsf{ts}} +b$.*

Proof. Assume $+a, +b \in \mathcal{A}$ and $+a \xrightarrow{\mathsf{ins}} +b$. Therefore either $+a \xrightarrow{\mathsf{pr}} +b$ or there exists a $-c$ with $+a \xrightarrow{\mathsf{pr}} -c \xrightarrow{\mathsf{vis}} +b$. Both these conditions are also preconditions of the definition of ts, which completes the proof. □

As we described in Appendix B we showed that we can ignore all `push-pop` pairs which have overlapping execution times. The proof is based on the insight that these `push-pop` pairs can always be added to a linearization of the other operations without introduction a violation of stack semantics. Therefore we assume in the following that for any $+a, -a \in \mathcal{A}$, if $+a \xrightarrow{\mathsf{val}} -a$, then also $+a \xrightarrow{\mathsf{pr}} -a$.

Next we assume the case where there exists a violating triple $(-a, +c, -b)$, and for all violating triples $(-a, +c_i, -b)$ there exists a $-c_i \in \mathcal{A}$ with $+c_i \xrightarrow{\mathsf{val}} -c_i$ and $-c_i \xrightarrow{\mathsf{rr}} -a$. We show that vis can be adjusted, resulting in a $\mathsf{vis'}$ which differs from vis only in ordering $+c \xrightarrow{\mathsf{vis}'} -a$ instead of $-a \xrightarrow{\mathsf{vis}} +c$, and which satisfies the following invariants (these are the original assumptions on vis):

- $\mathsf{pr} \cup \mathsf{vis}'$ is cycle-free.
- Let ts' be the ts relation derived from vis'. Let $+d, -d, +e \in \mathcal{A}$ with $+d \xrightarrow{\mathsf{val}} -d$, $+d \xrightarrow{\mathsf{pr}} -d$, and $+d \xrightarrow{\mathsf{ts}'} +e \xrightarrow{\mathsf{vis}'} -d$, then there exists a $-e \in \mathcal{A}$ with $+e \xrightarrow{\mathsf{val}} -e$ and $-e \xrightarrow{\mathsf{rr}} -d$.

Lemma 9. $\mathsf{pr} \cup \mathsf{vis}'$ *is cycle-free.*

Proof. If there exists a cycle in $\mathsf{pr} \cup \mathsf{vis}'$, then it has to contain the edge $+c \xrightarrow{\mathsf{vis}'} -a$ since all other edges existed already in vis and we assumed that $\mathsf{pr} \cup \mathsf{vis}$ is cycle-free. Therefore there has to exist a transitive edge from $-a$ to $+c$ in $\mathsf{pr} \cup \mathsf{vis}$. The transitivity of pr, and vis being alternating means that either

1. $-a \xrightarrow{\mathsf{pr}} +c$,
2. $-a \xrightarrow{\mathsf{vis}} +f \xrightarrow{\mathsf{pr}} +c$ for some $+f \in \mathcal{A}$,
3. $-a \xrightarrow{\mathsf{vis}} +f \xrightarrow{\mathsf{vis}} -e \xrightarrow{\mathsf{vis}} +c$ for some $+f, -e \in \mathcal{A}$, or
4. $-a \xrightarrow{\mathsf{pr}} -e \xrightarrow{\mathsf{vis}} +c$ for some $-e \in \mathcal{A}$.

Prove by case distinction. Assume item 1 – then there would exist the cycle $-a \xrightarrow{\mathsf{pr}} +c \xrightarrow{\mathsf{pr}} -c \xrightarrow{\mathsf{rr}} -a$, which would violate the assumption that $\mathsf{pre} \cup \mathsf{rr}$ is cycle-free.

Assume item 2 – then there exists a `push` operation $+f$ with $-a \xrightarrow{\mathsf{vis}} +f \xrightarrow{\mathsf{pr}} +c$. Since $\mathsf{pr} \cup \mathsf{vis}$ is cycle-free it holds that $+f \xrightarrow{\mathsf{vis}} -b$ and also $(-a, +f, -b)$ is a violating triple. According to our assumption it holds that there exists a $-f \in \mathcal{A}$ with $-f \xrightarrow{\mathsf{rr}} -a$. We only consider finite histories, therefore there exists a $+f' \in \mathcal{A}$ for which no such $+f$ exists, and we could deal with the violating triple $(-a, +f', -b)$ before we deal with the other violating triples. Eventually all violating triples can be resolved.

For item 3 apply the same argument as item 2.

