UNIVERSITÄT SALZBURG

Fast Concurrent Data-Structures Through Explicit Timestamping

Mike Dodds[a] Andreas Haas Christoph M. Kirsch

[a]University of York

Technical Report 2014-03 February 2014

Department of Computer Sciences

Jakob-Haringer-Straße 2
5020 Salzburg
Austria
www.cosy.sbg.ac.at

Technical Report Series

Fast Concurrent Data-Structures Through Explicit Timestamping

Mike Dodds[1], Andreas Haas[2], and Christoph M. Kirsch[2]

[1] University of York (firstname.lastname@york.ac.uk)
[2] University of Salzburg (firstname.lastname@cs.uni-salzburg.at)

Abstract. Concurrent data-structures, such as stacks, queues and deques, often implicitly enforce a total order over elements with their underlying memory layout. However, linearizability only requires that elements are ordered if the inserting methods ran sequentially. We propose a new data-structure design which uses explicit timestamping to avoid unwanted ordering. Elements can be left unordered by associating them with unordered timestamps if their insert operations ran concurrently. In our approach, more concurrency translates into less ordering, and thus less-contended removal and ultimately higher performance and scalability.
As a proof of concept, we realise our approach in a non-blocking double-ended queue. In experiments our deque outperforms and outscales the Michael-Scott queue by a factor of 4.2 and the Treiber stack by a factor of 2.8. It even outscales the elimination-backoff stack, the fastest concurrent stack of which we are aware, and the flat-combining queue, a fast queue more scalable than Michael-Scott.

1 Introduction

The main idea of our approach is to timestamp elements as they are added to a shared data-structure, and to use these timestamps to determine the order in which elements should be removed. The data-structure can be instantiated as a stack by removing the element with the latest timestamp, or as a queue by removing the element with the earliest timestamp. Both kinds of operation can be combined to give a *deque* – a double-ended queue. Listing 1.1 shows the high-level pseudocode for our Timestamped Deque (TS deque).

One might assume that generating a timestamp and adding an element to the data-structure has to be done together, atomically. This intuition is wrong: linearizability allows concurrent operations to take effect in any order within method boundaries – only sequential operations have to keep their order [13]. Therefore we need only order elements if the methods inserting them execute sequentially. Our approach exploits 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

Listing 1.1. TS deque algorithm – `insertL` / `removeL` are defined analogously. Implementations for TS buffer operations are given in Listing 1.2.

```
1  TS_Deque{
2    TS_Buffer buffer;
3
4    void insertR(Element element) {
5      item = buffer.insR(element);
6      t = buffer.newTimestamp();
7      buffer.setTimestamp(item,t);
8    }
9
10   Element removeR() {
11     do {
12       item = buffer.tryRemR();
13     } while (!item.isValid());
14     if (item.isEmpty())
15       return EMPTY;
16     else
17       return item.element;
18   }
19 }
```

be unordered, but *only* when the surrounding methods overlap, meaning the elements can legitimately be removed in any order. The only constraint is that elements of sequentially executed insert operations get ordered timestamps.

By separating timestamp creation from adding the element to the data-structure, the insert method can avoid two expensive synchronisation patterns – atomic-write-after-read (AWAR) and read-after-write (RAW). We take these patterns from [3], 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. The underlying data can be stored in many single-producer-multiple-consumer buffers. This also does not require strong synchronization in the insert operation. Therefore the whole insert operation does not require any 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, [3] proves that the remove operation for a deque cannot be implemented without strong synchronisation. Perhaps surprisingly, this problem can be mitigated by reducing the ordering between timestamps: intuitively, less ordering results in more opportunities for concurrent removal, and thus less contention. Our experiments show that data-structures based on partially-ordered timestamps can achieve performance and scalability comparable to or better than state-of-the-art concurrent data-structures. In particular, our TS stack performs better than any other stack of which we are aware.

Algorithm structure. Listing 1.1 shows the high-level structure of our TS deque. The abstract state of the deque is a sequence of elements, and there are four basic operations:

- `insertL` / `insertR` – add an element to the left or right end of the sequence.
- `removeL` / `removeR` – remove an element from the left or right end of the sequence, or return EMPTY.

To get a stack, we take two operations on the same end of the sequence (e.g. `insertR` and `removeR`). To get a queue, we take two operations on opposite ends (e.g. `insertL` and `removeR`).

To simplify the presentation of our algorithm, we define it using a lower-level structure called a *timestamped buffer* (TS-buffer). The abstract state of the buffer is a map from identifiers to values, optionally associated with timestamps. The TS-buffer offers the following operations:

- `insL` / `insR` – add an element to the buffer without attaching a timestamp, and return a reference to the item.
- `newTimestamp` – generate a timestamp later than any timestamp of an item already in the buffer.
- `setTimestamp` – attach a timestamp to a given item in the buffer.
- `tryRemL` / `tryRemR` – try to remove the leftmost / rightmost element of the buffer.

(Our TS-buffer implementation is discussed directly below, and in detail in Section 4.2. Pseudocode is given in Listing 1.2.)

With the TS-buffer operations defined, the structure of Listing 1.1 should be clear. To insert an element, the algorithm inserts an un-timestamped element into the buffer (line 5), generates a fresh timestamp (line 6), and sets the new element's timestamp (line 7). To remove an element, the algorithm tries to find and remove an appropriate element (line 12) until it succeeds or discovers the buffer is empty.

The necessary constraint to ensure linearizability holds straightforwardly: if two insert operations run sequentially, then the associated elements have ordered timestamps. Elements arising from concurrent operations can be ordered any way, so this suffices to ensure that elements are removed in a linearizable order. (In fact, subtleties arise when multiple insert and remove operations overlap – see Section 3 and Appendix A for details of the correctness argument).

Implementation approach. We implement the TS-buffer as a collection of single-producer multiple-consumer buffers (see Section 4.2, Listing 1.2 for pseudocode). Each thread is associated with a buffer into which it inserts, and `tryRemL` and `tryRemR` search through all the buffers to find the leftmost / rightmost element. Elements are removed using an atomic compare-and-swap (i.e. an AWAR) to write a '`taken`' flag; if the `CAS` fails, the `tryRem` operation fails. This contention means that our algorithm is lock-free but not wait-free; a thread can be forced to wait indefinitely by contending threads.

We have experimented with various implementations for timestamping itself. Most straightforwardly, we can use a strongly-synchronised fetch-and-increment counter. We can avoid strong synchronisation by using a vector of thread-local counters, meaning the counter may stutter (many threads get the same timestamp). We can also use a hardware timestamping operation – for example the RDTSCP instruction which is available on all modern x86 hardware. In the past, such instructions have largely been used for analysis and logging. In our experiments we found hardware timestamping to be by far the best-performing approach.

Elimination and interval timestamping. We have experimented with several optimisations, most importantly *elimination* and *interval timestamping.*

In the case of a stack, concurrent insert and remove operations can eliminate each other [10]. Therefore a thread can remove any concurrently-inserted element, not just the leftmost or rightmost one. Unlike [10], our mechanism for detecting elimination exploits the existance of timestamps. We read the current timestamp at the start of a remove; any element with a later timestamp has been inserted during the current remove, and can be eliminated. It is largely thanks to elimination that our TS stack is the fastest of our timestamped structures.

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

Pausing allows us to trade off the performance of insertion and removal: an increasing delay in insertion can reduce the number of retries in remove (see Section 5.3). Though pausing may make insertion slower than a single AWAR instruction, our experiments suggest what is expensive is not individual instructions, but rather contention that causes many instructions to be repeated. Our experiments show that by weakening the order of elements, interval timestamping can substantially increase overall throughput and decrease latency of removes.

Similarly, although interval timestamping increases the non-determinism of removal (i.e. the variance in which elements are returned), this need not translate into greater overall non-determinism compared to other state-of-the-art algorithms. A major source of non-determinism in existing concurrent data-structures is in fact contention [8]. While interval timestamping increases the potential for non-determinism in one respect, it decreases it in another.

Paper structure. §2 surveys the related work on concurrent data-structures. In §3 we argue for the correctness of our algorithm. In §4 we give more details of our algorithm implementation. In §5 we discuss our experiments, both with different

implementation choices, and with respect to other concurrent data-structures. §6 concludes.

2 Related Work

Our timestamping approach was inspired by Attiya et al.'s Laws of Order paper [3]. That paper 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 instead: the TS deque. The observation that enqueues need not take effect in the same order as their atomic operations is used in the basket queue [14], although unlike the TS deque that algorithm does not avoid strong synchronisation.

Our TS deque implementation reuses concepts from several concurrent data-structures. Storing elements in multiple partial data-structures instead of one global data-structure is used in the distributed queue [7] where insert and remove operations are distributed between partial queues using a load balancer. One can view the thread-local buffers of the TS-buffer as partial queues and the TS-buffer itself as the load balancer. The TS-buffer emptiness check also originates from the distributed queues. Our contribution is that the TS deque leverages the performance of distributed queues while preserving sequential deque semantics.

The generic, queue-specific, and stack-specific thread-local buffer implementations are similar to the ConcurrentLinkedDeque in OpenJDK [1], the Michael-Scott queue [20], and the Treiber stack [23], respectively. All these implementations are based on linked lists with one or two access pointers which are updated using `CAS`. However, unlike these implementations, insertion into our thread-local buffers does not require `CAS` or other strong synchronization.

There exist other lock-free concurrent deque implementations based on doubly-linked lists e.g. Michael [18], Sundell and Tsigas [22]. Generic deque implementations are generally slower than specialised queue and stack implementations (if this was not the case, then the faster deque could be used as a queue or stack). For example, the ConcurrentLinkedDeque in the OpenJDK adds about 40% overhead to the Michael-Scott queue. In our case, the TS deque adds 17% overhead to the hardcoded TS queue and 67% overhead to the hardcoded TS stack. Consequently we do not compare our TS deque to other deques, but rather to faster queue and stack implementations.

Gorelik and Hendler describe a similar timestamping approach to ours in their short paper announcing the asymmetric flat combining (AFC) queue [6]. In the AFC queue, enqueued items are timestamped and stored in thread-local buckets. Their approach differs in several respects, however. (1) Their remove relies on flat-combining-style consolidation rather than `CAS`, making their algorithm blocking; (2) their timestamps are Lamport clocks similar to TS-stutter, not hardware-generated intervals; (3) they define a queue rather than a deque. The fact that they use TS-stutter and define only a queue dramatically simplifies dequeuing in comparison to the TS deque (it also simplifies their proof of

linearizability). The fact they do not use interval time-stamping prevents them from trading off the cost of enqueuing and dequeuing. As a result, although inserts are faster, their removes are slower than other flat-combining queues.

The LCRQ queue [2] and the SP queue [11] both index elements using an atomic counter. The LCRQ queue uses indices to detect buffer overflows in a ring buffer, whereas the SP queue uses indices to identify elements in thread-local lists. In both queues elements are found and removed via independent indices generated by a second atomic counter. However, dequeue operations do not look for one of the oldest elements as in our TS deque, 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 deque need not be unique or even ordered, and the performance of the TS deque 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 was inspired by the paper on testing FIFO queues [8]. There the (unsychronised) RDTSC instruction was used to order the invocations of operations. RDTSCP has since been used in the design of an STM by Ruan et al. [24], who investigate the instruction's synchronisation behaviour on a single processor and between processors.

The elimination-backoff stack is based on a Treiber stack, but detects concurrent push and pop operations using a collision array [10]. Whenever it detects concurrent push, the pop operation returns the pushed element without modifying the stack. Our TS deque also performs elimination, detecting concurrent insert and remove operation by comparing timestamps. Note that in the case of the TS deque a remove operation which eliminates a concurrent insert is faster than a normal uncontended remove. 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.

In our experiments we also compare the TS deque with the flat combining queue [15]. This consists of a linked-list queue and a lock which controls the access to the linked list. A thread which acquires the lock not only executes its own operation on the linked-list queue but also the operations of all other threads which did not get the lock. Unlike the flat combining queue, our TS deque is lock-free.

3 Algorithm Correctness

3.1 Sequential Specifications

Deque. Assume a set of values Val. The set of abstract deque states DS is the set of sequences Val^*. Let $\sigma \in \mathsf{DS}$ be an arbitrary deque state. Then `insertR` and `removeR` have the following sequential specifications ('$+\!\!+$' means sequence concatenation):

– `insertR(v)` – Update the abstract state to $\sigma +\!\!+ [\mathtt{v}]$.
– `removeR()` – If $\sigma = []$, return EMPTY. Otherwise, σ must be of the form $\sigma' +\!\!+ [v']$ for some value v'. Update the abstract state to σ' and return v'.

`insertL` and `removeL` have analogous specifications.

TS-buffer. The TS-buffer is the underlying data-structure we use to implement our TS deque. Informally, a TS-buffer is a specialised index which associates a unique identifier with each stored value and timestamp, and which supports retrieval of leftmost and rightmost values.

