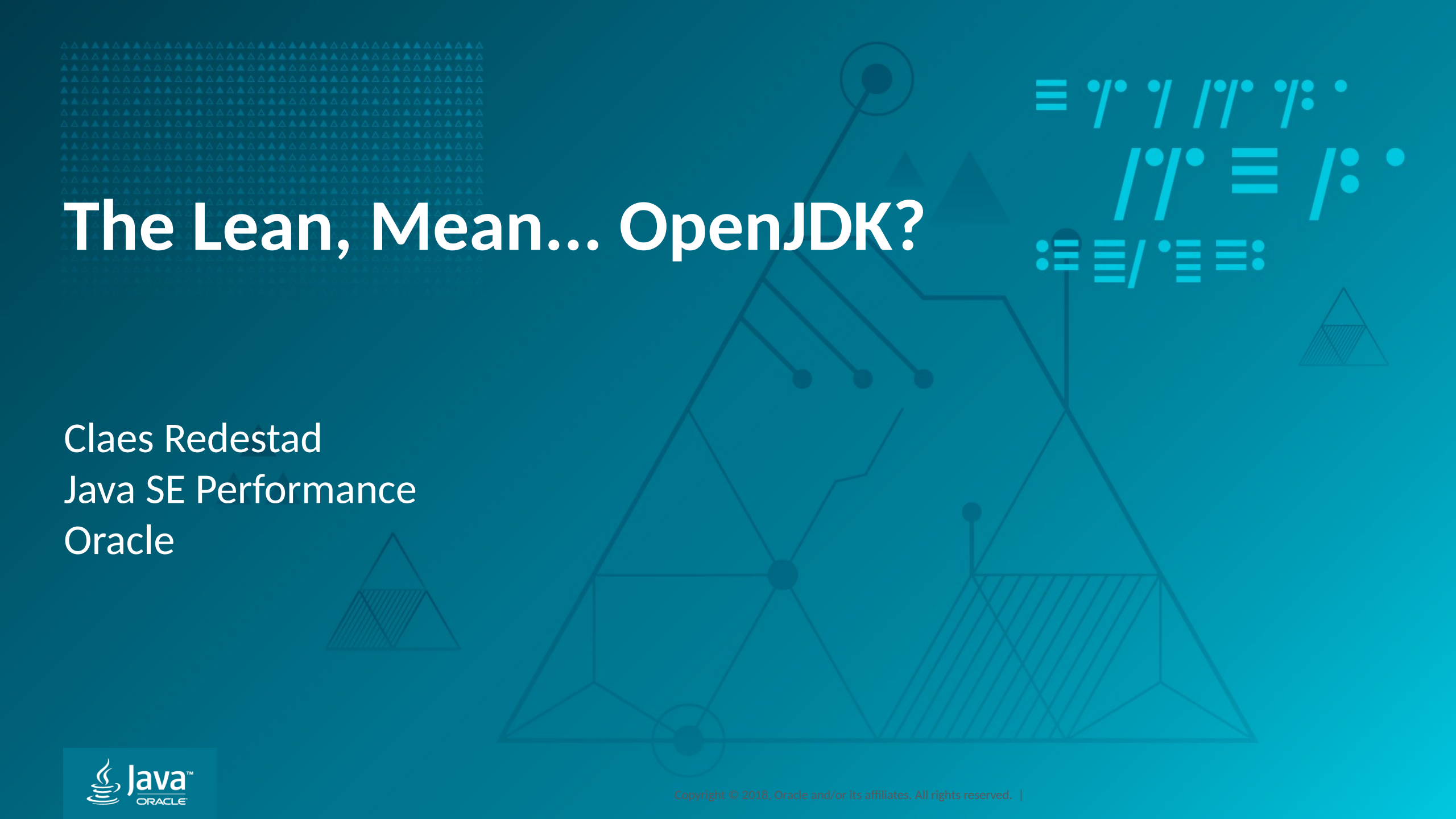

ORACLE®


Java™
ORACLE®

The Lean, Mean… OpenJDK?

