Java AOT Baselining

Gary Morrison

Principal Engineer @ Arm®

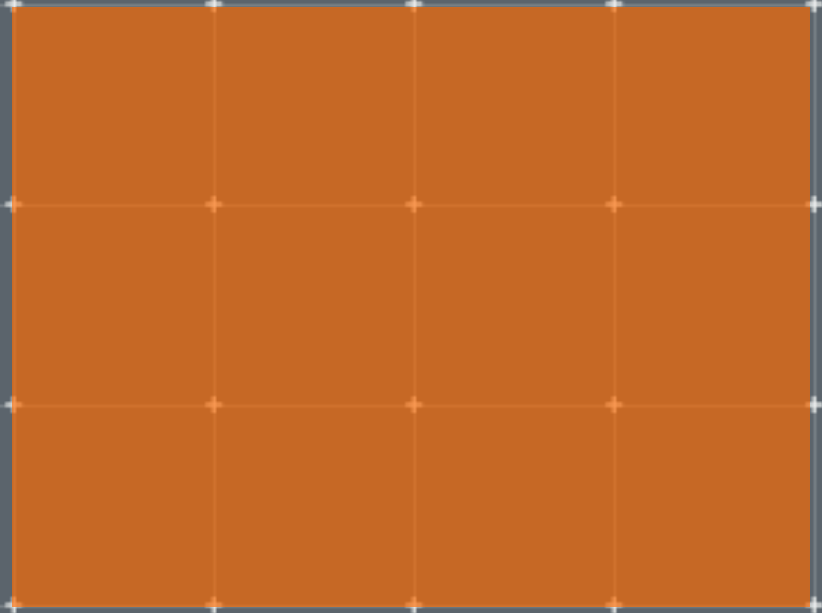
Open-Source Software, Austin, TX, USA

Linaro Connect, 2019/4/5

Copyright © 2019 Arm® Limited

Introduction

Java AOT: Wuzzat?



Copyright © 2019 Arm® Limited

arm

What is AOT? Well, *Without* AOT…

- The `javac` compiler compiles down to a portable, pseudo-machine code, called "bytecode."
- Initially bytecode is run interpreted. However, the JVM (Java Virtual Machine) contains two "JIT" compilers, called "C1" and "C2," that compile bytecode to native assembly.
  - C1 compiles faster than C2, but C2 produces better-optimized code.
  - C1 and C2 together form the *Tiered Compilation* system, called "Hotspot."
- Tiered compilation:
  - As a given method gets "hotter," meaning it *executes more frequently*, it gets compiled *increasingly optimally*.
  - Generally speaking, code compiled at Hotspot's "Tier 4" is more-optimized than code compiled at "Tier 1."
- Hotspot increases the optimization of code, at higher tiers, in two ways:
  - By using the better-optimizing, C2 compiler, and
  - By *profiling code* at lower Tiers, so that never- (or very-rarely-) executed code can be *optimized out entirely* at higher tiers.
- There is also a new, up-and-coming compiler, itself *written in Java*, called *Graal*.

Copyright © 2019 Arm® Limited

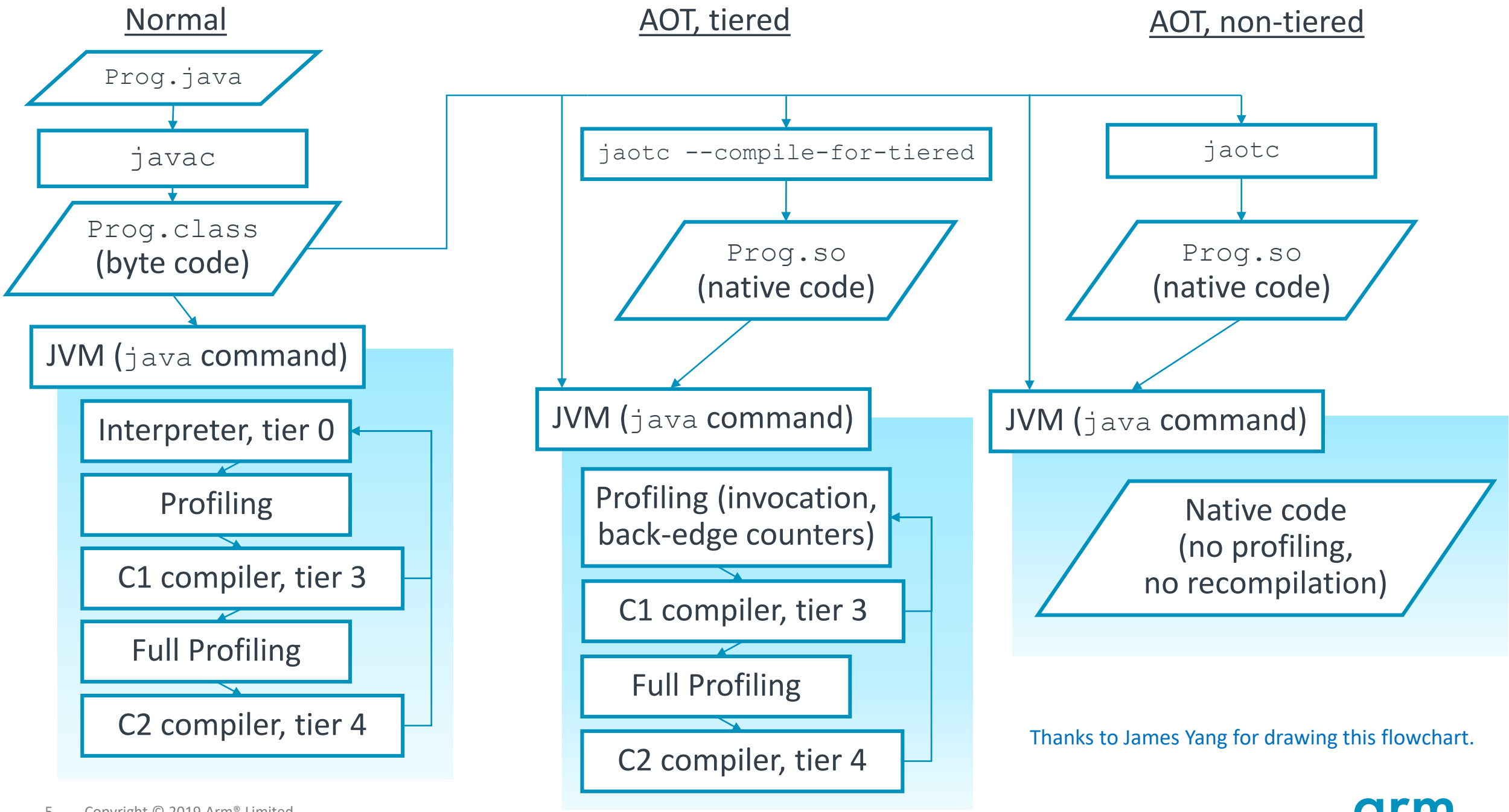
What is AOT? *With* AOT…

- Ahead-of-time ("AOT") compilation is, essentially, *compilation in the historical sense*.
  - Can think of it as an extension of the **`javac`** step, bytecode down to native assembly.
  - *JEP295* defines AOT.
    - *Primary goal*: Improve initial performance, by running native code as soon as possible.
    - Secondary goal: "Change the end user's work flow as little as possible."
  - You can AOT *with or without* `--compile-for-tiered`:
    - *With* `--compile-for-tiered`: "Tiered AOT" works like normal Hotspot, but profiling with *high-performance native code*.
    - *Without* `--compile-for-tiered`: with "untiered AOT," the AOT compilation is exactly what is run – no further Hotspot optimization.

- New tool = **`jaotc`** (Java ahead-of-time compiler):
  - **`jaotc`** operates upon the bytecode, *using **Graal**, to generate a* **`.so`** *library*.
  - To use these libraries, for example:

```
java -XX:+UseAOT -XX:AOTLibrary=./libHelloWorld.so,./libjava.base.so HelloWorld
```


Copyright © 2019 Arm® Limited



Thanks to James Yang for drawing this flowchart.

Copyright © 2019 Arm® Limited

arm

The Task: Baselining AArch64 Against x86

- Assignment = “baselining”:
  - Compare two Arm® cores, especially *with* AOT, to x86.
  - *How the AArch64 AOT implementation* compares to that for x86, factoring out microarchitecture differences.

- Sorry, but *I’ve been asked not to name the particular partner/core-type names*. I’ll call these two core types “A64#1” and “A64#2.” All three core types are *server-market*.

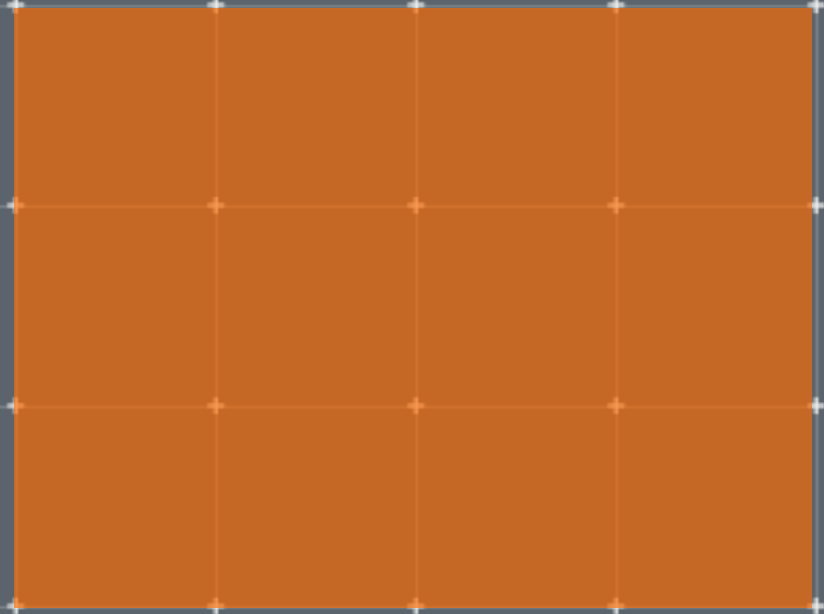
- Two sets of benchmarks used:
  - Initially, five specifically-chosen benchmarks – the “***I5B***”: three from Linaro, and two custom.
  - SPECjvm2008, including/especially the `startup.*` benchmarks, for characterizing startup and warmup.