Formally, we assume a set of *buffer identifiers*, ID representing individual buffer elements; a set of *timestamps*, TS, with partial order $<_{\mathsf{TS}}$ and top element $\top$; and a two-element set of *kinds*, $\{\mathsf{L}, \mathsf{R}\}$, recording whether the element was added to the left or right of the deque. Many elements in the buffer can be associated with the same timestamp.

Furthermore, we assume a set of abstract values called *bufferstamps*, BS, with total order $<_{\mathsf{BS}}$ and top element $\top$. Bufferstamps have no reality as values in the implementation, but rather record the order in which timestamps are *assigned to elements*. (This may differ from both the order elements are added to the buffer, and the order on the timestamps themselves). Bufferstamps are needed as part of the abstract specification because we in fact do not (quite) require that `tryRem` removes the strictly leftmost or rightmost element according to timestamp order; rather a `tryRem` can ignore an arbitrary subset of the elements inserted while it executes. The TS-buffer *does* respect the combination of timestamp and bufferstamp order. See the aside at the end of this section for more discussion of bufferstamps.

The abstract state of the TS-buffer is a partial map from identifiers to value-timestamp-bufferstamp-kind tuples:

$$\mathsf{Buf} \colon \mathsf{ID} \rightharpoonup (\mathsf{Val} \times \mathsf{TS} \times \mathsf{BS} \times \mathsf{Kind})$$

We define left and right orders $<^{\mathsf{L}}_{\mathsf{buf}}$ and $<^{\mathsf{R}}_{\mathsf{buf}}$ over timestamp-bufferstamp-kind tuples:

$$\begin{aligned}(x, a, k_1) <^{\mathsf{L}}_{\mathsf{buf}} (y, b, k_2) \overset{\Delta}{\iff}& \\ &(k_1 = k_2 = \mathsf{L} \wedge x <_{\mathsf{TS}} y \wedge a <_{\mathsf{BS}} b) \vee \\ &(k_1 = k_2 = \mathsf{R} \wedge y <_{\mathsf{TS}} x \wedge b <_{\mathsf{BS}} a) \vee \\ &(k_1 = \mathsf{R} \wedge k_2 = \mathsf{L})\end{aligned}$$

$$\begin{aligned}(x, a, k_1) <^{\mathsf{R}}_{\mathsf{buf}} (y, b, k_2) \overset{\Delta}{\iff}& \\ &(k_1 = k_2 = \mathsf{R} \wedge x <_{\mathsf{TS}} y \wedge a <_{\mathsf{BS}} b) \vee \\ &(k_1 = k_2 = \mathsf{L} \wedge y <_{\mathsf{TS}} x \wedge b <_{\mathsf{BS}} a) \vee \\ &(k_1 = \mathsf{L} \wedge k_2 = \mathsf{R})\end{aligned}$$

Intuitively, if two tuples are ordered $p <^{\mathsf{L}}_{\mathsf{buf}} q$, then the value associated with q is further to the left in the buffer. Similarly, $p <^{\mathsf{R}}_{\mathsf{buf}} q$ says q is further to the right. As elements are added to both sides of the buffer, these orders need to deal with elements inserted on both sides.

The TS-buffer functions have the following sequential specifications (B is the abstract prestate of the TS-buffer):

- `newTimestamp(v)` – pick a timestamp $t \neq \top$ such that for all timestamps $t' \neq \top$ already in B, $t' <_{\mathsf{TS}} t$. Note that this means many elements can be issued the same timestamp if the thread is preempted before writing it into the buffer.
- `insR(v)` – Pick an ID $i \notin \text{dom}(B)$. Update the abstract state to: $B[i \mapsto (v, \top, \top, \mathsf{R})]$ and return i.
- `setTimestamp(i,t)` – assume that $B(\mathtt{i}) = (v, \top, \top, k)$. Pick a bufferstamp $b \neq \top$ such that for all bufferstamps $b' \neq \top$ already in B, $b' <_{\mathsf{BS}} b$. Update the abstract state to $B[\mathtt{i} \mapsto (v, \mathtt{t}, b, k)]$.
- `tryRemR()` – There are three possible behaviours:
  1. Nondeterministically fail and return $\mathsf{invalid}$. In the implementation, this corresponds to a failed CAS caused by another thread pre-empting the removal.
  2. If the map is empty ($\text{dom}(B) = \emptyset$) return empty.
  3. Pick an ID i such that: $B(i) \mapsto (v, t, b, k)$, and

$$\begin{aligned}&\nexists i', v', t', b', k'.\ t' \neq \top \wedge \\ &\quad B(i') = (v', t', b', k') \wedge (t, b, k) <^{\mathsf{R}}_{\mathsf{buf}} (t', b', k')\end{aligned}$$

  Update the abstract state to $B[i \mapsto \bot]$ and return v. Because timestamps and bufferstamps need not be ordered, there may be many maximal elements with respect to $<^{\mathsf{R}}_{\mathsf{buf}}$ that can be chosen by `tryRemR`.

Aside: why are bufferstamps needed? The natural sequential specification of `tryRem` would be for it to remove a maximal element according to timestamp order, $<_{\mathsf{TS}}$. However, our implementation of `tryRem` is not linearizable with respect to this specification. We do not generate and set timestamps in a single atomic step during insertion, as this would require strong synchronisation. Instead, elements are inserted untimestamped, meaning `tryRem` may observe an element before it acquires its timestamp. Untimestamped elements cannot be safely ordered, so `tryRem` considers them maximal in $<_{\mathsf{TS}}$. Therefore, two elements may be ordered $a <_{\mathsf{TS}} b$ before b gets a timestamp, but then ordered $b <_{\mathsf{TS}} a$ afterwards. This lack of a consistent order prevents `tryRem` from linearizing with respect to the simple specification.

Instead we augment our specification with bufferstamps, recording the order that timestamps are assigned to elements. This ensures that if $a <_{\mathsf{buf}} b$ at one point, then $b \not<_{\mathsf{buf}} a$ at all subsequent points (a kind of monotonicity). This is sufficient to ensure that `tryRem` sees a consistent order, and thus allows linearizability. Weakening the specification in this way does not affect the correctness of the TS deque because any reordering with respect to $<_{\mathsf{TS}}$ can only happen in overlapping calls to `insert`. Thus, affected operations could be linearized in either order.

3.2 Correctness Argument

We now sketch the correctness argument for the TS deque. For a detailed correctness argument, see Appendix A.

We use linearizability [13], the standard correctness condition for concurrent algorithms. Proofs of linearizability commonly proceed by finding *linearization points*: the syntactic point in each method call where it appears to atomically take effect. This approach is tricky to apply to our algorithm as the order on `insert` operations is only fixed later in the trace by the order on `remove` operations. Instead, we show directly that for any trace we can build a corresponding total linearization order.

Assume we have a trace $\mathcal{T}$ arising from our algorithm. The trace is a sequence consisting of *operations*: calls and returns to deque methods and atomic calls to the TS-buffer methods. (We assume our TS-buffer is linearizable with respect to the specification given above – see Section 4.2 and Appendix B for a correctness argument.) As is standard, we assume all method calls in $\mathcal{T}$ have returned.

Building a linearization order. We start with the empty relation on methods in the trace, and build up the linearization order incrementally from $\mathcal{T}$. Loosely, the stages in the construction are as follows:

1. Order sequential methods according to the order from returns to calls.
2. Order `insert-remove` pairs where the value returned by the `remove` was added by the `insert`.
3. Order pairs of `remove` operations according to the order of their final, successful call to `tryRem`.
4. Order pairs of `insert` operations according to the order already fixed for their associated `remove` operations.
5. Order `remove-insert` pairs where the final `tryRem` in the `remove` is ordered before the `ins` in the `insert`.
6. Order `insert-remove` pairs which eliminate each other.
7. Order `remove-insert` pairs where it does not contradict the transitive closure of the already created order.

Lemma 1. *The constructed relation is acyclic.*

Proof. By case analysis and appeal to the sequential specification of TS-buffer. This is by far the trickiest proof – for details see Appendix A.

Order between any remaining unordered methods is irrelevant to the algorithm behaviour, so we pick as a candidate linearization order $<_{\mathsf{lin}}$ *any* total order which includes the constructed relation.

Lemma 2. $<_{\mathsf{lin}}$ *respects the order of sequentially-executing methods.*

Proof. By stage 1 of the construction.

Lemma 3. $<_{\mathsf{lin}}$ *satisfies the sequential deque specification.*

Proof. By contradiction. If $<_{\mathsf{lin}}$ does not satisfy the sequential specification, there must be a first method r that violates it. An `insert` method cannot violate the specification, so we assume without loss of generality that r is a `removeR` returning value e. Furthermore, assume the abstract state before r is σ. There are three possibilities:

1. e is not a value in σ; or
2. e is in σ, but the rightmost value is $e_2 \neq e$; or
3. $e = \mathsf{EMPTY}$ but σ is non-empty.

To complete the proof, we show for each case that the structure of the construction ensures a contradiction. For example, in the first case we appeal to the facts (1) TS-buffer allows each value to be removed at most once, and (2) $<_{\mathsf{lin}}$ orders a `remove` after the `insert` that added its value. This means that e must be in σ, contradicting the assumption. For full details see Appendix A.

Theorem 1. *The TS deque is linearizable with respect to the sequential deque specification.*

Proof. Witnessed for arbitrary trace $\mathcal{T}$ by the linearization order $<_{\mathsf{lin}}$ and Lemmas 2 and 3.

Theorem 2. *The TS deque is lock-free.*

Proof. Straightforward from the fact that the TS-buffer and thread-local buffers are lock-free. This follows from the structure of removal: an attempt to remove an element can only fail when another thread pre-empts it and succeeds.

4 Implementation Details

4.1 Timestamping Algorithms

The sequential specification of TS-buffer (Section 3.1) requires that a newly generated timestamp has a higher value than any contained in the buffer. To satisfy this, we use the fact that a timestamp has to be generated before it can appear in the buffer. All our implementations generate timestamps greater than or equal to all previously generated timestamps, thereby satisfying the sequential specification.

TS-atomic. The TS-atomic algorithm takes a timestamp from a global counter using an atomic fetch-and-increment instruction (such instructions are available on most modern processors). This guarantees that a new timestamp is greater than any timestamp generated before.

TS-hardware. The TS-hardware algorithm uses the x86 RDTSCP instruction [16] to read the current value of the TSC register. The TSC register counts the number of processor cycles since the last processor reset. Correctness of our algorithm depends on `newTimestamp` issuing a timestamp later than any currently held in the TS-buffer. Ruan et al [24] have tested RDTSCP on various x86 systems as part of their development of a transactional memory system. Our understanding of [24, 16] is that RDTSCP provides sufficient cross-processor synchronisation to ensure correctness, and in our experiments on multiprocessors, we have seen no anomalies arising from TSC behaviour. However, we are still corresponding with Ruan to better understand the precise synchronisation guarantees on TSC.

TS-stutter. The TS-stutter algorithm uses thread-local counters which are synchronized by the clock synchronization algorithm of Lamport [17]. To generate a new timestamp a thread first reads the values of all thread-local counters. It then increments the maximum value by one, stores the new value in its thread-local counter, and returns the value as the new timestamp. Note that the TS-stutter timestamping algorithm does not utilize AWAR or RAW synchronization [3].

The TS-stutter timestamping algorithm may return the same timestamp multiple times, but never returns a timestamp that already exists in the buffer. This is because whenever a timestamp exists in the buffer, there exists at least one thread-local counter with the same or greater timestamp. The TS-stutter algorithm would therefore read the value of that thread-local counter and return a greater timestamp.

TS-interval. The TS-interval algorithm does not return one value, but rather an interval timestamp 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 $b <_{\mathsf{TS}} c$. That is, if the two intervals overlap, the timestamps are unordered.

The TS-interval algorithm is correct because the upper limit of any interval timestamp in the buffer is less than the lower limit generated by the TS-interval algorithm. Therefore we can choose as the linearization point the generation of the first timestamp of the interval.

In our experiments we use the TS-hardware algorithm 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 5.3.

4.2 The TS-buffer

Listing 1.2 shows the pseudocode of our TS-buffer implementation, the underlying data-structure used to implement the TS deque (see Listing 1.1).