Claes Redestad
Java SE Performance
Oracle



Copyright © 2018, Oracle and/or its affiliates. All rights reserved. |

Safe Harbor Statement

The following is intended to outline our general product direction. 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. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

The Lean, Mean… OpenJDK?

Who am I?

- Performance engineer at Oracle since 2012
- OpenJDK: redestad
- Blog: https://cl4es.github.io
- Twitter: @cl4es

Java™ ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

What is OpenJDK?

- The OpenJDK project started in 2006 as an open sourcing effort of the Sun JDK
- OpenJDK has been the basis of all Sun/Oracle proprietary JDK distributions since then
- Starting with JDK 11, OpenJDK and the proprietary Oracle JDK have fully converged: proprietary and/or commercial features that were only in the Oracle JDK are now *freely* available and part of the OpenJDK

Java™ ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

Did you say free?

Yes! Oracle provides OpenJDK builds for *free* here:

https://jdk.java.net/

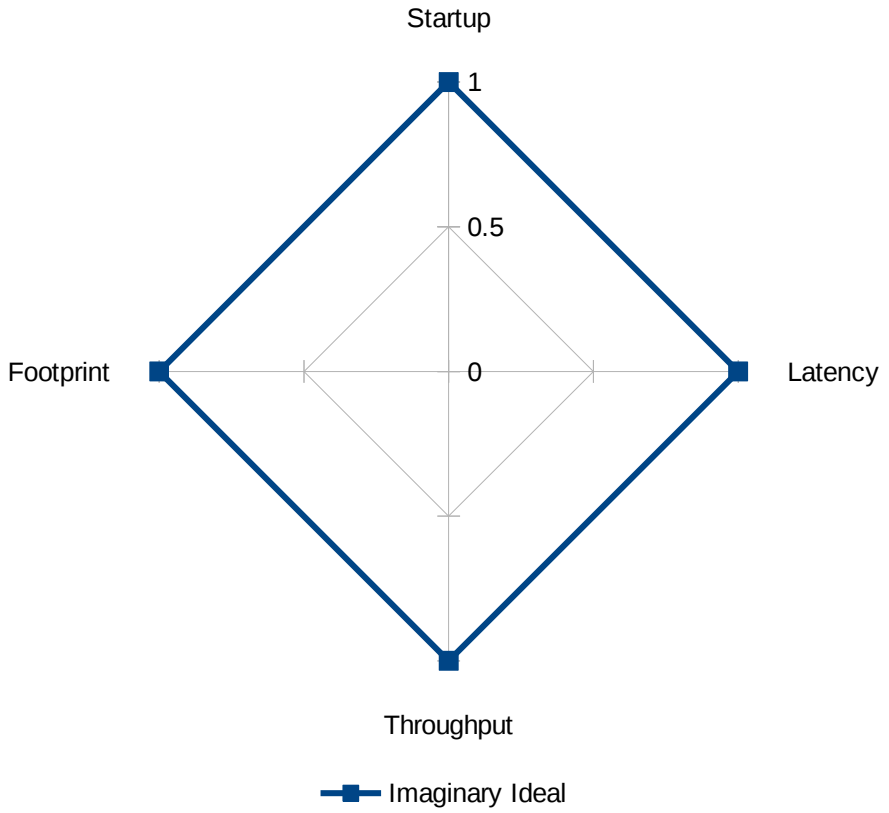
The latest release will continue to be free and unrestricted

https://blogs.oracle.com/java-platform-group/

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

What is performance?

