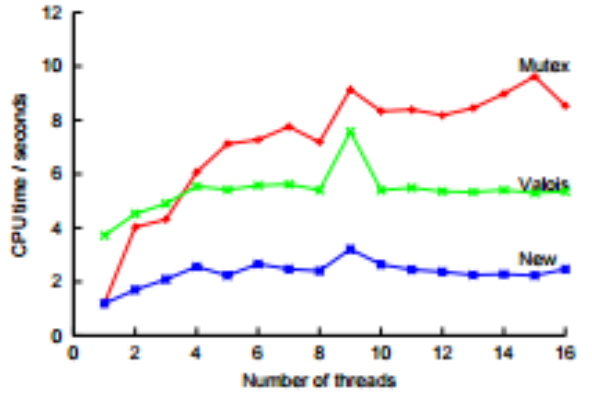
Five ways not to fool yourself

Tim Harris

23-Jun-18

Five ways not to fool yourself



(*b*) Non-reference-counted algorithms with keys 0 . . . 255.

The test harness was parameterized on the algorithm to use, the number of concurrent threads to operate and the range of keys that might be inserted or deleted. In each case every thread performed 1 000000 operations. Figure 6 shows the CPU accounted to the process as a whole for each of the algorithms tested on a variety of workloads.

"A pragmatic implementation of non-blocking linked lists", Tim Harris, DISC 2001

Five ways not to fool yourself

1. Measure as you go

Starting and stopping work

- How much work to do?

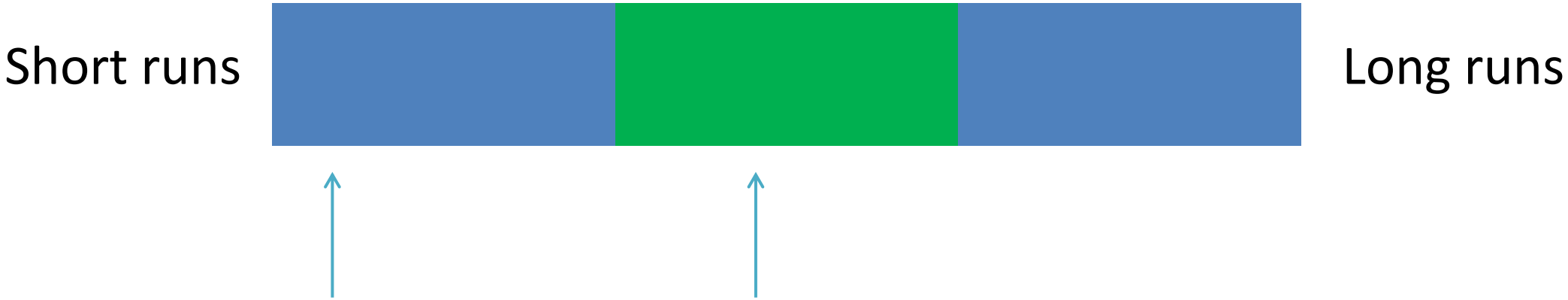
Short runs

Long runs

Too little: results dominated by start-up effects. Normalized metrics vary as you vary the duration.

Starting and stopping work

- How much work to do?



Starting and stopping work

- How much work to do?

