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 if their associated insert operations ran concurrently, and order imposed as necessary at the eventual removal.

We realise our approach in a new non-blocking data-structure, the TS (timestamped) stack. Using the same approach, we can define corresponding queue and deque data-structures. 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 linearizable. Our theorem constitutes a new, generic proof technique for concurrent stacks, and it paves the way for future weakly ordered data-structure designs.

Categories and Subject Descriptors D.1.3 [*Programming languages*]: Programming techniques – concurrent programing; E.1 [*Data Structures*]: Lists, stacks, and queues; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords concurrent stack; linearizability; timestamps; verification

POPL '15, January 15–17, 2015, Mumbai, India.
Copyright © 2015 ACM 978-1-4503-3300-9/15/01. . . $15.00.
http://dx.doi.org/10.1145/2676726.2676963

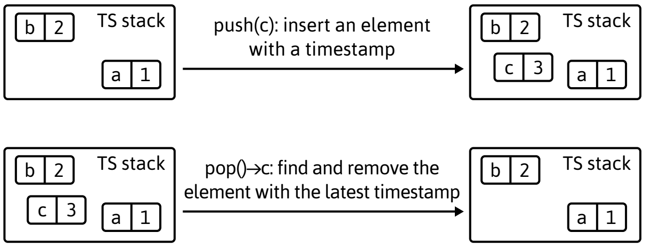

b 2
TS stack
a 1
push(c): insert an element with a timestamp
b 2
TS stack
c 3
a 1
b 2
TS stack
c 3
a 1
pop()→c: find and remove the element with the latest timestamp
b 2
TS stack
a 1

Figure 1: TS stack push and pop operations.

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 youngest timestamp, or as a queue by removing the element with the oldest 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 variant (the TS queue / deque variants are discussed briefly in §7). The TS stack push and pop are illustrated in Figure 1.

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 [14]. 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 can 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 pools. Such pools 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 [12]. Rather than a total order, we need only generate an order from push to pop operations, and vice versa, which avoids certain violations. This order *can* be generated from syntactic points in the TS stack code, allowing us to show that it is correct. Our stack theorem is generic, not tied to the TS stack. By generalising away from linearization points, it paves the way for future concurrent data-structures which weaken internal ordering.

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 stacks.

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 on the ACM digital library, and are also available from the TS stack webpage:

`http://scal.cs.uni-salzburg.at/tsstack/`

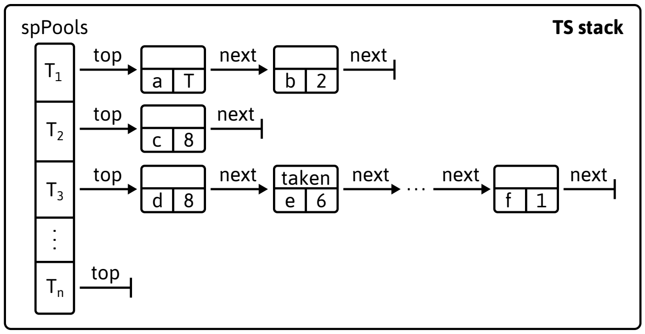


Figure 2: The TS stack data-structure.

Paper structure. §2 describes the key ideas behind the TS stack, then in §3 we describe the algorithm in detail. In §4 we describe our proof technique, while in §5 we use it to establish that the TS stack is linearizable. In §6 we discuss our experiments. §7 discusses TS queue and deque variants. §8 surveys the related work. §9 concludes.

Longer proofs and other auxiliary material are included in supplementary appendices, available on the ACM digital library. Appendix A discusses the Isabelle proof of our core stack theorem. Appendix B gives further details about our TS stack linearizability proof. Appendix C gives a proof of correctness for our intermediate TS buffer data-structure.

2. Key Ideas

Algorithm structure. The TS stack marks elements with timestamps recording the order they were pushed. Elements are popped according to this timestamp order. Figure 2 shows the stack's internal structure. Each thread T_1–T_n which is accessing the stack has an associated single-producer multi-consumer pool, implemented as a linked list (we call these *SP pools*). These are linked from a common array `spPools`. Every element on the stack is stored in the SP pool of the thread that pushed it.

Pushing to the TS stack involves (1) adding a node to the head of the thread's SP pool, (2) generating a new timestamp, and (3) attaching the timestamp to the node. Thus un-timestamped nodes are visible to other threads – we write these in Figure 2 as the maximal value, ⊤. As nodes are only added by one thread to each SP pool, elements in a single pool are totally ordered by timestamp, and no synchronisation is needed when pushing.

Popping from the TS stack involves (1) searching all the SP pools for an element with a maximal timestamp, and (2) attempting to remove it. This process repeats until an element is removed successfully. As each SP pool is ordered, searching only requires reading the head of each pool in turn. Timestamps in different pools may be mutually unordered – for example when two timestamps are equal. Thus more than one element may be maximal, and in this case, either can be chosen. To remove a node, the thread writes to a flag in the node marking it as taken. Multiple threads may try to remove the same node, so an atomic compare-and-swap (CAS) instruction ensures at most one thread succeeds.

Accessing the heads of multiple per-thread pools 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 unnecessary increments by using a compare-and-swap to detect when the counter has already been incremented. We can avoid strong synchronisation entirely by using a vector of thread-local counters, meaning the counter may stutter (many elements 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. Our benchmarks show that hardware timestamping provides the best push performance. However, the picture is more complicated in the presence of optimisation. See §6 for our experiments.

Optimisations. Timestamping enables several optimisations of the TS stack, most importantly elimination (a standard strategy in the literature), and interval timestamping (a 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 time 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, the algorithm includes a delay 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 §6.2). Though pausing may appear as an unnecessary overhead to a push, our experiments show that optimal delay times (4 µs - 8 µs) are actually shorter than e.g. an atomic fetch-and-inc on a contended memory location. 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 (§6.1) 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 separate back-off code.

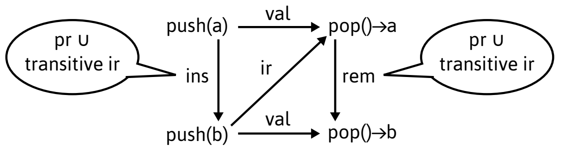


Figure 3: Non-LIFO behaviour forbidden by stack theorem.

Algorithm correctness. Intuitively, the TS stack is correct because any two push operations that run sequentially receive ordered timestamps, and are therefore removed in LIFO order. Elements arising from concurrent push operations may recieve unordered timestamps, and may be removed in either order, but this does not affect correctness.

To formally prove that a stack is correct, i.e. linearizable with respect to stack semantics, we need to show that for any execution there exists a total linearization order. However, proving this 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 pushes.

Instead, we use a new *stack theorem* which removes the need to find a total linearization order. Intuitively, our theorem requires that if two elements are on the stack, then the younger element is popped first (i.e. LIFO ordering).

It is important that all operations take a consistent view on which elements are on the stack. To express this, our stack theorem require a relation which states whether a push takes logical effect before or after a pop. We call this ir, for 'insert-remove'. The ir relation is the crux of the linearization order: surprisingly, our theorem shows that it is the only part that matters. Thus, the stack theorem relieves us of the need to resolve problematic ordering between parallel push or parallel pop operations.

The stack theorem also uses two other relations: *precedence*, pr, which relates methods that run in sequence; and *value*, val, which relates a push to the pop removing the associated value. Loosely, the theorem has the following form:

> *If for every execution there exist* ir*,* pr*,* val *relations that are order-correct, then the algorithm is linearizable with respect to sequential stack semantics.*[1]

Order-correctness rules out non-LIFO behaviour. Intuitively, the situation shown in Figure 3 is forbidden: if push(a) and push(b) are ordered, and push(b) is related to pop()→a in ir, then the two pops cannot also be ordered pop()→a before pop()→b – this would violate LIFO ordering.