Listing 1.2. TS-buffer algorithm (`insL` / `tryRemL` / `isMoreL` are defined analogously). The implementation of the thread-local buffer is described in Appendix C, Listing 1.3.

```
1  TS_Buffer{
2   ThreadLocalBuffer tlBuffers[numThreads];
3
4   TimestampedItem insR(Element element) {
5    TimestampedItem item = createItem(element);
6    threadID = getThreadID();
7    tlBuffers[threadID].insRtl(item);
8    return item;
9   }
10
11  TimestampedItem tryRemR() {
12   TimestampedItem rightMostItem;
13   int startTime=newTimestamp();
14   int containingBuffer;
15   start=random();
16   for (i=0 to numThreads-1) {
17    TimestampedItem item=
18      tlBuffer[(start+i)%numThreads].getRtl();
19    if (isMoreR(item, rightMostItem)) {
20     rightMostItem=item;
21     containingBuffer=(start+i)%numThreads;
22    }
23   }
24   if (empty())  // Emptiness check.
25     return emptyItem;
26   if (rightMostItem.wasAddedRight()) {
27    if (tlBuffer[containingBuffer].
28         tryRemRtl(rightMostItem))
29     return rightMostItem;
30   } else {
31    if (startTime>=rightMostItem.timestamp) {
32     if (tlBuffer[containingBuffer].
33          tryRemRtl(rightMostItem))
34      return rightMostItem;
35    }
36   }
37   return invalidItem;
38  }
39
40  bool isMoreR (TimestampedItem item1,
41                TimestampedItem item2) {
42   if (item2.wasAddedLeft()) {
43     if(item1.wasAddedRight())
44     return true;
45    return (item1.timestamp<item2.timestamp)
46   } else {
47    if(item1.wasAddedLeft())
48     return false;
49    return (item1.timestamp>item2.timestamp)
50   }
51  }
52 }
```

Thread-local buffers. We implement the TS-buffer using a collection of thread-local buffers. Each thread inserts elements into its own buffer but may remove elements from any thread's buffer. We assume the thread-local buffers support the following operations:

- `insRtl` / `insLtl` – insert an element on the right or left of the buffer.
- `getRtl` / `getLtl` – return the identifier of the rightmost or leftmost element according to $<_{\mathtt{TS}}$.
- `tryRemRtl` / `tryRemLtl` – try to remove the identified element from the buffer.

Our thread-local buffer implementation is a doubly-linked list design discussed in Appendix C. Only one thread inserts elements into a thread-local buffer so no strong synchronization is necessary in `insRtl` / `insLtl`. Removal is a two-stage process: first the thread marks a `taken` flag [9] to indicate the nodes has been removed, then marked nodes are unlinked lazily, either immediately or by later operations.[3]

If the TS deque is to be used solely as a queue or stack, the thread-local buffers can be optimised. Both queue- and stack-specific thread-local buffers require only a singly-linked list, and the queue-specific thread-local buffer also does not require a taken-flag.

The current implementation of the TS deque supports only a static number of threads. Support for a dynamic number of threads is future work.

TS-buffer operations. All the thread-local buffers are linked from a single array `tlBuffers`, which we assume is indexed by thread IDs. We assume a function `getThreadID()` which retrieves the current thread's identifier.

The `insR` operation first retrieves the thread-local buffer of the executing thread and then inserts the element into the thread-local buffer using its `insRtl` operation.

Starting at a random index, the `tryRemR` operation searches all thread-local buffers for a right-most elements, using the `isMoreR` operation (lines 16–23). The random start index increases the chance that concurrent `tryRemR` operations endup with different elements and thus avoid contention. If the discovered element was inserted using an `insR` operation, then `tryRemR` tries to remove it from its thread-local buffer using `tryRemRtl` (lines 26–29). However, if the element was inserted using an `insL` operation, then `tryRemR` first checks that the element was inserted before `tryRemR` started. When `tryRemR` begins, it records a starting timestamp (line 13) and then it compares this start time with the candidate element (line 31). Only if the element was inserted before `tryRemR` was called does `tryRemR` try to remove it from the thread-local buffer (line 32). If `tryRem` could remove an element inserted after it was called, then there could exist an element with an earlier timestamp that was missed while iterating through the thread-local buffers.

[3] To avoid polluting our benchmarks with memory management effects, unlinked nodes are not reclaimed. In a real-world implementation, it would be straightforward to use garbage collection or hazard pointers [19] to manage reclamation.

The isMoreR operation approximates the order $<^R_{\mathsf{buf}}$ defined in Section 3.1. If both elements were inserted at the right, then the element with the later timestamp is more right. If both elements were inserted at the left, then the element with the earlier timestamp is more right. Otherwise the element inserted at the right is more right.

The TS-buffer insL, tryRemL, and isMoreL work analogously to insR, tryRemR, and isMoreR.

To simplify the presentation, we have omitted the emptiness check from Listing 1.2. Our implementation use the same approach as it is used and proven correct in [7]. That is, each thread has two thread-local arrays the size of the number of thread-local buffers. Whenever a thread encounters an empty buffer, it stores the left and right pointer of the buffer in these arrays. If in two subsequent executions of tryRemR no thread-local buffer returns an element, and the left or right pointers of all thread-local buffers have not changed, then tryRemR returns EMPTY.

Correctness argument. Here we only sketch a correctness argument for TS-buffer – for details, see Appendix B. The concrete state of the TS-buffer is a partial mapping from a thread ID and buffer identifier to a value-timestamp-bufferstamp-kind tuple:

$$\mathsf{LBuf} \colon (\mathsf{Thr} \times \mathsf{ID}) \rightharpoonup (\mathsf{Val} \times \mathsf{TS} \times \mathsf{BS} \times \mathsf{Kind})$$

Note that bufferstamps do not exist as values in the concrete state, since these are a product of execution order. However, we can easily reconstruct them by examining the order that timestamps are assigned in the trace. To build the corresponding set of abstract states, we:

1. erase the thread identifiers.
2. erase a subset of buffer identifiers. This reflects the fact that the linearization point for the removal can be pushed earlier than the point an element is removed from the thread-local buffer.

Theorem 3. *The TS-buffer implementation is linearizable with respect to the specification given in Section 3.1.*

Proof. All the operations aside from tryRemR / tryRemL are atomic, either because they consist of a single call to other linearizable modules (newTimestamp, insR, insL) or because they are implemented as atomic assignment (setTimestamp). Consequently, we simply show that the atomic calls return appropriate values and update the abstract state correctly.

The tricky part of the proof lies in showing that a call to tryRemR / tryRemL satisfies the specification. By careful case analysis, we show that any overlapping calls to other tryRem operations must be linearizable in a way which allows the call to return the rightmost or leftmost element at some point during its execution. See Appendix B for details.

5 Performance Analysis

Our experiments compare the performance and scalability of the TS deque with several state-of-the-art algorithms:

- the Michael-Scott (MS) queue [20] because it is the de-facto standard lock-free queue implementation;
- the flat-combining (FC) queue [15] because it is a very fast queue;
- the Treiber stack [23] because it is the de-facto standard lock-free stack implementation; and
- the elimination-backoff (EB) stack [10] because it is the fastest concurrent stack we are aware of.[4]

We ran our experiments on two 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 memory running Linux 3.2.0-38; and
- an AMD-based server with four 6-core 2.1GHz AMD Opteron processors (24 cores), 6MB shared L3-cache, and 110GB of memory running Linux 3.2.0-29.

We show results for the 40-core machine in the paper body. Results for the 24-core machine are in Appendix D.

All measurements were done in the Scal Benchmarking Framework [5]. To avoid measurement artifacts unrelated to the benchmarked data-structures the framework uses a custom memory allocator which performs cyclic allocation [21] in preallocated thread-local buffers for objects smaller than 4096 bytes. Larger objects are allocated with the standard allocator of glibc. All memory is allocated cache-aligned when it is beneficial to avoid cache artifacts. The framework is written in C/C++ and compiled with gcc 4.7.2 and -O3 optimizations.

Scal provides implementations of the MS queue, the FC queue, the Treiber stack, and the EB stack. Unlike the description of the EB stack in [10] we access the elimination array before the stack because this improves scalability in our experiments. We configured the elimination array to a size of 8 in the high contention benchmarks and to a size of 12 in the low contention benchmarks. Operations wait in the elimination array for $13\mu s$ and $20\mu s$ in the high contention benchmarks and in the low contention benchmarks, respectively. These values were optimal in our benchmarks when exercised with 80 threads on the 40-core machine, but may be suboptimal for lower numbers of threads. Similarly, the TS deque configurations we discuss later are selected to be optimal for 80 threads on the 40-core machine and may be suboptimal for other configurations. On the 24-core machine the size of the elimination array is either 8 or 2 and the

[4] We decided against benchmarking the DECS stack [4] because (1) no implementation is available for our platform and (2) according to their experiments, in peak performance it is no better than a FC stack. We decided against benchmarking the FC stack because the EB stack outperforms it when configured to access the backoff array before the stack itself.

waiting time is either $5\mu s$ or $2\mu s$, depending on the contention. These values were optimal in our benchmarks when exercised with 24 threads on the 24-core machine.

We benchmark the TS deque either as a queue (TS Queue) by using insertL to insert elements and removeR to remove them; as a stack (TS Stack) by using the insertR and removeR; and as a deque (TS Deque) by randomly inserting and removing elements on both sides.

We compare the data-structures in producer-consumer microbenchmarks where each thread is either a dedicated producer which inserts 1000000 elements into the data-structure or a dedicated consumer which removes 1000000 elements from the data-structure. All figures show the performance in successful operations per millisecond, averaged over 5 executions. To avoid measuring empty removal, remove 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.

Comment: the LCRQ queue. We also experimented with the LCRQ queue, using the implementation available at `http://mcg.cs.tau.ac.il/projects/lcrq`[5]. Although in general the LCRQ outperforms the TS deque, its speed seems to be at the cost of extremely variable performance. Execution times on our benchmarks can vary by more than 100% in identical situations (all other implementations vary by less than 10%). Because of this, we felt we were unable to get meaningful results from the LCRQ and do not show it in our graphs.

5.1 Performance and Scalability

Figures 1 and 2 show performance and scalability in a producer-consumer benchmark where half of the threads are producers and half of the threads are consumers. We present only the results of the TS deque using TS-hardware and TS-interval timestamping because they provide the fastest insert and remove operations, respectively. The performance of TS-atomic and TS-stutter timestamping is discussed in Section 5.2.

For TS-interval timestamping we use the optimal delay when exercised with 80 threads, derived from the experiments in Section 5.3. The delay thus depends on the machine and benchmark, e.g. for the TS deque benchmark, we use $25\mu s$ on the 40-core machine and $5\mu s$ on the 24-core machine. The impact of different delays on performance is discussed in Section 5.3. In addition to the generic TS deque implementation we also measured implementations which use hardcoded queue-specific (Hardcoded TS Queue) and stack-specific (Hardcoded TS Stack) thread-local buffers.

[5] This version fixes a bug in [2] which lets the LCRQ drop enqueued elements.

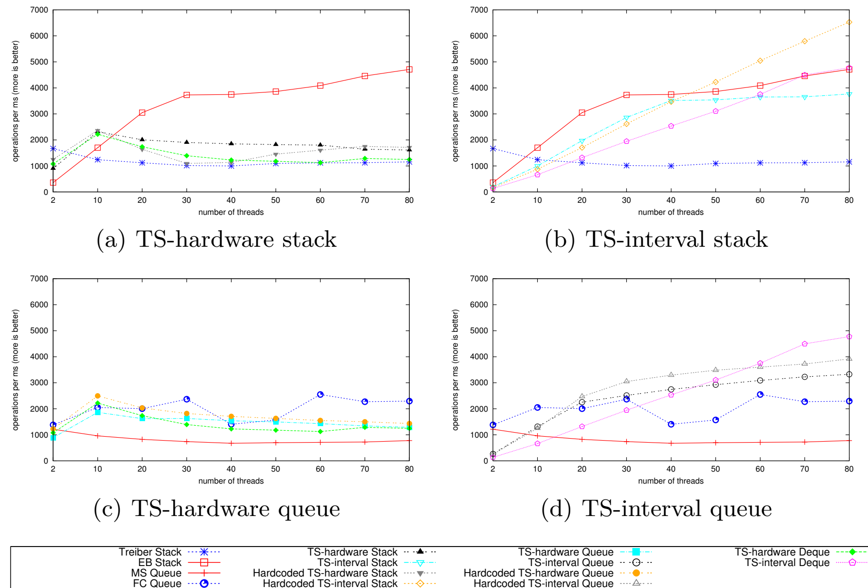

Fig. 1. High contention producer-consumer microbenchmarks on the 40-core machine.

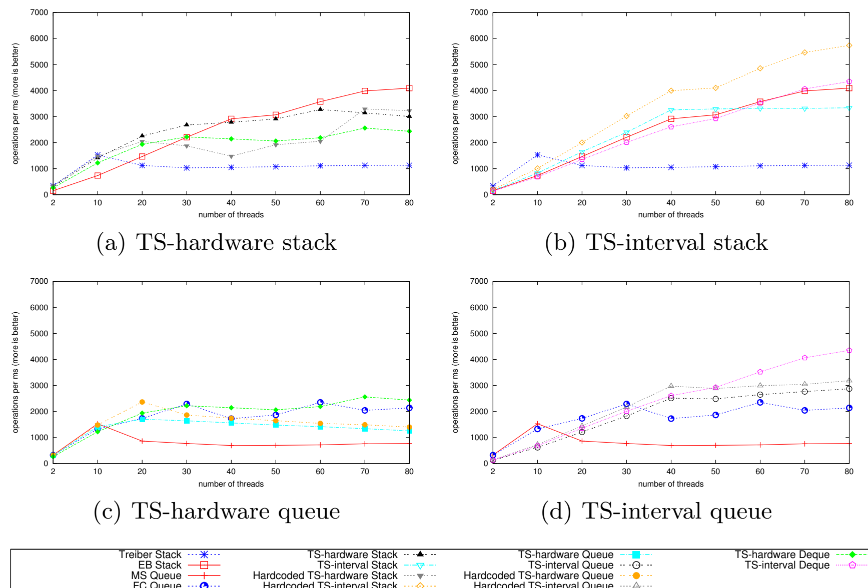

Fig. 2. Low contention producer-consumer microbenchmarks on the 40-core machine.