- x86 ran on JDK11, whereas AArch64 ran with our *latest-at-that-time* Graal fixes.

- Neither of these benchmark sets turned out ideal, and ideal AOT workloads are proving elusive. Nevertheless some interesting observations came up.

Copyright © 2019 Arm® Limited

Initial-Five-Benchmark "I5B" Results



Copyright © 2019 Arm® Limited

arm

Initial Five Benchmarks ("I5B")

- Running under JMH seemed a bit overkill. (Also, this is mostly about warm-up time, whereas JMH is nominally designed to *factor out* warm-up time.)

- Instead, each benchmark's essential workload was placed into a "big outer loop," and each loop iteration was timed (in ns).

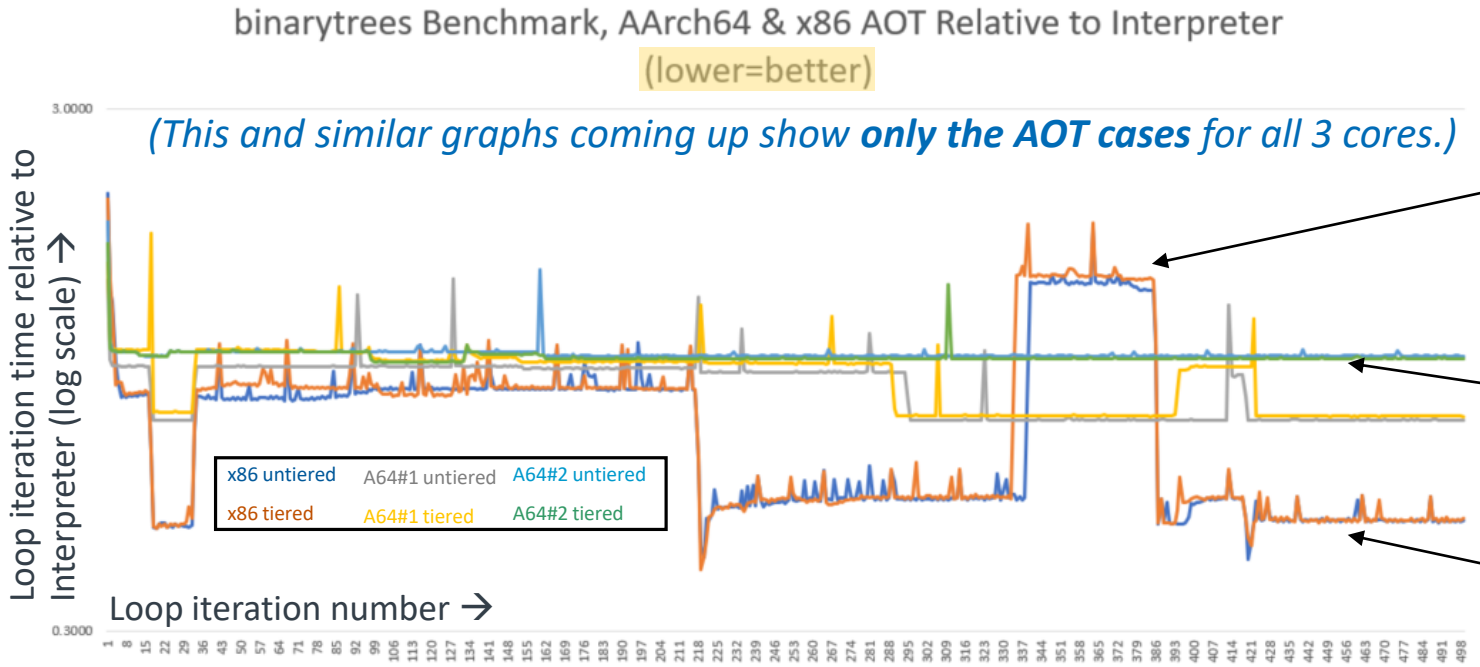
- The "I5B" graphs coming up are *scaled relative to the performance of* ***non-AOT*** *loop iterations 2-9, running in the interpreter*. Also, these graphs show only the AOT runs.

- In these graphs, unlike the SPECjvm graphs later, so ***lower is better***. They are also plotted on a ***logarithmic*** *Y-axis* (execution time) scale.

Copyright © 2019 Arm® Limited

"I5B" `binarytrees` Benchmark

- This Linaro benchmark was included for its use of *recursion*. Recursion complicates in-lining, and just generally has distinctive performance characteristics.



Why does (only) x86 briefly slow down well below its initial plateau?

Only A64#2 showed a "flat-line" performance, expected for untiered AOT.

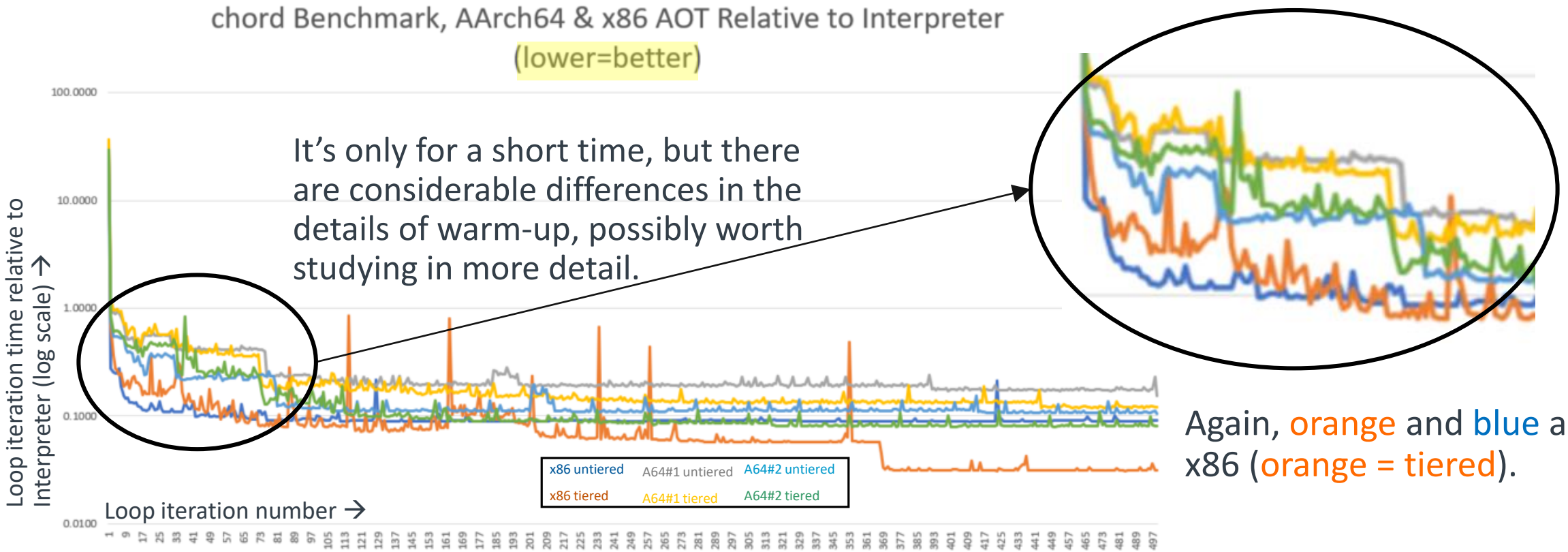
Orange and blue are x86 (tiered vs. untiered).

- ***Interestingly,*** `--compile-for-tiered` ***made very little difference*** on any of the three cores.
- The x86 JVM implementation was able to optimize something AArch64 doesn't – most likely, part of `java.base` that could not be AOTed.

Copyright © 2019 Arm® Limited
arm

"I5B" `chord` Benchmark

- This benchmark doesn't exercise anything in particular; it's just "fairly-average code."