As pr and val can easily be extracted from an execution, establishing linearizability amounts to showing the existence of a consistent ir. This is analogous to finding the linearization order, but ir can be constructed much more easily for the TS stack. We use a modified version of the linearization point method, but rather than a single order, we identify two and combine them to build ir.

- vis (for 'visibility'). A push(a) and pop are ordered by vis if the value a inserted by push was in a SP pool when the pop started, and so could have been visible to it.

[1] Additionally, the full theorem requires the algorithm is linearizable with respect to sequential set semantics. This guarantees non-LIFO properties such as absence of duplication.

- rr (for ‘remove-remove’). Two operations pop()→a and pop()→b are ordered in rr if elements a and b are removed in order from the underlying SP pools.

Building ir is more than just merging these two relations because they may contradict one another. Instead, ir is built by taking vis as a core, and using rr to correct cases that contradict LIFO order. Our proof of correctness (§5) shows that this is always possible, which establishes that the TS stack is linearizable.

3. The TS Stack in Detail

We now present our TS stack algorithm in detail. Listing 1 shows the TS stack code. This code manages the collection of thread-specific SP pools linked from the array spPools. We factor the SP pool out as a separate data-structure supporting the following operations:

- insert – insert an element without attaching a timestamp, and return a reference to the new node.
- getYoungest – return a reference to the node in the pool with the youngest timestamp, together with the top pointer of the pool.
- remove – tries to remove the given node from the pool. Return true and the element of the node or false and NULL depending whether it succeeds.

We describe our implementation of the SP pool in §3.1. Listing 1 also assumes the timestamping function newTimestamp – various implementations are discussed in §3.2.

We can now describe Listing 1. To push an element, the TS stack inserts an un-timestamped element into the current thread's pool (line 13), generates a fresh timestamp (line 14), and sets the new element's timestamp (line 15).

A pop iteratively scans over all SP pools (line 35-54) and searches for the node with the youngest timestamp in all SP pools (line 48-53). The binary operator $<_{\mathsf{TS}}$ is timestamp comparison. This is just integer comparison for non-interval timestamps – see §3.2. If removing the identified node succeeds (line 63) then its element is returned (line 26). Otherwise the iteration restarts.

For simplicity, in Listing 1 threads are associated statically with slots in the array spPools. To support a dynamic number of threads, this array can be replaced by a linked list for iteration in pop, and by a hashtable or thread-local storage for fast access in push.

Elimination and emptiness checking. Code in gray in Listing 1 handles elimination and emptiness checking.

Elimination [9] is an essential optimisation in making our stack efficient. It is permitted whenever a push and pop execute concurrently. To detect opportunities for elimination, a pop reads the current time when it starts (line 20). When searching through the SP pools, any element with a later timestamp must have been pushed during the current pop, and can be eliminated immediately (lines 46-47).

To check whether the stack is empty, we reuse an approach from [8]. When scanning the SP pools, if a pool indicates that it is empty, then its top pointer is recorded (lines 40-43). If no candidate for removal is found then the SP pools are scanned again to check whether their top pointers have changed (lines 56-60). If not, the pools must have been empty between the first and second scan. The linearizability of this emptiness check has been proved in [8].

Listing 1: TS stack algorithm. The SP pool is defined in Listing 2 and described in §3.1, timestamps are discussed in §3.2. The gray highlighted code deals with the emptiness check and elimination.

```
 1 TSStack{
 2  Node{
 3    Element element,
 4    Timestamp timestamp,
 5    Node next,
 6    Bool taken
 7  };
 8
 9  SPPool[maxThreads] spPools;
10
11  void push(Element element){
12    SPPool pool=spPools[threadID];
13    Node node=pool.insert(element);
14    Timestamp timestamp=newTimestamp();
15    node.timestamp=timestamp;
16  }
17
18  Element pop(){
19    // Elimination
20    Timestamp startTime=newTimestamp();
21    Bool success;
22    Element element;
23    do{
24      <success,element>=tryRem(startTime);
25    } while (!success);
26    return element;
27  }
28
29  <Bool,Element> tryRem(Timestamp startTime){
30    Node youngest=NULL;
31    Timestamp timestamp=-1;
32    SPPool pool;
33    Node top;
34    Node[maxThreads] empty;
35    for each(SPPool current in spPools){
36      Node node;
37      Node poolTop;
38      <node,poolTop>=current.getYoungest();
39      // Emptiness check
40      if(node==NULL){
41        empty[current.ID]=poolTop;
42        continue;
43      }
44      Timestamp nodeTimestamp=node.timestamp;
45      // Elimination
46      if(startTime <TS nodeTimestamp)
47        return current.remove(poolTop,node);
48      if(timestamp <TS nodeTimestamp){
49        youngest=node;
50        timestamp=nodeTimestamp;
51        pool=current;
52        top=poolTop;
53      }
54    }
55    // Emptiness check
56    if(youngest==NULL){
57      for each(SPPool current in spPools){
58        if(current.top!=empty[current.ID])
59          return <false,NULL>;
60      }
61      return <true,EMPTY>;
62    }
63    return pool.remove(top,youngest);
64  }
65 }
```

Listing 2: SP pool algorithm. The gray highlighted code deals with the unlinking of taken nodes.

```
66 SPPool{
67  Node top;
68  Int ID; // The ID of the owner thread.
69
70  init(){
71    Node sentinel=
72      createNode(element=NULL,taken=true);
73    sentinel.next=sentinel;
74    top=sentinel;
75  }
76
77  Node insert(Element element){
78    Node newNode=
79      createNode(element=element,taken=false);
80    newNode.next=top;
81    top=newNode;
82    Node next=newNode.next; // Unlinking
83    while(next->next!=next && next.taken)
84      next=next->next;
85    newNode.next=next;
86    return newNode;
87  }
88
89  <Node,Node> getYoungest(){
90    Node oldTop=top;
91    Node result=oldTop;
92    while(true){
93      if(!result.taken)
94        return <result,oldTop>;
95      else if (result.next==result)
96        return <NULL,oldTop>;
97      result=result.next;
98    }
99  }
100
101  <Bool,Element> remove(Node oldTop, Node node){
102    if(CAS(node.taken,false,true)){
103      CAS(top,oldTop,node); // Unlinking
104      // Unlink nodes before node in the list.
105      if(oldTop!=node)
106        oldTop.next=node;
107      // Unlink nodes after node in the list.
108      Node next=node.next;
109      while(next->next!=next && next.taken)
110        next=next->next;
111      node.next=next;
112      return <true,node.element>;
113    }
114    return <false,NULL>;
115  }
116 }
```

3.1 SP Pool

The SP pool (Listing 2) is a singly linked list of nodes accessed by a `top` pointer. A node consists of a `next` pointer for the linked list, the `element` it stores, the `timestamp` assigned to the element, and a `taken` flag. The singly linked list is closed at its end by a sentinel node pointing to itself (line 73). Initially the list contains only the sentinel node. The `taken` flag of the sentinel is set to `true` indicating that the sentinel does not contain an element. The `top` pointer is annotated with an ABA-counter to avoid the ABA-problem [13].

Elements are inserted into the SP pool by adding a new node (line 78) at the head of the linked list (line 80-81). To remove an element the `taken` flag of its node is set atomically with a `CAS` instruction (line 102). `getYoungest` iterates over the list (line 90-98) and returns the first node which is not marked as taken (line 94). If no such node is found, `getYoungest` returns `<NULL,oldTop>` (line 96).

Unlinking taken nodes. Nodes marked as `taken` are considered removed from the SP pool and are therefore ignored by `getYoungest` and `remove`. However, to reclaim memory and to reduce the overhead of iterating over `taken` nodes, nodes marked as `taken` are eventually unlinked either in `insert` (line 82-85) or in `remove` (line 103-111).

To unlink we redirect the `next` pointer from a node a to a node b previously connected by a sequence of `taken` nodes (line 85, line 106, and line 111). In `insert` the nodes between the new node and the next un-`taken` node are unlinked, and in `remove` the nodes between the old top node and the removed node, and between the removed node and the next un-`taken` node are unlinked. Additionally `remove` tries to unlink all nodes between `top` and the removed node (line 103). By using `CAS`, we guarantee that no new node has been inserted between the `top` and the removed node.

