Fast Concurrent Data-Structures Through Explicit Timestamping

Mike Dodds, Andreas Haas, Christoph M. Kirsch
mike.dodds@york.ac.uk, ahaas@cs.uni-salzburg.at, ck@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.

Key Ideas

- Order elements in the data-structure only partially by using explicit timestamping.
- Store elements in thread-local buffers to avoid synchronization of insert operations.
- Use the RDTSCP CPU instruction for highly-scalable timestamp generation.
- Use intervals as timestamps to reduce the order on timestamps while still providing linearizability.

TS Deque Pseudo Code

```
TS_Deque {
  TS_Buffer buffer;
  void insertR(Element element) {
    item = buffer.insR(element);
    timestamp = buffer.newTimestamp();
    buffer.setTimestamp(item, timestamp);
  }
  Element removeR() {
    do {
      item = buffer.tryRemoveR();
    } while (!item.isValid());
    if (item.isEmpty(){
      return EMPTY;
    else
      return item.element;
  }
}
```

insertL and **removeL** are defined analogously.

Correctness

- The TS deque is linearizable with respect to the sequential specification of a deque.
- The remove operations of the TS deque are lock-free, the insert operations are wait-free.

Additional Informantion

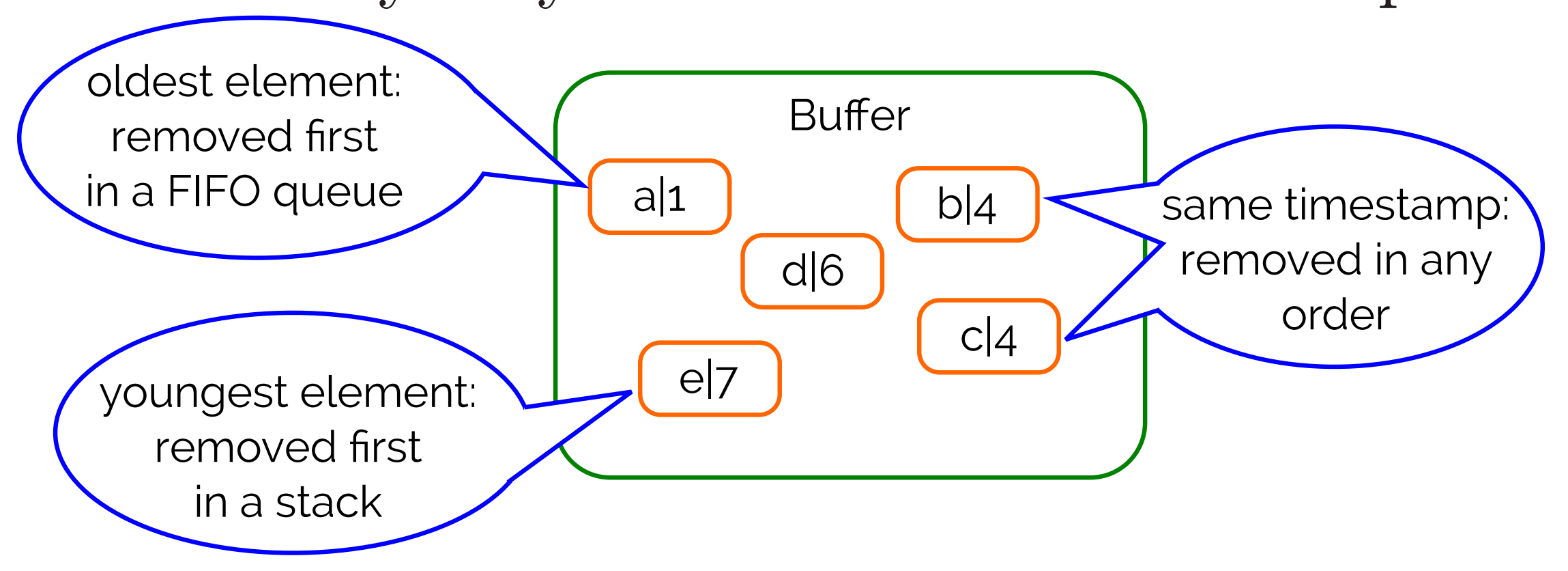
http://scal.cs.uni-salzburg.at/tsdeque

Acknowledgements

This work has been supported by the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund S11404-N23).