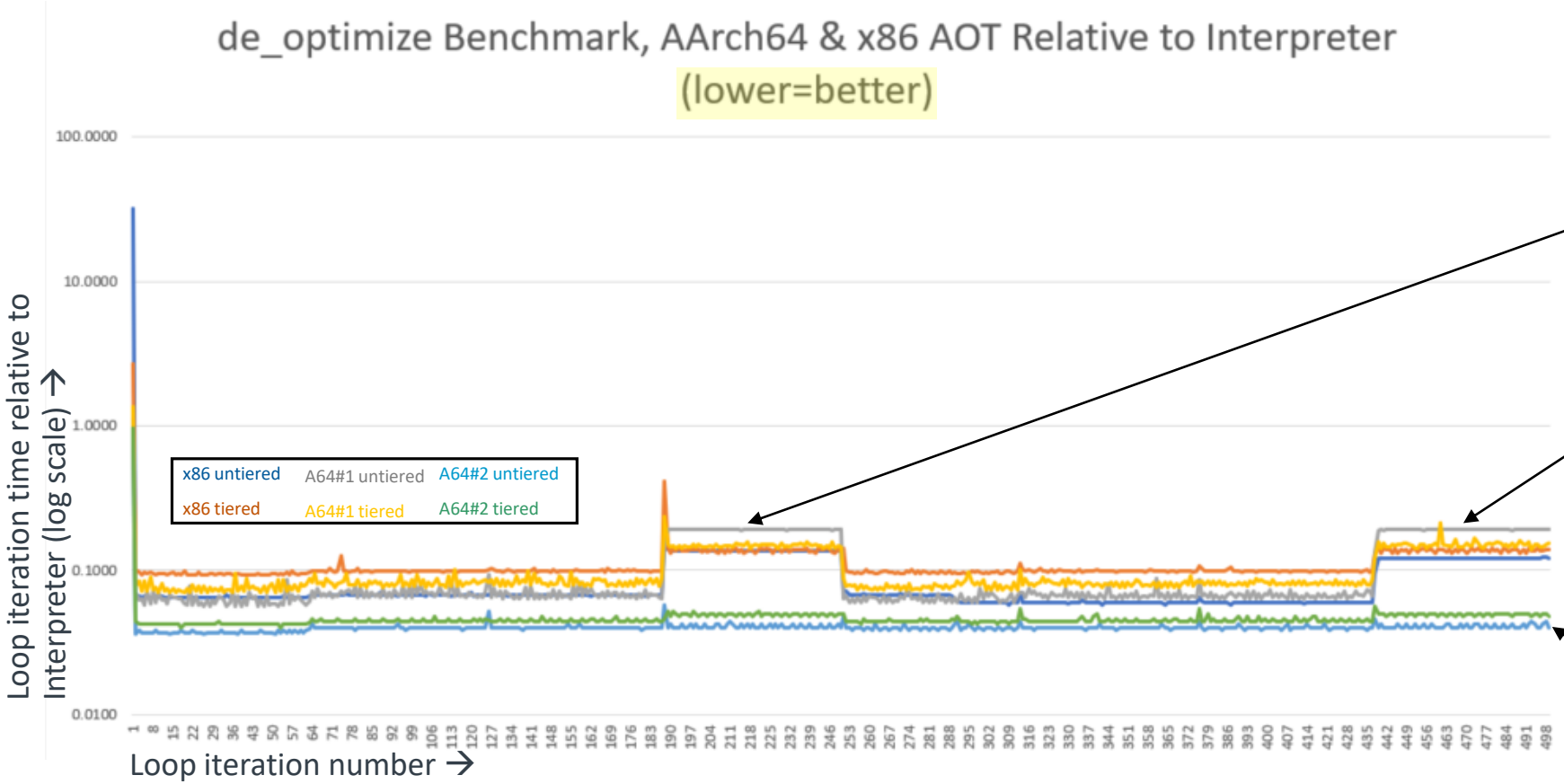


- Admittedly, this benchmark may not tell us a lot, except that x86 with `--compile-for-tiered` warmed up quickly, and then its performance improved somewhat way at the end (hmmm… why?).

Copyright © 2019 Arm® Limited
arm

"I5B" de_optimize Benchmark

- This benchmark constructs an array containing a run of subclass1 objects, then a run of subclass2 objects, subclass3 (etc.) and then executes a polymorphic method from each.



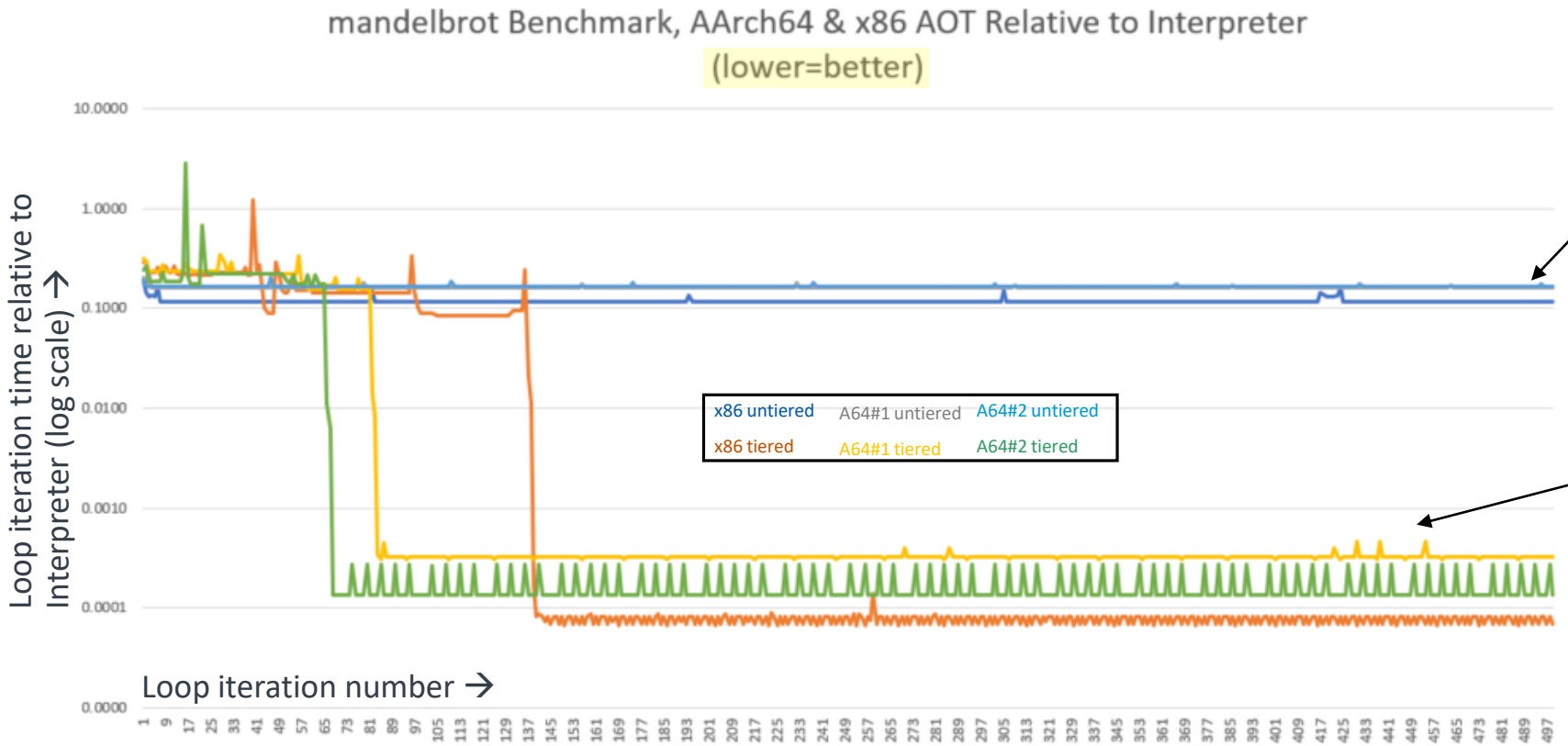
These upward bumps (slow-downs) on the fourth subclass is not related to the JVM implementation. The workload performs a FP division, where the other methods perform +, -, and *.

A64#2 did very well, especially on the FP divides!

Copyright © 2019 Arm® Limited
arm

"I5B" `mandelbrot` Benchmark

- This benchmark was originally included as a floating-point intensive benchmark, but turned out intriguing for a probably-even-more-interesting reason: *Hotspot optimized out a large chunk of code*, and how this happened turned out interesting!



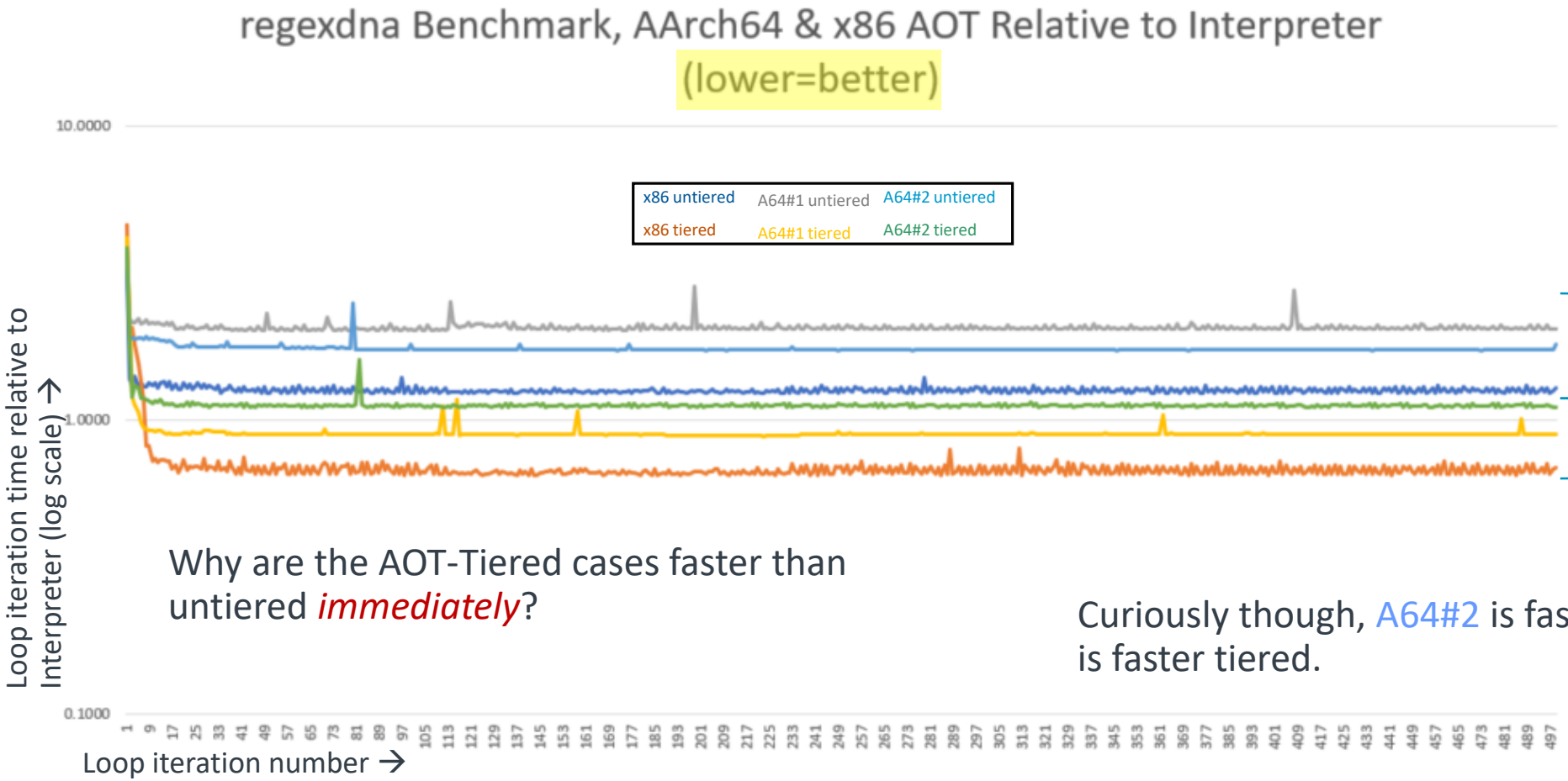
Not surprisingly, these are the non-tiered cases.

