Testing for Linearizability

Gavin Lowe

Linearizability

Linearizability is a well-established and accepted correctness condition for concurrent datatypes.

Informally, a concurrent datatype C is linearizable with respect to a sequential (specification) datatype S if every history c of C (i.e. a sequence of operation calls and returns) is linearizable to a history s of S: c can be reordered to produce s, but this reordering respects the order of non-overlapping operation calls.

Testing for linearizabilty

But how can we be sure that C is linearizable with respect to S?

Standard answer: formal verification. But this is hard work and time-consuming, and often requires various (sometimes dubious) abstractions.

Alternative answer: testing.

- This is quick (creating a testing framework takes a few minutes; bugs are normally found within 20 seconds);
- It's easy (students have no difficulty with it);
- It gives quite strong guarantees of correctness (but not as strong as for verification).

Testing framework

Basic idea:

- Create an immutable sequential specification datatype S, often by adapting a standard datatype in an API; for each operation `op : A` on the concurrent datatype, we need a corresponding function `seqOp : S => (A, S)`;
- Run several workers on the concurrent datatype C, recording the history of operation calls and returns;
- Apply a suitable algorithm to decide whether the history is a linearization of a history of S.
- Repeat.

Example testing program

```
object QueueTest{
  type C = LockFreeQueue[String]; type S = scala.collection.immutable.Queue[String]
  def seqEnqueue(x: String)(q: S) : (Unit, S) = ((), q.enqueue(x))
  def seqDequeue(q: S) : (String, S) = if(q.isEmpty) (null,q) else q.dequeue

  def worker(me: Int, tester : LinearizabilityTester [S, C]) = for(i <- 0 until 200)
    if(Random.nextFloat <= 0.3){
      val x = Random.nextInt(20).toString
      tester.log(me, _.enqueue(x), "enqueue("+x+")", seqEnqueue(x)) }
    else tester.log(me, _.dequeue, "dequeue", seqDequeue)

  def main(args: Array[String]) = for(i <- 0 until 1000){
    val concQueue = new LockFreeQueue[String] // The shared concurrent queue
    val seqQueue = Queue[String]() // The sequential specification queue
    val tester = new LinearizabilityTester (seqQueue, concQueue, 4, worker, 800)
    assert(tester() > 0) }
}
```

The Linear Tester algorithm

The Linear Tester algorithm takes a history of the concurrent datatype C, and tests whether it is linearizable with respect to S.

It traverses the history linearly, maintaining a set of *configurations* consistent with the history to date. Here, a configuration is a tuple $(s, calls, rets)$, where

- s is a state of the sequential specification object;
- *calls* records the set of operation calls that have been made but not yet been linearized;
- *rets* records the set of operation calls that have been linearized but not returned.

The Linear Tester Algorithm

The algorithm traverses the recorded history.

- When the algorithm encounters a call event, it is added to *calls* of each configuration in the current set;
- When the algorithm encounters a return event of operation *op*, for each configuration in the current set:
  - If *op* has already been linearized, it checks that it gave the recorded result; if not, the configuration is removed.
  - Otherwise, the algorithm chooses some other pending calls to linearize in some order (in all possible ways), updating the sequential specification *s*. It then tests whether *op* can be linearized, at this point, updating the configuration if so. Thus this operation is linearized just before it returns (a form of partial-order reduction).

  If no configuration remains, the history is not linearizable.

The Linear Tester Algorithm: optimizations

For efficiency, the Linear Tester memoizes:

- Results of equality tests between sequential specification objects, using a union-find datatype;
- Results of inequality tests;
- Results of previously performed operations on sequential objects.

The Wing & Gong Algorithm

Wing and Gong[a] presented an algorithm for linearizability testing.

The algorithm assumes a mutable sequential object with undo-ing of previous operations.

The algorithm performs an exhaustive search: it (potentially) considers *every* sequential history consistent with the concurrent history (regarding ordering of operations), and tests whether it is a valid sequential execution.

[a] J. M. Wing and C. Gong. Testing and verifying concurrent objects. *Journal of Parallel and Distributed Computing*, 17:164–182, 1993.

The Wing & Gong Algorithm

The algorithm maintains a sequential object corresponding to the sub-history linearized so far, and a linked list representing the sub-history not yet linearized.

At each step, it selects an suitable operation from the history to linearize next; if it gave the same result as on the sequential history, it is linearized, and the events removed from the linked list.

If no operation can be linearized next, the algorithm back-tracks, undoing each operation on the sequential object, and reinserting it into the linked list.

Depth-First Search Algorithm

The Wing & Gong Algorithm sometimes performs very poorly. The reason for this is that it fails to identify when it returns to a previously seen configuration; it therefore repeats previous work (sometimes to an exponential degree).

The Depth-First Search Algorithm overcomes this by storing previously-seen configurations in a hash table.

In order for this to work, configurations must not share sequential objects. Instead, we use the same approach as for the Linear Tester: we require the sequential object to be immutable and for operations on it to return a resulting sequential object.

In addition, the Depth-First Search tester uses the same memoization optimisations as the Linear Tester.

Competition Parallel

The Competition Parallel Tester runs the Wing & Gong and Depth-First Search Algorithms in parallel. When one terminates, the other is interrupted.

Experimental set-up

Each *run* constitutes a single test on the concurrent datatype, with some number of workers performing some number of operations.