Short runs

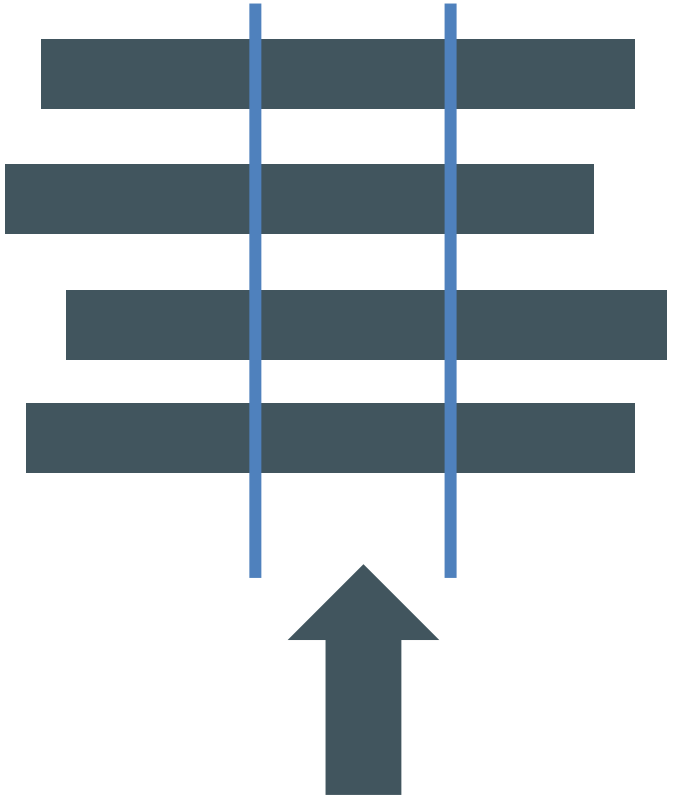
Long runs

Too little: results dominated by start-up effects. Normalized metrics vary as you vary the duration.

OK: results not sensitive to the exact choice of settings. Confirm this: double / halve duration with no change.

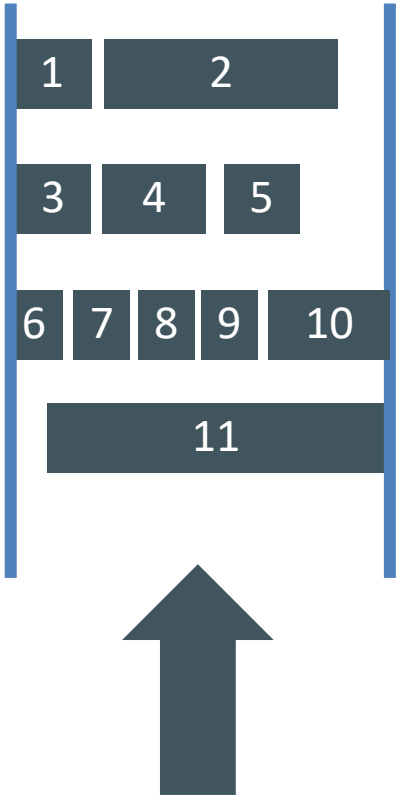
Unnecessarily long – deters experimentation, and risks errors from mixing up results from different runs

Constant load



Fixed set of threads active throughout the measurement interval. Measure the work they do.

Constant work



Fixed amount of work (e.g., loop iterations). Measure the time taken to perform it. Vary the number of threads.

Plot what you measure, not what you configure

“Bind threads 1 per socket”	Have each thread report where it is running
“Run for 10s”	Record time at start & end
“Use 50% reads”	Measured #reads/#ops
“Distribute memory across the machine”	Actual locations and page sizes used

Five ways not to fool yourself

1. Measure as you go
2. Include lightweight sanity checks

Be skeptical about the results

Be skeptical about the results

- Is the harness running what you intend it to run?
  - Incorrect algorithms are often faster
  - Good practice: do not print any output until you have confidence in the result

