

ORACLE®

Benchmarking Concurrent Data Structures

(or: “Do these numbers mean what you think they mean?”)

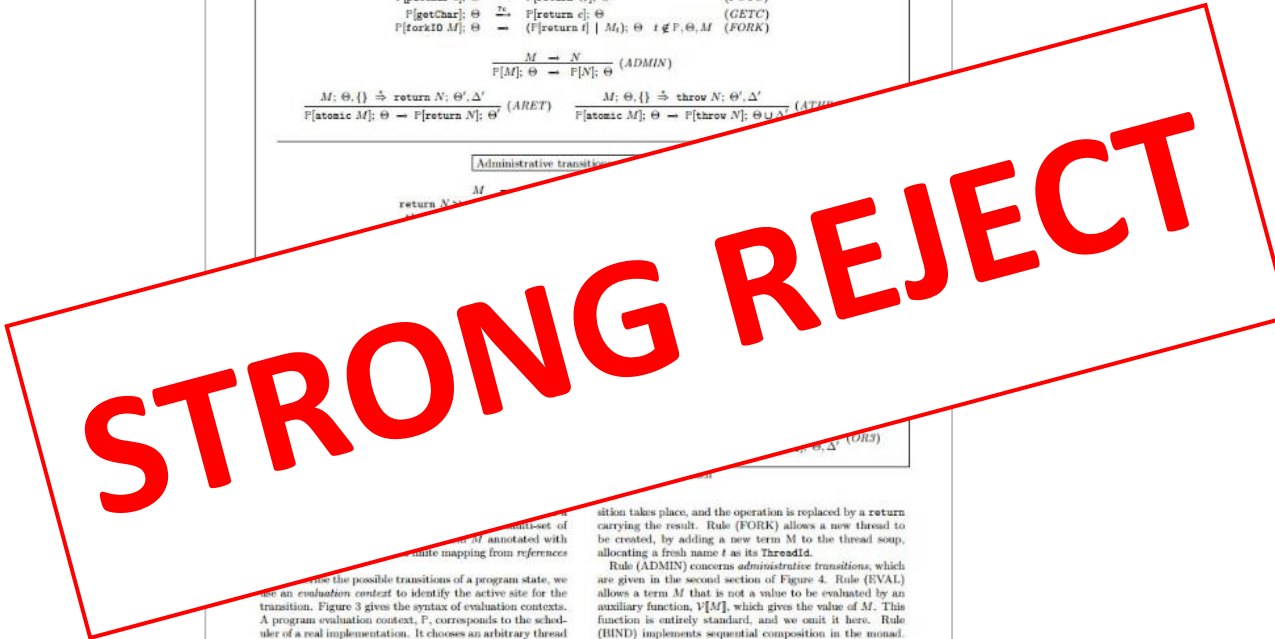
Tim Harris

11 October 2017

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

Why do we care about perf evaluation?

Why do we care about perf evaluation?



I/O transitions $P; \Theta \xrightarrow{\alpha} Q; \Theta'$

$$\frac{}{P[\text{putChar } c]; \Theta \xrightarrow{\alpha} P[\text{return } ()]; \Theta} \text{ (PUTC)}$$
$$\frac{}{P[\text{getChar}]; \Theta \xrightarrow{\alpha} P[\text{return } c]; \Theta} \text{ (GETC)}$$
$$\frac{}{P[\text{forkIO } M]; \Theta \xrightarrow{\alpha} (P[\text{return } t] \mid M_t); \Theta \quad t \notin P, \Theta, M} \text{ (FORK)}$$
$$\frac{M \xrightarrow{\alpha} N}{P[M]; \Theta \xrightarrow{\alpha} P[N]; \Theta} \text{ (ADMIN)}$$
$$\frac{M; \Theta, \{\} \xrightarrow{\alpha} \text{return } N; \Theta', \Delta'}{P[\text{atomic } M]; \Theta \xrightarrow{\alpha} P[\text{return } N]; \Theta'} \text{ (ARET)}$$
$$\frac{M; \Theta, \{\} \xrightarrow{\alpha} \text{throw } N; \Theta', \Delta'}{P[\text{atomic } M]; \Theta \xrightarrow{\alpha} P[\text{throw } N]; \Theta \cup \Delta'} \text{ (ATHROW)}$$

Administrative transitions

$$\frac{M \xrightarrow{\alpha} N}{\text{return } N; \Theta \xrightarrow{\alpha} \text{return } N; \Theta'} \text{ (OR3)}$$

5.2 Operational semantics

Now we are ready to discuss the transition rules of Figure 4. First we treat the *I/O transitions*, in the top part of the figure, which can have arbitrary input/output effects. The first two rules deal with input and output. If the active term is a `putChar` or `getChar` the appropriate labelled transition takes place, and the operation is replaced by a `return` carrying the result. Rule (FORK) allows a new thread to be created, by adding a new term M to the thread soup, allocating a fresh name t as its `ThreadId`.

Rule (ADMIN) concerns *administrative transitions*, which are given in the second section of Figure 4. Rule (EVAL) allows a term M that is not a value to be evaluated by an auxiliary function, $V[M]$, which gives the value of M . This function is entirely standard, and we omit it here. Rule (BIND) implements sequential composition in the monad. The rules (THROW), (CATCH1) and (CATCH2) implement exceptions in the standard way. All of these rules are, as we shall see, used in both the IO monad and the STM monad, which is why we keep them in a separate group.

Everything so far is quite standard. The new part starts with rules (ARET) and (ATHROW). The former describes how an atomic transaction takes place: the term M makes zero or more transitions of the STM relation, \Rightarrow , which takes the following form:

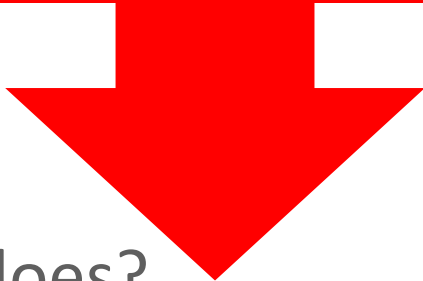
$$M; \Theta, \Delta \Rightarrow N; \Theta', \Delta'$$

Why do we care about perf evaluation?

- How does a new algorithm compare with alternatives?
 - Is it better?
 - When is it better?
 - Faster without contention?
 - Higher throughput under contention?
 - Lower tail latencies?
 - Improved characteristics e.g. plateau rather than fall off as load rises?
 - Are there surprises?

Why do we care about perf evaluation?

Most of the talk is about this point. What kinds of experiments can we run to show ourselves and demonstrate to others that a perf improvement is due to some aspects of our work?



- Why does it perform as it does?
 - Can we explain the trends seen?
 - Can we relate them to design choices in the algorithm?
 - What can we learn about designing other algorithms?
 - What hypotheses can we form to test (disprove!) our explanation?

Overview

1

Sound experimental practices

2

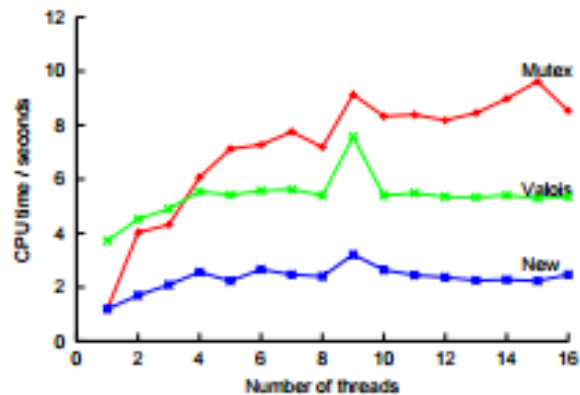
Understanding simple cases

3

Exploring performance trends

Starting and stopping work

Starting and stopping work



(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 1 000 000 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

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?

Short runs



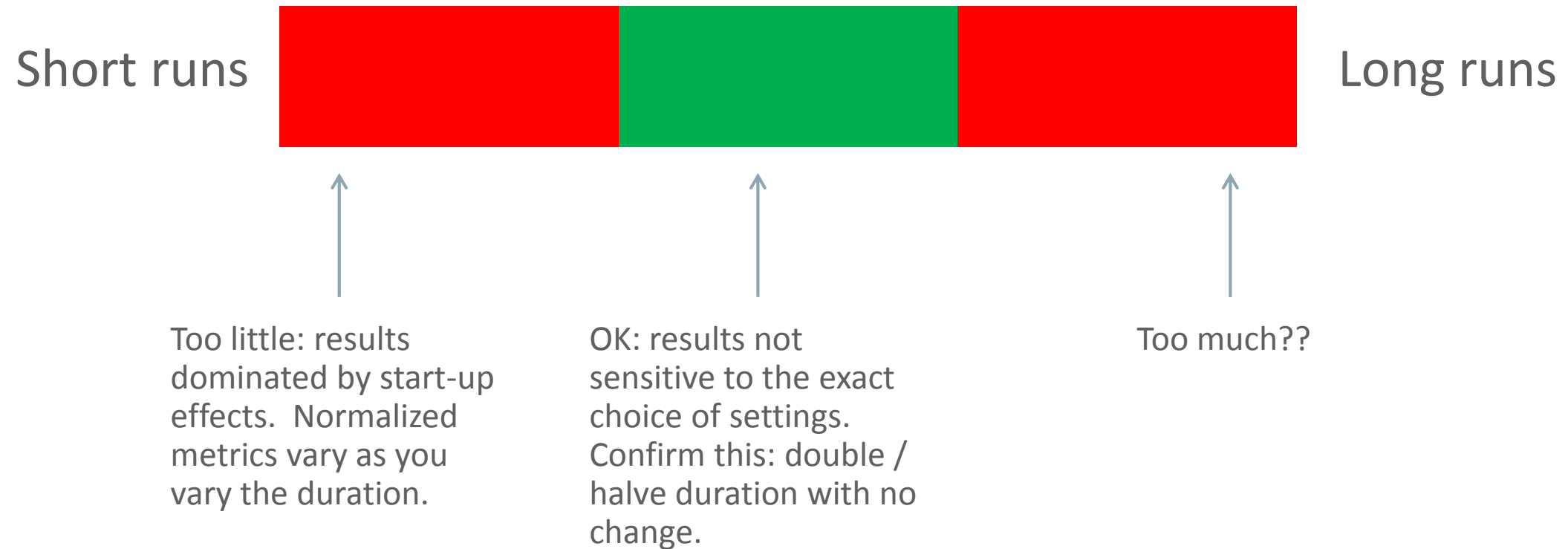
Long runs

↑
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.

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.

↑

Too much??

Deters experimentation if turnaround time is long (e.g. >> overnight)

Harder to separate resource re-use policy from the rest of the expt.