3.2 Timestamping Algorithms

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 reset.

TSC was not originally intended for timestamping, so an obvious concern is that it might not be synchronised across cores. In this case, relaxed-memory effects could lead to stack-order violations. We believe this is not a problem for modern CPUs. Ruan et al. [20] have tested `RDTSCP` on various x86 systems as part of their transactional memory system. Our understanding of [20] and the Intel x86 architecture guide [16] is that `RDTSCP` should provide sufficient synchronisation on recent multi-core and multi-socket machines. We have also observed no violations of stack semantics in our experiments across different machines. Aside from `RDTSCP`, we use C11 sequentially consistent atomics throughout, forbidding all other relaxed behaviours.

However, we have anecdotal reports that `RDTSCP` is not synchronised on older x86 systems. Furthermore, memory order guarantees offered by multi-processors are often underspecified and inaccurately documented – see e.g. Sewell et. al.'s work on a formalized x86 model [21] (which does not cover `RDTSCP`). Implementors should test `RDTSCP` thoroughly before using it to generate timestamps on substantially different hardware. We hope the TS stack 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 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 only if these timestamps were generated concurrently.

Listing 3: TS-CAS algorithm. The gray highlighted code is an optimisation to avoid unnecessary `CAS`.

```
117 TS_cas{
118  int counter=1;
119
120  Timestamp newTimestamp(){
121    int timestamp=counter;
122    pause(); // delay optimisation.
123    int timestamp2=counter;
124    if(timestamp != timestamp2)
125      return [timestamp,timestamp2-1];
126    if(CAS(counter,timestamp,timestamp+1))
127      return [timestamp,timestamp];
128    return [timestamp,counter-1];
129  }
130 }
```

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 for any two interval timestamps $[a, b]$ and $[c, d]$, if these intervals are generated sequentially, then b is generated before c and therefore $b < c$, as discussed above.

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 6.2.

TS-CAS: This algorithm is an optimisation of TS-atomic combined with interval timestamps. It exploits the insight that the shared counter needs only be incremented by *some* thread, not necessarily the current thread. In a highly concurrent situation, many threads using TS-atomic will increment the counter unnecessarily. The TS-CAS algorithm instead uses CAS failure to detect when the counter has been incremented. CAS failure without retrying is comparatively inexpensive, so this scheme is fast despite using strong synchronisation.

Source-code for TS-CAS is given in Listing 3. The algorithm begins by reading the counter value (line 121). If the CAS in line 126 succeeds, then the timestamp takes the counter's original value as its start and end (line 127). If the CAS fails, then another concurrent call must have incremented the counter, and TS-CAS does not have to. Instead it returns an interval starting at the original counter value and ending at the new value minus one (line 128). This interval will overlap with concurrent calls to `newTimestamp`, but will not overlap with any intervals created later.

Similar to TS-interval, adding a small delay between reading the counter value and attempting the CAS can improve performance. Here this not only increases the number of overlapping intervals, but also reduces contention on the global counter. Contention is reduced further by line 123-125, a standard CAS optimisation. If the value of `counter` changed during the delay, then the CAS in line 126 is guaranteed to fail. Instead of executing the CAS we can immediately return an interval timestamp. Our experiments show that in high-contention scenarios the performance of TS-CAS with a delay is up to 3x faster than without a delay.

4. Correctness Theorem for Stacks

Linearizability [14] is the *de facto* standard correctness condition for concurrent algorithms.[2] 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.* Note that our formulation is specialised to pool-like data-structures, because the val relation embeds the connection between an operation which inserts a value, and the operation that receives it (e.g. a push and corresponding pop).

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 a total order.*

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 call and return events. Therefore, a history captures all interactions between algorithm and context. We thus define a data-structure 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* $\langle \mathcal{A}, \mathsf{pr}^T, \mathsf{val} \rangle \in \mathcal{S}$*.*

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

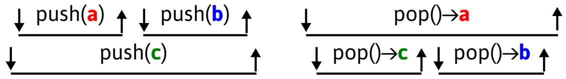
The problem with linearization points. Proving that a concurrent algorithm is linearizable with respect to a sequential specification amounts to showing that, for every possible execution, there exists a total linearization order. The standard strategy is a simulation-style proof where the implementation and specification histories are constructed in lock-step. The points where the specification 'takes effect' are known as *linearization points* – to simplify the proof,

[2] Our formulation of linearizability differs from the classic one [14]. Rather than have a history record the total order on calls and returns, we convert this information into a strict partial order pr. Likewise, linearizability between histories is defined by inclusion on orders, rather than by reordering call and return events. This approach, taken from [4], is convenient for us because our stack theorem is defined by constraints on orders. However, the two formulations are equivalent.

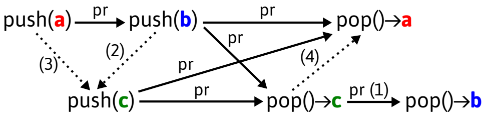
these are often associated with points in the implementation's syntax. Conceptually, when a linearization point is reached, the method is appended to the linearization order.

It has long been understood that linearization points are a limited approach. Simulation arguments work poorly for many non-blocking algorithms because the specification history is not precisely determined by the implementation. Algorithms may have linearization points dictated by complex interactions between methods, or by non-deterministic 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 elements 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 calls, and ↑ returns.



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 identify 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. [12] 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. (The exception is emptiness checking, which also requires special treatment in our approach – see below.)

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 in Val^*. Let $\sigma \in \mathsf{Val}^*$ be an arbitrary state. In Stack, push and pop have the following sequential behaviour ('$\cdot$' means sequence concatenation):

- push(v) – Update the abstract state to $\sigma \cdot [\mathrm{v}]$.
- 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'.

4.1 The Stack Theorem

We have developed stack conditions sufficient to ensure linearizability with respect to Stack. Unlike [12], our conditions are not expressed using only pr (indeed, we believe this would be impossible – see §4.2). Rather we require an auxiliary insert-remove relation ir which relates pushes to pops and vice versa, but that does not relate 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 relation* 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 $-a \xrightarrow{\mathsf{ir}} +c \xrightarrow{\mathsf{ir}} -b$.*

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 a stack.

In our formulation 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 is more convenient in verifying the TS stack.

Definition 4 (alternating)**.** *We call a relation 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 relation* ir *on $\mathcal{A}$, and derived orders* ins *and* rem*, such that:*

1. *ir $\cup$ 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) is at the heart of our proof approach. It forbids the non-LIFO behaviour illustrated in Figure 3.

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). For properties (1)-(3) it is trivial why this is sufficient. For emptiness checking, any history satisfying Set can be split into sub-histories free of pop-empty. As a pop-empty can only occur when no elements are in the data-structure, any such partitioning is also valid in Stack. Thus, the correctness of emptiness checking can be established separately from LIFO ordering.

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 five stages of the proof. For full details see supplementary Appendix A. (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 relation 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. □

For the TS stack, the advantage of Theorem 1 is that problematic orderings 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 relation *can* be extracted from the TS stack using an adapted version of the linearization point method.

Our stack theorem is generic, not tied to the TS stack. It characterises the internal ordering sufficient for an algorithm to achieve stack semantics. As well as sound, it is complete – for any linearizable stack, ir can be trivially projected from the linearization order. For CAS-based stacks such as Treiber's famous non-blocking stack [22] it is simple to see intuitively why the theorem applies. If two pushes are ordered, then their CASes are ordered. As a result their elements will be ordered in the stack representation and removed in order.

Intuitively, our theorem seems close to the lower bound for stack ordering. The next section (§4.2) provides evidence for this by ruling out the class of weaker formulations without ir. Intuitively, we would expect any concurrent stack to enforce orders as strong as the ones in our theorem. Thus, Theorem 1 points towards fundamental constraints on the structure of concurrent stacks.