Partially Ordered Elements

Elements with timestamps are stored in an unordered buffer. Elements which were inserted concurrently may have the same timestamp.

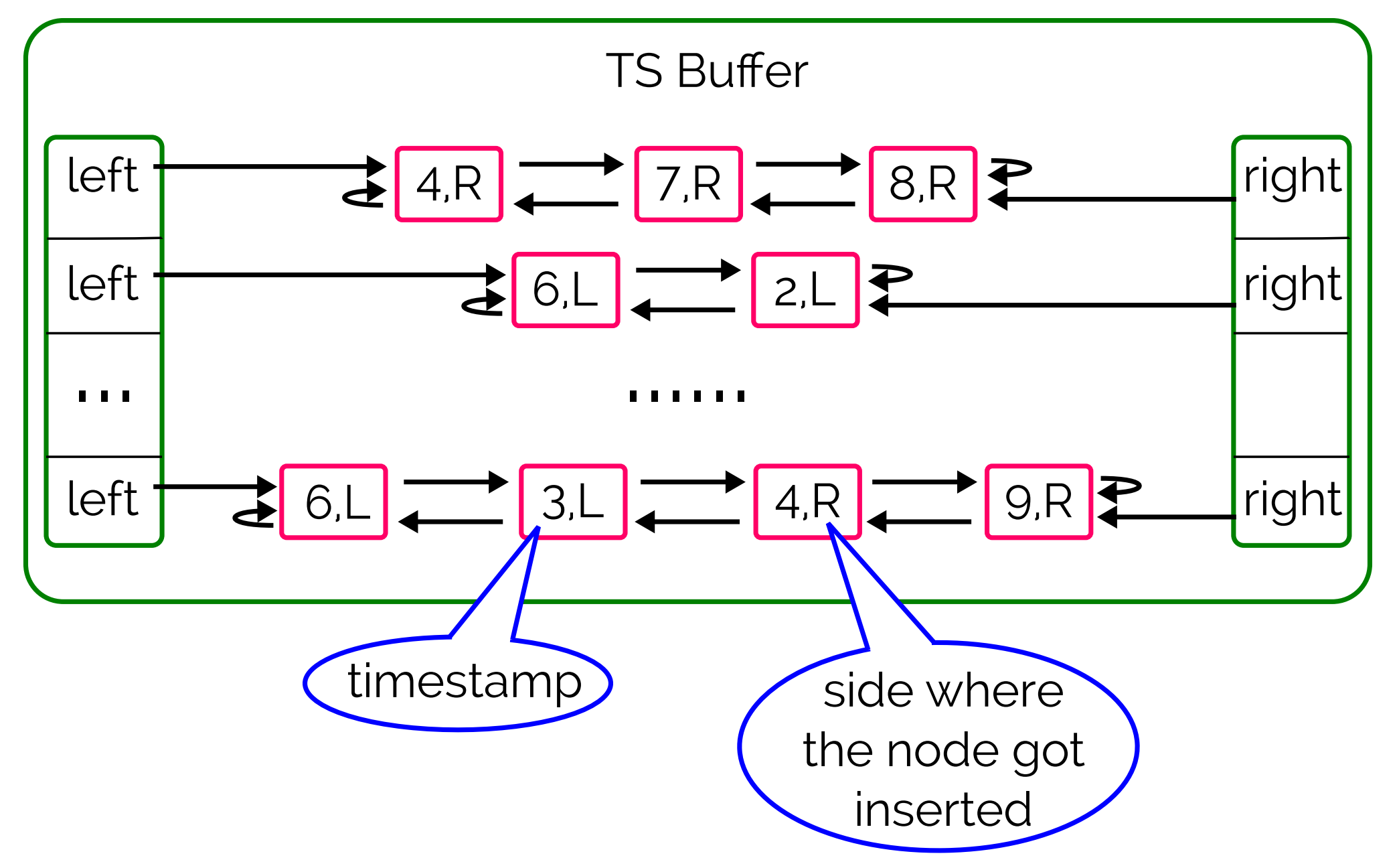

oldest element: removed first in a FIFO queue
Buffer
a|1
b|4
d|6
c|4
e|7
same timestamp: removed in any order
youngest element: removed first in a stack