Finally assume that item 4 holds and there exists a $-e \in \mathcal{A}$ with $-a \xrightarrow{\mathsf{pr}} -e \xrightarrow{\mathsf{vis}} +c$. We assume that there exists a $-c$ with $+c \xrightarrow{\mathsf{val}} -c$ and therefore $+c \xrightarrow{\mathsf{pr}} -c$. Applying Lemma 7 means that either $-a \xrightarrow{\mathsf{pr}} -c$ or $+c \xrightarrow{\mathsf{pr}} -e$. With the former there exists the cycle $-a \xrightarrow{\mathsf{pr}} -c \xrightarrow{\mathsf{rr}} -a$, and with the later there exists the cycle $+c \xrightarrow{\mathsf{pr}} -e \xrightarrow{\mathsf{vis}} +c$, both violating our assumptions that $\mathsf{pr} \cup \mathsf{rr}$ and $\mathsf{pr} \cup \mathsf{vis}$ are cycle-free. Therefore $\mathsf{vis'}$ is cycle-free. □

Lemma 10. *Let ts' be the ts relation derived from vis'. Let $+d, -d, +e \in \mathcal{A}$ with $+d \xrightarrow{\mathsf{val}} -d$, $+d \xrightarrow{\mathsf{pr}} -d$, and $+d \xrightarrow{\mathsf{ts}'} +e \xrightarrow{\mathsf{vis}'} -d$, then there exists a $-e \in \mathcal{A}$ with $+e \xrightarrow{\mathsf{val}} -e$ and $-e \xrightarrow{\mathsf{rr}} -d$.*

Proof. The current lemma holds for vis, and the only pair which is different in vis and in $\mathsf{vis'}$ is the $+c \xrightarrow{\mathsf{vis}'} -a$. Therefore we only have to show that $+c \xrightarrow{\mathsf{vis}'} -a$ is not part of a violation of this lemma. There exist two ways how $+c \xrightarrow{\mathsf{vis}'} -a$ could be part of a violation:

1. $+a \xrightarrow{\mathsf{val}} -a$, $+a \xrightarrow{\mathsf{pr}} -a$, $+a \xrightarrow{\mathsf{ts}'} +c \xrightarrow{\mathsf{vis}'} -a$, and either no $-c \in \mathcal{A}$ exists with $+c \xrightarrow{\mathsf{val}} -c$ or such a $-c$ exists and $-a \xrightarrow{\mathsf{rr}} -c$. This cannot be true because we assume that there exists a $-c \in \mathcal{A}$ with $+c \xrightarrow{\mathsf{val}} -c$ and $-c \xrightarrow{\mathsf{rr}} -a$.
2. There exist $+e, +f, -e \in \mathcal{A}$ with $+e \xrightarrow{\mathsf{val}} -e$, $+e \xrightarrow{\mathsf{pr}} -e$, $+e \xrightarrow{\mathsf{pr}} +c \xrightarrow{\mathsf{vis}'} -a \xrightarrow{\mathsf{vis}'} +f$ and therefore $+e \xrightarrow{\mathsf{ts}'} +f$, $+f \xrightarrow{\mathsf{vis}'} -e$, and either there does not exist a $-f \in \mathcal{A}$ with $+f \xrightarrow{\mathsf{val}} -f$ or such a $-f$ exists and $-e \xrightarrow{\mathsf{rr}} -f$.
   It cannot be the case that $+e \xrightarrow{\mathsf{ts}} +f$ because then the violation would have already existed in vis. As $-b \xrightarrow{\mathsf{vis}} +f$ would, however, construct $+e \xrightarrow{\mathsf{ts}} +f$, we know that $+f \xrightarrow{\mathsf{vis}} -b$. Therefore $(-a, +f, -b)$ is also a violating triple and could be dealt with before $(-a, +c, -b)$.

Therefore with both cases no violation is possible, which completes the proof. □

Next we assume the case where all violating triples $(-a, +c, -b)$ are resolved for which it holds that for all violating triples $(-a, +c_i, -b)$ there exists a $-c_i \in \mathcal{A}$ with $+c_i \xrightarrow{\mathsf{val}} -c_i$ and $-c_i \xrightarrow{\mathsf{rr}} -a$. Therefore it holds that for any remaining violating triple $(-a, +c, -b)$ there exists a violating triple $(-a, +d, -b)$ such that either there does not exist a $-d \in \mathcal{A}$ with $+d \xrightarrow{\mathsf{val}} -d$, or there exists a $-d$ and $-a \xrightarrow{\mathsf{rr}} -d$. We show that vis can then be adjusted such that $+b$ is ordered the same as $+d$ in the adjusted vis'', and all other operations are ordered the same as in vis. We prove now that the adjusted vis'' relation satisfies the following invariants (the same properties which hold for the final $\mathsf{vis'}$):