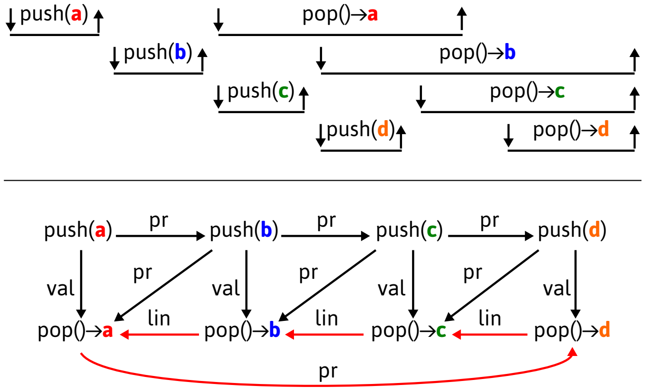


Figure 4: Top: example non-linearizable execution exhibiting non-local behaviour. Bottom: corresponding graph projecting out pr, val, and lin relations.

Mechanisation. We have mechanised Theorem 1 in the Isabelle theorem prover. The source files for this proof are provided in supplementary file `stackthm.tgz`. Supplementary Appendix A discusses the structure of our mechanisation alongside with an informal proof of the theorem.

4.2 Why the Insert-Remove Relation is Necessary

Theorem 1 builds on a similar theorem for queues proved by Henzinger et al. [12]. As in our definition of order-correctness (Definition 5), their theorem forbids certain bad orderings between operations. However, their conditions are defined purely in terms of precedence, pr, and value, val – they do not require the auxiliary insert-remove relation 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 relation is needed is the execution shown in Figure 4 (top). This execution as a whole is not linearizable – this can be seen more clearly in corresponding graph in Figure 4 (bottom), which projects out the pr and val relations. Here lin is the linearization order forced by LIFO ordering. The pr∪lin edges form a cycle, contradicting the requirement that linearization order is acyclic and 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 condition 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 condition can define linearizability for stacks. Our insert-remove relation introduces just enough extra structure to let us define a local stack theorem.

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

5. Proving the TS Stack Correct

We now prove the TS Stack correct. We use a two-level argument to separate concerns in the proof. By verifying a lower-level structure first, we hide the complexities of the data-structure from the higher-level proof.

1. Prove the linearizability of an intermediate structure called the *TS buffer*. This shows that the SP pools combine to form a single consistent pool, but does not enforce LIFO ordering.
2. Use our stack theorem (Theorem 1) to prove the TS stack is linearizable with respect to LIFO stack semantics. Linearizability lets us use the lower-level TS buffer in terms of its sequential specification.

5.1 TS Buffer Linearizability

The TS buffer is a 'virtual' intermediate data-structure, i.e. a proof convenience that does not exist in the algorithm syntax. (It would be easy to add, but would make our code more complex). The TS buffer methods are the lines in push and pop which modify the `spPools` array and thread-specific pools. Proving the TS buffer linearizable means these lines can be treated as atomic. We name the TS buffer operations as follows – line numbers refer to Listing 1. Note that where possible these names coincide with names in Listing 1.

- `ins` – inserts an element into an SP pool (line 13).
- `newTimestamp` – generates a new timestamp (line 14).
- `setTimestamp` – assign a timestamp to a SP pool element (line 15).
- `getStart` – record the current time at the beginning of a pop (line 20).
- `tryRem` – search through the SP pools and try to remove the element with the youngest timestamp (line 24).

Note that `newTimestamp` and `getStart` have the same underlying implementation, but different abstract specifications. This is because they play different roles in the TS stack: respectively, generating timestamps for elements, and controlling elmination.

The abstract state of the TS buffer hides individual SP pools by merging all the elements into a single pool. As elements may be eliminated depending on when the method started, the abstract state also records snapshots representing particular points in the buffer's history.

As with Stack and Set, we define the sequential specification 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 strict partial order $<_{\mathsf{TS}}$ and top element $\top$. 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.

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

We implicitly assume that all timestamps in the buffer were previously generated by `newTimestamp`.

Snapshots are used 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 `t` was generated by `newTimestamp()`. If $B(\mathtt{i}) = (v, \top)$, then update the abstract state to $(B[\mathtt{i} \mapsto (v, \mathtt{t})], S)$. If $B(\mathtt{i}) = \bot$, do nothing.
- `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.* Non-deterministically fail and return $\langle \mathsf{false}, \mathsf{null} \rangle$. This corresponds to a failed SP pool `remove` preempted by another thread.
  2. *emptiness check.* If the map is empty (i.e. $\mathrm{dom}(B) = \emptyset$) then return $\langle \mathsf{true}, \mathsf{EMPTY} \rangle$.
  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 unremoved elements from the snapshot, 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 $\langle \mathsf{true}, v_i \rangle$. Note that there may be many maximal elements that could be returned.
  4. *elimination.* Pick an ID i such that $i \in \mathrm{dom}(B)$, and either $i \notin \mathrm{dom}(S(\mathtt{t}))$ and $B(i) \mapsto (v, \_)$; or $S(\mathtt{t})(i) \mapsto (v, \top)$. Update the abstract state to $(B[i \mapsto \bot], S)$ and return $\langle \mathsf{true}, v \rangle$.

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

Theorem 2. *TS buffer operations are linearizable with respect to the specification* TSbuf.

Proof. The concrete state of the TS buffer consists of the array `spPools`, where each slot points to a SP pool, i.e. a linked list of nodes. For the abstract state, the mapping Buf is easily built by erasing taken nodes. We build $\mathsf{Snapshots}$ by examining the preceding trace. Snapshots are generated from the state of the buffer at any point `getStart` is called.

`ins` and `setTimestamp` are SP pool operations which take effect atomically because they build on atomic operations, i.e. the assignment to `top` in line 81, and to `timestamp` in line 15, respectively.

`newTimestamp` and `getStart` both build on the same timestamping operation. Only concurrent timestamp requests can generate overlapping timestamps. As timestamps have to be generated and then added to the buffer separately, at the call of `newTimestamp` and `getStart` an overlapping timestamp cannot be in the buffer. For `getStart`, the snapshot is correctly constructed automatically as a consequence of the mapping from concrete to abstract state.

The most complex proof is for `tryRem`, where the linearization point is in the call to `remove`. `tryRem` always removes a valid element because any element in the snapshot is guaranteed to be contained in one of the SP pools before `tryRem` starts its search for the youngest element. Any removed element not in the snapshot must have been added since the start of the search, and thus satisfies the elimination case of the specification. Further details are given in supplementary Appendix C. □

5.2 TS Stack Linearizability

We now prove that the TS stack is correct. We first define two orders vis and rr on push and pop operations. These orders are extracted from executions using a method analogous to linearization points, except that we generate two, possibly conflicting orders. The points chosen correspond to TS buffer operations. In §5.1 we proved that the TS buffer is linearizable, so we can treat these points as atomic.

- vis ('visibility' – the element inserted by a push was visible to a pop). A push and non-empty pop are ordered in vis if SP pool insertion in the push (line 13) is ordered before recording the current time in the pop (line 20).
- rr ('remove-remove' – two pops removed elements in order). Two non-empty pop operations are ordered in rr if their final successful `tryRem` operations (line 24) are similarly ordered in the execution.

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 vis 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 $-c, +d \in \mathcal{A}$.*

To apply Theorem 1 and to show that the TS stack is correct we need to show that any history arising from the TS stack is *order-correct* (Definition 5). The following lemma connects vis, rr and ts to this notion of order-correctness.

Lemma 3. *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. The proof works by using vis, rr and ts to construct a relation ir witnessing that $\mathcal{H}$ is order-correct. Either vis is such a witness, or vis can be adjusted locally such that it becomes a witness. Adjustment works iteratively by identifying triples of operations which violate order-correctness, then pushing one of the operations earlier or later in the relation to remove the violation. The detail of the proof consists of a case analysis showing that for any execution, such adjustments are always possible and eventually terminate. Further details are given in supplementary Appendix B. □

Lemma 4. *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 5. *TS stack is linearizable with respect to* Stack.