TS-Buffer Design

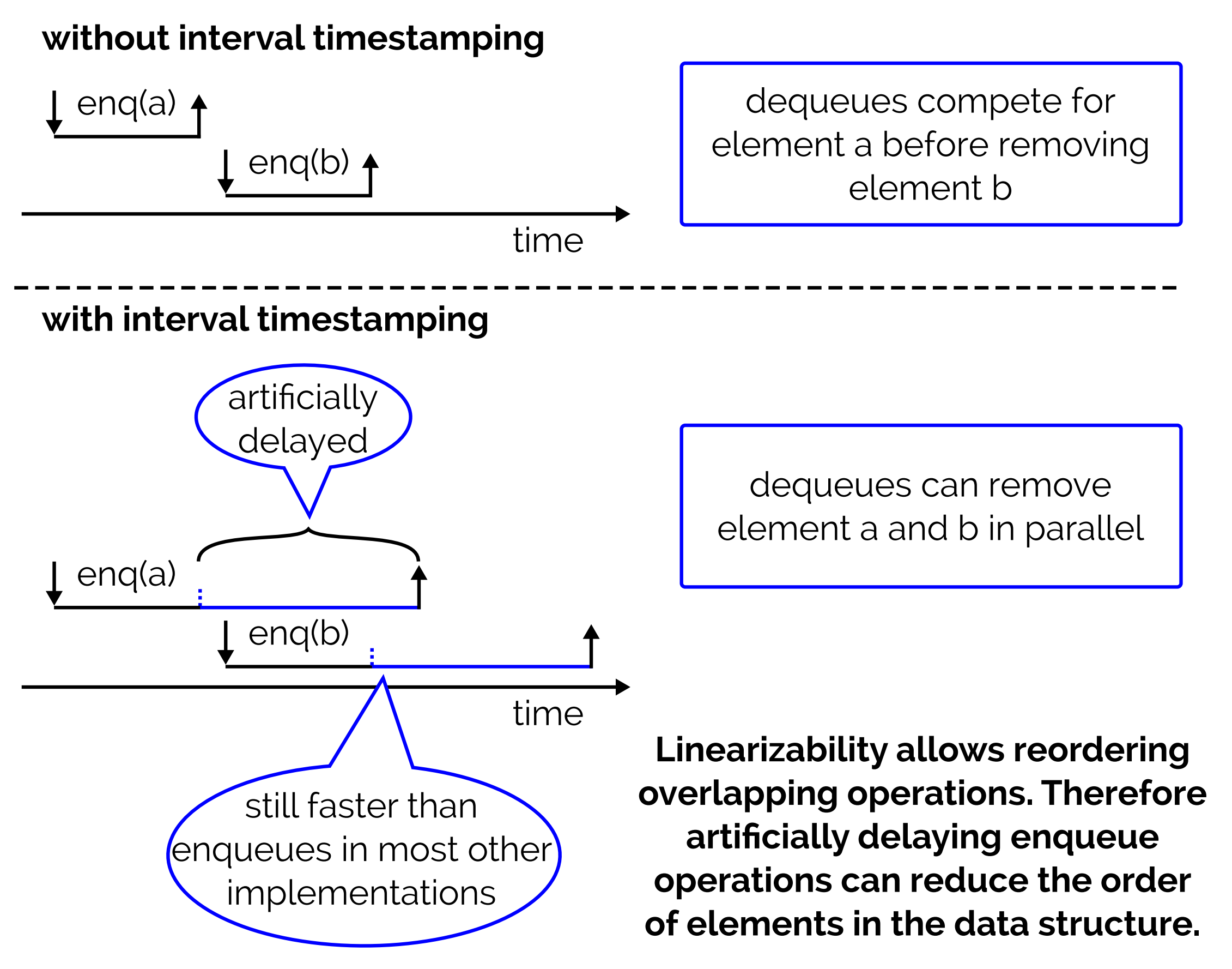
insR: Each thread inserts at the right side of its own linked list.

tryRemoveR: A thread searches through the right ends of all linked lists for the right-most element to remove. Nodes are removed logically first by setting a removed flag. Unlinking occurs in later insert or remove operations. If setting the removed flag fails, then **tryRemoveR** returns an invalid item.

insL/tryRemoveL: Analogous to **insR** and **tryRemoveR**, but the left side of the linked lists is accessed.