- $\mathsf{pr} \cup \mathsf{vis}''$ is cycle-free.
- Let ts'' be the ts relation derived from vis''. Let $+s, -s, +e \in \mathcal{A}$ with $+s \xrightarrow{\mathsf{val}} -s$, $+s \xrightarrow{\mathsf{pr}} -s$, and $+s \xrightarrow{\mathsf{ts}''} +e \xrightarrow{\mathsf{vis}''} -s$, then there exists a $-e \in \mathcal{A}$ with $+e \xrightarrow{\mathsf{val}} -e$ and $-e \xrightarrow{\mathsf{rr}} -d$.
- Let $(-s, +f, -e)$ be a violating triple for vis'', then there exists a violating triple $(-s, +f', -e)$ such that there does not exist a $-f' \in \mathcal{A}$ with $+f' \xrightarrow{\mathsf{val}} -f'$, or there exists such a $-f' \in \mathcal{A}$ and $-s \xrightarrow{\mathsf{rr}} -f'$.

Lemma 11. $\mathsf{pr} \cup \mathsf{vis}''$ *is cycle-free.*

Proof. Assume a cycle exists in $\mathsf{pr} \cup \mathsf{vis}''$, then the cycle has to contain $+b$. As $+b$ is ordered the same as $+d$ in vis'', and vis is cycle-free, there exists either a $-f \in \mathcal{A}$

with $+b \xrightarrow{\mathsf{pr}} -f \xrightarrow{\mathsf{vis}} +d$, or there exist $+g, -f \in \mathcal{A}$ with $+b \xrightarrow{\mathsf{pr}} +g \xrightarrow{\mathsf{vis}} -f \xrightarrow{\mathsf{vis}} +d$. In both cases $+b \xrightarrow{\mathsf{ts}} +d$ and therefore $+b \xrightarrow{\mathsf{ts}} +d \xrightarrow{\mathsf{vis}} -b$ and therefore according to our assumptions there exists a $-d \in \mathcal{A}$ with $-d \xrightarrow{\mathsf{rr}} -b$. By the transitivity of rr it holds that $-d \xrightarrow{\mathsf{rr}} -a$, which contradicts our assumptions. Therefore no cycle in $\mathsf{pr} \cup \mathsf{vis}''$ is possible. □

For the other two assumptions we first show a helping lemma. In the following, let ts'' be the ts relation derived from vis''.

Lemma 12. *Let $+s, -s, +e \in \mathcal{A}$ with $+s \xrightarrow{\mathsf{val}} -s$, $+s \xrightarrow{\mathsf{pr}} -s$, and $+s \xrightarrow{\mathsf{ts}''} +e \xrightarrow{\mathsf{vis}''} -s$, then either also $+s \xrightarrow{\mathsf{ts}} +e \xrightarrow{\mathsf{vis}} -s$, or $+e = +b$ and $+s \xrightarrow{\mathsf{ts}} +d \xrightarrow{\mathsf{vis}} -s$.*

Proof. First we observe that the only difference between vis and vis'' is that $+b$ has been moved backwards. Therefore it holds for any $+g, -h \in \mathcal{A}$ that if $+g \xrightarrow{\mathsf{vis}''} -h$, than also $+g \xrightarrow{\mathsf{vis}} -h$. Therefore, if $+s \xrightarrow{\mathsf{ts}''} +e \xrightarrow{\mathsf{vis}''} -s$, it also holds that $+s \xrightarrow{\mathsf{ts}''} +e \xrightarrow{\mathsf{vis}} -s$, and in the case of $+e = +b$ also $+d \xrightarrow{\mathsf{vis}} -s$. Therefore it only remains to show that if $+s \xrightarrow{\mathsf{ts}''} +e$, then either also $+s \xrightarrow{\mathsf{ts}} +e$, or $+e = +b$ and $+s \xrightarrow{\mathsf{ts}} +d$. We do a case distinction on the definition of ts''.