Proof. Follows by applying our stack theorem. Lemma 4 deals with the first clause of Theorem 1. The other clause requires the existence of an ir relation that satisfies order-correctness. It suffices to show that vis, rr, and ts satisfy the preconditions of Lemma 3. The first requirement that $\mathsf{pr} \cup \mathsf{vis}$ and $\mathsf{pr} \cup \mathsf{rr}$ are cycle-free, follows from the fact that the instructions used to define vis and rr are linearization points of the TS buffer. The second requirement for the lemma follows from the fact that ordering in ts implies ordering in $<_{\mathsf{TS}}$, and the fact that the TS stack removes elements in an order that respects $<_{\mathsf{TS}}$. Further details are given in supplementary Appendix B. □

Theorem 6. *The TS stack is lock-free.*

Proof. Straightforward from the structure of `tryRem`: removal can only fail when another thread succeeds. □

6. 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.[3] We configured the Treiber and EB stacks to perform as well as possible on our test machines: see below for the parameters used.

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 [19] 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.

[3] 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.

	40-core machine	64-core machine
high-contention:		
TS-interval stack	7 µs	6 µs
TS-CAS stack	6 µs	10.5 µs
low-contention:		
TS-interval stack	4.5 µs	4.5 µs
TS-CAS stack	3 µs	9 µs

Table 1: Benchmark delay times for TS-interval / TS-CAS.

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 µs in the high-contention benchmark, and of size 12 with a delay of 18 µs in the low contention benchmark. On the 64-core machine the elimination array is of size 32 with a delay of 21 µs in the high-contention benchmark, and of size 16 with a delay of 18 µs 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 of 300 µs in the high-contention experiments and a constant delay of 200 µs in the low-contention experiments, 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 µs on the 40-core machine.

6.1 Performance and Scalability Results

Figures 5a and 5b 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 7 in the supplementary material.

For TS-interval timestamping and TS-CAS 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 6.2. The delay thus depends on the machine and benchmark. The delay times we use in the benchmarks are listed in Table 1. The impact of different delay times on performance is discussed in Section 6.2.

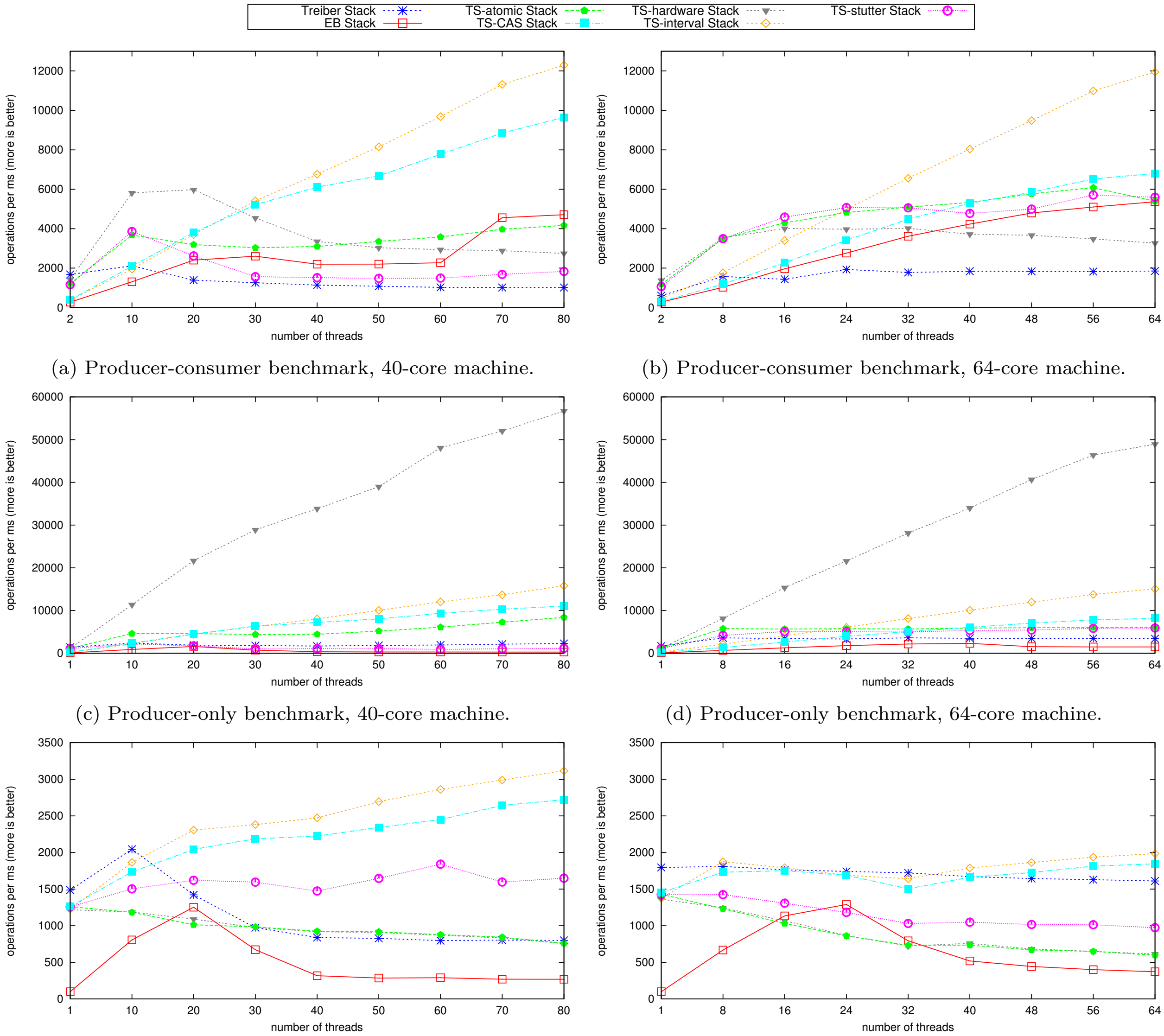
Comparison between implementations. TS-interval is faster than the other timestamping algorithms in the producer-consumer benchmarks with an increasing number of threads. Interestingly the TS-atomic stack is faster than the TS-hardware stack in the high-contention producer-consumer benchmark. The reason 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 2 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 (see Figure 5c). On the 40-core machine the TS-CAS stack is much faster than the TS-hardware stack, TS-atomic stack, and TS-stutter stack, on the 64-core machine it is slightly faster. The reason is that on the 64-core machine a `CAS` is slower in comparison to other instructions than on the 40-core machine.

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 and the TS-CAS stack outperform the EB stack in the high-contention producer-consumer benchmark with a maximum number of threads, on the 64-core machine also the TS-stutter stack and the TS-atomic stack are sightly faster than the EB stack.

We believe TS-interval's and TS-CAS'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 for example 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 and the TS-CAS stack use the same delay as in the high-contention producer-consumer benchmark, see Table 1:

Figure 5c and Figure 5d 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



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

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.

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 5e and Figure 5f 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 performance of TS-CAS is close to the performance of TS-interval. The TS-stutter stack is faster than the TS-atomic and TS-hardware stack 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 less and sometimes as expensive as removing elements sequentially from multiple lists (TS stack).

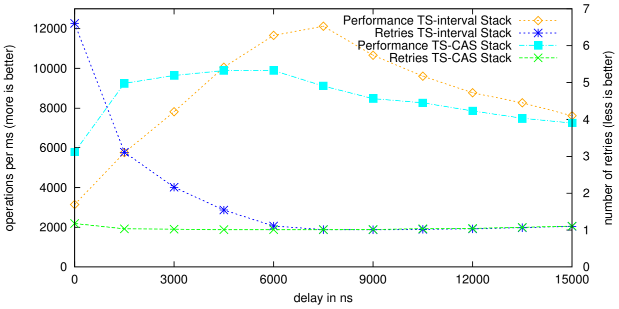


Figure 6: High-contention producer-consumer benchmark using TS-interval and TS-CAS timestamping with increasing delay on the 40-core machine, exercising 40 producers and 40 consumers.