Constant load

- Fixed number of threads active
 - E.g., data structure micro-benchmarks
 - Look at how the structure under test behaves under varying loads
- Keep all threads active throughout experiment. Typically:
 - Create threads
 - Perform warm-up work in each thread
 - Barrier
 - Actual measurement interval
 - Main thread signals request to exit to others
- Investigate and report differences in actual work completed by threads

Constant work

- Fixed amount of work to perform
 - Share it among a set of threads – e.g., OpenMP parallel loop
 - Aim to use threads to complete the work more quickly
 - Measure from when the work is started until when it is all complete
- Show results for
 - Strong scaling: same amount of work as you vary the number of threads
 - Weak scaling: increase the work proportional to the threads
- Investigate and report differences in
 - Load imbalance (do threads finish early?)
 - Actual amount of work completed by threads (do some threads work faster?)

Lightweight correctness checks

- Be skeptical about the results

Lightweight correctness checks

- 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

Lightweight correctness checks

- 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
- Does the data structure pass simple checks?
 - Start with N items, insert K , delete L , check that we have $N+K-L$ 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?

Overview

1

Sound experimental practices

2

Understanding simple cases

3

Exploring performance trends

Understand simple cases first

- Why? Almost without exception:
 - There are bugs in the test harness
 - There are bugs in the data processing scripts (grep, cut-n-paste, ...)
 - There are unexpected factors influencing the results

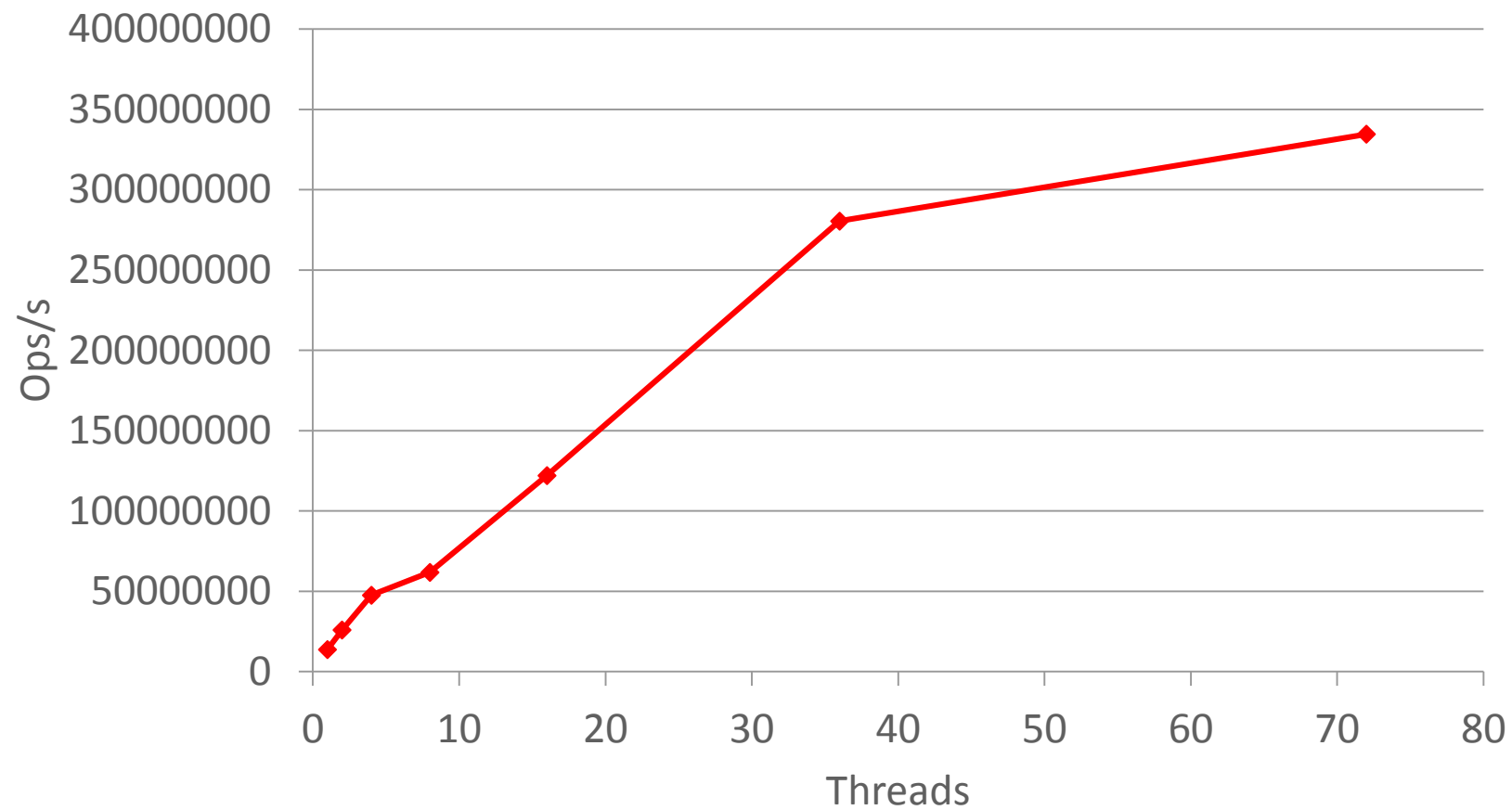
Understand simple cases first

- Why? Almost without exception:
 - There are bugs in the test harness
 - There are bugs in the data processing scripts (grep, cut-n-paste, ...)
 - There are unexpected factors influencing the results
- Before paying any attention to actual results, try to identify simple test cases that should have known behavior
 - (Even if you do not care about them, or they are contrived)
 - Do they behave as expected?
 - Can you completely explain them? (“Memory system effects” is not an answer – see Trevor Brown’s talk later today)

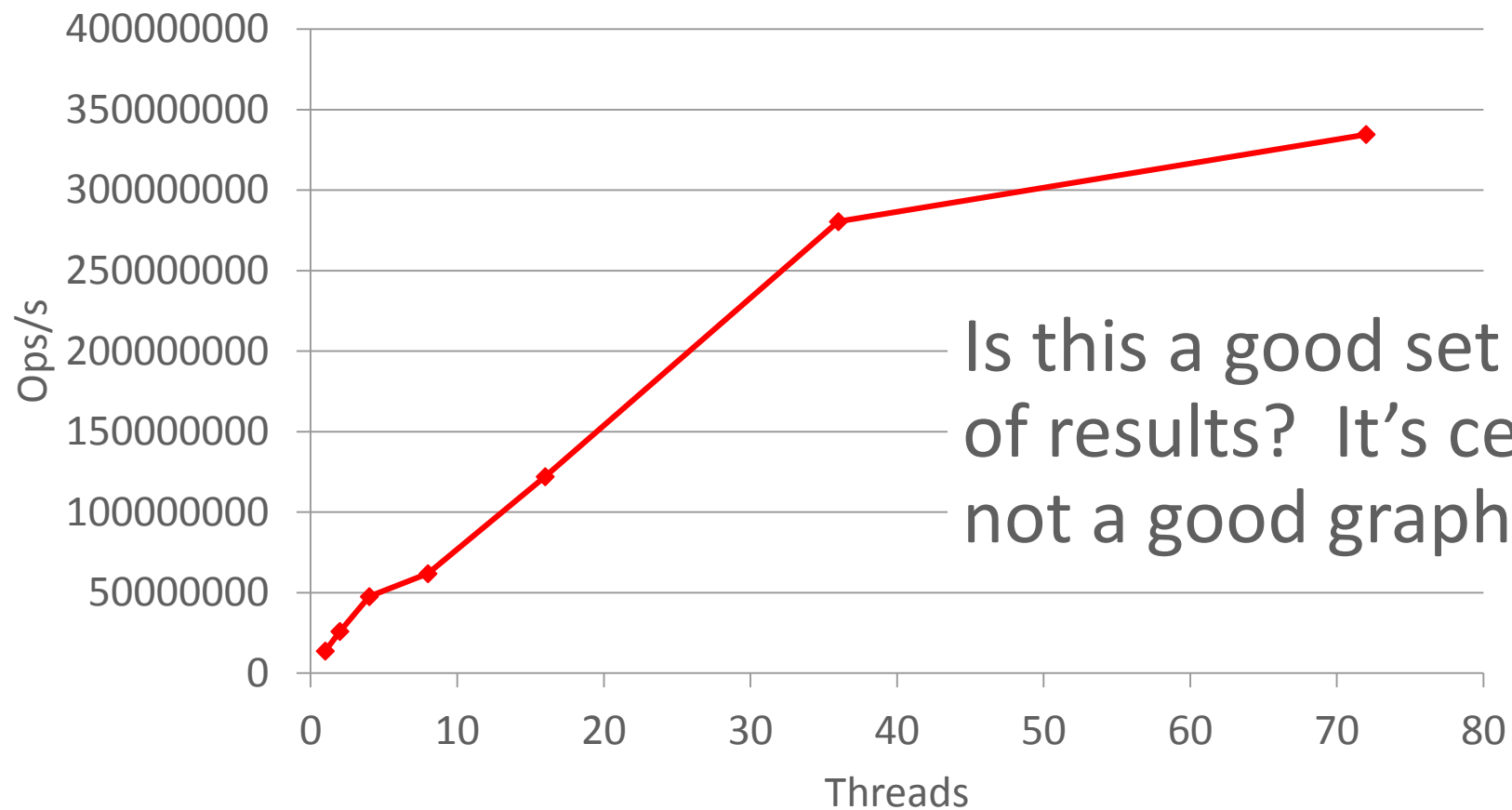
Basic stats to watch

- Elapsed time, system time, CPU time, context switch counts
- Should the workload be 100% user mode?
 - Confirm this with “top”, check that “strace” is quiet (no system call activity)
- Where are the threads running?
- Where is the memory they access located?
- What do profiling tools show?
 - Can you use with optimized builds? If not, check impact of disabling optimization
 - Look at simpler 1-thread workloads – as expected?
 - Increase thread count and look for trends