Comparison between implementations. The TS data-structures that use TS-interval timestamping perform better than the TS data-structures that use TS-hardware timestamping. TS-interval timestamping even scales up to 80 threads on the 40-core machine. On the 24-core machine only the TS stack and the TS deque with TS-interval timestamping scale. The TS queue and the hardcoded TS queue stagnate between 2000 and 2500 operations per millisecond. In general the TS stack and the TS deque are faster than the TS queue because remove operations can process elements of concurrent insert operations without checking the timestamp of the elements, as discussed in Section 4.2. The TS deque is faster than the TS queue and the TS stack in the high contention producer-consumer benchmark because elements are inserted and removed on both sides of the deque, resulting in less contention on each side.

Comparison with other data-structures. With more than 20 threads all TS queues are faster than the MS queue. The FC queue is faster than the TS queues with TS-hardware timestamping but slower than the TS queue with TS-interval timestamping when the benchmark is exercised with more than 40 threads. The TS stacks with TS-interval timestamping are faster than the Treiber stack. The EB stack is faster then the TS deque when used as a stack but is slower than the hardcoded TS-interval Stack.

5.2 Analysis of the Timestamping Algorithms

In this section we compare the different timestamping algorithms described in Section 4.1 by using producer-only and consumer-only benchmarks. All timestamping algorithms are measured with the TS deque used as queue, because the performance of a queue does not depend on the presence of concurrent insert and remove operations. TS-interval timestamping is configured to use the same delay as in the experiments in Section 5.1. We include the results of the MS queue and the Treiber stack as baselines.

Producer-only benchmarks. Figure 3 shows the results of the producer-only microbenchmark in the high contention scenario. The TS deque with TS-hardware timestamping and the TS deque with TS-interval timestamping do not perform any software synchronization in their insert operation. Therefore both of them scale linearly with an increasing number of threads. However, with TS-interval timestamping the TS deque is significantly slower because of the constant overhead introduced by the delay within the timestamping. Especially with a low number of threads the overhead introduced by the delay dominates performance. With a lower number of threads a shorter delay would be optimal. TS-atomic timestamping is faster than TS-stutter timestamping. With TS-atomic timestamping the TS deque is faster than both the Treiber stack and the MS queue whereas the TS deque with TS-stutter timestamping is slower than the Treiber stack but still faster than the MS queue on both machines.

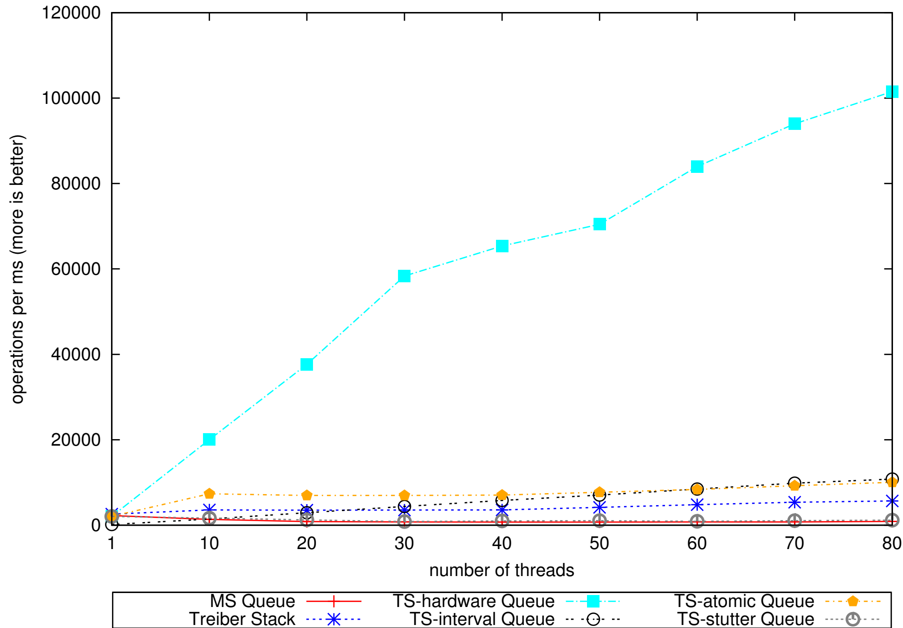

Fig. 3. High contention producer-only microbenchmarks on the 40-core machine.

Consumer-only benchmarks. Figure 4 shows the results of the consumer-only microbenchmark in the high contention scenario. At the beginning of the benchmark, the TS queue is pre-filled concurrently by the same number of threads as there are later consumers during the measured execution of the benchmark. Elements may thus be ordered less strictly when using TS-interval and TS-stutter timestamping.

On the 40-core machine the TS deque with TS-interval timestamping scales positively up to 80 threads. With the other timestamping techniques the TS deques scale negatively. Also the MS queue and the Treiber stack scale negatively. With an increasing number of threads all timestamping algorithms are faster than the MS queue. With TS-stutter timestamping the TS deque is even faster than the Treiber stack with more than 30 threads.

Note that TS-interval and TS-stutter timestamping only improve the performance of the remove operation if elements are inserted concurrently. If the TS deque was pre-filled sequentially, then the performance of the TS deque with TS-interval or TS-stutter timestamping would be the same as the performance of the TS deque with TS-atomic timestamping. An insert-sequentially, remove-concurrently workload is the worst case for the TS deque because elements cannot share timestamps. With TS-atomic timestamping all elements have unique timestamps, so the TS deque behaves the same whether inserting sequentially or concurrently.

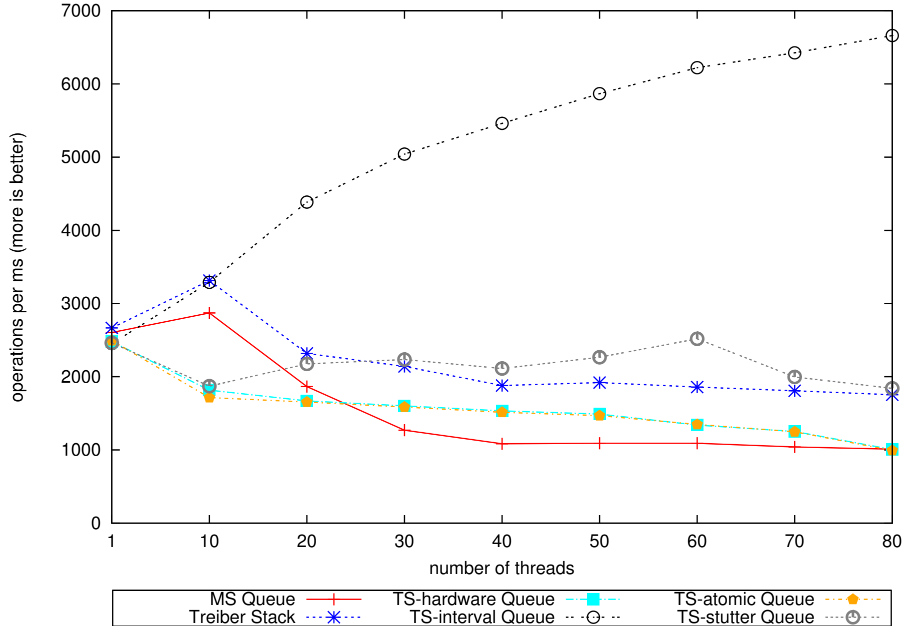

Fig. 4. High contention consumer-only microbenchmarks on the 40-core machine.

5.3 Analysis of TS-Interval Timestamping

Figure 5 shows the performance of the TS data-structures along with the average number of `CAS` retries needed in each `remove`. These figures were collected with TS-interval timestamping and an increasing delay 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 5.1. For each machine / benchmark combination, we used the lowest delay which was within 3% of the maximum performance of any delay.

Initially the performance of the TS data-structures increases with an increasing delay, but beyond $10\mu s$ the performance stagnates for the TS queue, hardcoded TS queue, and TS stack. Beyond $30\mu s$ the insert operation becomes slower than remove and the overall performance declines again. For the TS deque it takes longer to reach the optimal performance because elements are inserted on both sides of the deque and therefore there is less contention on each side. Unlike the other TS data-structures, the hardcoded TS stack has one optimal delay which is significantly better than other delays.

The figure also shows that high performance correlates strongly with a drop in `CAS` retries. We conclude from this that the impressive performance we achieve with interval timestamping arises from reduced contention in `remove`.

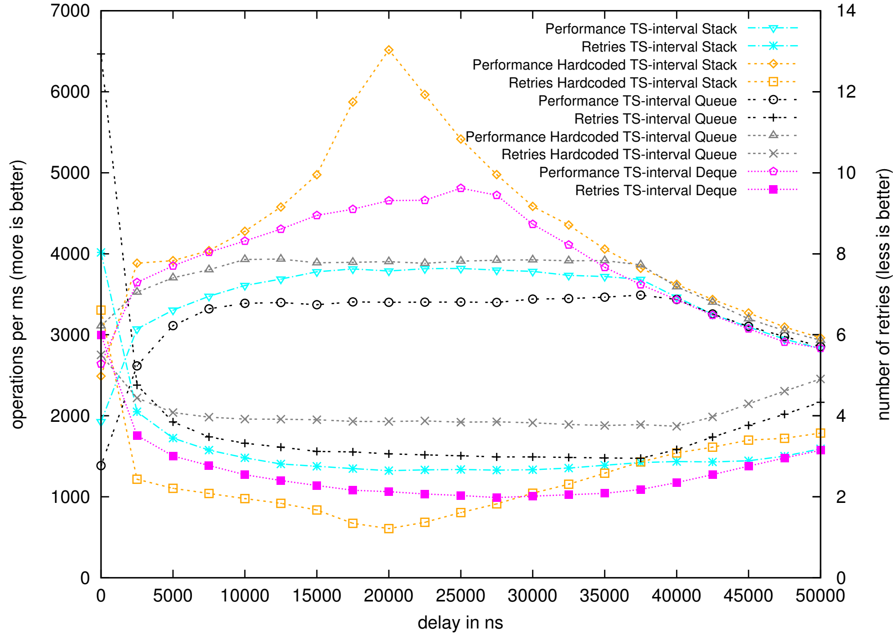

Fig. 5. High contention producer-consumer benchmark using TS-interval timestamping with increasing delay on the 40-core machine, exercising 40 producers and 40 consumers.

Results for the 24-core machine show similar behaviour (see Appendix D). The main difference is that the maxima on the 24-core machine is shifted to the left: for lower numbers of threads shorter delays are sufficient to avoid contention problems.

6 Conclusions

We present a novel approach to implementing ordered concurrent data-structures like queues, stacks, and deques. Elements are timestamped upon insertion and removed according to the order of their timestamps. Optimal performance is achieved by weakening timestamp order so that elements are unordered if their inserting methods overlap. Our experiments demonstrate that our implementations are competitive or even faster than the state of the art.

Our work represents a first step in designing data-structures based on weakly-ordered timestamps. It is quite surprising to us that we have been able to relax internal orders so drastically while preserving a linearizable specification. In future work, we plan to experiment with relaxing other internal ordering constraints, with dynamically adjusting the level of order in response to contention, and with exploiting relaxations in the underlying memory model.

References

1. OpenJDK: An open-source implementation of the Java platform, standard edition.
2. Y. Afek and A. Morrison. Fast concurrent queues for x86 processors. In *PPoPP*. ACM, 2013.
3. 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 *POP*, 2011.
4. G. Bar-Nissan, D. Hendler, and A. Suissa. A dynamic elimination-combining stack algorithm. In *OPODIS*, 2011.
5. Computational Systems Group, University of Salzburg. Scal framework.
6. M. Gorelik and D. Hendler. Brief announcement: an asymmetric flat-combining based queue algorithm. In *PODC*, pages 319–321, 2013.
7. 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 *Computing Frontiers*. ACM, 2013.
8. A. Haas, C. Kirsch, M. Lippautz, and H. Payer. How FIFO is your concurrent FIFO queue? In *RACES*. ACM, 2012.
9. T. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*. Springer, 2001.
10. D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*. ACM, 2004.
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. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.
13. M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3), 1990.
14. M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. In *OPODIS*, pages 401–414. Springer, 2007.
15. D. H. I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, pages 355–364. ACM, 2010.
16. Intel. Intel 64 and ia-32 architectures software developer's manual, volume 3b: System programming guide, part 2, 2013.
17. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications ACM*, 21, July 1978.
18. M. Michael. CAS-based lock-free algorithm for shared deques. In *Euro-Par*. Springer, 2003.
19. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15:491–504, 2004.
20. M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275. ACM, 1996.
21. H. H. Nguyen and M. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *ISMM*. ACM, 2007.
22. H. Sundell and P. Tsigas. Lock-free deques and doubly linked lists. *Journal of Parallel and Distributed Computing*, 68, 2008.
23. R. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, April 1986.
24. Y. L. Wenjia Ruan and M. Spear. Boosting timestamp-based transactional memory by exploiting hardware cycle counters. In *TRANSACT*, 2013.