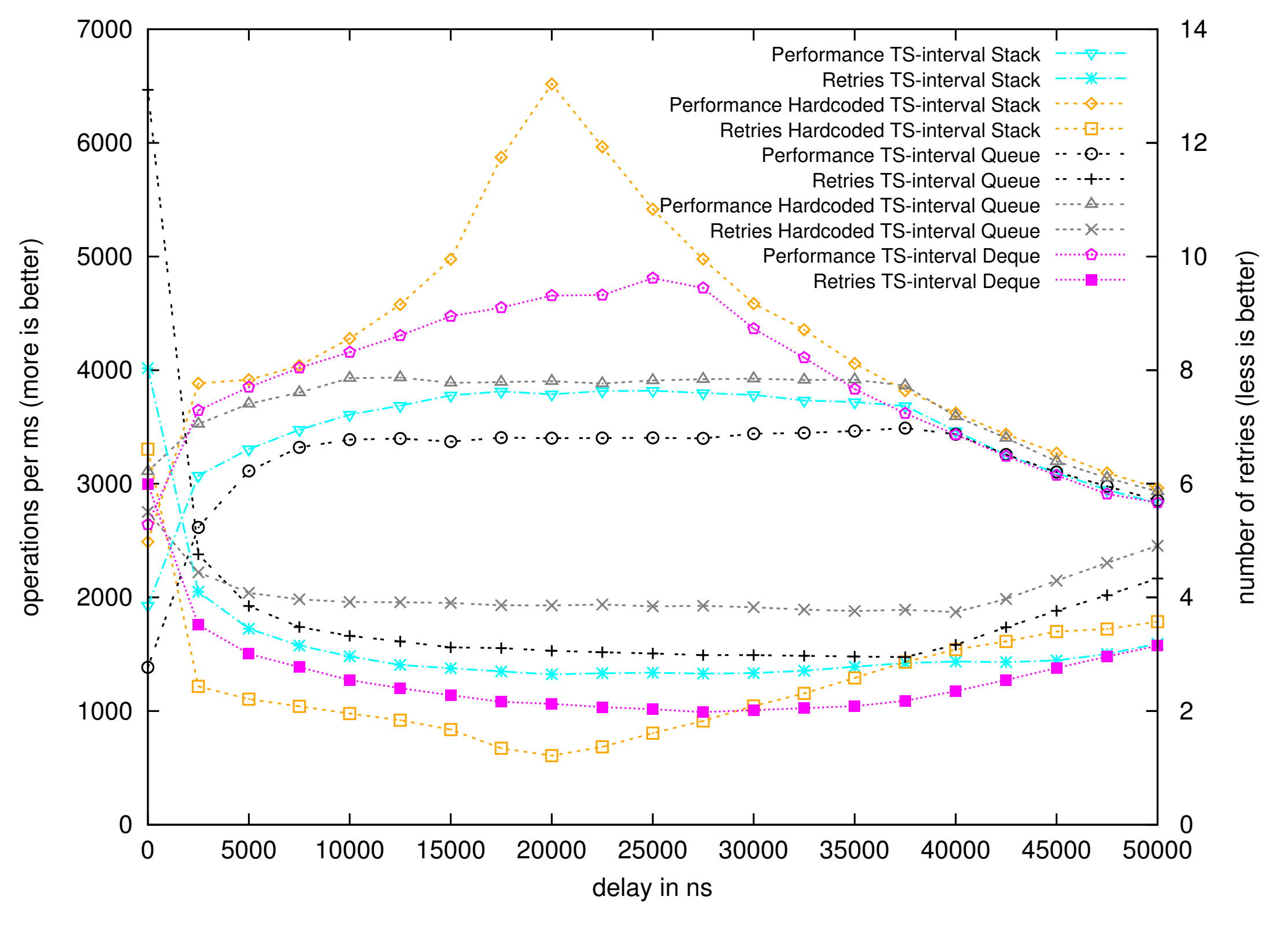


Interval Timestamping


without interval timestamping
enq(a)
enq(b)
time
dequeues compete for element a before removing element b
with interval timestamping
artificially delayed
enq(a)
enq(b)
time
dequeues can remove element a and b in parallel
still faster than enqueues in most other implementations
Linearizability allows reordering overlapping operations. Therefore artificially delaying enqueue operations can reduce the order of elements in the data structure.