The relative timings of when the different versions optimize out the code are interesting. In this case, AArch64 did better. Orange is x86.

Copyright © 2019 Arm® Limited
arm

"I5B" regexdna Benchmark

- This benchmark does a lot of regexes, just since regex matching is very common.



These are the untiered versions, x86 fastest.

These are the tiered versions, x86 fastest.

Why are the AOT-Tiered cases faster than untiered *immediately*?

Curiously though, A64#2 is faster untiered, whereas A64#1 is faster tiered.

Copyright © 2019 Arm® Limited

arm

“I5B” Findings Potentially Worthwhile to Explore Further

- On **`binarytrees`**, x86 optimized *something* AArch64 could not, to improve performance (e.g., at the end).
- On **`chord`**, several curious differences in the details of warm-up might prove enlightening to investigate.
- On **`mandelbrot`**, a huge optimization occurred at very different times, and to somewhat different levels. In this particular case, AArch64 did better – optimized it sooner – than x86. What exactly are we doing right?
- On **`regexdna`**, not surprisingly, AOT tiered did better than untiered, but it seemed to warm up seemingly impossibly quickly.

Copyright © 2019 Arm® Limited

SPECjvm2008 Results

Copyright © 2019 Arm® Limited

arm

SPECjvm2008 and Methodology

- Ran SPECjvm2008 on the same core types – x86, A64#1, and A64#2 – and same AOT usage – non-AOT, AOT-untiered, and AOT-tiered.
- SPECjvm suite includes several, real-world-ish benchmarks.
- SPECjvm benchmark structure:
  - An "iteration" of many "operations" can then be run for a specified number of seconds.
  - You can then tell it to run a specified number of such timed iterations.
- ***Or***… `startup` tests:
  - SPECjvm `startup` tests run just one "operation" of a given benchmark, in a new JVM.
  - *Startup* tests *with AOT* run immediately in native code, instead of in the interpreter.
- Iterations ran for only 15 seconds to reduce Hotspot optimization, *for greatest contrast AOT vs. not*.

Copyright © 2019 Arm® Limited

Overall Observations from SPECjvm Results

- Not surprising: Untiered AOT was typically 10-70% slower than non-AOT, or tiered AOT.
- Rather surprising: Tiered AOT was often *a few % slower* than non-AOT!
- A handful were considerably *more than a few % slower*.
- All in all, performance (relative to non-AOT) seemed to be more a function of the particular benchmark than of the particular CPU type.
- Generally speaking, the SPECjvm results were only so-so informative, but as with the "I5B," the results give some hints for what to explore.

Copyright © 2019 Arm® Limited

Too Many Dimensions

- There are far too many combinations of factors to present all raw-data graphs:
  - Every CPU type, for every
  - Benchmark (e.g., `compress` vs. `crypto` vs. `scimark`…), for every…
  - Sub-benchmark (e.g., `scimark.fft` vs. `scimark.sor`…), for every…
  - The iteration number, for every
  - *Startup* (single operation) for a given benchmark, vs. the more-continuous run.

  Altogether ***many*** data sets! ☺
- The following slides will therefore examine each of the five "observations" from the previous slide, concentrating mostly upon:
  - *x86 vs. AArch64*, since that's what we're baselining and want to improve,
  - SPECjvm sub/benchmarks that showed the *most variation* between AOT and non-AOT, and
  - Greater interest in the `startup.*` benchmarks, since AOT *should* affect them more.
- First, a quick note: In contrast with the "I5B," the numbers shown here for SPECjvm are operations per minute, *performance* numbers, so ***higher is better***.

Copyright © 2019 Arm® Limited
arm

Untiered AOT was typically 10-70% slower than non-AOT, and Tiered AOT was often a few % slower than non-AOT

- Using A64#2 as an example (other CPUs were similar in this regard):

A64#2

Operations per minute
Higher is Better

Benchmark	No AOT	Untiered AOT	Untiered-to-No Δ%	Tiered AOT	Tiered-to-No Δ%
compress	42	20	-53%	42	-1%
crypto	95	69	-27%	97	2%
derby	61	25	-59%	59	-3%
mpegaudio	28	19	-34%	27	-2%
scimark.large	16	8	-50%	16	1%
scimark.small	53	28	-47%	50	-4%
serial	21	8	-62%	23	10%
startup	31	25	-18%	30	-4%
sunflow	42	25	-40%	36	-14%
xml	64	55	-13%	62	-3%

- However, note that `startup` cases averaged “only” 18% slower.

Copyright © 2019 Arm® Limited
arm

Untiered AOT was typically 10-70% slower than non-AOT, and Tiered AOT was *mostly* just a few % slower than non-AOT (ctd.)

- The differences for startup in particular were less extreme but more varied (also not surprising):

Operations per minute
Higher is Better

Startup Benchmark	No AOT	Untiered AOT	Untiered-to-No Δ%	Tiered AOT	Tiered-to No Δ%
startup.helloworld	134	130	-3%	123	-8%
startup.compiler.compiler	136	126	-8%	124	-9%
startup.compiler.sunflow	140	125	-10%	119	-15%
startup.compress	30	17	-42%	29	-4%
startup.crypto.aes	11	10	-9%	11	-2%
startup.crypto.rsa	54	70	29%	58	8%
startup.crypto.signverify	58	62	7%	67	17%
startup.mpegaudio	17	16	-5%	18	10%
startup.scimark.fft	55	49	-11%	54	-3%
startup.scimark.lu	50	28	-43%	51	3%
startup.scimark.monte_carlo	30	11	-62%	17	-43%
startup.scimark.sor	23	19	-17%	23	-1%
startup.scimark.sparse	23	23	1%	24	4%
startup.serial	12	7	-43%	13	8%
startup.sunflow	20	20	1%	23	15%
startup.xml.transform	1	1	-3%	1	-3%
startup.xml.validation	18	18	-1%	18	-4%

Copyright © 2019 Arm® Limited
arm

startup.scimark.monte_carlo

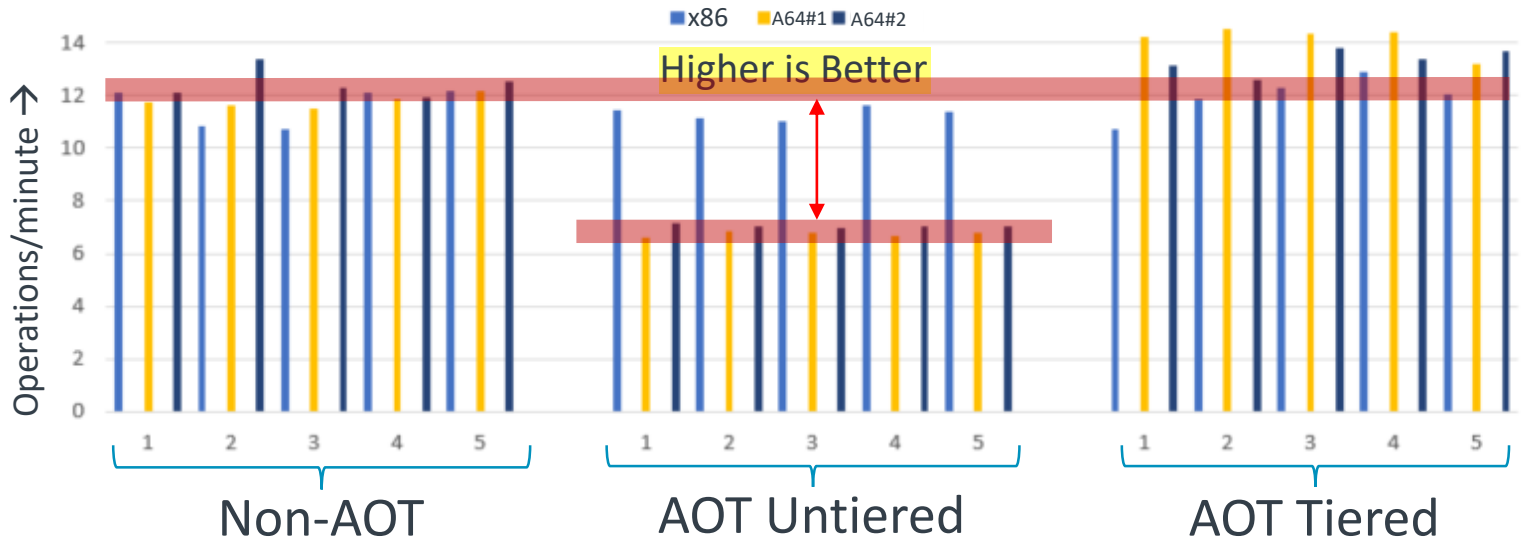
- Startup benchmarks `startup.scimark.monte_carlo`, and `startup.serial` have interesting characteristics worth showing in detail, and perhaps investigating further.