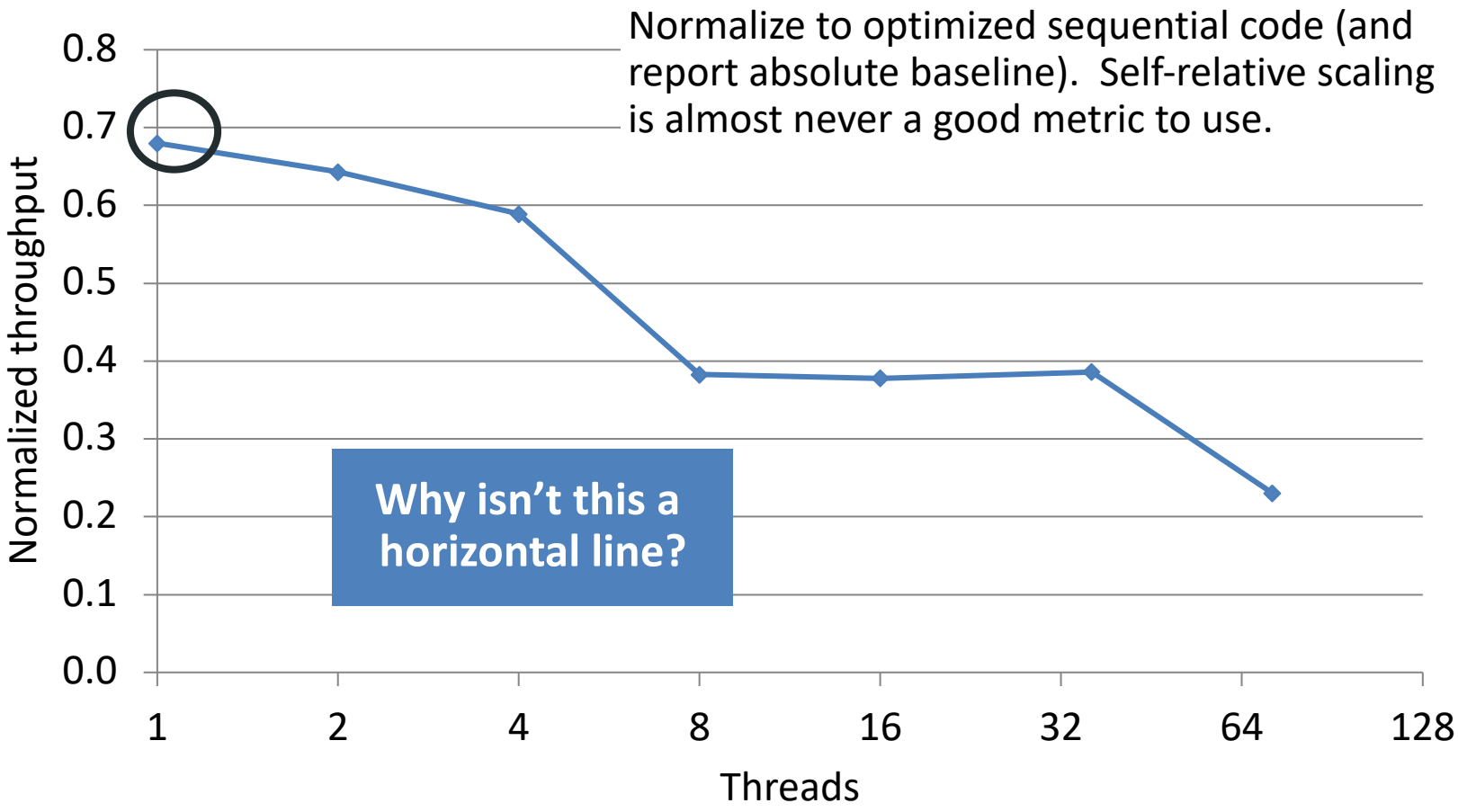
Be skeptical about the results

- Does the data structure pass simple checks?
  - Start with N items, insert P, delete M, check that we have N+P-M at the end
  - Suppose we are building a balanced binary tree – is it actually balanced at the end?
  - Suppose we have a vector of N items and swap pairs of items – do we have N distinct items at the end?