A Correctness Argument for Deque

Assume we have a trace $\mathcal{T}$ arising from our algorithm. The trace is a sequence consisting of *operations*: calls and returns to deque methods and atomic calls to the TS-buffer methods. As is standard in linearizability, we assume $\mathcal{T}$ is *complete* – every method call has an associated return. We write $a <_{\mathcal{T}} b$ if operations a and b are ordered in the trace. To make the notation easier, we assume that each method instance (consisting of the method's call, return, and internal calls to TS-buffer) has a unique identifier, a, b, etc. Given two methods a and b, we write $a \xrightarrow{\mathsf{val}} b$ if a is an `insert`, b is a `remove`, and the value returned by b was originally inserted by a. By the semantics of TS-buffer, each `insert` is associated with at most one `remove`.

Building a linearization order. We now build a linearization order, beginning by building an acyclic relation $<_{\mathsf{lin}}$ over methods in $\mathcal{T}$. We start with the empty relation $<^0_{\mathsf{lin}}$, and iteratively build $<^i_{\mathsf{lin}}$ by applying each stage of the following construction until it no longer applies. $<_{\mathsf{lin}}$ is the relation that results once none of the stages applies.

1. Precedence and value orders:
   (a) Let a_{ret} be the return operation of one method, and b_{call} be the call operation of another. If $a_{\mathsf{ret}} <_{\mathcal{T}} b_{\mathsf{call}}$, then $a <^{i+1}_{\mathsf{lin}} b$.
   (b) If $a \xrightarrow{\mathsf{val}} b$ then $a <^{i+1}_{\mathsf{lin}} b$.
2. Let r_1, r_2 be a pair of `remove` operations (either right or left) that are unordered in $<^i_{\mathsf{lin}}$. If their final `tryRem` operations are ordered $\mathtt{tryRem}_1 <_{\mathcal{T}} \mathtt{tryRem}_2$, then these operations are ordered $r_1 <^{i+1}_{\mathsf{lin}} r_2$.
3. `insertR` operations (`insertL` operations are the same with L and R inverted):
   (a) Let a_1 be an `insertR` operation with associated $\mathtt{setTS}_1$, let r_1 be a `removeR` operation with associated $\mathtt{tryRem}_1$, and let a_2 be an `insertR` operation with associated $\mathtt{ins}_2$ operation. If $a_1 \xrightarrow{\mathsf{val}} r_1$ and $\mathtt{setTS}_1 <_{\mathcal{T}}$ tryRem_1, and $\mathsf{tryRem}_1 <_{\mathcal{T}} \mathsf{setTS}_2$ then a_1 and a_2 are ordered $a_1 <^{i+1}_{\mathsf{lin}} a_2$.
   (b) Let a_1 be an `insertR` operation with associated $\mathtt{setTS}_1$, let r_1 be a `removeL` operation with associated $\mathtt{tryRem}_1$, and let a_2 be an `insertR` operation with associated setTS_2. If $a_1 \xrightarrow{\mathsf{val}} r_1$ and $\mathtt{tryRem}_1 <_{\mathcal{T}} \mathsf{setTS}_2$, then a_1 and a_2 are ordered $a_1 <^{i+1}_{\mathsf{lin}} a_2$.
   (c) Let a_1, a_2 be a pair of `insertR` operations that are unordered in $<^i_{\mathsf{lin}}$.
      i. Suppose there exists no `remove` operation r_2 such that $a_2 \xrightarrow{\mathsf{val}} r_2$, and there exists a `removeR` operation r_1 with $a_1 \xrightarrow{\mathsf{val}} r_1$. Let $\mathtt{setTS}_1$ be associated with a_1 and $\mathtt{tryRemR}_1$ be associated with r_1. If $\mathtt{setTS}_1 <_{\mathcal{T}}$ $\mathtt{tryRemR}_1$, then $a_2 <^{i+1}_{\mathsf{lin}} a_1$
      ii. If there exists no `remove` operation r_2 such that $a_2 \xrightarrow{\mathsf{val}} r_2$, and there exists a `removeL` operation r_1 with $a_1 \xrightarrow{\mathsf{val}} r_1$ then the pair is ordered $a_1 <^{i+1}_{\mathsf{lin}} a_2$.

iii. Suppose there exists a `removeR` operation r_2 and a `remove` operation r_1 (either left or right) with $a_1 \stackrel{val}{\to} r_1$ and $a_2 \stackrel{val}{\to} r_2$ and $r_2 <^i_{\mathsf{lin}} r_1$ and $\mathtt{setTS}_2 <_{\mathcal{T}} \mathtt{tryRem}_2$, where $\mathtt{setTS}_2$ is the `setTS` operation of a_2 and tryRem_2 is the `tryRem` operation of r_2. Then the pair is ordered $a_1 <^{i+1}_{\mathsf{lin}} a_2$
iv. Suppose there exists a `removeL` operation r_1 and a `remove` operation r_2 (either left or right) with $a_1 \stackrel{val}{\to} r_1$ and $a_2 \stackrel{val}{\to} r_2$ and $r_1 <^i_{\mathsf{lin}} r_2$. Then the pair is ordered $a_1 <^{i+1}_{\mathsf{lin}} a_2$.

4. `insert` / `remove` pairs: A `remove` operation r_1 and an `insert` operation a_2 are ordered $r_1 <_{\mathsf{lin}} a_2$ if the final $\mathtt{tryRem}_1$ of r_1 and the `ins` operation ins_2 of a_2 are ordered $\mathtt{tryRem}_1 <_{\mathcal{T}} \mathtt{ins}_2$ in the trace.
5. Elimination for `insertR` operations (`insertL` operations are the same with L and R inverted):
   (a) Let a_1 be an `insertR` operation and let r_1 be a `removeR` operation such that $a_1 \xrightarrow{\mathsf{val}} r_1$. Let $\mathtt{setTS}_1$ be associated with a_1 and let $\mathtt{tryRemoveR}_1$ be associated with r_1, and let $\mathtt{tryRemoveR}_1 <_{\mathcal{T}} \mathtt{setTS}_1$. Let x be another method (either `insert` or `remove`) with $x <^i_{\mathsf{lin}} r_1$. Then $x <^{i+1}_{\mathsf{lin}} a_1$.
   (b) Let a_1 be an `insertR` operation and let r_1 be a `removeR` operation such that $a_1 \xrightarrow{\mathsf{val}} r_1$. Let $\mathtt{setTS}_1$ be associated with a_1 and let $\mathtt{tryRemoveR}_1$ be associated with r_1, and let $\mathtt{tryRemoveR}_1 <_{\mathcal{T}} \mathtt{setTS}_1$. Let x be another method (either `insert` or `remove`) with $r_1 <^i_{\mathsf{lin}} x$. Then $a_1 <^{i+1}_{\mathsf{lin}} x$.

Lemma 4. *After Stage 1 the linearization relation $<^1_{\mathsf{lin}}$ does not contain cycles.*

Proof. By construction, any cycle between in $<^1_{\mathsf{lin}}$ necessarily correspond to a cycle in $<_{\mathcal{T}}$. Therefore $<^i_{\mathsf{lin}}$ is acyclic after Stage 1.

Lemma 5. *After Stage 2 the linearization relation $<^2_{\mathsf{lin}}$ does not contain cycles.*

Proof. $<^2_{\mathsf{lin}}$ is a total order on remove operations because it is based on the total order $<_{\mathcal{T}}$ on `tryRem` calls. As also $<^1_{\mathsf{lin}}$ is an order based on the total order $<_{\mathcal{T}}$, the union of both orders is again an order.

Next we show that no cycle is constructed in Stage 3.

Lemma 6. *Let a_1, a_2 be two* `insertR` *operations, let* ret_1 *be the return operation of a_1, let* call_2 *be the call of a_2, and let* $\mathsf{ret}_1 <_{\mathcal{T}} \mathsf{call}_2$. *Then after Stage 3 $a_2 \not<^3 i_{\mathsf{lin}} a_1$.*

Proof. If a_1, a_2 are ordered $a_2 <^3_{\mathsf{lin}} a_1$, then this order has been created in one of the stages in Stage 3. We will consider now each stage:

– Stage 3a and Stage 3b: Let setTS_1 belong to a_1, ins_2 belong to a_2, let r_2 be the `remove` operation with $a_2 \xrightarrow{val} r_2$, and let tryRem_2 belong to r_2. Assuming $\mathsf{ret}_1 <_{\mathcal{T}} \mathsf{call}_2$ and $a_2 <^3_{\mathsf{lin}} a_1$, then $\mathsf{tryRem}_2 <_{\mathcal{T}} \mathsf{setTS}_1 <_{\mathcal{T}} \mathsf{ret}_1 <_{\mathcal{T}} \mathsf{call}_2 <_{\mathcal{T}} \mathsf{ins}_2$. However, this means that an element is removed from the TS-Buffer before it is inserted, which violates the sequential specification of the TS-Buffer.

– Stage 3(c)i and Stage 3(c)iii: Let insR_1, setTS_1 belong to a_1 and newTS_2, setTS_2 belong to a_2. Let v_1 be the buffer ID returned by insR_1, and let v_2 be the buffer ID returned by insR_2. Let r_1 be the `removeR` operation with $a_1 \xrightarrow{val} r_1$, and let $\mathsf{tryRemR}_1$ belong to r_1. As a_1, a_2 were not ordered in Stage 3a, $\mathsf{setTS}_2 <_{\mathcal{T}} \mathsf{tryRemR}_1$. Assuming $\mathsf{ret}_1 <_{\mathcal{T}} \mathsf{call}_2$ and $a_2 <^3_{\mathsf{lin}} a_1$, then $\mathsf{setTS}_1 <_{\mathcal{T}} \mathsf{ret}_1 <_{\mathcal{T}} \mathsf{call}_2 <_{\mathcal{T}} \mathsf{newTS}_2 <_{\mathcal{T}} \mathsf{setTS}_2 <_{\mathcal{T}} \mathsf{tryRemR}_1$. Because of the sequential specification of `newTS` and `setTS` $B(v_1) <^R_{\mathsf{buf}} B(v_2)$ and according to the sequential specification $\mathsf{tryRemR}_1$ would not return the value of v_1 but the value of v_2, which contradicts the assumption that $a_1 \xrightarrow{val} r_1$.
– Stage 3(c)ii and Stage 3(c)iv: Let setTS_1 belong to a_1, and newTS_2, setTS_2 belong to a_2. Let v_1 be the buffer ID returned by insR_1, and let v_2 be the buffer ID returned by insR_2. Let r_2 be the `removeL` operation with $a_2 \xrightarrow{val} r_2$, and let $\mathsf{tryRemL}_2$ belong to r_2. As a_1, a_2 were not ordered in Stage 3b, $\mathsf{setTS}_2 <_{\mathcal{T}} \mathsf{tryRemL}_1$. Assuming $\mathsf{ret}_1 <_{\mathcal{T}} \mathsf{call}_2$ and $a_2 <^3_{\mathsf{lin}} a_1$, then $\mathsf{setTS}_1 <_{\mathcal{T}} \mathsf{ret}_1 <_{\mathcal{T}} \mathsf{call}_2 <_{\mathcal{T}} \mathsf{newTS}_2 <_{\mathcal{T}} \mathsf{setTS}_2 <_{\mathcal{T}} \mathsf{tryRemL}_2$. Because of the sequential specification of `newTS` and `setTS` $B(v_2) <^L_{\mathsf{buf}} B(v_1)$ and according to the sequential specification $\mathsf{tryRemL}_2$ would not return the value of v_2 but the value of v_1, which contradicts the assumption that $a_2 \xrightarrow{val} r_2$.

Lemma 7. *Assume there exists a cycle $x_1 <^3_{\mathsf{lin}} x_2 <^3_{\mathsf{lin}} \cdots <^3_{\mathsf{lin}} x_n <^3_{\mathsf{lin}} x_1$, then there exist $1 \le j < k < l \le n$ such that x_j, x_k, x_l are* `insertR` *operations (or* `insertL` *operations).*

Proof. Assume the cycle $x_1, \ldots, x_n$ has minimal size, i.e. there does not exist a cycle with less operations. There was no cycle in $<^2_{\mathsf{lin}}$, so the cycle was closed by ordering two $\mathsf{insertR}$ operations $a_1 <^3_{\mathsf{lin}} a_2$. In stage 3 any pair of $\mathsf{insertR}$ operations is ordered at most once, so $a_2 \not<^3_{\mathsf{lin}} a_1$. Let call_1 belong to a_1, lets setTS_2, ret_2 belong to a_2. Let x_s be the operation in $x_1, \ldots, x_n$ with $x_s <^3_{\mathsf{lin}} a_1$. Let call_s and ret_s belong to x_s.

If x_s is an `insertR` operation, then we are done because a_1, a_2, x_s are the three operations x_j, x_k, x_l.