- x86 shows considerably *less* reduction in performance from non-AOT to AOT-tiered: ~6% from ~38.6 to ~36.2 ops/min. A64#1 goes down ~20% from ~28.3 to ~22.5, and A64#2 drops ~42% from ~29.9 to ~17.3.
- This may suggest that what `monte_carlo` does is especially sub-optimal under Graal.

Copyright © 2019 Arm® Limited
arm

`startup.serial`

- Similarly, here's `startup.serial`:



- `startup.serial` is the only startup case, under AOT-***untiered***, where AArch64 performance went down considerably (~42%), but x86 performance didn't change much at all.
- *This probably suggests performance-improvement opportunities in the AArch64 Graal implementation*.
- However, curiously, both AArch64 AOT-***tiered*** cases went up considerably (12% and 23%). In many *other* cases, there was some loss of performance, rather than improvement, from non-AOT to AOT-tiered cases.

Copyright © 2019 Arm® Limited
arm

Overall Conclusions

Copyright © 2019 Arm® Limited

arm

Interesting AOT workloads seem to be hard-to-find

- SPECjvm results especially, but “I5B” results as well, *don’t clearly show*…
  - … huge *warm-up-time benefit* for AOT, nor
  - … what the *best AOT settings* are.
- Still, there are differences between the x86 and AArch64 timings that, upon further investigation, hint about what to investigate further.
- Why is *Tiered-AOT often slightly slower than non-AOT*? *Conceptually*, especially on startup tests, it should be *considerably faster*! Hypothesis: profiling differences.
- Everybody seems to agree that benchmarks of these sorts aren’t likely to show big differences, and that finding workloads that do is “challenging.” *Hotspot is **good***!
- However, *AOT-untiered does reduce overall CPU loading by turning off JIT compiles*!
- Feel free to Email questions to *gary.morrison@arm.com*.

Copyright © 2019 Arm® Limited
arm

Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
תודה

Confidential © 2018 Arm® Limited

Backup…

Copyright © 2019 Arm® Limited

arm

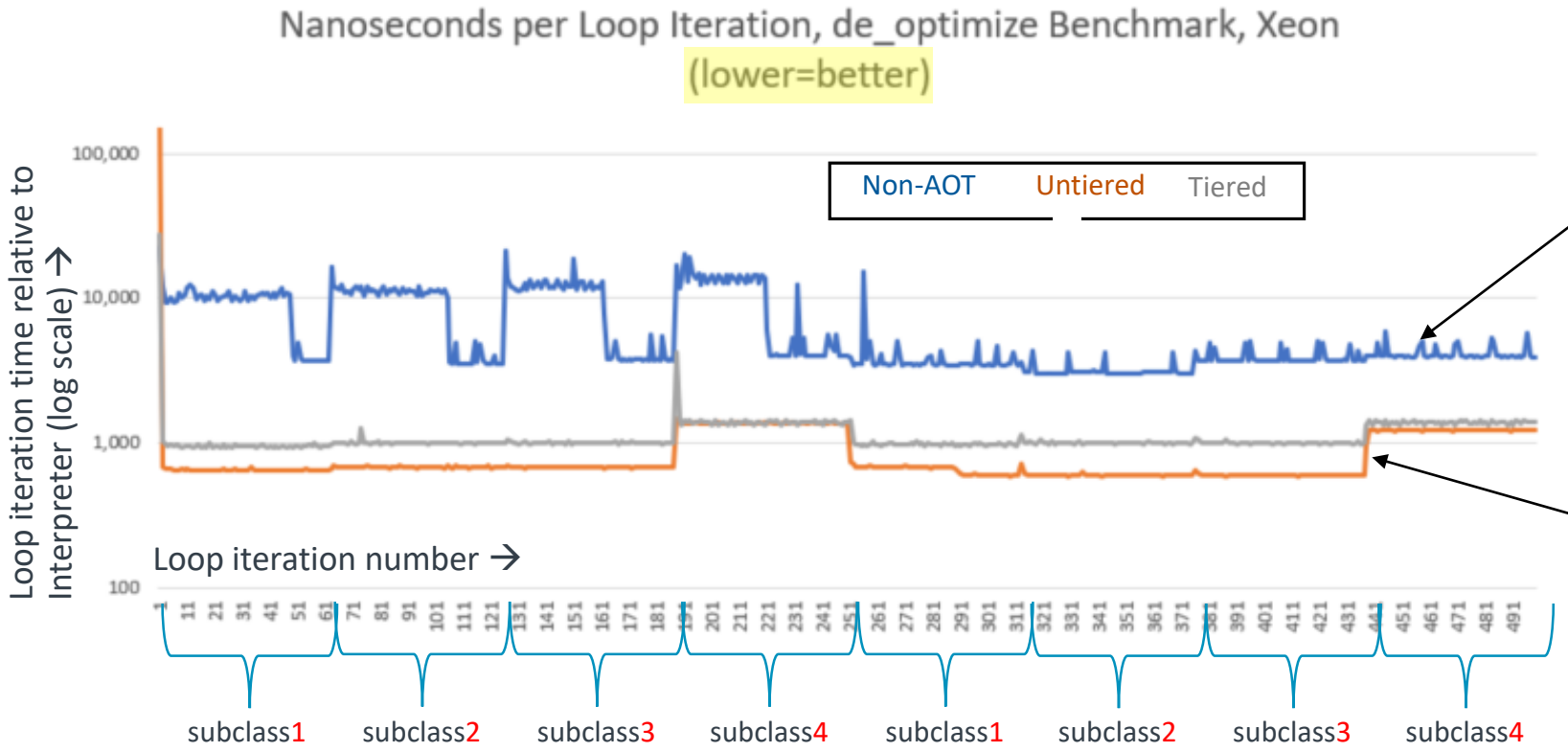
What to Look for in Benchmarks

- Class loads and/or unloads (changing polymorphism), since changes to the class hierarchy forces de-optimization, and in the case of AOT, my understanding is that it should drop back to the version in the `.so`, rather than dropping back to the interpreter as without AOT.
- Something involving long or complexly linked lists, trees, or other linked structures. Those will exercise the de/optimization related to NULL-pointer checks.
- Something floating-point-intensive – perhaps along the lines of transforming a set of coordinates through a projection matrix, or thereabouts.
- Not sure what exactly yet, but something involving recursion – perhaps something the lines of searching a tree.

Copyright © 2019 Arm® Limited

"I5B" `de_optimize` Benchmark

- This benchmark constructs an array containing a run of subclass1 objects, then a run of subclass2 objects, subclass3 (etc.) and then executes a polymorphic method from each.
- *Looking first at **x86 only***, and ***unlike** the other graphs*, showing non-AOT too:



Blue is *without AOT*; can see it running slowly until it optimizes to monomorphic, then deoptimizes back, re-optimizes, etc.

With AOT, not surprisingly, it can't optimize much, but it does get good performance anyway.

Copyright © 2019 Arm® Limited
arm

A64#2 was also run for iteration lengths of the default 240 seconds

- The operation/minute results, with only a few exceptions, were within a few percent of the 15s runs:

Percent difference in operations/minute for the 15s runs compared to that for the 240s runs

A64#2, %Diff. 15s to 240s Runs

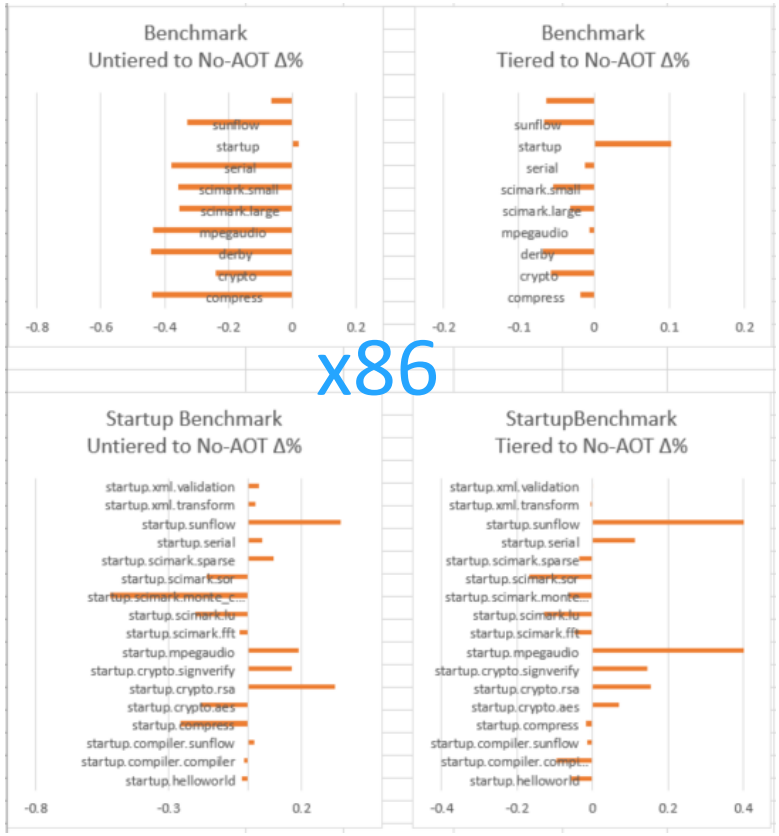
Benchmark	No AOT	Untiered AOT	Tiered AOT
compress	1%	0%	-2%
crypto	-2%	22%	-4%
derby	-1%	-1%	-4%
mpegaudio	0%	0%	0%
scimark.large	-2%	0%	0%
scimark.small	-3%	1%	-4%
serial	-10%	-1%	4%
startup	0%	0%	0%
sunflow	5%	1%	-1%
xml	-11%	-12%	-9%