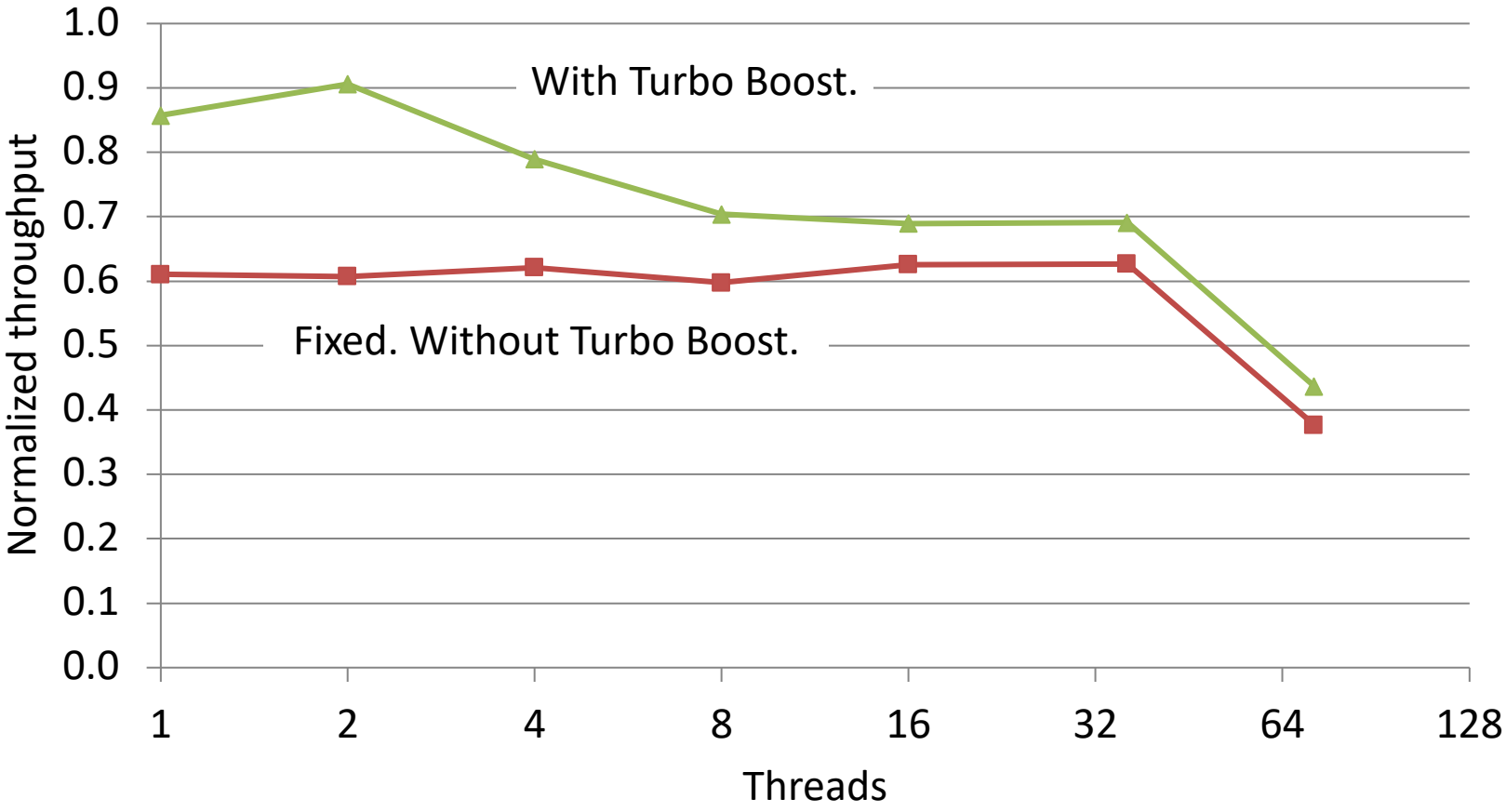
Five ways not to fool yourself

1. Measure as you go
2. Include lightweight sanity checks
3. Understand the simple cases first

Skip-list, 100 % read only, 2*Haswell



Skip-list, 100 % read only, 2*Haswell



Five ways not to fool yourself

1. Measure as you go
2. Include lightweight sanity checks
3. Understand the simple cases first
4. Look beyond timing

Look beyond timing

- Try to link:
  - Performance measurements from an experiment
  - Measurements of resource use during the experiment
  - Differences between the algorithms being executed

Resource utilization

- Examine the use of significant resources in the machine
  - Bandwidth to and from memory
  - Bandwidth use on the interconnect
  - Instruction execution rate
- Clock frequency and power settings
- Look for evidence of bad behavior
  - High page fault rate (i.e., going to disk)
  - High TLB miss rate

Thread placement

- Choice between OS-control threading versus pinning
- Real workloads run with OS-controlled threading
  - …but OS-controlled threading can be sensitive to blocking / wake-up behavior, thread creation order, prior machine state, ….
- Deliberately explore different pinned placements, and quantify impact
  - Are differences between algorithms consistent across these runs?
- In experiments compare:
  - OS (report version)
  - Different pinning choices (how many sockets used, how many cores per socket, what order are h/w threads used)?