Now assume that x_s is not an `insertR` operation. The order $x_s <^i_{\mathsf{lin}} a_1$ can only be constructed in Stage 1a because of precedence. Therefore $\mathsf{ret}_s <_{\mathcal{T}} \mathsf{call}_1$. As a_1 and a_2 are not ordered in Stage 1a it holds that $\mathsf{call}_s <_{\mathcal{T}} \mathsf{ret}_s <_{\mathcal{T}} \mathsf{call}_1 <_{\mathcal{T}} \mathsf{ret}_2$. Therefore a_2 and x_s are not ordered by precedence in Stage 1a. If a_2 and x_s were ordered by value in Stage 1b, then Stage 3a or Stage 3b would apply and construct the order $a_2 <^3_{\mathsf{lin}} a_1$. Therefore there has to exist at least one more operation x_t in $x_1, \ldots, x_n$ with $a_2 <^3_{\mathsf{lin}} x_t$. Let call_t belong to x_t. The operations a_2 and x_t were ordered either in Stage 1a because of precedence or in Stage 1b because of a value relation. Assume a_2, x_t were ordered in Stage 1a. Together with Lemma 6 this means that $\mathsf{ret}_s <_{\mathcal{T}} \mathsf{call}_1 <_{\mathcal{T}} \mathsf{ret}_2 <_{\mathcal{T}} \mathsf{call}_t$. However, this would mean that x_s and x_t would already have been ordered $x_s <^3_{\mathsf{lin}} x_t$ in Stage 1a and therefore there would be a smaller cycle without a_1 and a_2 in it.

Now assume that a_2 and x_t were ordered in Stage 1b, and assume that x_t is a `remove` operation. If $x_t <^3_{\mathsf{lin}} x_s$, then they were either ordered in Stage 1a,

in which case x_s could be removed from the cycle because of the transitivity of precedence, or they were ordered in Stage 2, in which case Stage 3a would construct the order $a_2 <^3_{\mathsf{lin}} a_1$ or a_1 and a_2 were not ordered at all. The same holds if there are only `remove` operations between x_t and x_s in the cycle.

Therefore we assume now that x_u is the first `insertL` operation x_u in the cycle after x_t, and x_{u-1} is the `remove` operation which precedes x_u in the cycle. As x_s precedes a_1 and x_{u-1} precedes x_u, either the order $x_s <^3_{\mathsf{lin}} x_u$ or $x_{u-1} <^3_{\mathsf{lin}} a_1$ was constructed in Stage 1a. Both would allow a smaller cycle, and therefore no cycle is possible with only two `insertR` operations.

Lemma 8. *Assume there exists a cycle $x_1 <^3_{\mathsf{lin}} x_2 <^3_{\mathsf{lin}} \cdots <^3_{\mathsf{lin}} x_n <^3_{\mathsf{lin}} x_1$ of minimal size, with $1 \leq j < k < l \leq n$ such that x_j, x_k, x_l are* `insertR` *operations (or* `insertL` *operations).*

Let setTS_j ,setTS_k, setTS_l belong to x_j, x_k, x_l, respectively, and let r_j, r_k, r_l be three `remove` *operations with $x_j \xrightarrow{val} r_j$, $x_k \xrightarrow{val} r_k$, and $x_l \xrightarrow{r}_l$. Let tryRem_j, tryRem_k, tryRem_l belong to r_j, r_k, r_l, respectively. Then for any $f \in \{j, k, l\}$, if r_f is a* `removeR` *operation, then $\mathsf{setTS}_f <_{\mathcal{T}} \mathsf{tryRem}_f$, i.e. any pair of operations x_j, x_k, x_l is ordered after Stage 3.*

Proof. Let ins_j, ins_k, ins_l belong to x_j, x_k, x_l, respectively.

No cycle is created before Stage 3. Therefore the cycle is created in Stage 3 and without loss of generality $k = j + 1$, i.e. x_k is a direct successor of x_j and $x_j \not<^{1a}_{\mathsf{lin}} x_k$.

It cannot be that r_j is a `removeR` operation and $\mathsf{tryRem}_j <_{\mathcal{T}} \mathsf{setTS}_j$ because there does not exist a stage in Stage 3 which would order $x_j <^3_{\mathsf{lin}} x_k$.

Now assume that r_k is a `removeR` operation and $\mathsf{tryRem}_k <_{\mathcal{T}} \mathsf{setTS}_k$. As x_k is not ordered before any other operation in Stage 3, x_{k+1} is either r_k, or x_k and $x_k + 1$ were ordered in Stage 1a.

Assume r_k is x_{k+1}. As only Stage 3a Stage 3b can order $x_j <^3_{\mathsf{lin}} x_k$, and because of the sequential specification of the TS-buffer, it holds that $\mathsf{ins}_j <_{\mathcal{T}} \mathsf{tryRem}_j <_{\mathcal{T}} \mathsf{tryRem}_k <_{\mathcal{T}} \mathsf{setTS}_k$. Hence also $r_j <^3_{\mathsf{lin}} r_k$, and x_k could be replaced by r_j in the cycle, which contradicts Lemma 7.

Now assume that x_k, x_k+1 were ordered in Stage 1a. With a similar argument as in the proof of Lemma 7 this would mean that x_j and x_k could be excluded from the cycle, which contradicts the assumption that the cycle has minimal size.

Now assume that for both x_j and x_k with $x_k = x_{j+1}$ it holds that $\mathsf{setTS}_j <_{\mathcal{T}} \mathsf{tryRem}_j$ and $\mathsf{setTS}_k <_{\mathcal{T}} \mathsf{tryRem}_k$. We already handled the case that $x_l = x_{k+1}$ and $\mathsf{tryRem}_k <_{\mathcal{T}} \mathsf{setTS}_k$. Therefore $x_{k+1} \neq x_l$, which is also a case that is already handled in the proof of Lemma 7.

Lemma 9. *If there exists a cycle $x_1 <^3_{\mathsf{lin}} x_2 <^3_{\mathsf{lin}} \cdots <^3_{\mathsf{lin}} x_n <^3_{\mathsf{lin}} x_1$, then there exists also a cycle $a_1 <^3_{\mathsf{lin}} a_2 <^3_{\mathsf{lin}} a_3 <^3_{\mathsf{lin}} a_1$ consisting only of* `insertR` *operations a_1, a_2, a_3 (or* `insertL` *operations).*

Proof. With Lemma 8 it is guaranteed that in the cycle $x_1 <^3_{\mathsf{lin}} x_2 <^3_{\mathsf{lin}} \cdots <^3_{\mathsf{lin}} x_n <^3_{\mathsf{lin}} x_1$ there exist at least three `insertR` operations which are all related by $<^3_{\mathsf{lin}}$.

We can then pick any pair of $\mathsf{insertR}$ operations a_1, a_2 with $a_1 <^3_{\mathsf{lin}} a_2$ where a_2 is not the successor of a_1 in the cycle. As $a_1 <^3_{\mathsf{lin}} a_2$, we can shrink the cycle by removing all operations between a_1 and a_2 without breaking the cycle. We repeat shrinking the cycle until no more shrinking is possible and we end up with a cycle of size 3.

Lemma 10. *After Stage 3 the linearization relation $<^3_{\mathsf{lin}}$ does not contain cycles.*

Proof. With Lemma 9 we only have to consider all possible ways to construct a cycle $a_1 <^3_{\mathsf{lin}} a_2 <^3_{\mathsf{lin}} a_3 <^3_{\mathsf{lin}} a_1$ of `insertR` operations a_1, a_2, a_3, which are finitely many.

In any case we end up either in an order which does not contain a cycle, or the sequential specification of the TS-buffer is violated, or the order in which the stages have to be used is violated. Therefore $<^3_{\mathsf{lin}}$ does not contain cycles.

Lemma 11. *After Stage 4 the linearization relation $<^4_{\mathsf{lin}}$ does not contain cycles.*

Proof. Assume Stage 4 creates a cycle, and assume it is a cycle of minimal size. Therefore there exists a `remove` operation r_1 and an `insert` operation a_2 such that $r_1 <^4_{\mathsf{lin}} a_2$. Without restriction of generality assume that a_2 is an `insertR` operation. There cannot exist a cycle $r_1 <^4_{\mathsf{lin}} a_2 <^4_{\mathsf{lin}} r_1$ because the order $a_2 <^4_{\mathsf{lin}} r_1$ is not created in Stage 1. Therefore exist other operations x_1, $\ldots$, x_n in the cycles.

Assume x_i is a `remove` operation and assume that tryRem_x belongs to x_i, tryRem_1 belongs to r_1, and ins_2 belongs to a_2. If $\mathsf{tryRem}_x <_{\mathcal{T}} \mathsf{tryRem}_1$, then there exists an order $x_i <^4_{\mathsf{lin}} a_2$ and a smaller cycle would be possible which does not contain r_1. If $\mathsf{tryRem}_1 <_{\mathcal{T}} \mathsf{tryRem}_x$, then $r_1 <^4_{\mathsf{lin}} x_i$ and all operations between r_1 and x_i could be removed from the cycle. Therefore r_1 is the only `remove` operation in the cycle of minimal size.

Next, assume that x_i is a `insert` operation, assume that ins_x belongs to x_i, and assume that tryRem_1 belongs to r_1. If $\mathsf{tryRem}_1 <_{\mathcal{T}} \mathsf{ins}_x$, then $r_1 <^4_{\mathsf{lin}} x_i$ and therefore a smaller cycle would be possible which does not contain a_2. Therefore the `ins` operation of all x_i operations are ordered before tryRem_1 in $<_{\mathcal{T}}$.

From the total order of $\mathcal{T}$ it follows that only one pair of operations x_i, x_{i+1} is ordered by precedence in Stage 1a. If there existed a second pair x_j, x_{j+1} which is also ordered by precedence in Stage 1a, then also $x_i <^{1a}_{\mathsf{lin}} x_{j+1}$ or $x_i <^{1a}_{\mathsf{lin}} x_{j+1}$. In both cases a smaller cycle is possible.

As only Stage 1a orders `insertR` operations and `insertL` operations, there can exist only one `insertR` operation x_i in the cycle such that x_{i+1} is a `insertL` operation. Since the ins_{i+1} operation of x_{i+1} is ordered $\mathsf{ins}_{i+1} <_{\mathcal{T}} \mathsf{tryRem}_1$, the ret_i of x_i is also ordered before tryRem_1. However, in that case there would exist a rule in Stage 3 which creates the order $x_i <^4_{\mathsf{lin}} a_2$ and would therefore create a smaller cycle, or the cycle would require a second pair of operations being ordered in Stage 1a, which would also create the possibility of a smaller cycle. Therefore all operations x_i are `insertR` operations.

For a similar reason as in the proof of Lemma 8 all but one `insertR` operation in the cycle are totally ordered in $<^4_{\mathsf{lin}}$. Hence, if there exists a cycle, then it

contains at most three `insertR` operations. However, with only three `insertR` operations and one `remove` operation it is not possible to create a cycle using only Stage 1 to Stage 4 in order without some of the TS-buffer operations violating their sequential specification. Therefore Stage 4 does not create a cycle.

Lemma 12. *After Stage 5 the linearization relation $<^5_{\mathsf{lin}}$ does not contain cycles.*

Proof. Let a_1 be an `insert` operation a_1, let r_1 be a remove operation with $a_1 \xrightarrow{val} r_1$, and let x be a second operation. Assume a cycle is created because a_1, x are ordered $a_1 <^5_{\mathsf{lin}} x$ in Stage 5. Then there would exist a second cycle which can be created by replacing a_1 by r_1, and the second cycle would have already existed after Stage 4. However, this contradicts Lemma 11.

The constructed relation $<_{\mathsf{lin}}$ is acyclic because it is the same as $<^5_{\mathsf{lin}}$.

Now pick any total order $<^t_{\mathsf{lin}}$ that includes $<_{\mathsf{lin}}$. As $<_{\mathsf{lin}}$ is acyclic, such an order must exist.

Lemma 13. $<^t_{\mathsf{lin}}$ *agrees with sequential method order in $\mathcal{T}$.*

Proof. By construction (stage 1a above).

Lemma 14. *The method order $<^t_{\mathsf{lin}}$ satisfies the sequential specification.*

Proof. By contradiction. Assume $<^t_{\mathsf{lin}}$ does not satisfy the sequential specification. As `insert` operations cannot violate the specification, there must be a first `remove` operation which does. Assume without loss of generality that it is a `removeR` method r_1 returning value e_1, and that the abstract state before the operation is the sequence σ. As the sequential specification has been violated, one of the following must hold:

1. e_1 is not a value in σ; or
2. e_1 is in σ, but the rightmost value is $e_2 \neq e_1$; or
3. $e_1 = \mathsf{EMPTY}$ but σ is non-empty.

Consider these three possibilities in order.

$e_1 \notin \sigma$. By the semantics of the TS-buffer, there must be an `insert` i_1 for value e_1 with which r_1 is uniquely associated $i_1 \xrightarrow{\mathsf{val}} r_1$. By the construction of $<^t_{\mathsf{lin}}$, it must be true that $i_1 <^t_{\mathsf{lin}} r_1$ (Stage 1b). Thus e_1 must be a member of σ, ruling out this case.