- Throughput
  - The total amount of work a system can do in some given time
- Latency
  - The time it takes to do some unit of work
- Footprint
  - Memory and storage requirements
- Startup
  - The time and resources needed to get ready for action



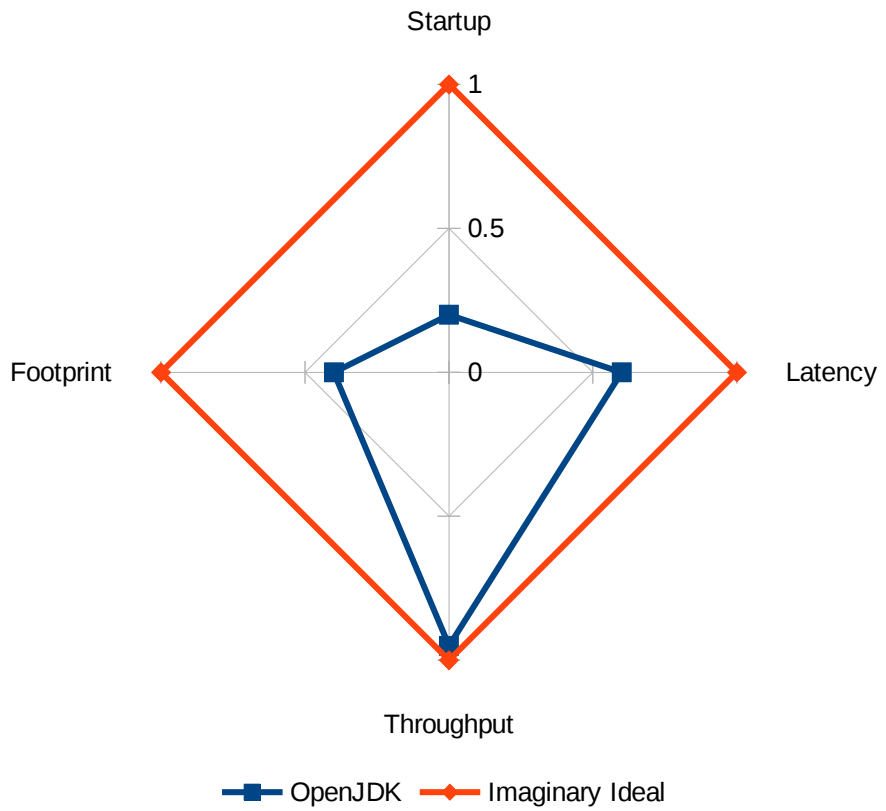
Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

Trade-offs and the ergonomic JDK

- Out-of-the-box we ergonomically seek to strike a good balance between all performance concerns

- Historically the JDK has favored peak throughput
  - Some industry shift towards favoring low latency, especially as workloads scale up
  - To some extent tuning allow users to choose different trade-offs

… goal to make (most) tuning unnecessary



Source: Data and qualified guesses

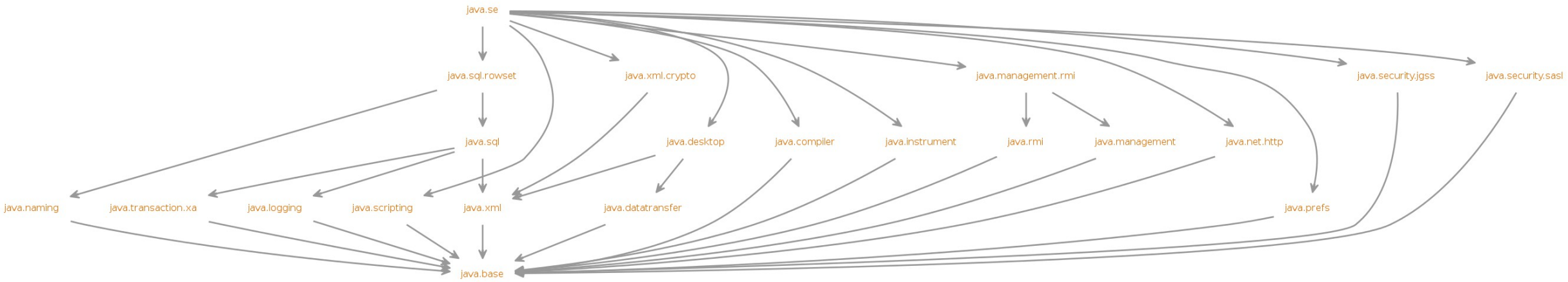
Java™ ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

Where to start?

