How FIFO is Your Concurrent FIFO Queue?

Authors: Andreas Haas, Christoph M. Kirsch, Michael Lippautz, Hannes Payer

Presenter: Cameron Hobbs
Overview

- Goal of the paper
- Definitions
  - Sequential and Concurrent Histories
  - Linearizations and Zero-time Linearizations
  - Element- and Operation-fairness
- Experimental Results
- Conclusions
Goal of the Paper

- To improve algorithms for concurrent data structures, the requirements for the data structure (and thus, synchronization) are relaxed.
- This improves performance but at the cost of adhering to semantics.
- ...or does it? How do we decide?
Goal of the Paper

- To measure adherence to semantics, the paper introduces the metrics of element- and operation-fairness.
- Something to think about: Do those metrics sound reasonable?
- It uses them to show that even scalable, nonlinearizable algorithms can adhere to semantics just as well as an algorithm that strictly enforces ordering.
Why does it matter?

• As you relax your requirements when programming concurrently, the order of operations performed on a data structure can be any of various orders.

• What orderings are acceptable? or, as this paper hopes to define, What orderings are better?

• If we can define this quantitatively, we can evaluate how well a concurrent algorithm adheres to the semantics of a data structure.
Definitions

- Sequential History
- Concurrent History
- Linearizable History
- Zero-time Linearization
- Element-fairness
  - Element-lateness
  - Element-age
- Operation-fairness
A sequential history for an object is a series of operations performed on that object.

For example:

enq(a), enq(b), deq(a), deq(b)
A concurrent history for an object is a series of operations performed on that object, noting their invocation and response times.

The history:
enq(a)_i, enq(b)_i, enq(a)_r, enq(b)_r, deq(a)_i, deq(a)_r, deq(b)_i, deq(b)_r
A sequential history is a linearization of a concurrent one when:

- If a concurrent operation $m$ returns before the invocation another operation $n$ in the concurrent history, then $m$ appears before $n$ in the sequential history.
- Only operations invoked in the concurrent history occur in the sequential history.

These rules mean that there can be multiple linearizations, but they can only disagree about the order of overlapping concurrent operations.
A Linearizable History

- Linearizable and semantically correct
- But this still can be argued as “wrong” from the perspective of the caller
- `enq(a)` was called before the other operations, but it still took effect after them
Zero-time Linearization

- Ideally, operations on a data structure will complete instantly.
- If we consider that, then the order of the calls mirrors the order that operations take effect, fixing the “problem” from the previous slide.
- But if we consider the operations as taking zero-time, then the otherwise legitimate history from before doesn't satisfy FIFO queue semantics.
Zero-time linearization:
enq(a), enq(b), enq(c), deq(b), deq(c), deq(a)

(Note that this now violates FIFO semantics!)
Zero-time Linearization

• Since in a zero-time linearization we are considering operations to take zero time, we can define the zero-time linearization formally as:

The linearization where an invocation of one operation $m$ preceding another invocation of operation $n$ means $m$ precedes $n$

• Corresponds to the intuitive idea of just looking at invocation times of the history
Element-fairness

Zero-time linearization:
enq(a), enq(b), enq(c), deq(b), deq(c), deq(a)

- **Element-lateness**
  - $a$ is enqueued first, but is dequeued 2 operations later than it should ($b$ and $c$ 'overtake' it)
  - $a$'s element-lateness is 2
Element-fairness

Zero-time linearization:
enq(a), enq(b), enq(c), deq(b), deq(c), deq(a)

• Element-age
  - $b$ and $c$ each overtake $a$ (1 element) when compared to the zero-time linearization
  - $b$ and $c$'s element-age is each 1
Element-fairness

- Together element-lateness and element-age determine element-fairness.
- The lower these values are, the more element-fair the algorithm is for a given concurrent history.
- The formalization is defined by finding the cardinality (size) of the set of elements that overtake (lateness) or are overtaken by (age) the one we are interested in.
Operation-fairness

- Similar to element-fairness, but for operations rather than elements
- Compare invocation time (when the zero-time linearization has the operation take effect) with when the operation actually takes effect (with respect to the concurrent history)
- Stricter algorithms' operations tend to take more time due to failed attempts, reducing operation-fairness
Experiments

- The paper compares three strict implementations with three relaxed implementations.
- Using the metrics described before, they show that the relaxed implementations have not only better performance than the strict ones, but also good semantic performance under their metrics of element- and operation-fairness.
Relaxed implementations are generally better when it comes to speed.

(a) Up to 80 threads on the 40-core machine with a computational load of 2000 pi-calculation iterations between two queue accesses.

(b) Up to 24 threads on the 24-core machine with a computational load of 2000 pi-calculation iterations between two queue accesses.
• Some relaxed implementation actually perform better than strict ones!
Operation-fairness Comparison

- Only the strict implementations were measured here due to tool limitations

(a) Maximum operation-age of all enqueue operations.

(b) Maximum operation-lateness of all enqueue operations.
In general, strict implementations were not very operation-fair.

Hard to compare without relaxed implementation results, though.
What does this mean?

- Practically, you can have efficient concurrent algorithms that adhere to semantics as well as or better than strict ones.
- This is even though a strict algorithm can guarantee an 'acceptable' ordering while the relaxed algorithm does not.
- How does this happen?
  - By not being as strict, the algorithms can execute more quickly.
  - Speed keeps the ordering intact.
  - Being too strict has a negative effect on speed, which in turn negatively affects ordering.
Conclusions

- Paper concludes that relaxed implementations can adhere just as well or better to semantics as strict implementations.
- This is at odds with the perhaps more intuitive view that strict implementations adhere better but are less efficient.
- So using a relaxed implementation will often bring efficiency benefits with no semantic cost.