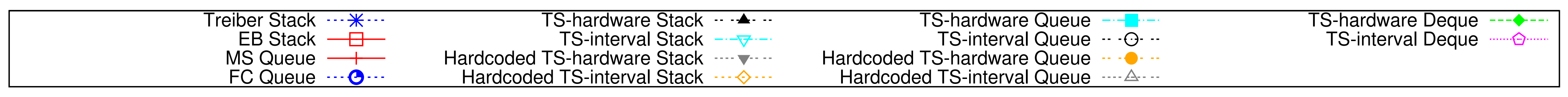
Interval Length

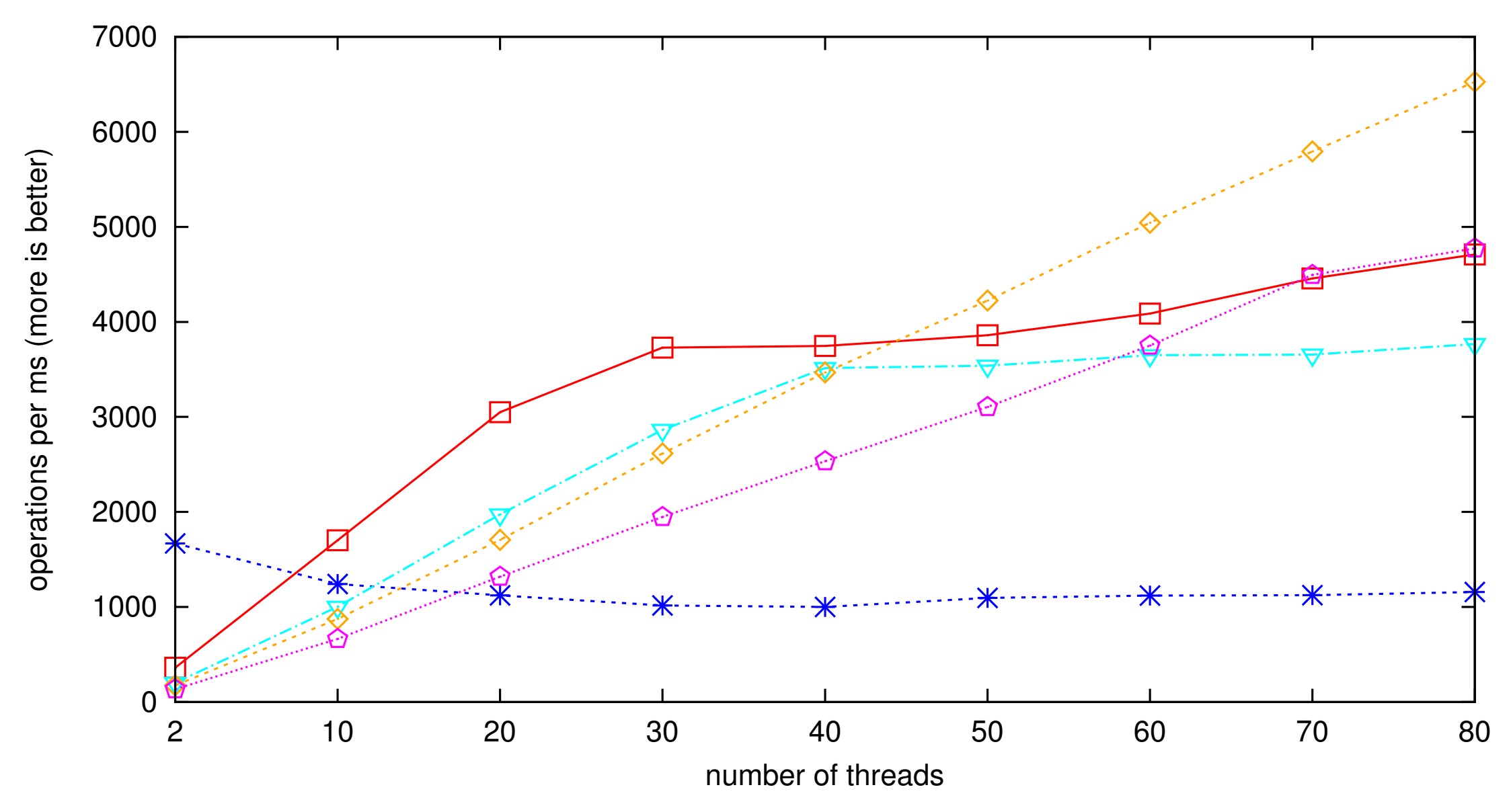
Performance of the TS deque with interval timestamping with an increasing delay in a high-contention producer-consumer benchmark. All measurements are done with 80 threads on a 40-core (2 hyperthreads per core) server machine. The left y-axis shows the average number of operations (both insert and remove operations) per millisecond, the right y-axis shows the average number of CAS retries per remove operation.