Startup Benchmark	No AOT	Untiered AOT	Tiered AOT
startup.helloworld	-1%	2%	0%
startup.compiler.compiler	2%	-4%	-3%
startup.compiler.sunflow	3%	-1%	4%
startup.compress	0%	0%	-1%
startup.crypto.aes	16%	0%	0%
startup.crypto.rsa	2%	-1%	-4%
startup.crypto.signverify	-1%	1%	-8%
startup.mpegaudio	6%	1%	0%
startup.scimark.fft	0%	1%	1%
startup.scimark.lu	-4%	-1%	-1%
startup.scimark.monte_carlo	1%	0%	0%
startup.scimark.sor	0%	-1%	1%
startup.scimark.sparse	-2%	1%	-1%
startup.serial	0%	-1%	7%
startup.sunflow	-4%	-1%	1%
startup.xml.transform	1%	2%	-4%
startup.xml.validation	0%	1%	1%

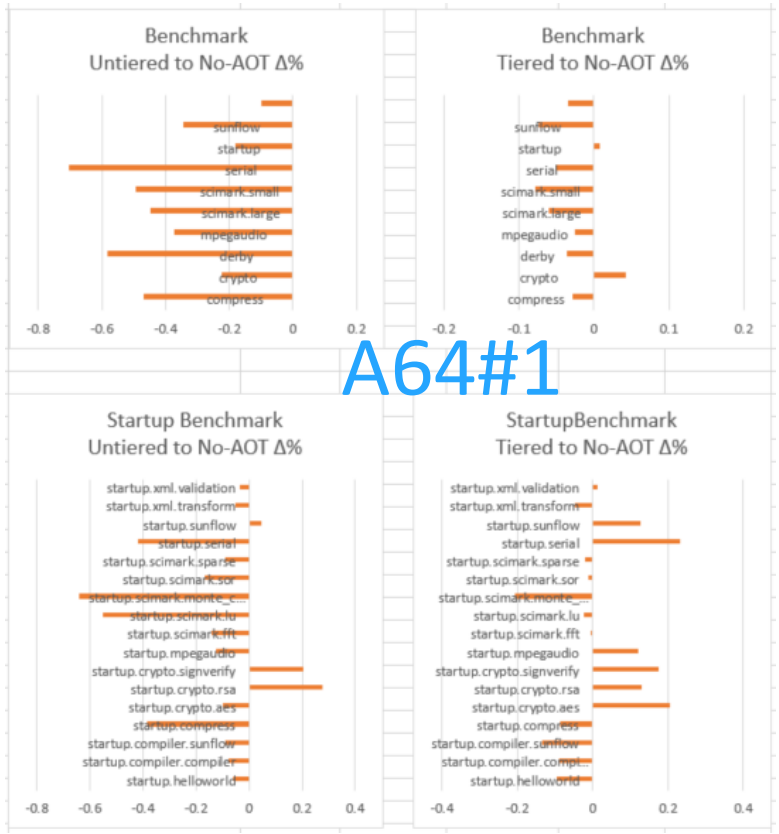
Copyright © 2019 Arm® Limited
arm

Percent timing differences, relative to Non-AOT, are *probably* more related to the benchmark than to micro/architecture

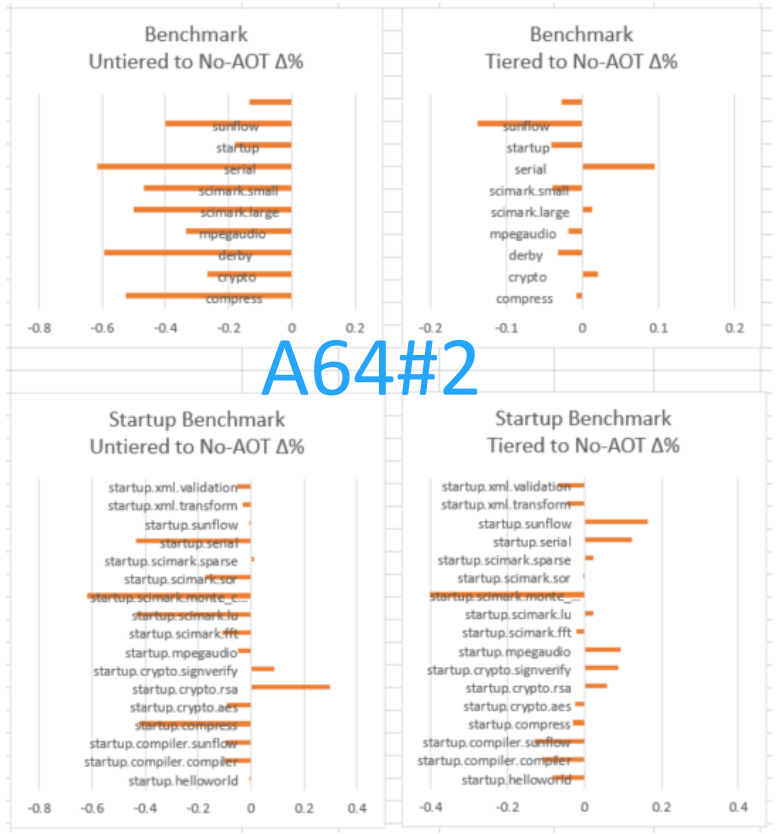
- *This is debatable*, but if true, it means there may be “only so much” to learn from this regarding how to improve our AOT implementation.