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

The modular JDK



JDK 9 *modularized* the JDK

Modules enable better control for developers to encapsulate internals

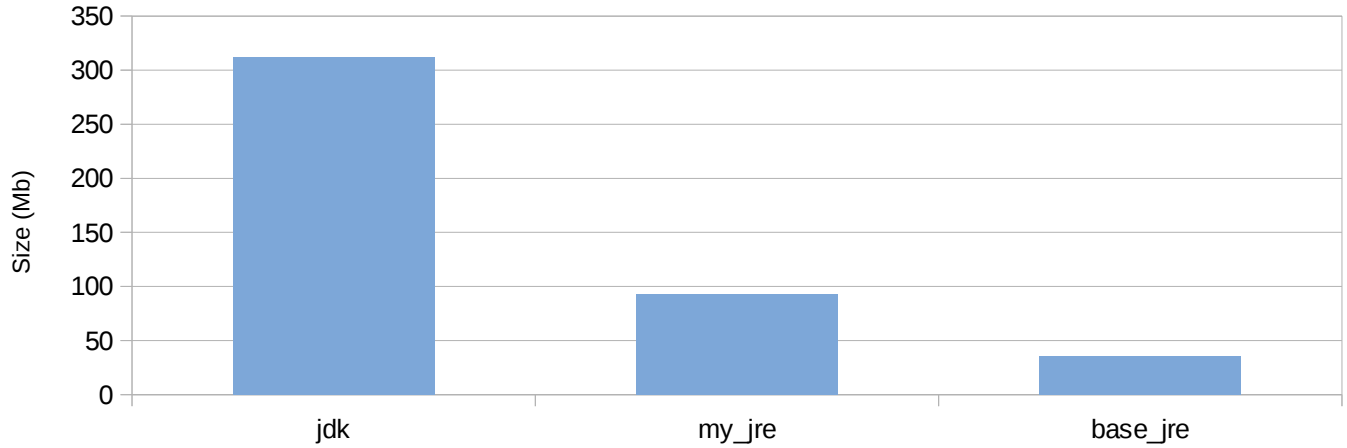
Consolidate embedded JDKs projects into the mainline

Java™ ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

The modular JDK allows us to scale down…

jlink can be used to build custom JRE images from a subset of JDK modules, down to the bare minimum



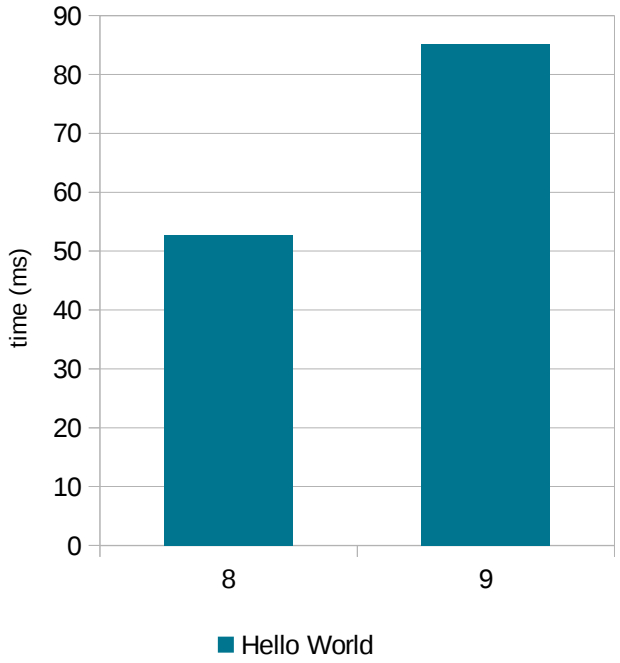

```
$ bin/jlink --add-modules java.se,jdk.jfr --module-path jmods --output my_jre

$ bin/jlink --compress=2 --add-modules java.base --module-path jmods --output base_jre
```

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

Hello World(s)!