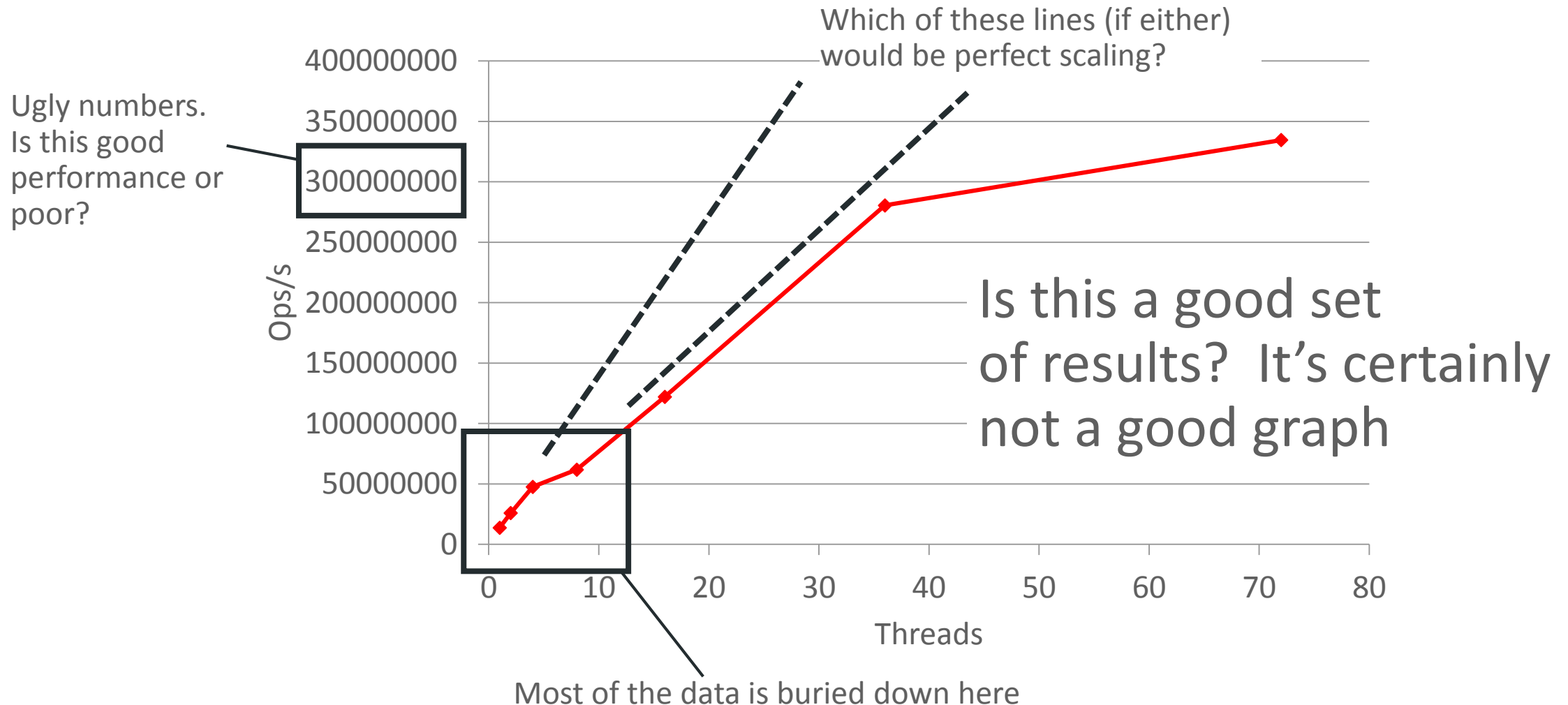
Synchrobench, Fraser skip-list, 100 % read only, 2*Haswell



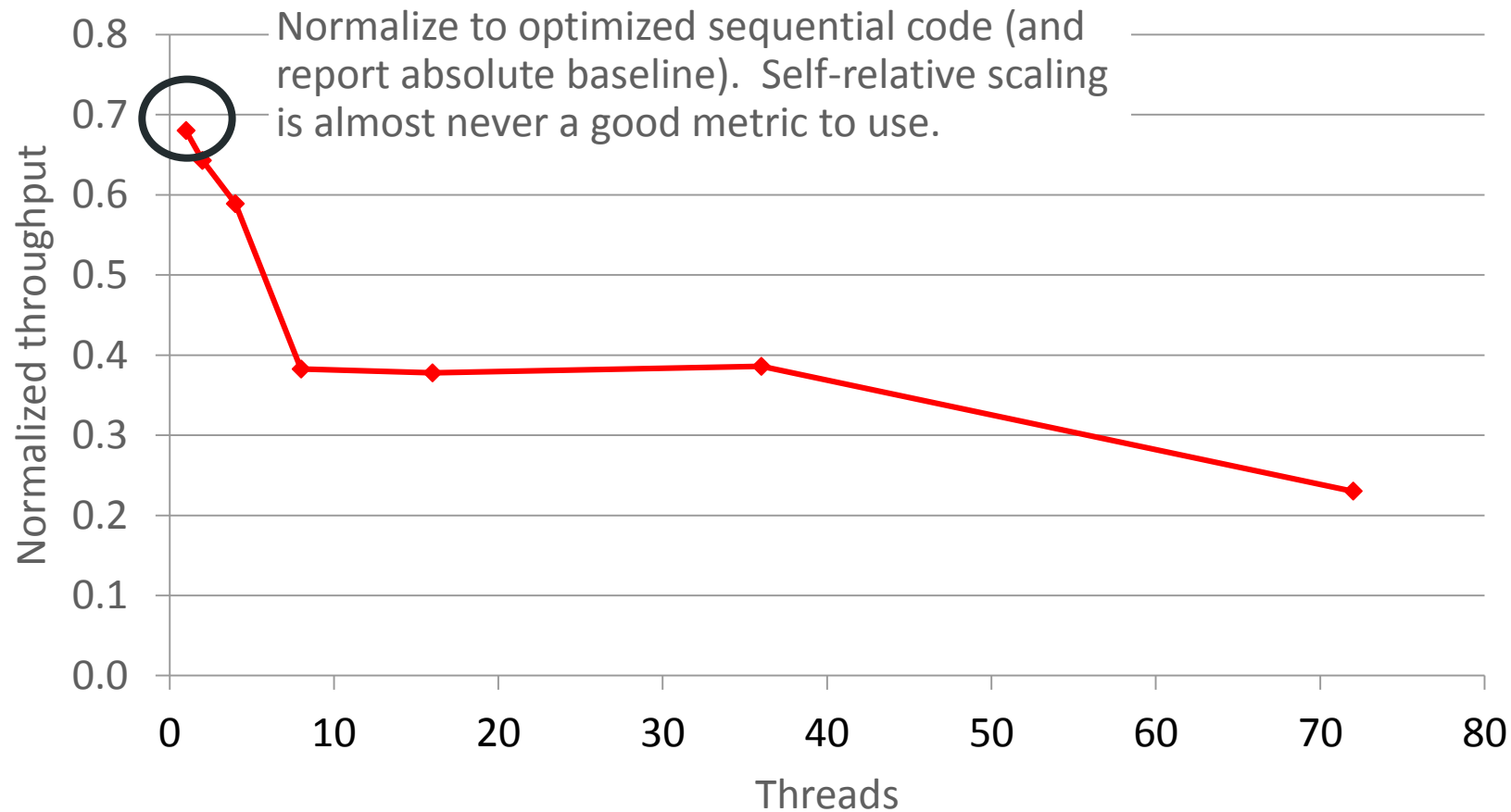
Synchrobench, Fraser skip-list, 100 % read only, 2*Haswell



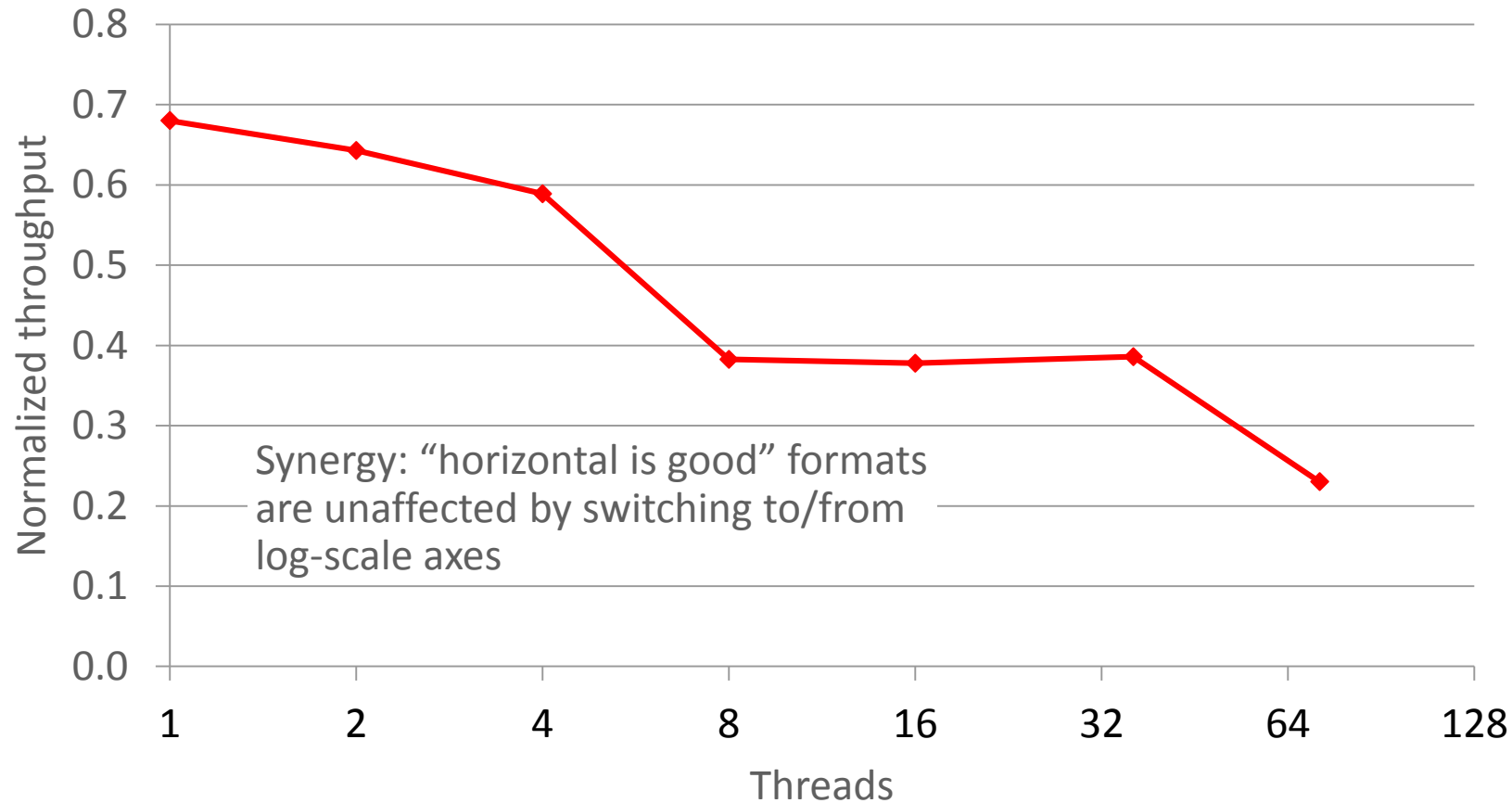
Synchrobench, Fraser skip-list, 100 % read only, 2*Haswell