6.2 Analysis of Interval Timestamping

Figure 6 shows the performance of the TS-interval stack and the TS-CAS 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 6.1.

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

For the TS-interval stack the 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 `tryRem`. For the optimal delay time we have 1.009 calls to `tryRem` per pop, i.e. less than 1% of pop calls need to scan the SP pools array more than once. In contrast, without a delay the average number of retries per pop call is more than 6.

The performance of the TS-CAS stack increases initially with an increasing delay time. However this does not decrease the number of `tryRem` retries significantly. The reason is that without a delay there is more contention on the global counter. Therefore the performance of TS-CAS with a delay is actually better than the performance without a delay. However, similar to TS-interval timestamping, with a delay time beyond 6 µs the performance decreases again. This is the point where an average push operation becomes slower than an average pop operations.

7. 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 pool such that `getYoungest` 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 is the second fastest queue we know of. In our experiments the TS-interval queue outperforms the Michael-Scott queue [18] and the flat-combining queue [10] but the lack of elimination means it is not as fast as the LCRQ [1].

The TS-interval deque is the fastest deque we know of, although it is slower than the corresponding stack / queue. However, it still outperforms the Michael-Scott and flat-combining queues, and the Treiber and EB stacks.

8. Related Work

Timestamping. Our approach was initially 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 [15] 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 elements 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 §3.2). 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 [11] 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 calls. (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. [20], 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 [12]. 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 relation between inserts and removes. We suspect it is impossible to define such a theorem for stacks without an auxiliary insert-remove relation (see §4.2).

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 pools as partial queues and the TS stack itself as the load balancer. The TS stack 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.

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.

Acknowledgments

We thank Ana Sokolova for feedback, Frank Zeyda for help with the Isabelle formalization, and Michael Lippautz for help with Scal. We also thank the POPL referees for their thoughtful comments. This work has been supported by the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund (FWF): S11404-N23).

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

[11] 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.

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

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

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

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

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

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

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

[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.

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 A 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.

A. 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 $\mathsf{emp}(e)$ 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:*

$$\forall e, a \in \mathcal{A}.\, \mathsf{emp}(e) \wedge \mathsf{isPush}(a) \wedge a \xrightarrow{\mathsf{pr}} e \implies \exists b.\, a \xrightarrow{\mathsf{val}} b \wedge b \xrightarrow{\mathsf{pr}} e$$

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

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 $\mathsf{pr}^{\mathrm{pop}}$ 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 ir. We adjust the ir relation to become part of the final linearization order. Informally we do this by ordering push operations as late as possible, even if that contradicts the original ir relation. 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 $+a \xrightarrow{\mathsf{val}} -a$, $+c \xrightarrow{\mathsf{val}} -c$, $+c \xrightarrow{\mathsf{ir}'} -b$, $-b \xrightarrow{\mathsf{pr}^{\mathrm{pop}}} -c \xrightarrow{\mathsf{pr}^{\mathrm{pop}}} -a$, 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.

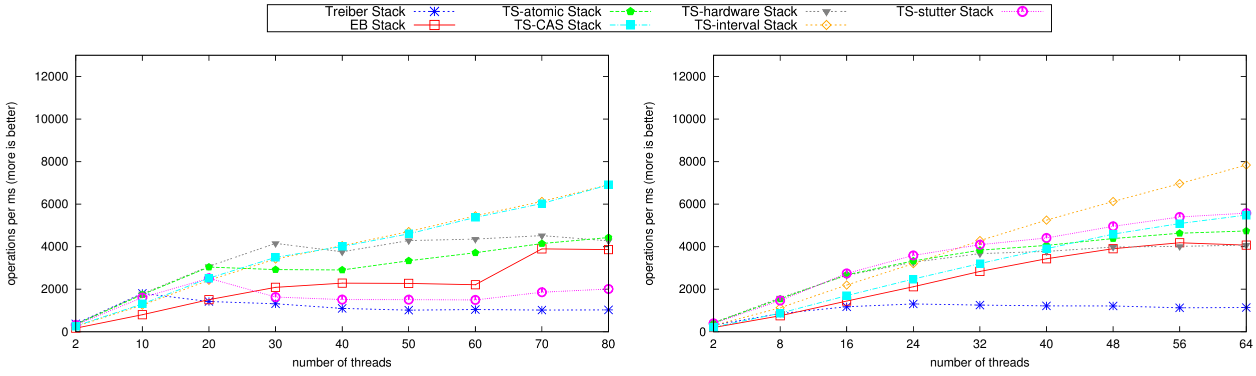
To simplify the proof structure, we assume that in every execution, all elements which are pushed also get popped. We justify this by observing that any concurrent history where some elements do not get popped can be extended to one where all elements get popped. If the extended history is linearizable,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

B. Proving the TS Stack Linearizable

Lemma 3. *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.



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

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

Figure 7: Performance of the TS stack in the low-contention scenario.

	40-core machine	64-core machine
EB stack	92%	76%
TS-atomic stack	89%	93%
TS-hardware stack	43%	64%
TS-stutter stack	96%	88%
TS-interval stack	99%	98%
TS-CAS stack	99%	99%

Table 2: Percentage of elements removed by elimination in the high-contention producer-consumer benchmark.

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

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 $+a \xrightarrow{\mathsf{ts}} +b$ (see Lemma 8). 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*.

Violating triples are resolved iteratively. In each iteration we adjust vis, resulting in vis', where at least one violating triple is resolved. There exist two ways to resolve violating triples $(-a, +c, -b)$: $+c$ can be "moved forwards", which means that in vis' $+c \xrightarrow{\mathsf{vis}'} -a$, and all other operations remain unchanged, i.e. $x \xrightarrow{\mathsf{vis}'} y$ for all $x, y \in \mathcal{A}$ with $x \xrightarrow{\mathsf{vis}} y$, $x \neq -a$, and $y \neq +c$. We call violating triples which can be resolved by moving $+c$ forwards *forward triples*.

Violating triples $(-a, +c, -b)$ can also be resolved by "moving $+b$ backwards", where $+b \xrightarrow{\mathsf{val}} -b$. Moving $+b$ backwards means for all pop operations $-d \in \mathcal{A}$, $-d \xrightarrow{\mathsf{vis}'} +b$ if and only if $-d \xrightarrow{\mathsf{vis}} +c$. Otherwise $+b \xrightarrow{\mathsf{vis}'} -d$. All other operations remain unchanged. Such violating triples are called *backward triples*. Any violating triple which is not a forward triple is a backward triple.

Violating triples $(-a, +c, -b)$ are resolved in the following order:

- If for all $+c_i, -c_i \in \mathcal{A}$ with $+c_i \xrightarrow{\mathsf{val}} -c_i$ where $(-a, +c_i, -b)$ is a violating triple it holds that $-c_i \xrightarrow{\mathsf{rr}} -a$, then $(-a, +c, -b)$ is a forward triple. Forward triples are always resolved before backward triples.
- Let $(-a, +c, -b)$ be a forward triple, and let $-a, +c', -b)$ be another forward triple. If $+c' \xrightarrow{\mathsf{pr}} +c$, or if there exists a $-d \in \mathcal{A}$ with $+c' \xrightarrow{\mathsf{vis}} -d \xrightarrow{\mathsf{vis}} +c$, then $(-a, +c', -b)$ is resolved before $(-a, +c, -b)$.
- Let $(-a, +c, -b)$ and $(-a, +f, -b)$ be two forward triples with $+c \overset{\mathsf{pr}}{\not\rightarrow} +f$, $+f \overset{\mathsf{pr}}{\not\rightarrow} +c$, and there does not exist a $-g \in \mathcal{A}$ with $+c \xrightarrow{\mathsf{vis}} -g \xrightarrow{\mathsf{vis}} +f$ or $+f \xrightarrow{\mathsf{vis}} -g \xrightarrow{\mathsf{vis}} +c$. Then $(-a, +c, -b)$ and $(-a, +f, -b)$ are resolved at the same time.