- Out of the box, the module system caused some bootstrap regressions in JDK 9
- Especially running on a JRE with all JDK modules

```
Hello World:

 System.out.println("Hello World!");
```

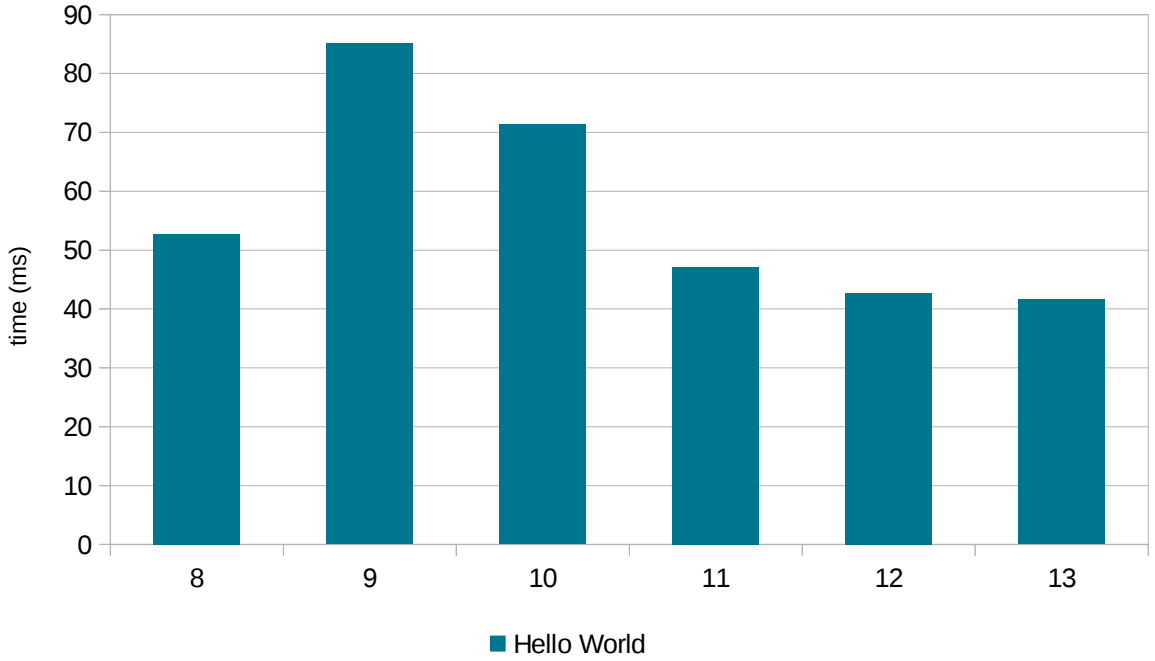


Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

Hello World(s)!

- We fixed many of those regressions…
- *… and kept on fixing*
  - 120+ startup-related enhancements resolved in JDK 10 through JDK 13

```
Hello World:

 System.out.println("Hello World!");
```