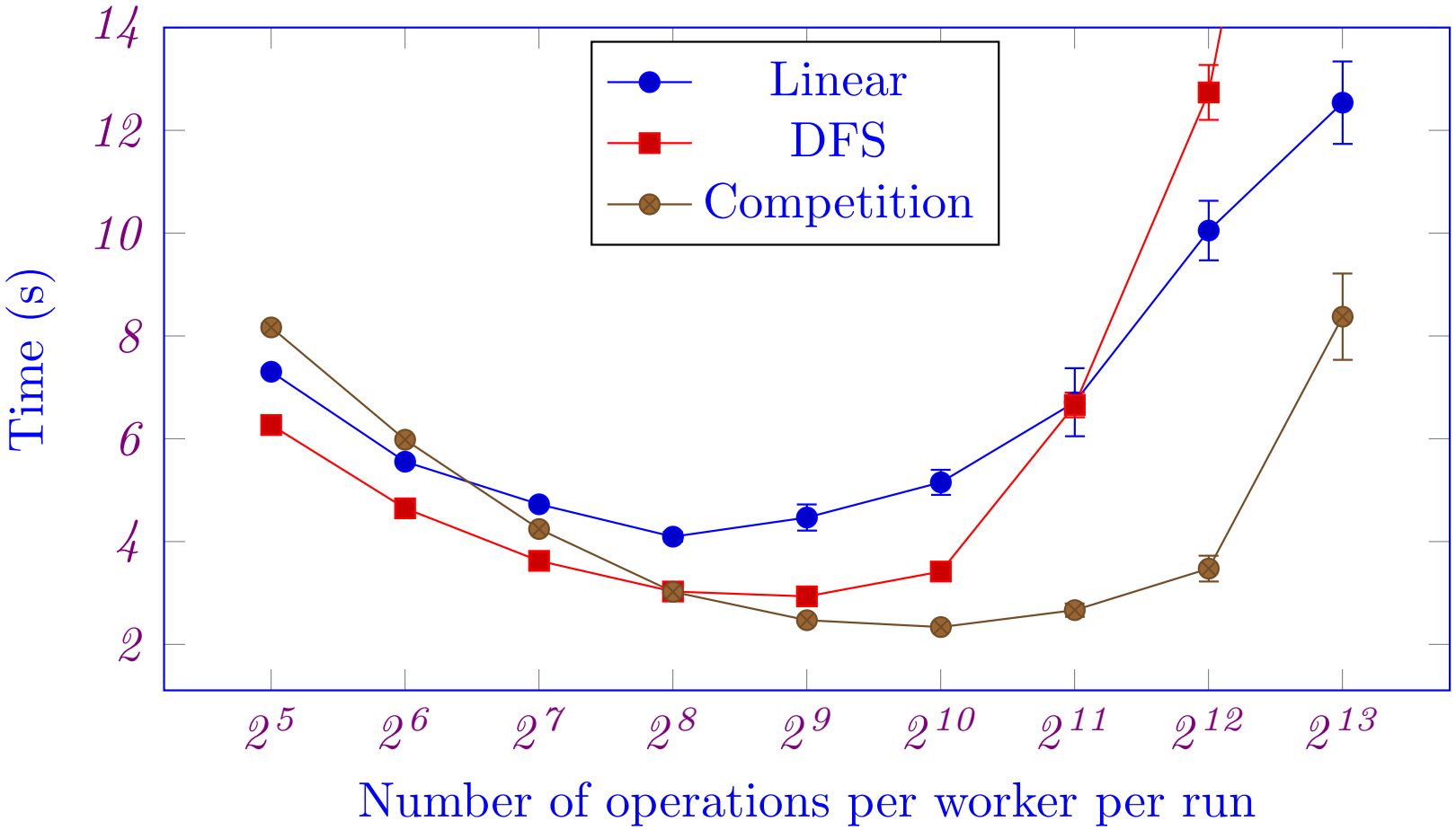
Each *observation* constituted a number of runs, chosen to be close to a typical use case.

Multiple observations were made and timed; 95% confidence intervals were calculated.

Unfortunately the Wing & Gong Algorithm proved impossible to profile in a meaningful way. In most observations, it was faster than the other algorithms. However, sometimes it was much slower, sometimes failing to terminate after several hours! The reason is that it normally gets lucky, and finds a correct linearization early in its search space. But when it doesn't get lucky (or the history is not linearizable) it explores a huge amount of the search space, with an exponential blow-up.

Experiment on a queue

Four worker threads, each performing 2^5–2^{13} operations per run; each observation constituted runs with a total of 2^{20} operations.



Experimental results

Other experiments show similar results.

The Competition Parallel algorithm seems to work well.

- On most runs, the Wing & Gong Algorithm terminates quickly.
- When the Wing & Gong Algorithm encounters a bad case, the DFS Algorithm still finishes within a reasonable time.

Thus the two algorithms complement one another well.

Finding bugs

The approach can find some quite subtle bugs. Here's the relevant part of the output when the debugger is run on a map. All five operations seem to be necessary.

```
369  1 invokes update(0, 0)
370  0 invokes delete 0
371  0 returns ()
372  0 invokes update(0, 2)
373  0 returns ()
374  0 invokes delete 0
375  0 returns ()
376  1 returns ()
378  1 invokes getOrElse(0, X)
379  -- Previous event not linearized
380  1 returns 2
381  -- Previous event not linearized
382  -- Allowed return values: 0, X
```

Finding bugs

The approach is fast, particularly with well-crafted tests.

The bug on the previous slide is found in average time 12.4 ± 2.1s.

A different bug in a hash map, related to resizing, is found in average time 15.7 ± 2.5s.

A bug in a skiplist, related to hash collisions, is found in average time 100 ± 22ms.

A bug in a sharded map with lock-free reads is found in average time 499 ± 71ms.

Future directions

- Can the Linear Algorithm be made more depth-first, to allow it to "get lucky", at least some of the time?
- Use task parallelism in the testing framework, running several tests concurrently.
- Would more erratic scheduling allow bugs to be found faster?
- On datatypes like sets and maps, it is enough to test for linearizability of each key separately.
- Support nondeterministic specifications.