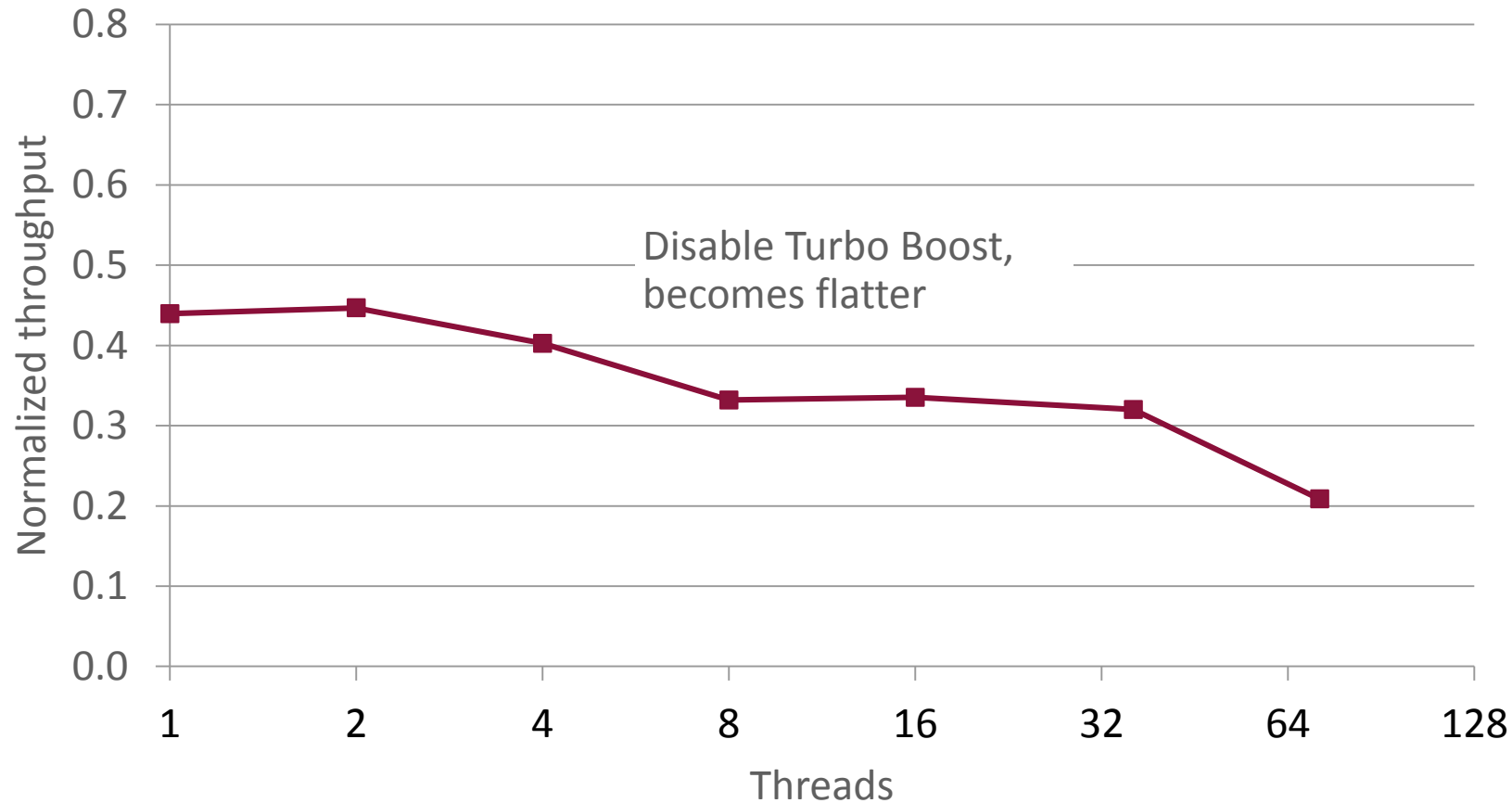
Synchrobench, Fraser skip-list, 100 % read only, 2*Haswell



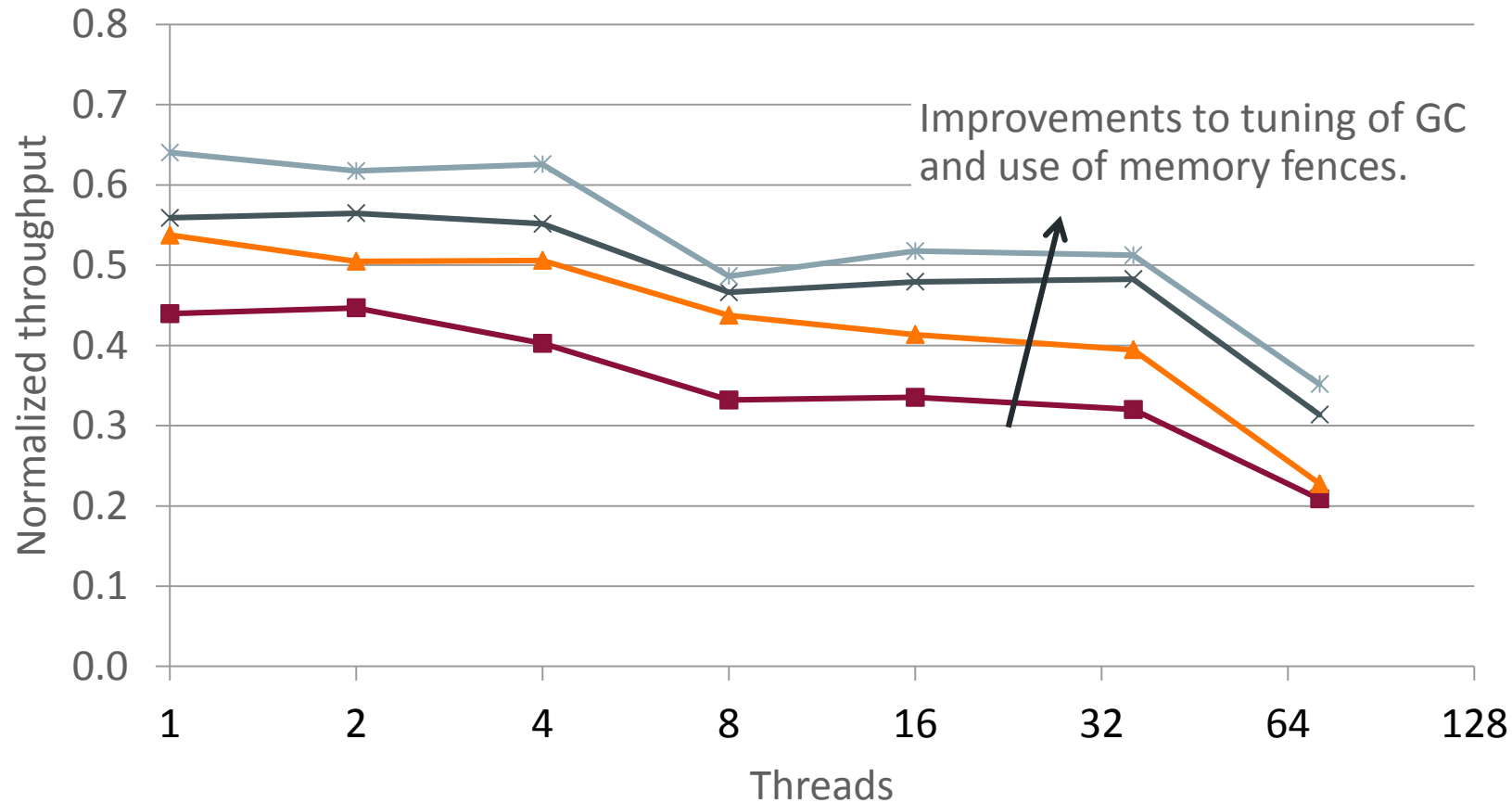
Synchrobench, Fraser skip-list, 100 % read only, 2*Haswell



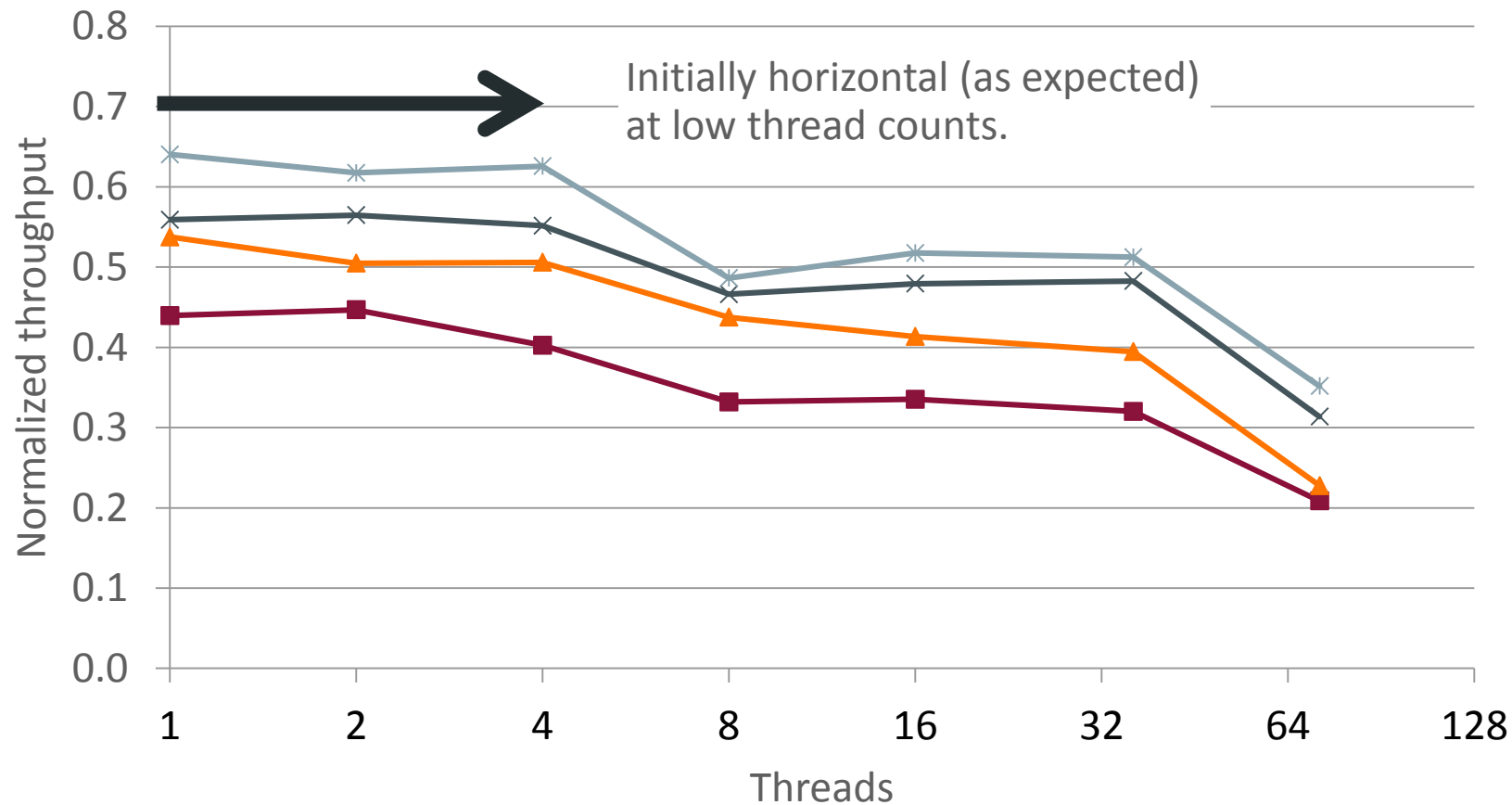
Synchrobench, Fraser skip-list, 100 % read only, 2*Haswell