$e_1 \in \sigma$, *but rightmost value is $e_2 \neq e_1$.* As $e_1, e_2 \in \sigma$, there must exist `insert` operations i_1, i_2 associated with e_1 and e_2, such that $i_1 <^t_{\mathsf{lin}} r_1$ and $i_2 <^t_{\mathsf{lin}} r_1$. Furthermore, as $e_2 \in \sigma$, either i_2 is not associated with a `remove` operation, or the associated `remove` r_2 is ordered $r_1 <^t_{\mathsf{lin}} r_2$.

If both i_1 and i_2 are `insertR` operations, then Stage 3(c)i, Stage 3(c)iii, or Stage 5 in our construction obliges us to order them $i_2 <^t_{\mathsf{lin}} i_1$. However, this means that e_1 is further to the right in σ than e_2, contradicting our assumption.

A similar argument using Stage 3(c)ii, Stage 3(c)iv, and Stage 5 applies if i_1 and i_2 are both `insertL` operations.

If $i_1 = \texttt{insertR}$ and $i_2 = \texttt{insertL}$, then by the sequential semantics, e_1 must be further right in σ than e_2, contradicting our assumption.

If $i_1 = \texttt{insertL}$ and $i_2 = \texttt{insertR}$, then the `insR` call in i_2 must be ordered in $\mathcal{T}$ after the successful `tryRemR` in r_1. Otherwise by the TS-buffer semantics, r_1 could not have returned e_1. However in this situation, stage 4 of our construction obliges us to order $r_1 <^t_{\mathsf{lin}} i_2$, meaning $e_2 \notin \sigma$ and contradicting our assumption.

$e_1 = \mathsf{EMPTY}$*, but* σ *is non-empty.* If σ is non-empty, there must be an `insert` operation i_2 such that $i_2 <^t_{\mathsf{lin}} r_1$ and either (1) the associated remove operation r_2 is ordered $r_1 <^t_{\mathsf{lin}} r_2$, or (2) there is no associated remove operation. If r_2 exists, by stage 2 the `tryRem` in r_2 must be ordered in $\mathcal{T}$ after the `tryRem` in r_1. Consequently the `ins` call in i_2 must be ordered in $\mathcal{T}$ after the `tryRem` in r_1 – otherwise r_1 would not report the queue as empty. By stage 4 we are thus obliged to order the methods $r_1 <^t_{\mathsf{lin}} i_2$, which contradicts our assumption.

Theorem 4. *The deque is linearizable.*

Proof. Application of Lemma 13 and Lemma 14.

B Correctness of TS-buffer

All the operations aside from `tryRemR` / `tryRemL` are atomic, either because they consist of a single call to other linearizable modules (`newTimestamp`, `insR`, `insL`) or because they are implemented as atomic assignment (`setTimestamp`). Consequently, we only need to show that the atomic calls return appropriate values and update the abstract state correctly:

- `newTimestamp` – By assumption, this returns a timestamp greater than any previously call to `newTimestamp`. All the timestamps in the buffer have been generated by `newTimestamp`. Thus the call generates a timestamp greater than any in the buffer. The concrete state is unmodified.
- `insR` / `insL` – Satisfies the specification trivially by appeal to the specification of the thread-local buffer.
- `setTimestamp` – We assume that we have reconstructed the bufferstamps correctly, i.e. so that they strictly increase in the trace with calls to `setTimestamp`. The call then satisfies the specification trivially by appeal to the semantics of assignment.

The tricky part of the proof is showing that `tryRemR` / `tryRemL` satisfy the specification. Without loss of generality, we assume the call is a `tryRemR` call r_1. If the call fails, i.e. returns invalid, then the concrete state is unmodified and the specification is trivially satisfied. If r_1 returns an element e_1, we only need to consider three cases:

1. e_1 was the rightmost element at the invocation of r_1; or

2. e_1 was assigned its timestamp after the invocation of r_1; or
3. any element e_2 that was further to the right of e_1 at the invocation of r_1 was removed within the execution of r_1.

The first possibility is that, at the invocation of r_1, for all elements e_2 in the abstract state it holds that $e_1 \not<^R_{\mathsf{buf}} e_2$. If this holds we can satisfy the sequential specification by picking the invocation as the linearization point. The abstract state is updated to remove e_1 from the buffer, but it remains in the concrete state. The subsequent calls in the buffer do not affect the abstract state. By assumption, r_1's final call to `tryRemRtl` removes e_1 from the concrete state.

The second possibility is that the `insR` operation i_1 that inserts e_1 is executed after the invocation of `tryRemR`. Then immediately after i_1, e_1 has a timestamp / bufferstamp of $\top$. Therefore it holds for all e_2 in the buffer that $e_1 \not<^R_{\mathsf{buf}} e_2$. Thus the linearization point can be added here, and the abstract state updated. Again, e_1 is removed from the concrete state in the final `tryRemRtl`.

Now consider the third possibility: at the time of the invocation r_1 there exists elements $e_2, e_3, \ldots$ in the buffer such that $e_1 <^R_{\mathsf{buf}} e_2$, $e_1 <^R_{\mathsf{buf}} e_3$, etc. at all points during r_1. As r_1 does not discover e_2 in any of the thread-local buffers, it must have been removed by method r_2 before its containing thread-local buffer is encountered by r_1. The same argument applies to e_3, e_4 etc. If no further elements further right than e_1 are added during r_1, we can place the linearization point for r_1 immediately after these thread-local removes have completed.

However, further elements may get inserted into the buffer during the execution of r_1. Suppose some element e'_2 is inserted before r_2, r_3 etc. have returned, and that $e_1 <^R_{\mathsf{buf}} e'_2$ at all points during the execution of r_1. If e'_2 is inserted after the invocations of all the removes r_2, r_3 etc, by the above argument their linearization points can be their invocation points. Thus we can linearize r_1 immediately afterwards.

Suppose e'_2 is inserted before the invocation of some remove r_2. If $e_2 <^R_{\mathsf{buf}} e'_2$ at all points, then e'_2 must have been removed before the return of r_2. Otherwise r_2 would have been obliged to remove it in favour of e_2.

Suppose instead $e_2 \not<^R_{\mathsf{buf}} e'_2$ at some point. This can happen if either (1) both e_2 and e'_2 were in the buffer without timestamps at the invocation of r_2, or (2) `setTimestamp` for e_2 interleaved after `setTimestamp` for e'_2. In the first case then according to the sequential specification r_1 and r_2 can ignore e'_2, and so r_2 can be linearized at its invocation point, and r_1 immediately afterwards. In the second case, `setTimestamp` for e_2 interleaved after `setTimestamp` for e'_2, meaning it also interleaved after the invocation of r_1 and can therefore be ignored at the beginning of r_1 and at the beginning of any r_3, r_4, ... which remove elements further right than e_1 at the invocation of r_1. Thus r_2 can be ignored, and the remaining methods can be linearized as discussed above.

C Thread-Local Buffer

The implementation of the thread-local buffer for the TS deque is similar to the implementation of the concurrent linked deque in the OpenJDK [1]. A thread-local buffer consists of a doubly-linked list of nodes which is accessed by a `left` pointer and a `right` pointer. Both pointers are annotated with ABA-counters to avoid the ABA-problem [12]. Each node contains a `left` pointer, a `right` pointer, a `data` field, a `taken` flag, and an `index` field. The `left` and `right` pointers point to the left neighbor and the right neighbor of the node, respectively, the `data` field stores the element, the `taken` flag indicates if the element of the node has been removed from the thread-local buffer, and the `index` field is used to order and identify nodes.

The doubly-linked list is closed at its ends by nodes which have themselves as their left or right neighbor. The doubly-linked list is initialized with a sentinel node. Initially both left and `right` pointer of the thread-local buffer as well as `left` and `right` pointer of the sentinel node itself point to the sentinel node. Moreover the `taken` flag is set to indicate that the node does not contain an element.

An element is contained in the thread-local buffer if (1) there exists a node in the doubly-linked list that contains the element in its `data` field, if (2) the `taken` flag of that node is not set, if (3) the node is reachable from the `left` pointer of the thread-local buffer following only `right` pointers, and if (4) the node is reachable from the `right` pointer of the thread-local buffer following only `left` pointers. If one of the four conditions does not hold, the element is not considered as contained in the thread-local buffer.

The nodes in the doubly-linked list are sorted by their indices. The right neighbor of any node in the doubly-linked list has a higher index than the node itself. To keep this invariant nodes that are inserted by an `insRtl` operation get positive indices which are greater than any index assigned earlier, and nodes that are inserted by an `insLtl` operation get negative indices which are less than any index assigned earlier. By using this order on the doubly-linked list it is guaranteed that the right-most element in the buffer is contained in the right-most node which is not marked as taken.

Listing 1.3 shows the pseudocode of the thread-local buffer. To insert an element at the right side of the buffer with an `insRtl` operation, first a new node is created with the next free index assigned to it and the element stored in its `data` field. Initially the `taken` flag is not set, and the `right` pointer is set to point to the new node itself. The `insRtl` operation then tries to find the right-most node that has not been marked as taken (line 21-24). In line 29-31 the new node is inserted at the right of that right-most node. Only after the new node is reachable from the `right` pointer of the thread-local buffer the element is considered to be contained in the buffer. If the thread-local buffer is empty, then the iteration in line 21-24 ends at the left-most node of the buffer. To consistently insert the new node, this left-most node becomes a new sentinel node (line 25-27) and the `left` pointer of the thread-local buffer is changed to point to this sentinel node. The new node is then inserted to the right of the

Listing 1.3. Thread-local buffer algorithm.

```
1 ThreadLocalBuffer {
2  Node {
3   Node *left,
4   Node *right,
5   TimestampedItem item,
6   int index,
7   bool taken
8  };
9  <Node*, int> *left; // Pointer with ABA counter.
10 <Node*, int> *right; // Pointer with ABA counter.
11 int nextIndex = 1;
12 void init() {
13  Node *sentinel = createNode(data=0, index=0, taken=true);
14  sentinel.left = sentinel.right = sentinel;
15  left = right = <sentinel, 0>;
16 }
17 void insRtl(TimestampedItem item) {
18  Node *newNode = createNode(item=item, index=nextIndex, taken=false);
19  newNode->right = newNode;
20  nextIndex = nextIndex + 1;
21  <Node*, int> <rightMost, rightAba> = right;
22  while (rightMost->left != rightMost && rightMost->taken) {
23   rightMost = rightMost->left;
24  }
25  if (rightMost = rightMost->left) {
26    <Node*, int> <oldLeft, leftAba> = left;
27    left = <rightMost, leftAba+1>;
28  }
29  newNode->left = rightMost;
30  rightMost->right = newNode;
31  right = newNode;
32 }
33 void insLtl(TimestampedItem item) {
34  // Analogous to insRtl, but the index of a new node is initialized with 'index=-nextIndex'.
35 }
36 <Node*, Node*, int> getRtl() {
37  <Node*, int> <oldLeft, leftAba> = left;
38  <Node*, int> <oldRight, rightAba> = right;
39  Node* result = oldRight;
40  while (true) {
41   if(result->index < oldLeft->index) return <NULL, oldRight, rightAba>;
42   if(!result->taken) return <result, oldRight, rightAba>;
43   if(result->left == result) return <NULL, oldRight, rightAba>;
44   result = result->left;
45  }
46 }
47 <T, Node*, int> getLtl() {
48  // Analogous to getRtl.
49 }
50 bool tryRemRtl(<Node*, int> <oldRight, aba>, Node *node) {
51  if (CAS(node->taken, false, true)) {
52   CAS (right, <oldRight, aba>, <node, aba>);
53   return true;
54  }
55  return false;
56 }
57 bool tryRemLtl(<Node*, int> <oldRight, aba>, Node *node) {
58  // Analogous to tryRemRtl.
59 }
60 }
```

sentinel node. By using the sentinel node we guarantee that both the `right` pointer and the `left` pointer of the thread-local buffer always point to the same doubly-linked list.

The `getRtl` operation first reads the `left` pointer and the `right` pointer of the thread-local buffer. Then it starts iterating over the doubly-linked list starting at the node stored in the `right` pointer and following the `left` pointers of the nodes. The iteration stops either at a node younger than the node the `left` pointer of the thread-local buffer was pointing to at the beginning of the `getRtl` operation, or at a node which has not been marked as taken, or at the end of the doubly-linked list where the `left` pointer of the node points to itself. If a node is found whose `taken` flag is not set, then the element of the node is returned. Otherwise, `getRtl` returns `NULL`. Additionally `getRtl` returns the value of the `right` pointer at the beginning of its execution. The value of the `right` pointer is then used in Line 60 in the second `CAS` of the `tryRemRtl` operation and in the emptiness check of the TS-buffer.