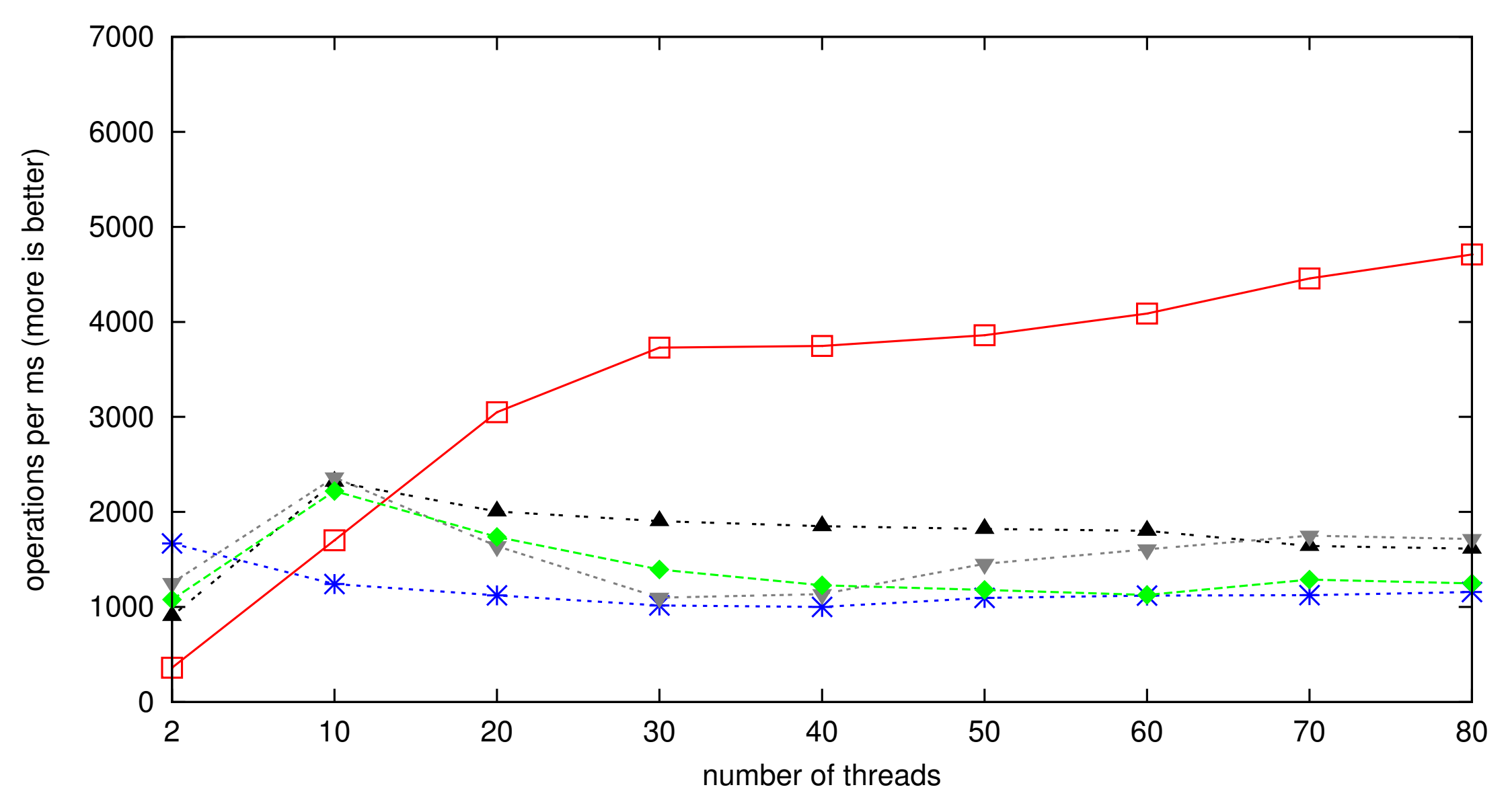
Performance in a Producer/Consumer Benchmark

Performance and scalability of the TS deque in a high-contention producer-consumer benchmark with an increasing number of threads. The baselines are a Treiber stack, an elimination-backoff stack (EB stack), a Michael-Scott queue (MS queue), and a flat-combining queue (FC queue). Stack-specific and queue-specific TS-Buffers (Hardcoded TS Stack and Hardcoded TS Queue) allow additional performance gains.