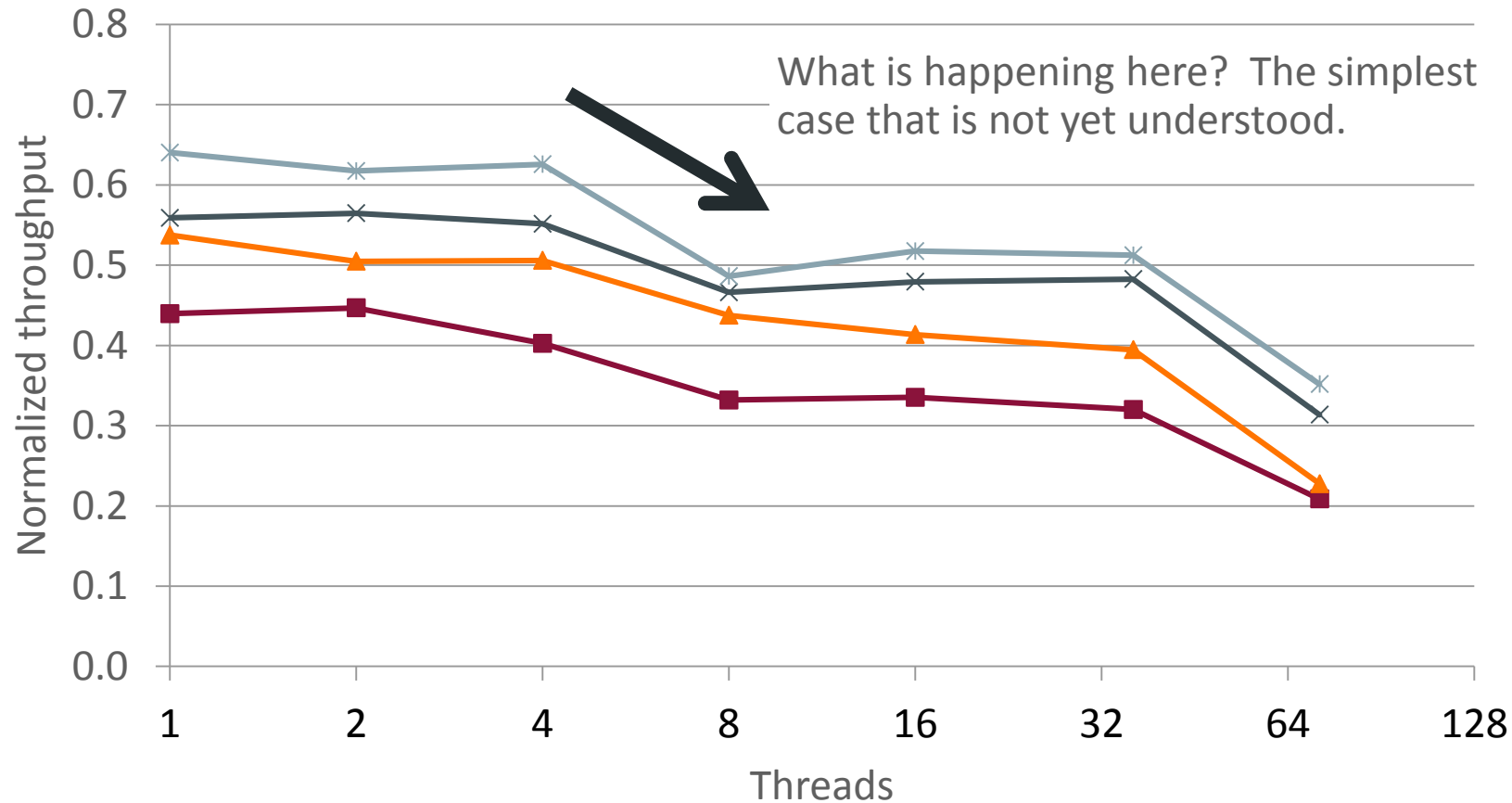
Synchrobench, Fraser skip-list, 100 % read only, 2*Haswell



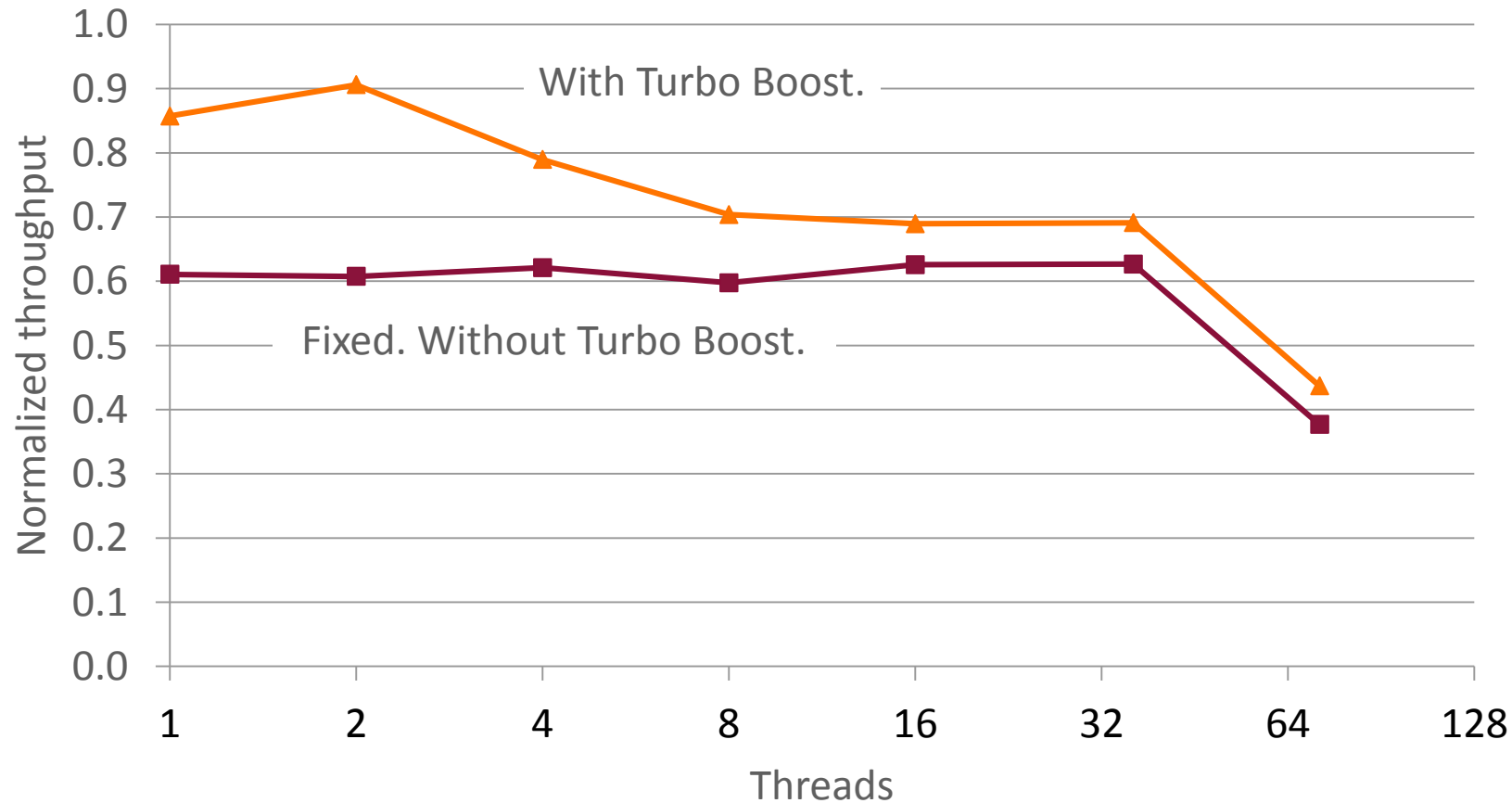
Synchrobench, Fraser skip-list, 100 % read only, 2*Haswell