The `tryRemRtl` operation tries to set the `taken` flag with a `CAS` and returns `true` if it succeeds. Otherwise the `tryRemRtl` operation returns `false`. After succeeding in the first `CAS` the operation additionally tries to adjust the `right` pointer of the thread-local buffer with a `CAS`. The purpose of that `CAS` is an optimization which is described below.

The `insLtl`, `getLtl`, and `tryRemLtl` operations work analogous to their counterparts `insRtl`, `getRtl`, and `tryRemLtl`. Except for swapping ‘right’ and ‘left’, the only difference is that negative indices are assigned when elements are inserted by an `insLtl` operation to keep the doubly-linked list sorted, and that the comparison of node indices is changed to account for the negative indices (i.e., ‘$<$’ is swapped with ‘$>$’ and ‘$>$’ is swapped with ‘$<$’).

Correctness. The correctness of the thread-local buffer is based on the invariant that the doubly-linked list is sorted by the insertion side and the insertion time of its nodes. Thereby the right-most or left-most element in the thread-local buffer can be found simply by finding the right-most or the left-most node in the doubly-linked list, respectively. As the thread-local buffer allows only a single thread to insert elements, we do not have to care about concurrent `insRtl` and `insLtl` operations. The atomicity of the `tryRemRtl` and `tryRemLtl` operations is guaranteed by using the `taken` flag to mark the element of a node as removed atomically.

The correctness of the `getRtl` operation is more subtle and in one aspect it does not conform to its specification. In Line 41 in Listing 1.3 the iteration stops at a node with a lower index than the node the `left` pointer of the thread-local buffer was pointing to at the beginning of the operation. However, there may be a node further to the left which was added to the doubly-linked list by an `insLtl` operation after `getRtl` reads the value of the `left` pointer. Stopping the iteration is an unsound optimization which, however, does not affect the behaviour of the TS-buffer. No matter if the element of a node further to the left was returned to the `tryRemR` operation of the TS-buffer, it would not be considered as the right-most element anyways because of the `if`-condition in

Line 31 in Listing 1.2. The condition says that elements will not be considered which were inserted by an `insLtl` operation after the invocation of the `tryRemR` operation. Therefore the behaviour of `tryRemR` of the TS-buffer does not change no matter if `getRtl` returns an element which gets inserted after the invocation of `getRtl` or if it returns `NULL` instead.

The optimization, however, makes the `insRtl` operation of the thread-local buffer much simpler. After Line 29 in the `insRtl` operation the new node is already reachable from the `left` pointer of the thread-local buffer but not from the `right` pointer. However, by stopping the iteration in the `getLtl` operation at a node older than the node which is referred to by the `right` pointer of the thread-local buffer, the new node is not found before also the `right` pointer of the thread-local buffer is updated. By making the new node reachable from both sides at the same time the `insRtl` method is linearizable with respect to its sequential specification because the new element becomes visible in the thread-local buffer by a single operation.

When a `getRtl` operation returns an element, then it is linearizable with respect to its specification because there always exists a point in time within the execution of `getRtl` where that element is stored in the right-most node in the doubly-linked list which is not marked as taken. If the element is stored in the right-most node at the time when `getRtl` is invoked, then the invocation time of `getRtl` is a valid linearization point. If there exist nodes at the invocation time of `getRtl` which are further right than the node which contains the returned element, then these nodes would have been encountered in the iteration except if they were removed by a `tryRemRtl` operation in the meantime. In that case, the linearization point of `getRtl` is right after the linearization point of the last `tryRemRtl` operation which removes one of the nodes which were further right when `getRtl` was invoked.

The linearization point of the `tryRemRtl` operation is the `CAS` which sets the `taken` flag of the node. The second `CAS` which adjusts the `right` pointer of the thread-local buffer is an optimization which tries to change the `right` pointer to a node closer to the right-most pointer in the doubly-linked list which is not marked as taken. The same optimization is done in the `insRtl` operation in Line 21-24 where the new node is not added at the right end of the doubly-linked list but next to the right-most node which is not marked as taken.

D Experiments on the 24-core machine

Figures 6 and 7 show performance and scalability in the producer-consumer benchmark on the 24-core server machine where half of the threads are producers and half of the threads are consumers.

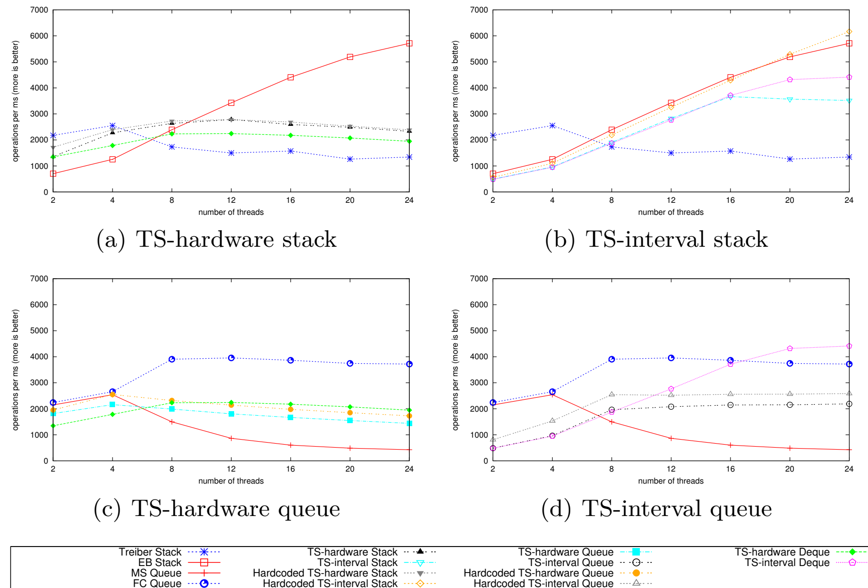

Fig. 6. High contention producer-consumer microbenchmarks on the 24-core machine.

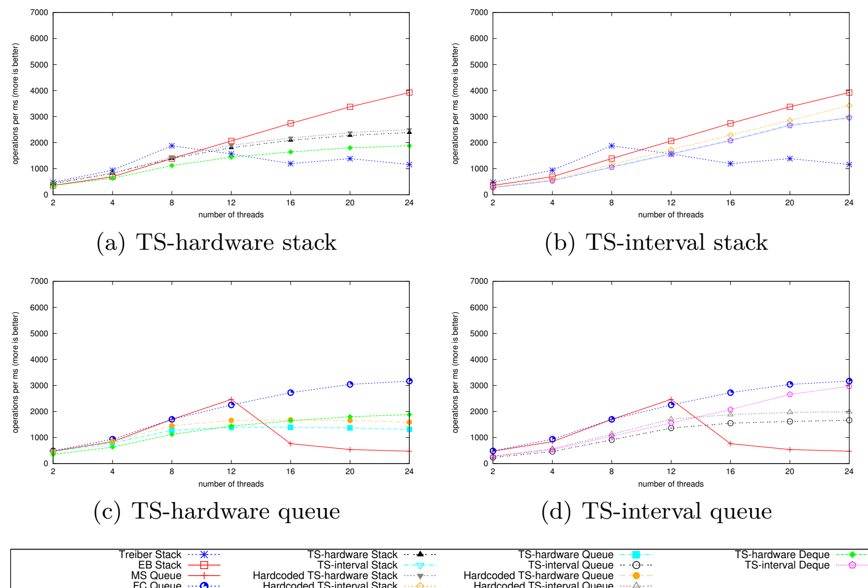

Fig. 7. Low contention producer-consumer microbenchmarks on the 24-core machine.

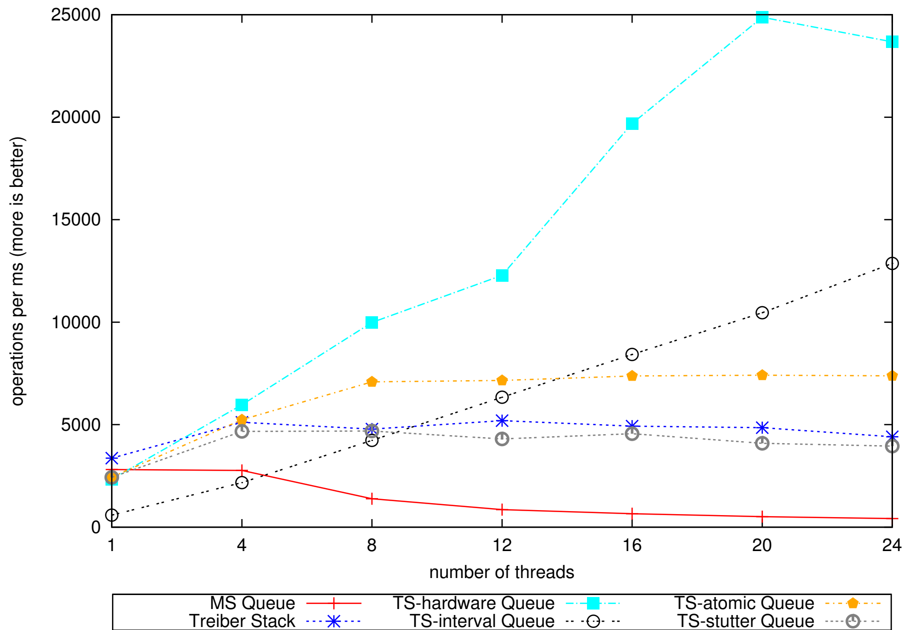

Fig. 8. High contention producer-only microbenchmarks on the 24-core server machine.

Figure 8 shows the performance of the TS deque used as a queue in a high contention producer-only benchmark with different timestamping algorithms.

Figure 9 shows the performance of the TS deque used as a queue in a high contention consumer-only benchmark with different timestamping algorithms.

Figure 10 shows the performance of the TS deque with TS-interval timestamping with an increasing delay in the high-contention producer-consumer benchmark on the 24-core server machine. All measurements are done with 12 threads inserting 1000000 elements into the TS deque and 12 threads removing 1000000 elements from the TS deque.

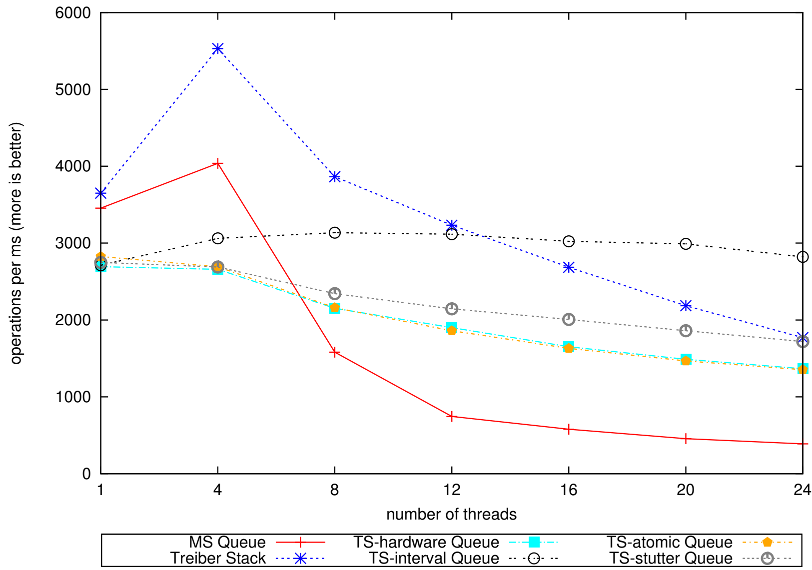

Fig. 9. High contention consumer-only microbenchmark on the 24-core server machine.

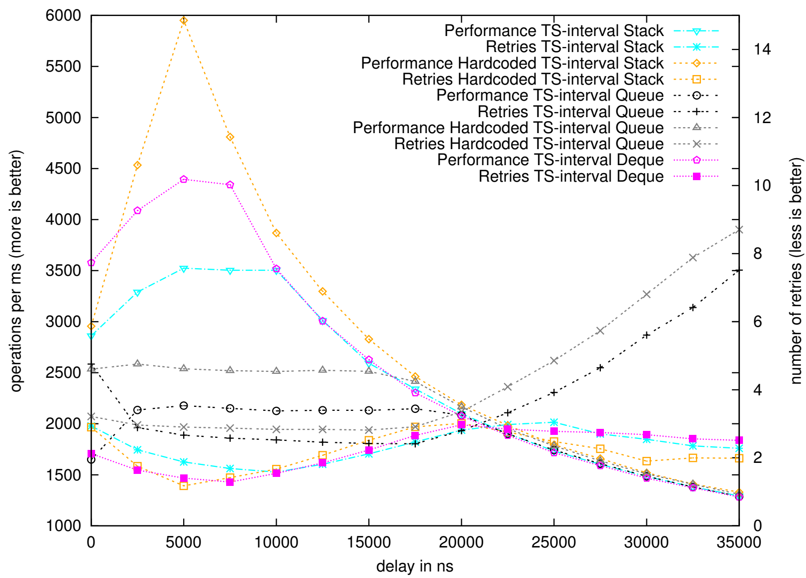

Fig. 10. High contention producer-consumer benchmark using TS-interval timestamping with increasing delay on the 24-core server maching, exercising 12 producers and 12 consumers.