x86



A64#1

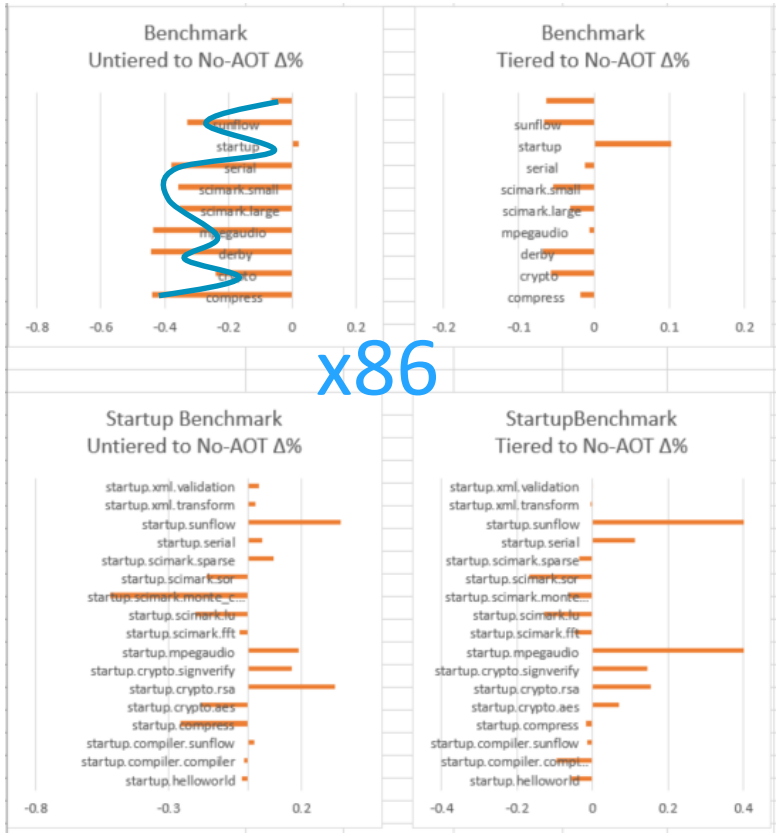


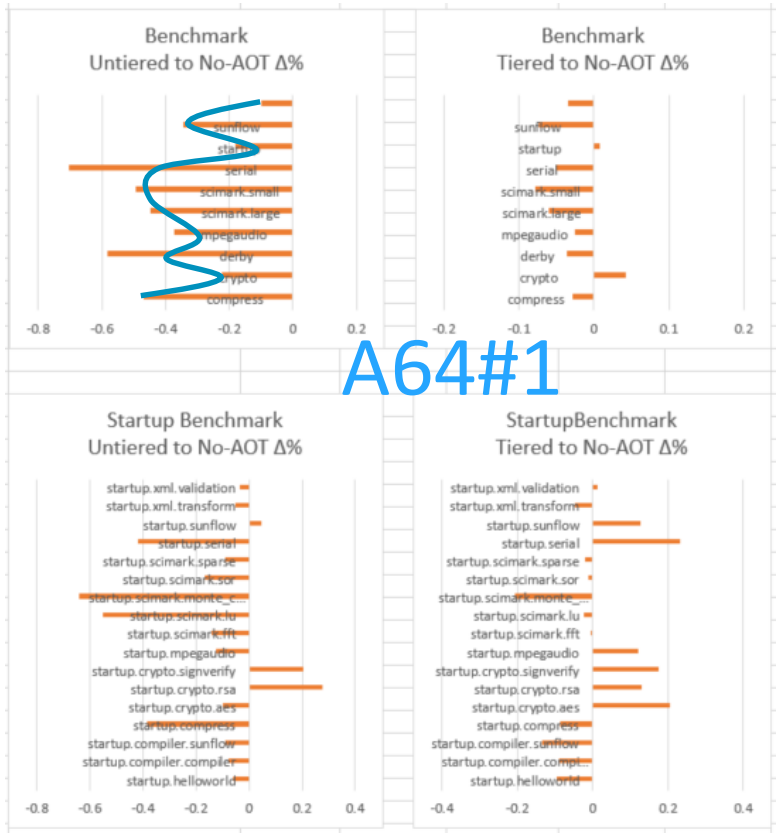
A64#2

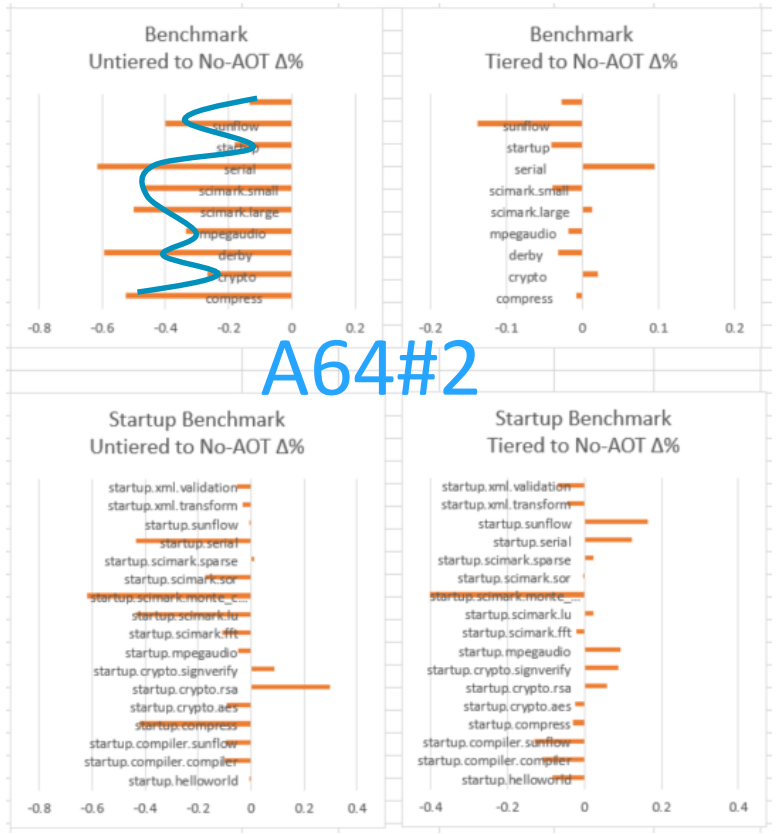
Copyright © 2019 Arm® Limited
arm

Percent timing differences, relative to Non-AOT, are *probably* more related to the benchmark than to micro/architecture

- This is debatable, but if so, this means there may be “only so much” to learn from this regarding how to improve our AOT implementation.



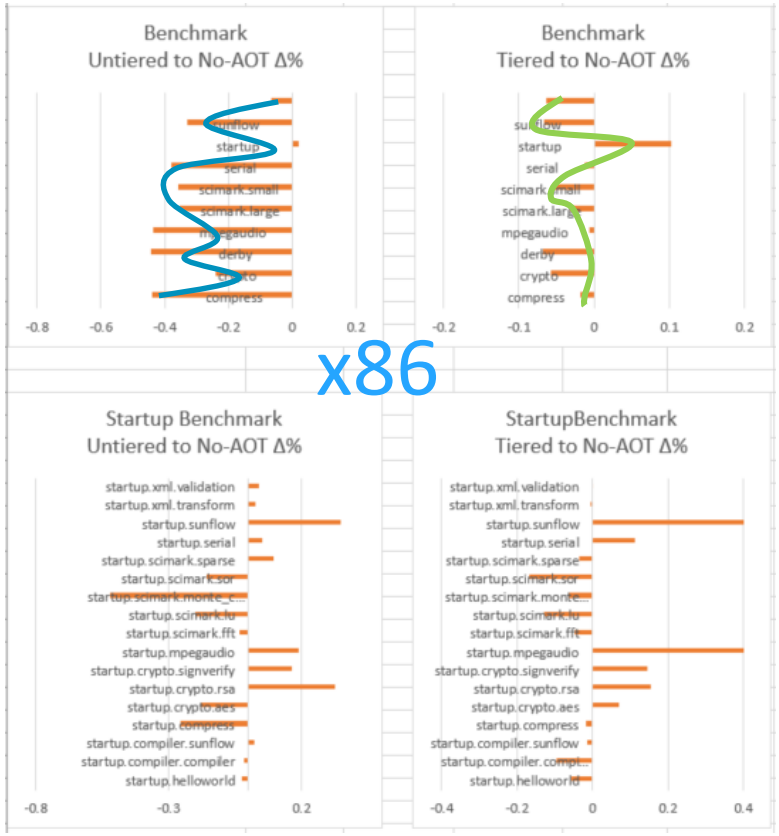




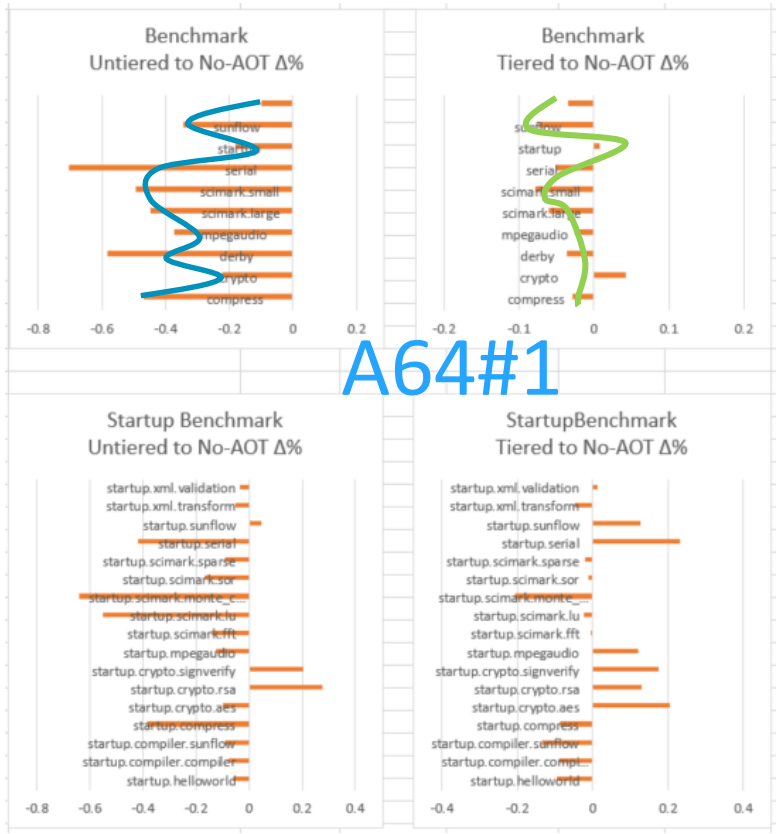
Copyright © 2019 Arm® Limited
arm

Percent timing differences, relative to Non-AOT, are *probably* more related to the benchmark than to micro/architecture

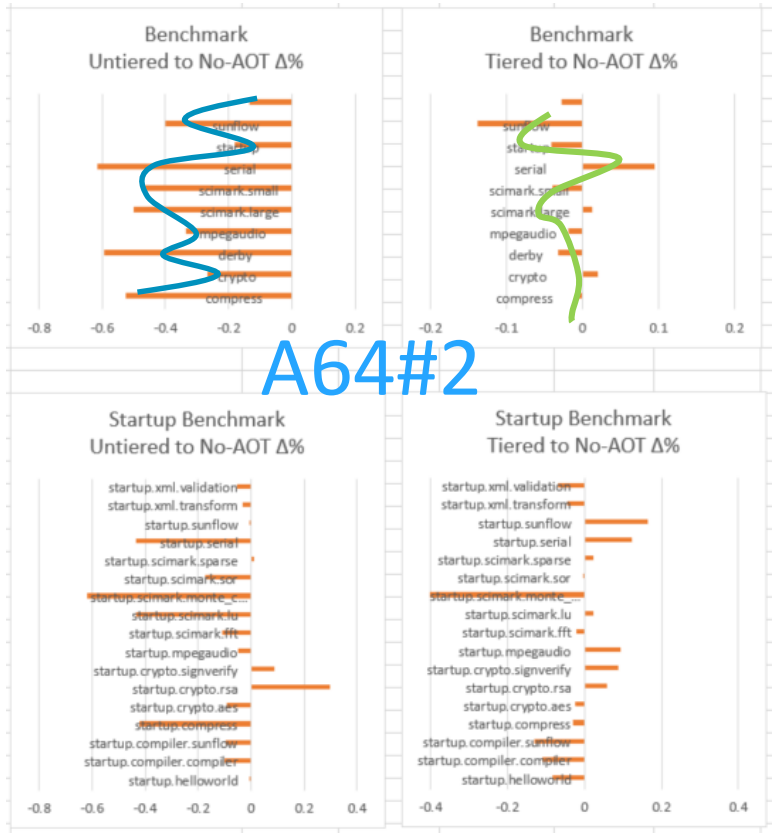
- This is debatable, but if so, this means there may be “only so much” to learn from this regarding how to improve our AOT implementation.