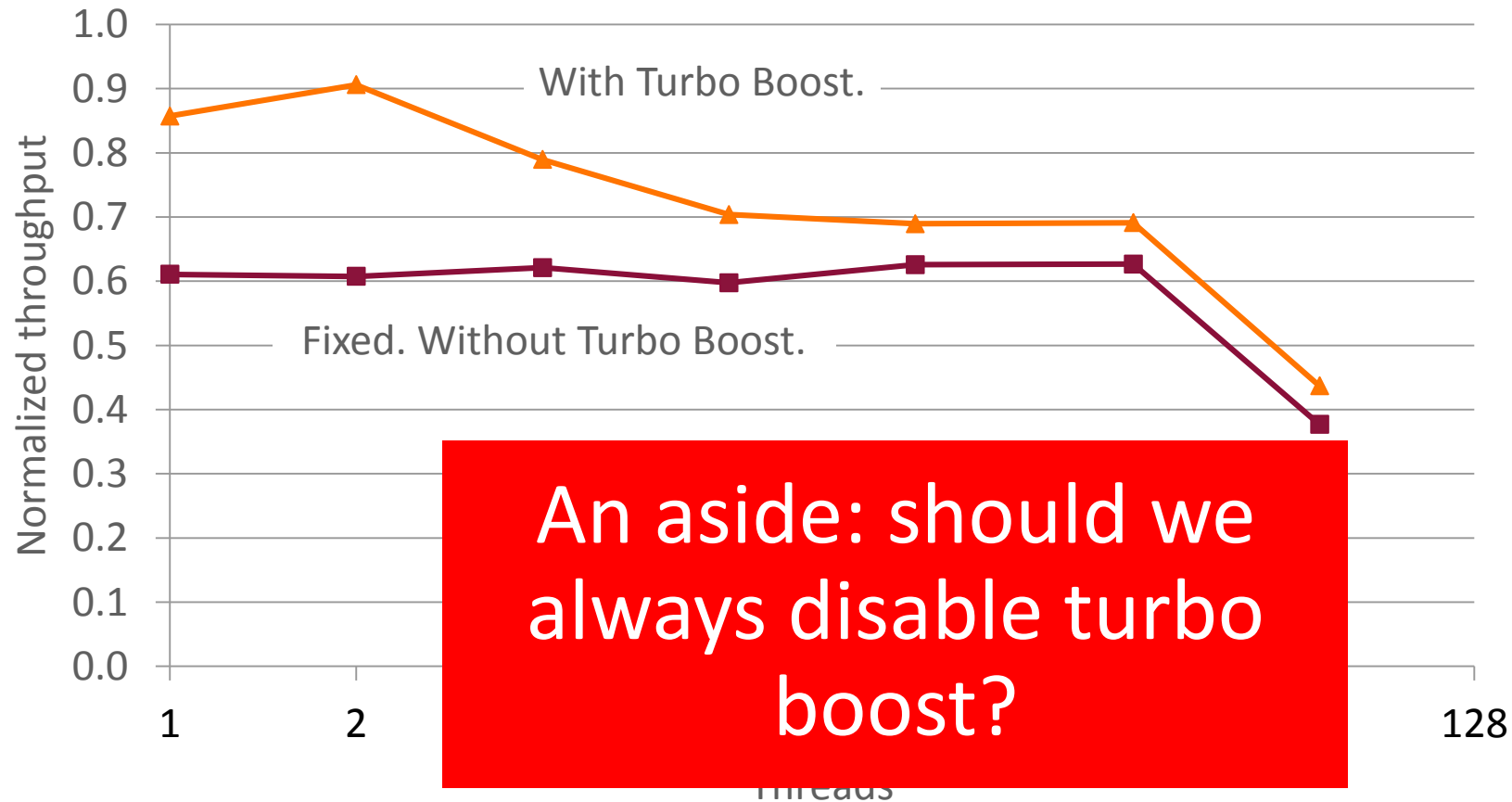
Synchrobench, Fraser skip-list, 100 % read only, 2*Haswell



(It was a stray process still running on the machine)



(It was a stray process still running on the machine)



Overview

1

Sound experimental practices

2

Understanding simple cases

3

Exploring performance trends

Try to identify why performance differs between algorithms

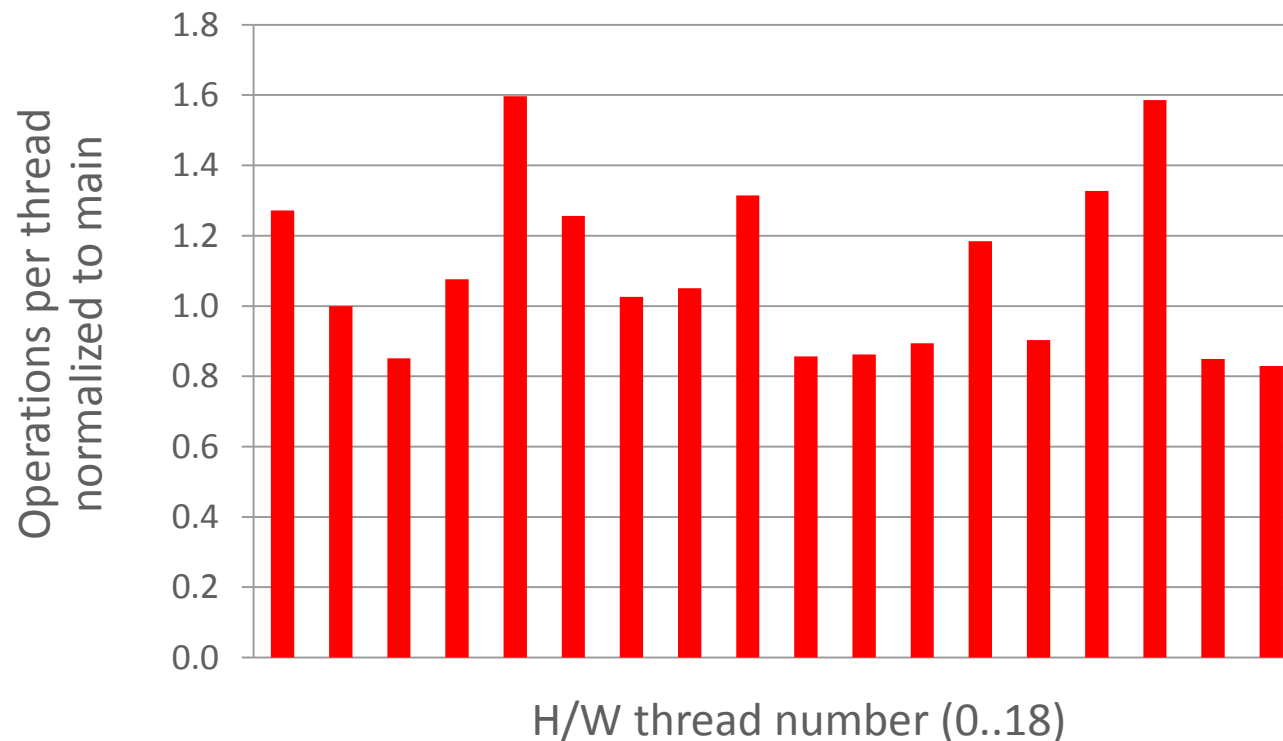
- Several “usual suspects”
 - Try to vary each of these factors in turn
 - Does the perf difference remain/grow/diminish?
- Hard to untangle cause and effect
 - Identifying factors which are significant vs insignificant helps understand behavior
- Four examples:
 - Unfairness
 - Thread placement
 - Memory placement
 - Resource utilization

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

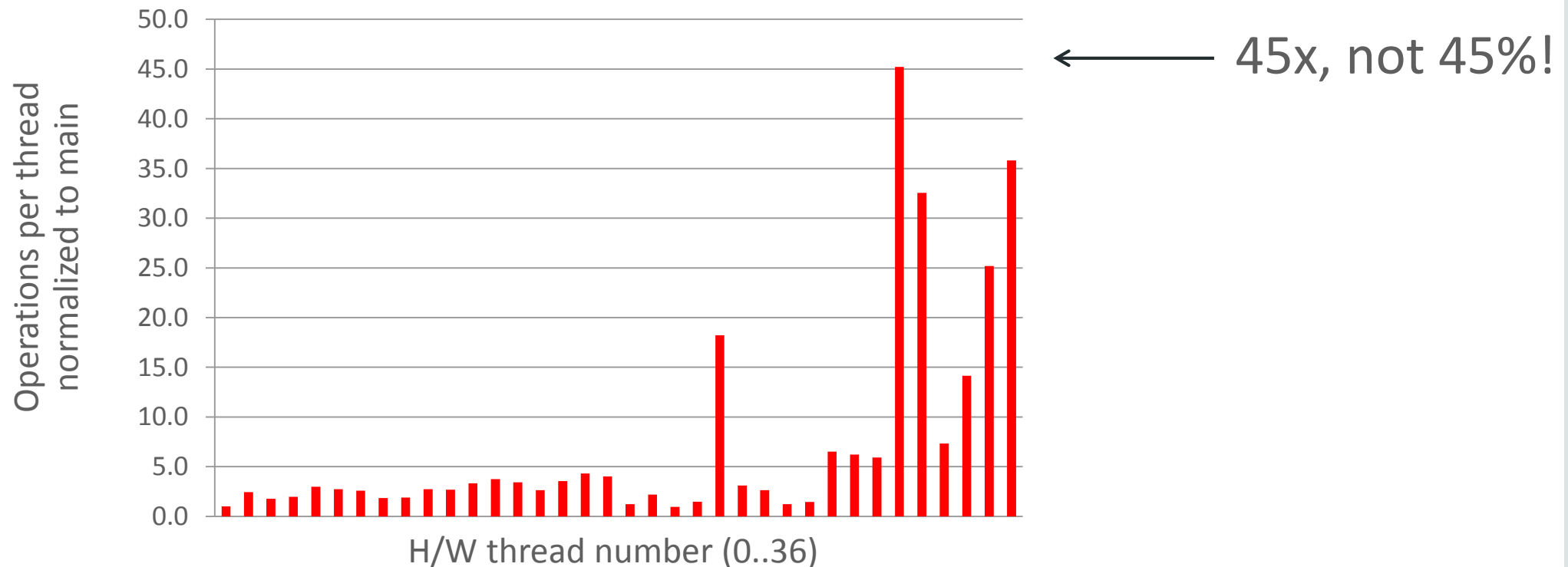
Unfairness: simple test-and-test-and-set lock

- Main thread runs a constant number of iterations, signals others to stop
- 2-socket Haswell, threads pinned sequentially to cores in 1 socket