- if $+s \xrightarrow{\mathsf{pr}} +e$, then also $+s \xrightarrow{\mathsf{ts}} +e$.
- if $+s \xrightarrow{\mathsf{pr}} -k \xrightarrow{\mathsf{vis}''} +e$ for some $-k \in \mathcal{A}$, and $+e \neq +b$, then also $-k \xrightarrow{\mathsf{vis}} +e$ because vis and vis'' only differ in the order of $+b$, and therefore $+s \xrightarrow{\mathsf{ts}} +e$. If $+e = +b$, then $+d$ is ordered the same in vis as $+b$ in vis'' and therefore $+s \xrightarrow{\mathsf{pr}} -k \xrightarrow{\mathsf{vis}} +d$ and $+s \xrightarrow{\mathsf{ts}} +d$.
- if $+s \xrightarrow{\mathsf{pr}} +l \xrightarrow{\mathsf{vis}''} -k \xrightarrow{\mathsf{vis}''} +e$ for some $+l, -k \in \mathcal{A}$, then, as we observed already before, it holds that $+s \xrightarrow{\mathsf{pr}} +l \xrightarrow{\mathsf{vis}} -k \xrightarrow{\mathsf{vis}''} +e$. If $+e \neq +b$, then also $-k \xrightarrow{\mathsf{vis}} +e$ because vis and vis'' only differ in the order of $+b$, and therefore $+s \xrightarrow{\mathsf{ts}} +e$. If $+e = +b$, then $+d$ is ordered the same in vis as $+b$ in vis'' and therefore $+s \xrightarrow{\mathsf{pr}} +l \xrightarrow{\mathsf{vis}} -k \xrightarrow{\mathsf{vis}} +d$ and $+s \xrightarrow{\mathsf{ts}} +d$.

□

Lemma 13. *Let ts'' be the ts relation derived from vis''. Let $+s, -s, +e \in \mathcal{A}$ with $+s \xrightarrow{\mathsf{val}} -s$, $+s \xrightarrow{\mathsf{pr}} -s$, and $+s \xrightarrow{\mathsf{ts}''} +e \xrightarrow{\mathsf{vis}''} -s$, then there exists a $-e \in \mathcal{A}$ with $+e \xrightarrow{\mathsf{val}} -e$ and $-e \xrightarrow{\mathsf{rr}} -d$.*

Proof. Let $+s, -s, +e \in \mathcal{A}$ with $+s \xrightarrow{\mathsf{val}} -s$, $+s \xrightarrow{\mathsf{pr}} -s$, and $+s \xrightarrow{\mathsf{ts}''} +e \xrightarrow{\mathsf{vis}''} -s$. Assume $+e \neq +b$, then according to Lemma 12 also $+s \xrightarrow{\mathsf{ts}} +e \xrightarrow{\mathsf{vis}} -s$, and according to our assumptions there exists a $-e \in \mathcal{A}$ with $+e \xrightarrow{\mathsf{val}} -e$ and $-e \xrightarrow{\mathsf{rr}} -s$.

Next assume that $+e = +b$. We have to show that $-s \xrightarrow{\mathsf{rr}} -b$ is impossible. We know from Lemma 12 that $+s \xrightarrow{\mathsf{ts}} +d \xrightarrow{\mathsf{vis}} -s$, and therefore there exists a $-d \in \mathcal{A}$ with $-d \xrightarrow{\mathsf{rr}} -s$. From our assumptions we know that $(-a, +d, -b)$ is a violating triple of vis and therefore $-b \xrightarrow{\mathsf{rr}} -a$, and also that $-a \xrightarrow{\mathsf{rr}} -d$. Now, if $-s \xrightarrow{\mathsf{rr}} -b$ would hold, then there existed the cycle $-s \xrightarrow{\mathsf{rr}} -b \xrightarrow{\mathsf{rr}} -a \xrightarrow{\mathsf{rr}} -d \xrightarrow{\mathsf{rr}} -s$, which contradicts our assumptions. Therefore $-b \xrightarrow{\mathsf{rr}} -s$, which completes the proof.

□

We also need to preserve the condition after the first proof step:

Lemma 14. *Let $(-s, +f, -e)$ be a violating triple for vis'', then there exists a violating triple $(-s, +f', -e)$ such that there does not exist a $-f' \in \mathcal{A}$ with $+f' \xrightarrow{\mathsf{val}} -f'$, or there exists such a $-f' \in \mathcal{A}$ and $-s \xrightarrow{\mathsf{rr}} -f'$.*

Proof. Let $(-s, +f, -e)$ be a violating triple for vis''. Therefore it holds that there exist $+s, +e \in \mathcal{A}$ with $+s \xrightarrow{\mathsf{val}} -s$, $+s \xrightarrow{\mathsf{pr}} -s$, and $+s \xrightarrow{\mathsf{ts}''} +e \xrightarrow{\mathsf{vis}''} -s \xrightarrow{\mathsf{vis}''} +f \xrightarrow{\mathsf{vis}''} -e$. If $+e \neq +b$, then according to Lemma 12 it holds that $+s \xrightarrow{\mathsf{ts}} +e \xrightarrow{\mathsf{vis}} -e$. If $+f \neq +b$, then also $-s \xrightarrow{\mathsf{vis}} +f \xrightarrow{\mathsf{vis}} -e$ and therefore there exists a $+f' \in \mathcal{A}$ such that $(-s, +f', -e)$ is a violating triple of vis and also of vis'', and either there exists no $-f' \in \mathcal{A}$ with $+f' \xrightarrow{\mathsf{val}} -f'$, or there exists such a $-f'$ and $-s \xrightarrow{\mathsf{rr}} -f'$.