Resolving violating triples preserves the following invariants. Let vis' be the relation resulting from resolving a violation triple in vis. Let ts' be derived from vis'.

- vis' is alternating.
- $\mathsf{pr} \cup \mathsf{vis}'$ is cycle-free.
- Let $+a, -a, +b \in \mathcal{A}$ with $+a \xrightarrow{\mathsf{val}} -a$ and $+a \xrightarrow{\mathsf{pr}} -a$, and $+a \xrightarrow{\mathsf{ts}'} +b \xrightarrow{\mathsf{vis}'} -a$, then there exists a $-b \in \mathcal{A}$ with $+b \xrightarrow{\mathsf{val}} -b$ and $-b \xrightarrow{\mathsf{rr}} -a$.

We prove these invariants in Lemma 9 and Lemma 10 for forward triples, and in Lemma 11 and Lemma 13 for backward triples.

Note that resolving violating triples can result in new violating triples. However, we prove by induction that eventually all violating triples are resolved. First we observe that vis defines a partial order on violating triples. A violating triple $(-a, +c, -b)$ is ordered before $(-d, +f, -e)$ if there exists a $-g \in \mathcal{A}$ with $+c \xrightarrow{\mathsf{vis}} -g \xrightarrow{\mathsf{vis}} +f$. We call this order the *violating triple order*. We prove by induction that for any N eventually the last N violating triples will be resolved forever.

Let $(-a, +c, -b)$ be one of the last violating triples in the violating triple order. Then in all following iterations $+c$ will never be moved backward. The reason is that moving $+c$ backwards would require a violating triple which is later than $(-a, +c, -b)$ in the violating triple order. However, since $(-a, +c, -b)$ is the last violating triple, no triple can be moved behind it, and $+c$ cannot be moved before another violating triple. Therefore, since $+c$ can only be moved forwards finitely many times, eventually $+c$ will not be part of any violating triple forever.

Now assume there exists a vis' where the last N violating triples have been resolved forever. In vis' there exists again a last violating triple $(-a, +c, -b)$. For the same reasons as above $+c$ cannot be moved backwards anymore, and after moving it forwards finitely many times, $+c$ cannot be part of a violating triple anymore. Thus, if the last N violating triples will eventually be resolved forever, also the last $N + 1$ will. Therefore eventually all violating triples will be resolved.

□

We continue with the helping lemmata of the proof of Lemma 3.

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. □

For the following lemmata we have the following assumption: let $\langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ be a history with an alternating relation vis, a total order on non-empty pop operations rr, and a derived order ts. We assume that

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 $-b \xrightarrow{\mathsf{rr}} -a$;

Let $(-a, +c, -b)$ be a forward triple such that there does not exist a forward triple $(-a, +f, -b)$ and $-g$ with $+f \xrightarrow{\mathsf{vis}} -g \xrightarrow{\mathsf{vis}} +c$ or $+f \xrightarrow{\mathsf{pr}} +c$. These restrictions correspond to the order in which violating triples are resolved. If such a forward triple $(-a, +f, -b)$ existed, then it would be resolved before $(-a, +c, -b)$.

Additionally let vis' be the modified vis relation which results from resolving the violating triple $(-a, +c, -b)$, and let ts' be derived from pr and vis'.

Without restriction of generality we assume that there does not exist a violating triple $(-a, +c', -b)$ with $+c \overset{\mathsf{pr}}{\not\to} +c'$, $+c' \overset{\mathsf{pr}}{\not\to} +c$, and there does not exist a $-g \in \mathcal{A}$ with $+c \xrightarrow{\mathsf{vis}} -g \xrightarrow{\mathsf{vis}} +c'$ or $+c' \xrightarrow{\mathsf{vis}} -g \xrightarrow{\mathsf{vis}} +c$. According to our rules such a violating triple $(-a, +c', -b)$ would be resolved at the same time as $(-a, +c, -b)$ anyways, not introducing any extra special cases.

Lemma 9. *With the assumptions stated above $\mathsf{pr} \cup \mathsf{vis}'$ 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. This, however, contradicts our assumption that there does not exist a violating triple $(-a, +f, -b)$ with $+f \xrightarrow{\mathsf{pr}} +c$. Thus this case is impossible. 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 vis' is cycle-free. □

Lemma 10. *With the assumptions above it holds for all $+d, -d, +e \in \mathcal{A}$ with $+d \xrightarrow{\mathsf{val}} -d$, $+d \xrightarrow{\mathsf{pr}} -d$, and $+d \xrightarrow{\mathsf{ts}'} +e \xrightarrow{\mathsf{vis}'} -d$ that there exists a $-e \in \mathcal{A}$ with $+e \xrightarrow{\mathsf{val}} -e$ and $-e \xrightarrow{\mathsf{rr}} -d$.*

Proof. We assumed that the current lemma holds for vis, and the only pair which is different in vis and in 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 $+e \xrightarrow{\mathsf{ts}} +f \xrightarrow{\mathsf{vis}} -e$ and the violation would have already existed in vis. As $+e \xrightarrow{\mathsf{ts}} +f$ would be constructed with any $-g$ with $+c \xrightarrow{\mathsf{vis}} -g \xrightarrow{\mathsf{vis}} +f$, it has to be true that for all $-g \in \mathcal{A}$, if $+c' \xrightarrow{\mathsf{vis}} -g$, then also $+f \xrightarrow{\mathsf{vis}} -g$. Therefore $(-a, +f, -b)$ would be a violating triple which is resolve at the same time as $(-a, +c, -b)$. Therefore $-a \xrightarrow{\mathsf{vis}'} +f$ is not possible.

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

In the following we assume that all forward triples have already been resolved. Therefore we assume that $(-a, +c, -b)$ is a backward triple with $+b \xrightarrow{\mathsf{val}} -b$ for some $+b \in \mathcal{A}$. Without restriction of generality we assume that either there does not exist a $-c \in \mathcal{A}$ with $+c \xrightarrow{\mathsf{val}} -c$, or such a $-c$ exists and $-a \xrightarrow{\mathsf{rr}} -c$. We can make this assumption because if there did not exist a $+c$ with these properties, then $(-a, +c, -b)$ would be a forward triple.

Additionally let vis' result from vis by resolving the backward triple $(-a, +c, -b)$, and let ts' be derived from pr and vis'.

Lemma 11. *With the assumptions stated above* $\mathsf{pr} \cup \mathsf{vis}'$ *is cycle-free.*

Proof. Assume a cycle exists in $\mathsf{pr} \cup \mathsf{vis}'$, then the cycle has to contain $+b$, as all other operations are ordered in vis' the same as in vis and $\mathsf{pr} \cup \mathsf{vis}$ is cycle-free. By construction vis' is cycle-free, and therefore a potential cycle in $\mathsf{pr} \cup \mathsf{vis}'$ has to contain a pr edge. As $+b$ is moved backwards, whenever for some operation $d \in \mathcal{A}$ there exists a transitive edge from $+b$ to d in $\mathsf{pr} \cup \mathsf{vis}'$, there also exists a transitive edge in $\mathsf{pr} \cup \mathsf{vis}$. Therefore a cycle has to contain a transitive edge from some $d \in \mathcal{A}$ to $+b$ in $\mathsf{pr} \cup \mathsf{vis}'$ which did not exist in $\mathsf{pr} \cup \mathsf{vis}$. Additionally, also the transitive edge from $+b$ to d which existed in $\mathsf{pr} \cup \mathsf{vis}$, is be preserved in $\mathsf{pr} \cup \mathsf{vis}'$, and is part of the cycle. With the transitivity and abcd property of pr, vis being alternating, and $\mathsf{pr} \cup \mathsf{vis}$ being cycle-free, any such transitive edge implies one of the following two kinds of transitive edges:

1. $+b \xrightarrow{\mathsf{pr}} -f \xrightarrow{\mathsf{vis}} +c$ for some $-f \in \mathcal{A}$; and
2. $+b \xrightarrow{\mathsf{pr}} +g \xrightarrow{\mathsf{vis}} -f \xrightarrow{\mathsf{vis}} +c$ for some $-f, +g \in \mathcal{A}$.