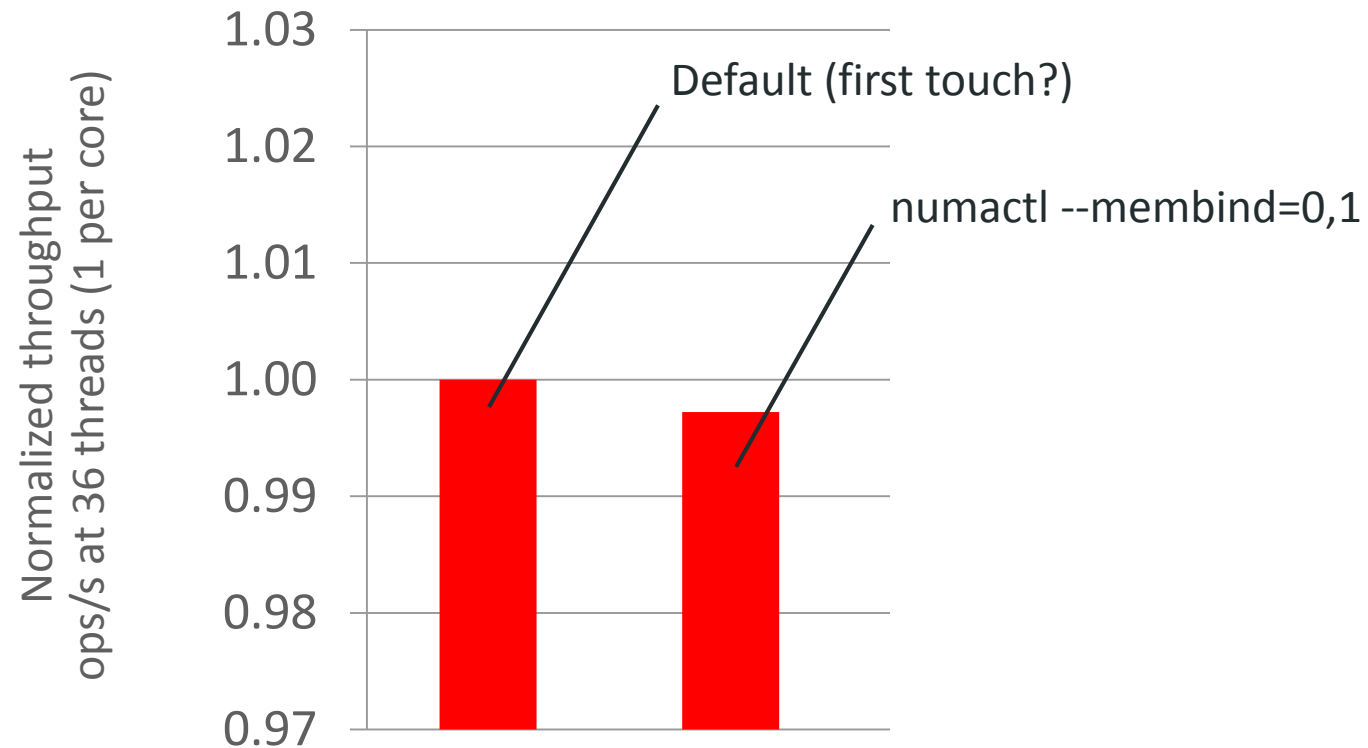


Unfairness: simple test-and-test-and-set lock

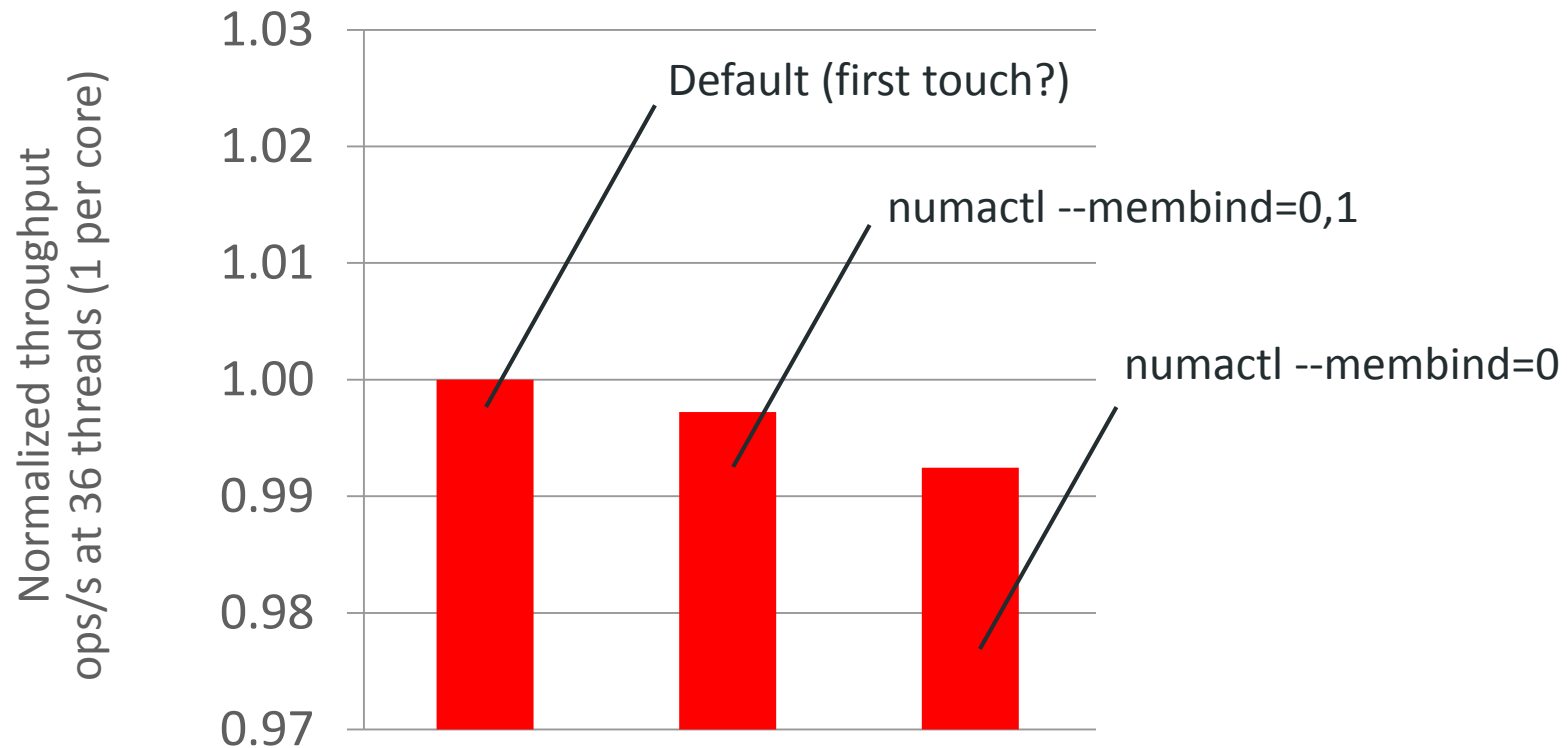
- Main thread runs a constant number of iterations, signals others to stop
- 2-socket Haswell, threads pinned sequentially to cores in both sockets



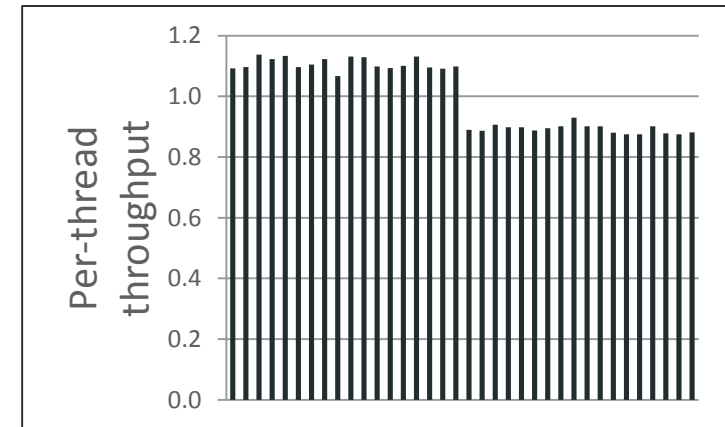
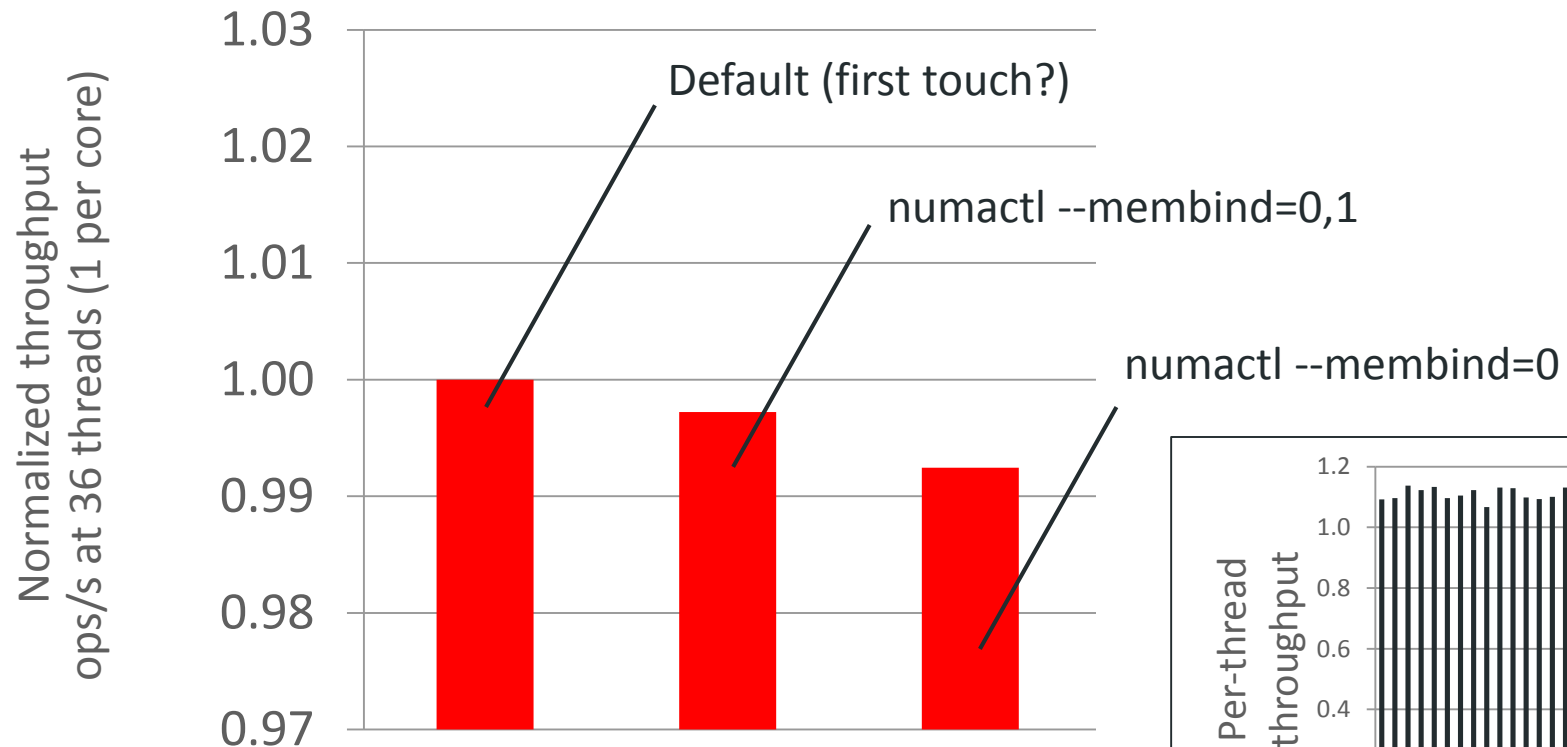
Unfairness: Synchrobench, Fraser skip list, read only