Next assume $+f = +b$, then also $-s \xrightarrow{\mathsf{vis}''} +b \xrightarrow{\mathsf{vis}''} -e$ and therefore also $-s \xrightarrow{\mathsf{vis}} +d \xrightarrow{\mathsf{vis}} -e$. This means that $(-s, +d, -e)$ is a violating triple of vis, and therefore there has to exist a $+f' \in \mathcal{A}$ such that $(-s, +f', -e)$ is a violating triple of vis and also of vis'', and either there exists no $-f' \in \mathcal{A}$ with $+f' \xrightarrow{\mathsf{val}} -f'$, or there exists such a $-f'$ and $-s \xrightarrow{\mathsf{rr}} -f'$.

Finally assume that $+e = +b$. Therefore it holds according to Lemma 12 that $+s \xrightarrow{\mathsf{ts}} +d \xrightarrow{\mathsf{vis}} -s$. If $-s \xrightarrow{\mathsf{vis}''} +f \xrightarrow{\mathsf{vis}''} -b$, then also $-s \xrightarrow{\mathsf{vis}} +f \xrightarrow{\mathsf{vis}} -d$ and there exists a $+f' \in \mathcal{A}$ such that $(-s, +f', -d)$ is a violating triple of vis, and either there exists no $-f' \in \mathcal{A}$ with $+f' \xrightarrow{\mathsf{val}} -f'$, or there exists such a $-f'$ and $-s \xrightarrow{\mathsf{rr}} -f'$. This violating triple has not been resolved in vis'' and therefore also $(-s, +f', -b)$ is a violating triple of vis'', which completes the proof.

□

F. Proof of correctness for the TS buffer

Theorem 6. *The TS buffer implementation is linearizable with respect to the specification* TSBUF *given in §5.*

Proof. Assume a concrete pre-state L and an abstract pre-state (B, S). We prove the correctness of each method separately:

`newTimeStamp`() generates a timestamp greater than those in the buffer because all such timestamps were originally generated by `newTimeStamp` and added by `setTimestamp`.

The linearization point of `ins(e)` is the call to `insSP`. This call inserts the new element into the SP buffer and thereby also to the abstract state of the TS buffer.

`setTimestamp(i,t)` consists of a single assignment which is its linearization point.

`getStart()` consists of a single call to the `getMaxTimestamp`, which is its linearization point. There are two cases for the generated timestamp: either $\mathtt{t} \in \mathrm{dom}(S)$ or not. In the former case, `t` was generated by a previous calls to `getStart`, and there must have been no intervening methods calls, meaning that $B = S(\mathtt{t})$ as required. Otherwise a new snapshot is generated in S.

The case of `tryRem()` returning INVALID is trivially correct and the correctness of `tryRem()` returning EMPTY has been shown in [8]. In normal removal (case 3) and elimination (case 4), the linearization points are successful calls to `tryRemSP`. Both cases remove an element atomically from the concrete state, so we only need to show that the correct element is removed.

First observe that any element in the snapshot $S(\mathtt{t})$ must have been in a SP buffer before `tryRem` begins, and that elements that are not in $S(\mathtt{t})$ are guaranteed to have timestamps younger than `t`.

Suppose that `tryRem` removes an element on line 36 in listing 2. Comparison with `startTime` on line 24 ensures that elements not in the snapshot are not considered for normal removal. Suppose `tryRem` removes an element a although there exists an element b in the snapshot and in B with $a <_{\mathsf{TS}} b$. Therefore b is contained in an SP buffer and `getSP` must have returned b or some even younger element c. In this case, b or c would be considered younger than a by the comparison code, and either removed or eliminated, contradicting our assumption. Note that b would also have been considered younger if b was timestamped after it got returned by `getSP`.

Suppose alternatively that `tryRem` removes an element on line 24. If $a \notin \mathrm{dom}(S(\mathtt{t}))$, then the removal of a is correct because of elimination. If $a \in \mathrm{dom}(S(\mathtt{t}))$, then the timestamp of a was generated after the snapshot or the timestamp was assigned to a after the snapshot. Therefore the timestamp of a in the snapshot is $\top$ and the removal is correct because of elimination. □