x86



A64#1

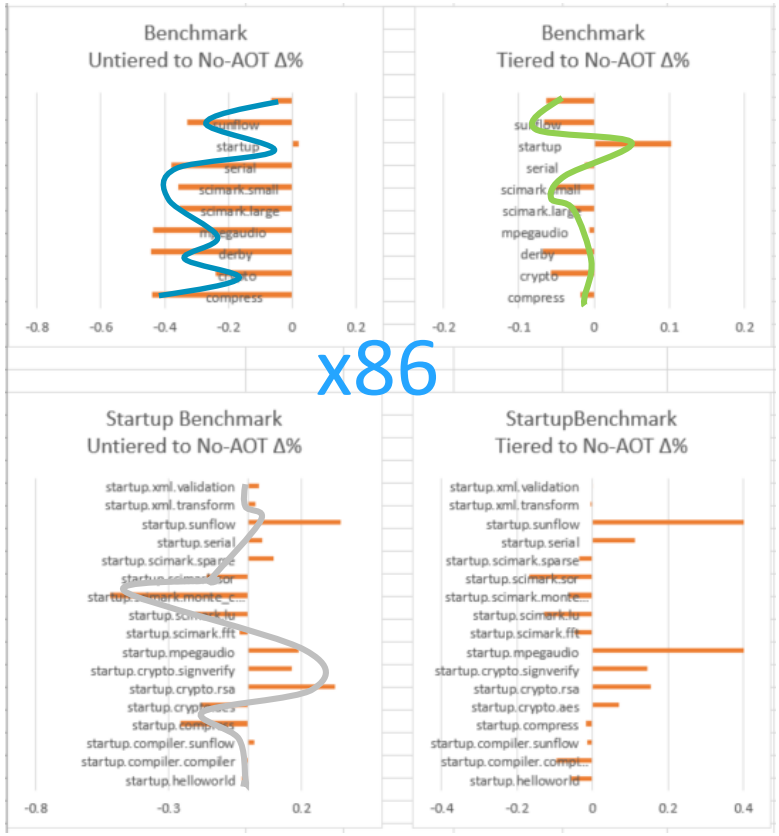


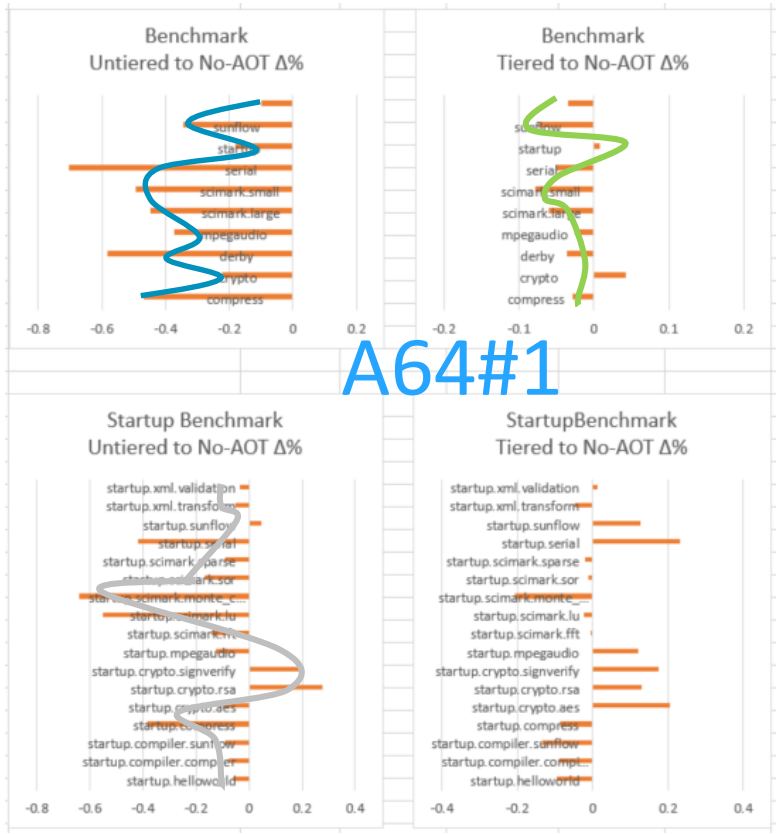
A64#2

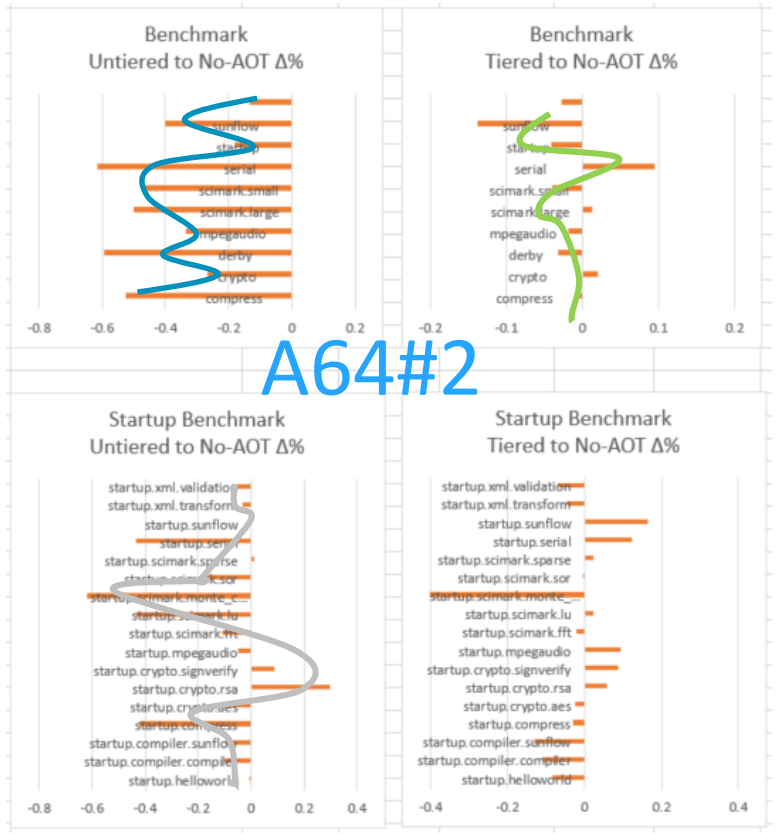
Copyright © 2019 Arm® Limited
arm

Percent timing differences, relative to Non-AOT, are *probably* more related to the benchmark than to micro/architecture

- This is debatable, but if so, this means there may be “only so much” to learn from this regarding how to improve our AOT implementation.



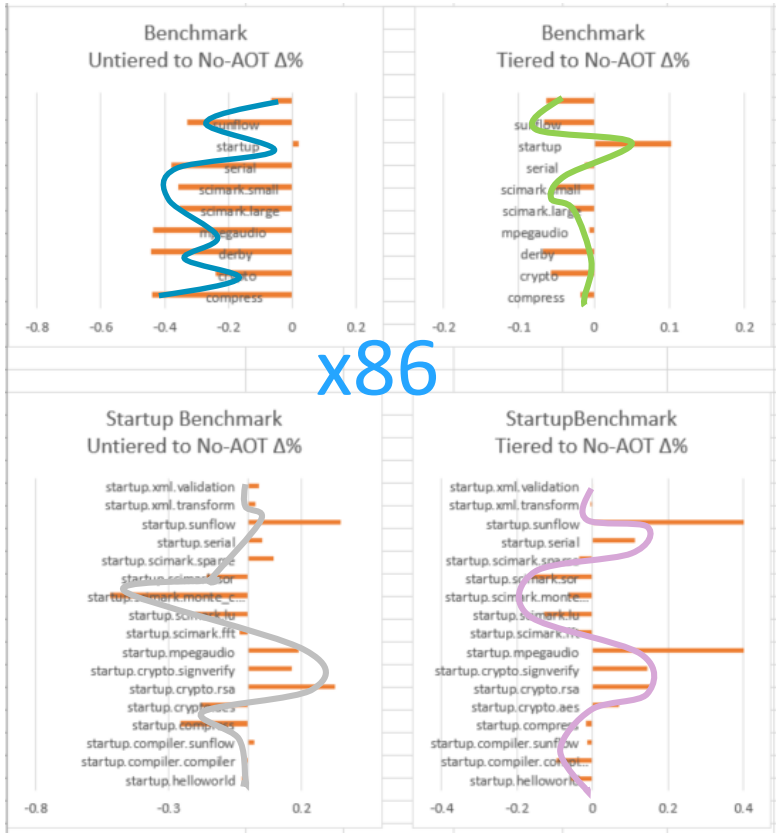


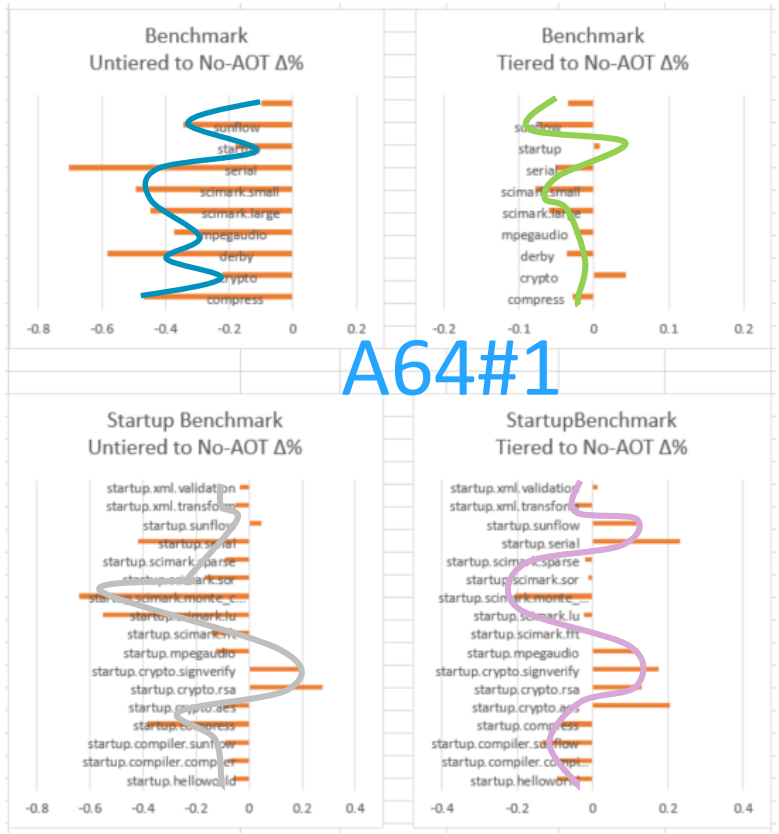


Copyright © 2019 Arm® Limited
arm

Percent timing differences, relative to Non-AOT, are *probably* more related to the benchmark than to micro/architecture

- This is debatable, but if so, this means there may be “only so much” to learn from this regarding how to improve our AOT implementation.





Copyright © 2019 Arm® Limited
arm