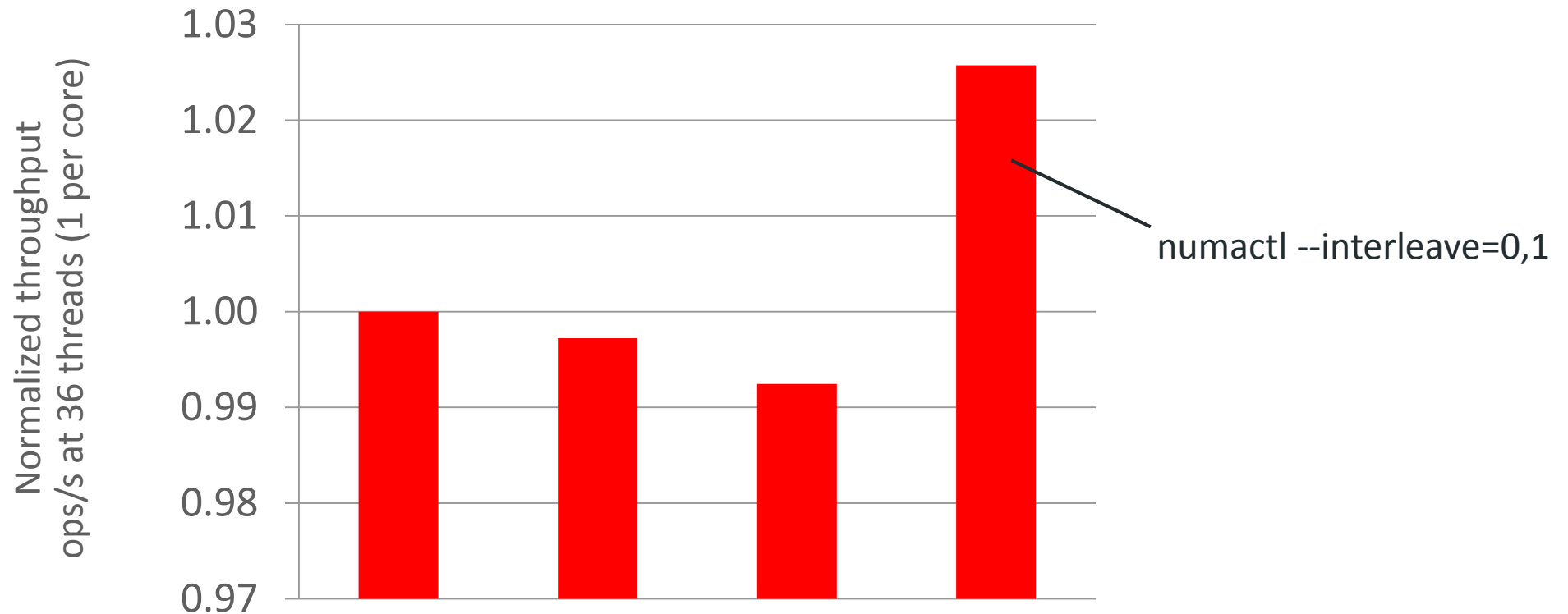
Unfairness: Synchrobench, Fraser skip list, read only



Unfairness: Synchrobench, Fraser skip list, read only



Unfairness: Synchrobench, Fraser skip list, read only



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

- Two aspects to this:
 - NUMA-related allocations – same socket vs different socket?

Memory placement

- Two aspects to this:
 - NUMA-related allocations – same socket vs different socket?
 - Re-use of memory – e.g., via hazard pointers, epochs, etc.

Memory placement

- Two aspects to this:
 - NUMA-related allocations – same socket vs different socket?
 - Re-use of memory – e.g., via hazard pointers, epochs, etc.
- Suppose we have a new GC technique, and execution goes faster
 - Is the GC running faster?
 - Is it giving back memory with better distribution over sockets?
 - Is it giving back memory which is still in the LLC?
- Try to separate out aspects of this behavior
 - Run algorithms with the new vs old GC, but never re-use the memory => only difference is the GC's work

Resource utilization

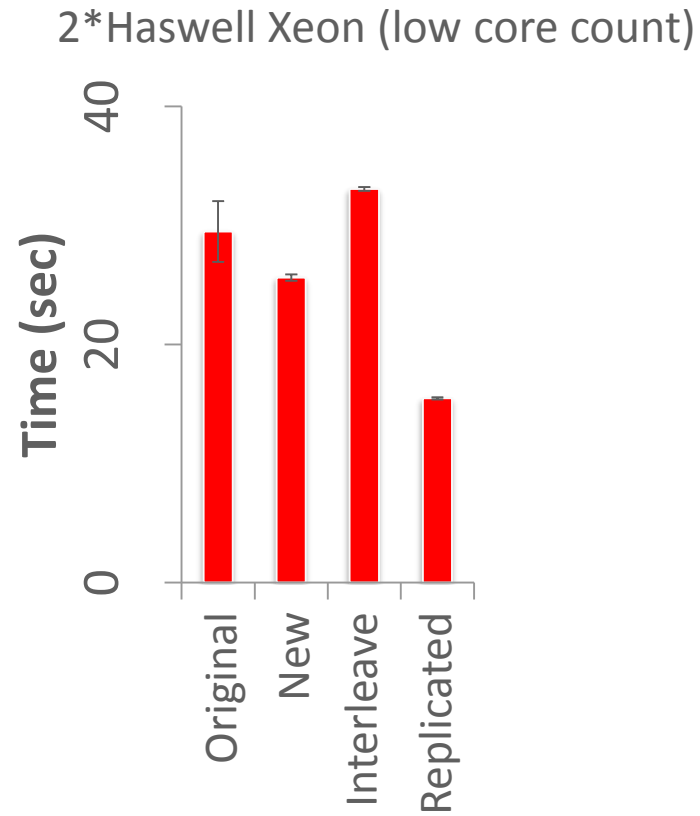
- Examine the use of significant resources in the machine
 - Bandwidth to and from memory
 - Bandwidth use on the interconnect
 - Instruction execution rate
- Clock frequency and power settings
- Look for evidence of bad behavior
 - High page fault rate (i.e., going to disk)
 - High TLB miss rate

Resource utilization

- Let's imagine that we see a difference in resource consumption between two algorithms
 - Threads are placed in the same way
 - Progress is distributed equally across the threads
- Can we relate the difference in resource consumption to the algorithm?
 - More cache misses?
 - More cache hits?
 - More / fewer floating point operations?
- How do these metrics compare with known resource limits on the machine? Have we reduced use of a bottleneck resource?

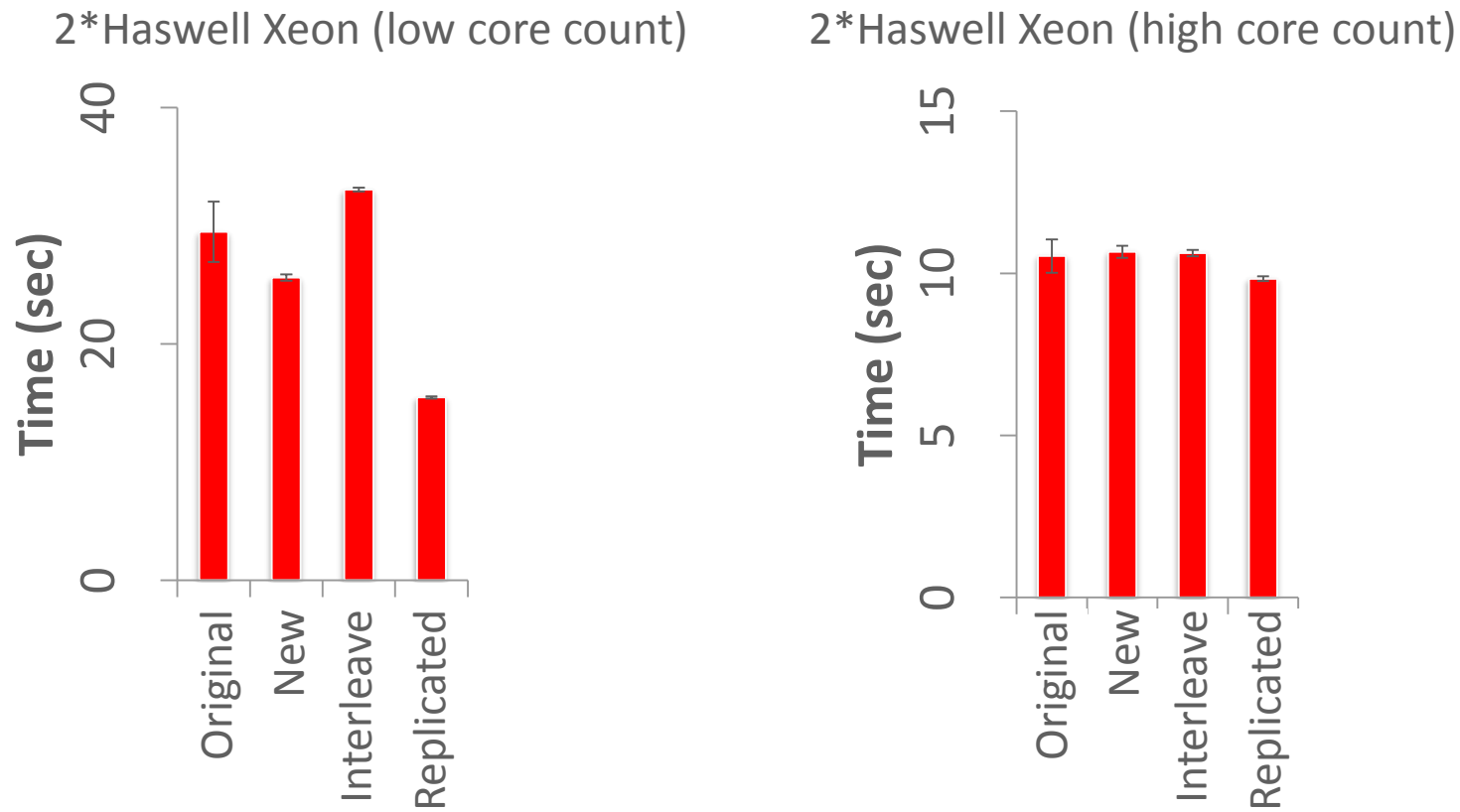
Resource utilization – summing two arrays

Should we duplicate the arrays in local memory on each socket?



Resource utilization – summing two arrays

Should we duplicate the arrays in local memory on each socket?



Overview

1

Sound experimental practices

2

Understanding simple cases

3

Exploring performance trends

Some final points

- 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”

Integrated Cloud

Applications & Platform Services

ORACLE®

Java has its own private ring of performance hell

- E.g., complexity from JIT (execution time, generated code quality)
- Complexity from GC
 - E.g., suppose a change makes a workload that incurred 4 major GCs take 3 or 5 instead
- Look at frameworks such as <http://openjdk.java.net/projects/code-tools/jmh/>
- Cliff Click's podcast <http://cliffc.org/blog/>
- Laurie Tratt's upcoming OOPSLA paper <https://arxiv.org/abs/1602.00602>