Java AOT Baselining

Gary Morrison
Principal Engineer @ Arm®
Open-Source Software, Austin, TX, USA
Linaro Connect, 2019/4/5
Introduction

Java AOT: Wuzzat?
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.
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  
    java -XX:+UseAOT -XX:AOTLibrary=./libHelloWorld.so,./libjava.base.so HelloWorld  
    ```
Prog.java

javac

Prog.class (byte code)

JVM (java command)

Interpreter, tier 0

Profiling

C1 compiler, tier 3

Full Profiling

C2 compiler, tier 4

Normal

AOT, tiered

jaotc --compile-for-tiered

Prog.so (native code)

JVM (java command)

Profiling (invocation, back-edge counters)

C1 compiler, tier 3

Full Profiling

C2 compiler, tier 4

AOT, non-tiered

jaotc

Prog.so (native code)

JVM (java command)

Native code (no profiling, no recompilation)

Thanks to James Yang for drawing this flowchart.
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.
Initial-Five-Benchmark “I5B” Results
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.
“I5B” binarytrees Benchmark

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

![Graph showing performance comparison between different architectures and configurations.](image)

- 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.
“15B” chord Benchmark

• This benchmark doesn’t exercise anything in particular; it’s just “fairly-average code.”

chord Benchmark, AArch64 & x86 AOT Relative to Interpreter
(lower=better)

It’s only for a short time, but there are considerable differences in the details of warm-up, possibly worth studying in more detail.

• 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?).

Again, orange and blue are x86 (orange = tiered).
“l5B” 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!
“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.
“15B” *regexdna* Benchmark

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

*regexdna* Benchmark, AArch64 & x86 AOT Relative to Interpreter

(lower = better)

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

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

These are the untiered versions, x86 fastest.

These are the tiered versions, x86 fastest.
“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.
SPECjvm2008 Results
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.
**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.
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.
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):

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

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

<table>
<thead>
<tr>
<th>Startup Benchmark</th>
<th>No AOT</th>
<th>Untiered AOT</th>
<th>Untiered-to-No Δ%</th>
<th>Tiered AOT</th>
<th>Tiered-to No Δ%</th>
</tr>
</thead>
<tbody>
<tr>
<td>startup.helloworld</td>
<td>134</td>
<td>130</td>
<td>-3%</td>
<td>123</td>
<td>-8%</td>
</tr>
<tr>
<td>startup.compiler.compiler</td>
<td>136</td>
<td>126</td>
<td>-8%</td>
<td>124</td>
<td>-9%</td>
</tr>
<tr>
<td>startup.compiler.sunflow</td>
<td>140</td>
<td>125</td>
<td>-10%</td>
<td>119</td>
<td>-15%</td>
</tr>
<tr>
<td>startup.compress</td>
<td>30</td>
<td>17</td>
<td>-42%</td>
<td>29</td>
<td>-4%</td>
</tr>
<tr>
<td>startup.crypto.aes</td>
<td>11</td>
<td>10</td>
<td>-9%</td>
<td>11</td>
<td>-2%</td>
</tr>
<tr>
<td>startup.crypto.rsa</td>
<td>54</td>
<td>70</td>
<td>29%</td>
<td>58</td>
<td>8%</td>
</tr>
<tr>
<td>startup.crypto.signverify</td>
<td>58</td>
<td>62</td>
<td>7%</td>
<td>67</td>
<td>17%</td>
</tr>
<tr>
<td>startup.mpegaudio</td>
<td>17</td>
<td>16</td>
<td>-5%</td>
<td>18</td>
<td>10%</td>
</tr>
<tr>
<td>startup.scimark.fft</td>
<td>55</td>
<td>49</td>
<td>-11%</td>
<td>54</td>
<td>-3%</td>
</tr>
<tr>
<td>startup.scimark.lu</td>
<td>50</td>
<td>28</td>
<td>-43%</td>
<td>51</td>
<td>3%</td>
</tr>
<tr>
<td>startup.scimark.monte_carlo</td>
<td>30</td>
<td>11</td>
<td>-62%</td>
<td>17</td>
<td>-43%</td>
</tr>
<tr>
<td>startup.scimark.sor</td>
<td>23</td>
<td>19</td>
<td>-17%</td>
<td>23</td>
<td>-1%</td>
</tr>
<tr>
<td>startup.scimark.sparse</td>
<td>23</td>
<td>23</td>
<td>1%</td>
<td>24</td>
<td>4%</td>
</tr>
<tr>
<td>startup.serial</td>
<td>12</td>
<td>7</td>
<td>-43%</td>
<td>13</td>
<td>8%</td>
</tr>
<tr>
<td>startup.sunflow</td>
<td>20</td>
<td>20</td>
<td>1%</td>
<td>23</td>
<td>15%</td>
</tr>
<tr>
<td>startup.xml.transform</td>
<td>1</td>
<td>1</td>
<td>-3%</td>
<td>1</td>
<td>-3%</td>
</tr>
<tr>
<td>startup.xml.validation</td>
<td>18</td>
<td>18</td>
<td>-1%</td>
<td>18</td>
<td>-4%</td>
</tr>
</tbody>
</table>
• 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.
**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.
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.
Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
תודה
Backup...
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.
"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:

  ![Graph](image)

  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.
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:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Percent difference in operations/minute for the 15s runs compared to that for the 240s runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>1%</td>
</tr>
<tr>
<td>crypto</td>
<td>-2%</td>
</tr>
<tr>
<td>derby</td>
<td>-1%</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>0%</td>
</tr>
<tr>
<td>scimark.large</td>
<td>-2%</td>
</tr>
<tr>
<td>scimark.small</td>
<td>-3%</td>
</tr>
<tr>
<td>serial</td>
<td>-10%</td>
</tr>
<tr>
<td>startup</td>
<td>0%</td>
</tr>
<tr>
<td>sunflow</td>
<td>5%</td>
</tr>
<tr>
<td>xml</td>
<td>-11%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>No AOT</th>
<th>AOT</th>
<th>Tiered AOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>startup.helloworld</td>
<td>-1%</td>
<td>2%</td>
<td>0%</td>
</tr>
<tr>
<td>startup.compiler.compiler</td>
<td>2%</td>
<td>-4%</td>
<td>-3%</td>
</tr>
<tr>
<td>startup.compiler.sunflow</td>
<td>3%</td>
<td>-1%</td>
<td>4%</td>
</tr>
<tr>
<td>startup.compress</td>
<td>0%</td>
<td>0%</td>
<td>-1%</td>
</tr>
<tr>
<td>startup.crypto.aes</td>
<td>16%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>startup.crypto.rsa</td>
<td>2%</td>
<td>-1%</td>
<td>-4%</td>
</tr>
<tr>
<td>startup.crypto.signverify</td>
<td>-1%</td>
<td>1%</td>
<td>-8%</td>
</tr>
<tr>
<td>startup.mpegaudio</td>
<td>6%</td>
<td>1%</td>
<td>0%</td>
</tr>
<tr>
<td>startup.scimark.fft</td>
<td>0%</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>startup.scimark.lu</td>
<td>-4%</td>
<td>-1%</td>
<td>-1%</td>
</tr>
<tr>
<td>startup.scimark.monte_carlo</td>
<td>1%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>startup.scimark.sor</td>
<td>0%</td>
<td>-1%</td>
<td>1%</td>
</tr>
<tr>
<td>startup.scimark.sparse</td>
<td>-2%</td>
<td>1%</td>
<td>-1%</td>
</tr>
<tr>
<td>startup.serial</td>
<td>0%</td>
<td>-1%</td>
<td>7%</td>
</tr>
<tr>
<td>startup.sunflow</td>
<td>-4%</td>
<td>-1%</td>
<td>1%</td>
</tr>
<tr>
<td>startup.xml.transform</td>
<td>1%</td>
<td>2%</td>
<td>-4%</td>
</tr>
<tr>
<td>startup.xml.validation</td>
<td>0%</td>
<td>1%</td>
<td>1%</td>
</tr>
</tbody>
</table>
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.
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.
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.
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.
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.