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

Before proving the last invariant we first show a helping lemma based on the assumptions we stated above.

Lemma 12. *Let* $+s, -s, +e \in \mathcal{A}$ *with* $+s \neq +b$, $+s \xrightarrow{\mathsf{val}} -s$, $+s \xrightarrow{\mathsf{pr}} -s$, *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* $+s \xrightarrow{\mathsf{ts}} +c \xrightarrow{\mathsf{vis}} -s$.

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 $+g \xrightarrow{\mathsf{vis}'} -h$, then also $+g \xrightarrow{\mathsf{vis}} -h$ (otherwise $+g$ would have been moved forwards before $-h$). Therefore, if $+e \xrightarrow{\mathsf{vis}'} -s$, it also holds that $+e \xrightarrow{\mathsf{vis}} -s$, and in the case of $+e = +b$ also $+c \xrightarrow{\mathsf{vis}} -s$. 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 $+s \xrightarrow{\mathsf{ts}} +c$. 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 $+b$. Therefore $+s \xrightarrow{\mathsf{ts}} +e$. If $+e = +b$, then $+c$ is ordered the same in vis as $+b$ in vis′ and therefore $+s \xrightarrow{\mathsf{pr}} -k \xrightarrow{\mathsf{vis}} +c$ and $+s \xrightarrow{\mathsf{ts}} +c$.
- 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 $+c$ is ordered the same in vis as $+b$ in vis′ and therefore $+s \xrightarrow{\mathsf{pr}} +l \xrightarrow{\mathsf{vis}} -k \xrightarrow{\mathsf{vis}} +c$ and $+s \xrightarrow{\mathsf{ts}} +c$.

□

We continue proving the last invariant:

Lemma 13. *Let the assumptions stated above hold, and 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* $-e \xrightarrow{\mathsf{rr}} -s$.

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 $+e = +b$. From the assumption that $(-a, +c, -b)$ is a violating triple we know that $-b$ exists and $-b \xrightarrow{\mathsf{rr}} -a$. Moreover we assumed that if a $-c$ with $+c \xrightarrow{\mathsf{val}} -c$ exists, then $-a \xrightarrow{\mathsf{rr}} -c$ and therefore by transitivity also $-b \xrightarrow{\mathsf{rr}} -c$ holds. We showed in Lemma 12 that $+s \xrightarrow{\mathsf{ts}} +c \xrightarrow{\mathsf{vis}} -s$ and therefore according to our assumptions it holds that $-c \xrightarrow{\mathsf{rr}} -s$. Again by the transitivity of rr it therefore holds that $-b \xrightarrow{\mathsf{rr}} -s$. As $-e = -b$ this means that $-e \xrightarrow{\mathsf{rr}} -s$, which completes the proof.

□

Theorem 5. *TS stack is linearizable with respect to* Stack.

Proof. Lemma 4 deals with one clause of the stack theorem, Theorem 1. To show that there exists an ir relation that satisfies the definition of order-correctness it suffices to show that the generated vis, rr, and ts orders satisfy the conditions of Lemma 3.

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 $\texttt{ts}$, 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 $\texttt{ts}$. 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 $\texttt{setTimestamp}$ operation of $+a$, and let $\texttt{ins}_{+b}$ and $\texttt{newTimestamp}_{+b}$ be the $\texttt{ins}$ and $\texttt{newTimestamp}$ 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 $+b \xrightarrow{\mathsf{vis}} -a$. 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 $\texttt{getStart}$ operation $\texttt{getStart}_{-a}$. Therefore either b is removed before $-a$ calls $\texttt{getStart}$ or b is within the snapshot generated by $\texttt{getStart}$.

If b is removed before $\texttt{getStart}_{-a}$ then there must exist a pop operation $-b$ such that its $\texttt{tryRem}$ operation $\texttt{tryRem}_{-b}$ precedes the $\texttt{tryRem}$ 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, $\texttt{tryRem}$ 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 $\texttt{tryRem}$ operation precedes $\texttt{tryRem}_{-a}$ in $\mathcal{T}$. This completes the proof. □

C. Proof of correctness for the TS buffer

Theorem 2. *The TS buffer implementation is linearizable with respect to the specification* TSBUF *given in §5.*

Proof. All methods except $\texttt{tryRem}$ are already discussed in detail in §5. Here we will focus solely on the linearizability of $\texttt{tryRem}$.

The cases of $\texttt{tryRem}$ returning `<false,NULL>` (line 59 in the emptiness check and line 114 when setting the $\texttt{taken}$ flag fails) are trivially correct. The linearizability of $\texttt{tryRem}$ returning `<true,EMPTY>` has been shown in [8]: if the recorded $\texttt{top}$ pointers have not changed, then there must have existed a point in time where the abstract state of the TS buffer contained no elements. In normal removal (case 3) and elimination (case 4), the linearization point is the successful $\texttt{CAS}$ to set the $\texttt{taken}$ flag of an item in line 102. With the $\texttt{taken}$ flag an element is removed atomically from the concrete state, so we only need to show that the correct element is removed.

We start with some properties of the elements in the buffer. Let $S(\texttt{t})$ be the snapshot of the TS buffer created by $\texttt{getStart}$. Any element in $S(\texttt{t})$ must have been in a SP pool before $\texttt{tryRem}$ begins, and elements that are not in $S(\texttt{t})$ are guaranteed to have timestamps younger than $\texttt{t}$.

Next we identify invariants of the SP pool.

1. If a node is reachable from the $\texttt{top}$ pointer of a SP pool once, then it remains reachable at least as long as its $\texttt{taken}$ flag is not set.
2. If a node b is reachable from a node a once by a sequence of $\texttt{next}$ pointers, then b remains reachable from a as long as its $\texttt{taken}$ flag is not set.
3. The nodes in the list are sorted by their timestamps. Thereby it is guaranteed that the node with the youngest timestamp in the SP pool is contained in the first node in the list which is not marked as $\texttt{taken}$.

With these invariants we show that the node returned by $\texttt{getYoungest}$ is either the element in the snapshot with the youngest timestamp, or the element of the node is not in $S(\texttt{t})$. Invariant 1 and invariant 2 guarantee that all nodes in $S(\texttt{t})$ are reachable from $\texttt{top}$ (line 90) by following $\texttt{next}$ pointers (line 97). Therefore the node with the youngest timestamp will be found eventually. With invariant 3 it is guaranteed that the node with the youngest timestamp in $S(\texttt{t})$ will be the first node encountered in the iteration which is not marked as $\texttt{taken}$. Therefore at the time the node is found it is indeed the node with the youngest timestamp in the SP pool which is also in $S(\texttt{t})$. Alternatively the first un-$\texttt{taken}$ node in the list may not be in $S(\texttt{t})$, it which case returning that node is also correct.

Now, suppose that $\texttt{tryRem}$ calls $\texttt{remove}$ of a SP pool in line 63 in listing 1. Comparison with $\texttt{startTime}$ on line 46 ensures that elements not in the snapshot are not considered for normal removal. Suppose $\texttt{tryRem}$ removes an element a although there exists an element b in the snapshot and in the TS buffer with $a <_{\mathsf{TS}} b$. Therefore b is contained in a SP pool and $\texttt{getYoungest}$ must have returned b or some even younger element c. In this case, b or c would be considered younger than a by the timestamp comparison (line 48), 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 $\texttt{getYoungest}$.

Suppose alternatively that $\texttt{tryRem}$ removes an element a on line 47. If $a \notin \mathrm{dom}(S(\texttt{t}))$, then the removal of a is correct because of elimination. If $a \in \mathrm{dom}(S(\texttt{t}))$, 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. □