Memory placement

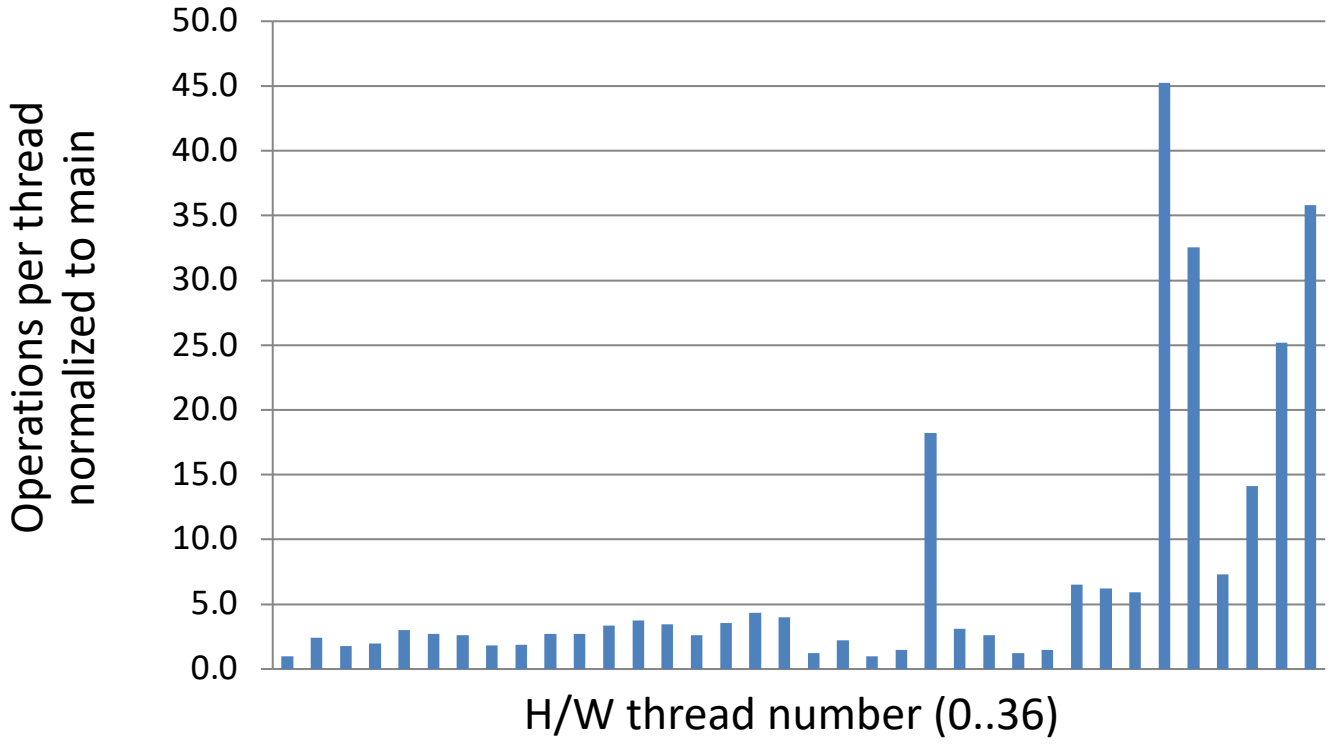
- How are we distributing memory across sockets?
- How is the load distributed over memory channels?
- How is memory being allocated / deallocated?

Unfairness

- Look across all of the threads: did they complete the same amount of work?
- Trade-offs between unfairness and aggregate throughput
  - Unfairness may correlate with better LLC behavior
  - Threads running nearby synchronize more quickly, and get to complete more work
- Whether we care about unfairness *in itself* depends on the workload
  - Threads serving different clients: may want even response time
  - Threads completing a batch of work: just care about overall completion time

Unfairness: simple test-and-test-and-set lock

- 2-socket Haswell, threads pinned sequentially to cores in both sockets



← 45x, not 45%!

Five ways not to fool yourself

1. Measure as you go
2. Include lightweight sanity checks
3. Understand the simple cases first
4. Look beyond timing
5. Move toward production settings

Concluding comments

- We optimize for what we measure, or measure what we optimized
  - Why pick specific workloads (read/write mix, key space, … ?)
  - Does the choice reflect an important workload?
  - Are results sensitive to the choice?
- Be careful about averages
  - As with fairness over threads, an average over time hides details
  - Even if you do not plot all the results, examine trends over time, variability, etc.
- Be careful about trade-offs
  - Is a new system strictly better, or exploring a new point in a trade-off?

Further reading

- Books
  - Huff & Geis – “How to Lie with Statistics”
  - Jain – “The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling”
  - Tufte – “The Visual Display of Quantitative Information”
- Papers and articles
  - Bailey – “Twelve Ways to Fool the Masses”
  - Fleming & Wallace – “How not to lie with statistics: the correct way to summarize benchmark results”
  - Heiser – “Systems Benchmarking Crimes”
  - Hoefler & Belli – “Scientific Benchmarking of Parallel Computing Systems”