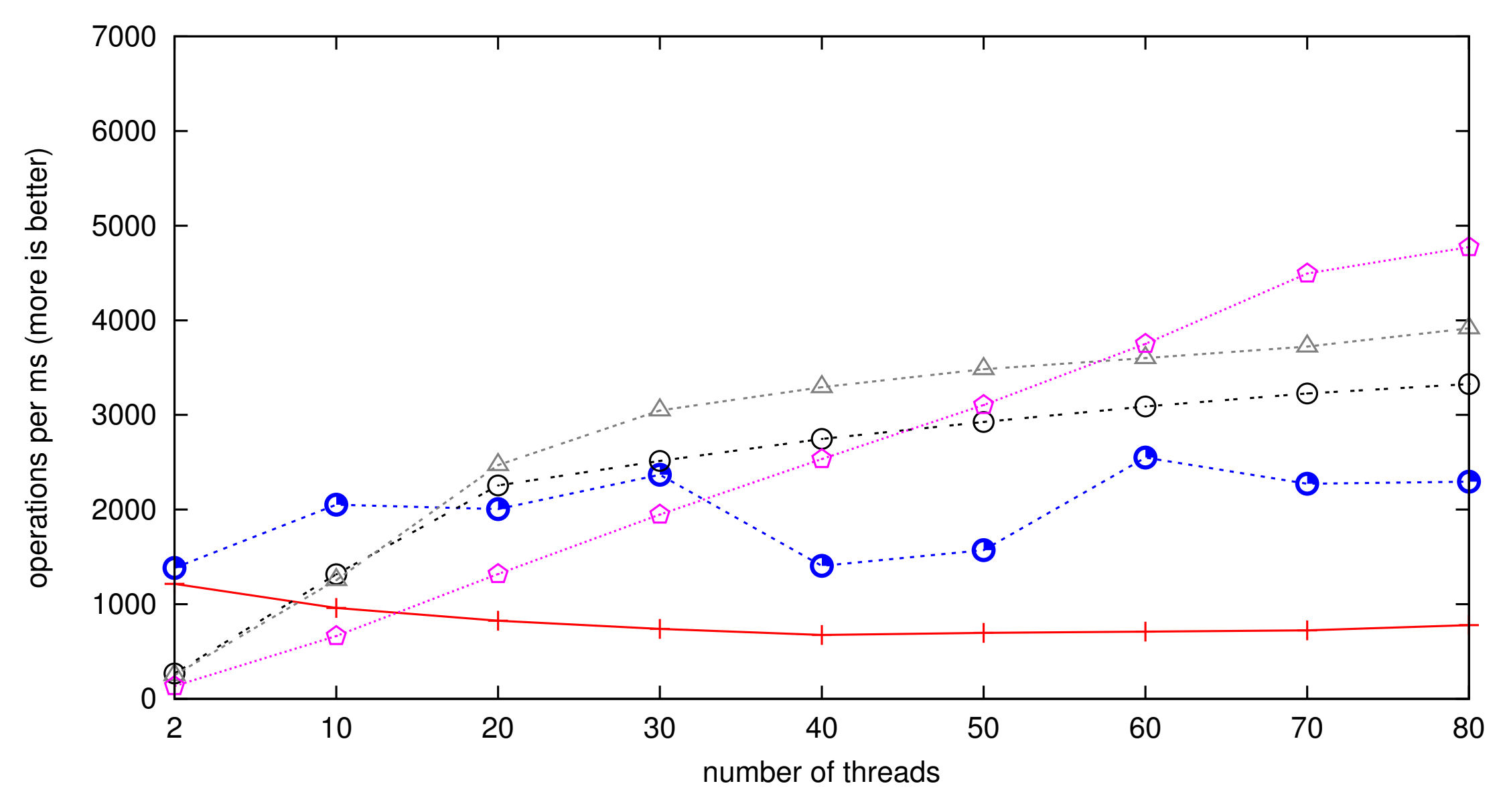




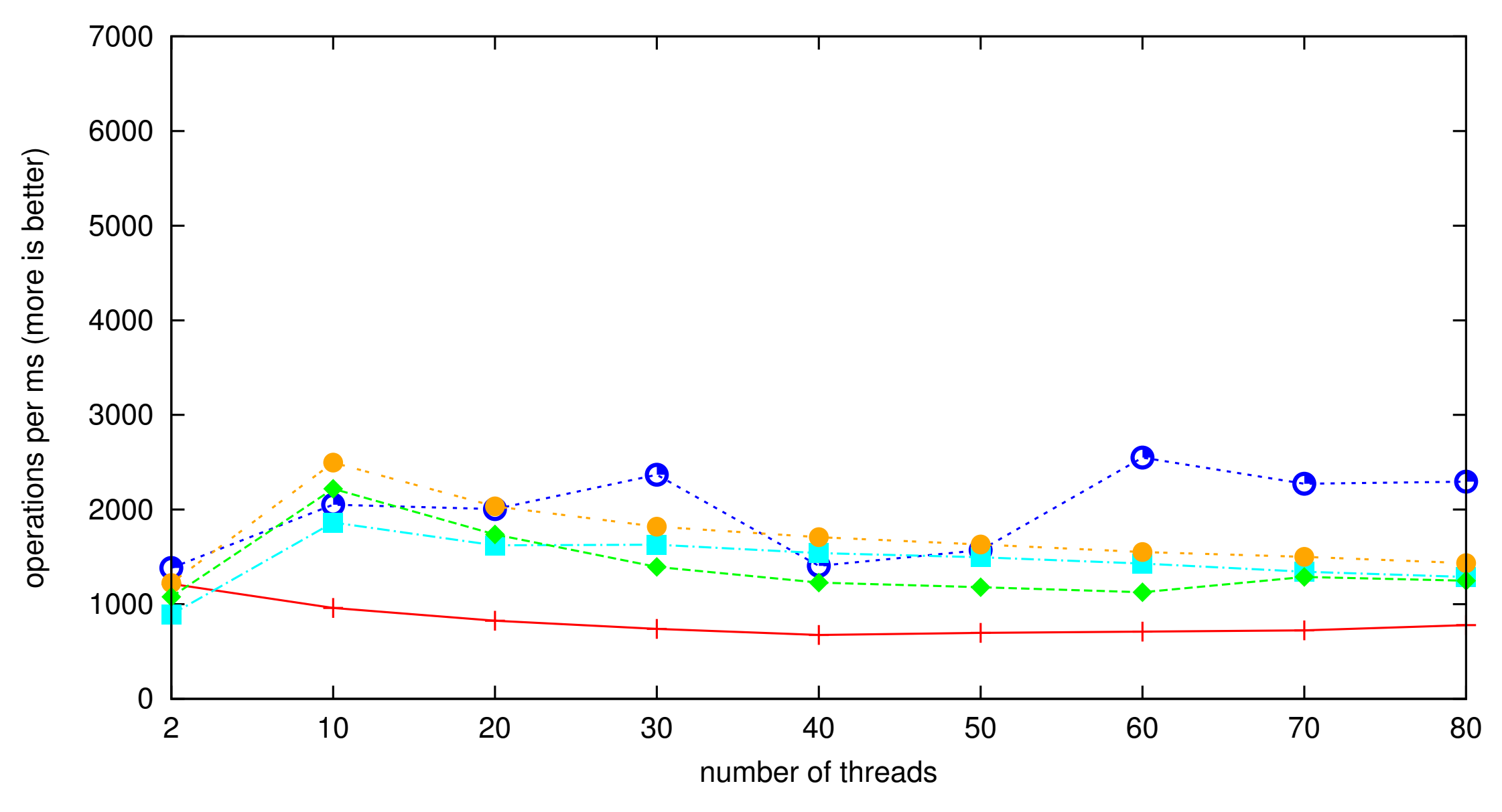
(a) TS stack with interval timestamping



(b) TS stack without interval timestamping



(c) TS queue with interval timestamping



(d) TS queue without interval timestamping