A Fast, Correct Time-Stamped Stack

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

[1] University of York, UK. mike.dodds@york.ac.uk
[2] University of Salzburg, Austria. firstname.lastname@cs.uni-salzburg.at

Order in concurrent data-structures. Efficient and scalable concurrent data-structures are key to high-performance multicore systems. Algorithms such as stacks and queues are widely used to handle synchronisation and distribute work between cores. In an ideal concurrent data-structure each method call executes in zero time, giving a sequential order on calls and avoiding the need for concurrent reasoning when using the data structure. Naturally such an implementation is impossible – however it can be simulated. Informally the correctness condition *linearizability* [4] requires that each call for the algorithm *appears to* take effect atomically between its invocation and response.

Many linearizable data-structures not only simulate a sequential execution, but also generate a total sequence internally, for example by arranging elements into a linked list. Linearizability is comparatively easy to prove for these structures, as it only requires that the internal sequence respects the data structure semantics. However, our work suggests that enforcing a total order internally misses opportunities for performance and scalability, especially in high-contention scenarios.

The time-stamped stack. We have developed a data-structure called the TS (time-stamped) stack [2] which avoids building a total order between elements. Elements that are pushed in parallel can be left internally unordered. Nonetheless, a total order *can* be reconstructed, meaning the TS stack is linearizable with respect to sequential stack semantics. Proving linearizability is challenging, however, because this linearization order cannot be derived directly from the implementation.

A TS stack consists of a collection of buffers, each allowing insertion, removal, and search for elements. Each buffer is independent, allowing threads to insert elements without expensive synchronization [1]. Elements are timestamped to record the insertion order, but elements inserted concurrently may get the same timestamp, i.e may be unordered. This is sound because such operations could be ordered either way in the simulated sequential order. We use the x86 RDTSCP [5, 6] instruction to generate timestamps.

Each pop operations searches through the buffers to find the elements with the latest timestamp, and then removes one of them. If there exist multiple elements with the same timestamp the pop operation tries to remove any of these elements. If a pop encounters an element timestamped after the search started, it tries to remove that element instead: in this case the corresponding push operation is executed concurrently, and therefore the operations can eliminate one another [3].

Far from presenting a problem, elements that share a common timestamp can *increase* performance, because such elements can be removed parallel. We can thus optimise the TS stack by deliberately increasing the amount of timestamp parallelism. To do this we associate elements with two timestamps, rather than one, and say that two elements are unordered if their timestamp intervals overlap. The amount of parallelism can be increased by adding a delay between the generation of the two timestamps.

Establishing linearizability. The linearizability of the TS stack is based on two properties: (a) if the timestamp of element A is later than of an element B, and the pop operation which removes B starts after A was inserted into the buffer, then A is removed before B; and (b) elements are timestamped after they are inserted into a buffer.

The first property guarantees that elements are not removed out-of-order, i.e. that any removed element was youngest in the buffers at some point during the search. The second property induces a kind of monotonicity on elements: if A is seen by a pop with its timestamp assigned and an element B is not seen, then the timestamp of B is later than that of A.

To establish linearizability, we must show that for any execution we can construct a linearization order which satisfies the stack specification. We do this in three steps.

1. Construct an order between push and pop operations. Element visibility gives an initial approximation, i.e. a push is ordered before a pop when it inserts its element before the pop starts searching. We refine this approximation to remove inconsistencies and give a total order.

2. Order pairs of pop operations according to the timestamps of removed elements. Property (b), the monotonicity of element timestamps, guarantees that the newly constructed order is transitive. Some pop operations may remain unordered, but we can show that these pop operations can be ordered arbitrarily.

3. Order push operations as late as possible while preserving stack semantics. With property (a) we can show that this is always possible.

We have completed a hand proof of linearizability, and a mechanisation in Isabelle is in progress.

Algorithm performance. In our high-contention producer-consumer benchmark the TS stack is about twice as fast as the elimination backoff stack [3], the fastest alternative we are aware of. An implementation of the TS stack is available at `http://scal.cs.uni-salzburg.at/`, and additional benchmark results can be found in the technical report [2].

References

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

[2] M. Dodds, A. Haas, and C. M. Kirsch. Fast concurrent data-structures through explicit timestamping. Technical Report 2014–03, University of Salzburg, 2014.

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

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

[5] Intel. Intel 64 and ia-32 architectures software developer's manual, volume 3b: System programming guide, part 2, 2013.

[6] W. Ruan, Y. Liu, and M. Spear. Boosting timestamp-based transactional memory by exploiting hardware cycle counters. *TACO*, 10, 2013.