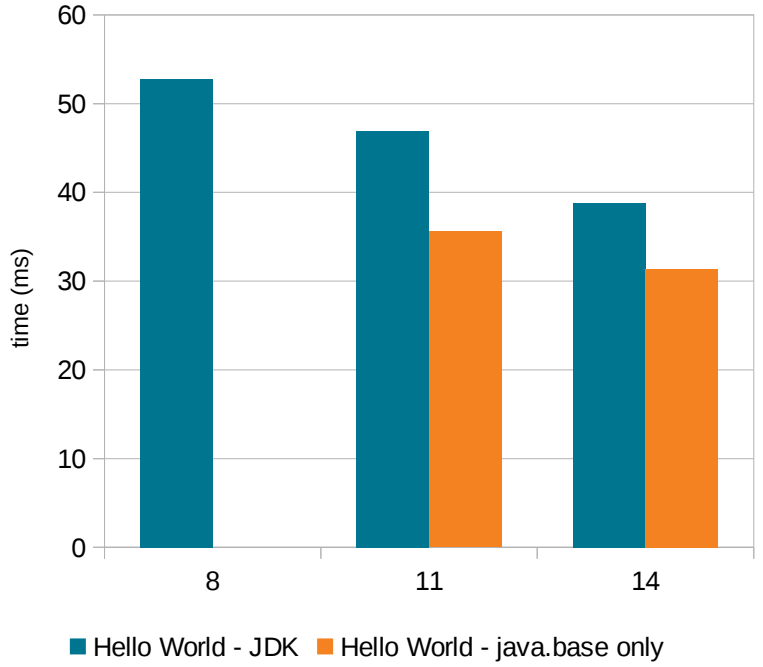


Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

Hello World(s)!

- **Bonus:** slightly better when leaving out unneeded modules

```
Hello World:

 System.out.println("Hello World!");
```

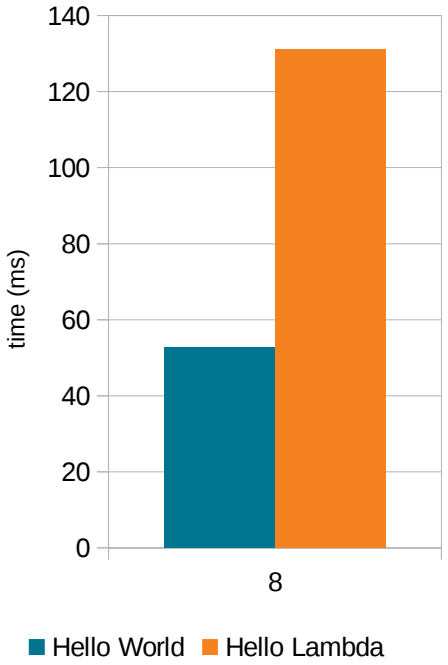


Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

Bootstraps, all the way up!

- In JDK 8, bootstrapping the first lambda expression took longer time than starting up the entire JVM(!)
- Early prototypes of the module system saw use of lambdas during bootstrap
- It seemed prudent to deal with this to avoid even larger *regressions*

```
Hello Lambda:

  Consumer<String> println =
    System.out::println;
  println.accept("Hello World!");
```

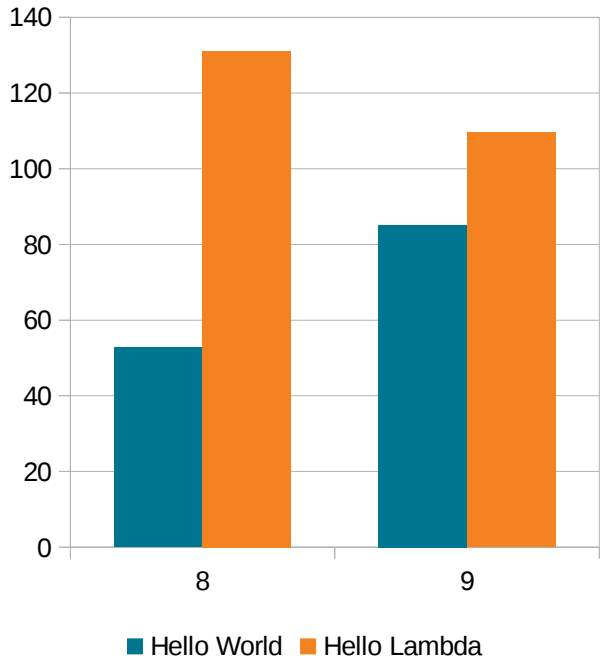


Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

Bootstraps, all the way up!

- For JDK 9, we spent some time cleaning things up
  - Just removing a few unnecessary things and making various things initialize more lazily got us quite far
- With the new **jlink** tool in the works we have a new means to move work from runtime to link time
  - A **jlink** plugin to generate some commonly used classes cut the overhead of lambda bootstrap roughly in half

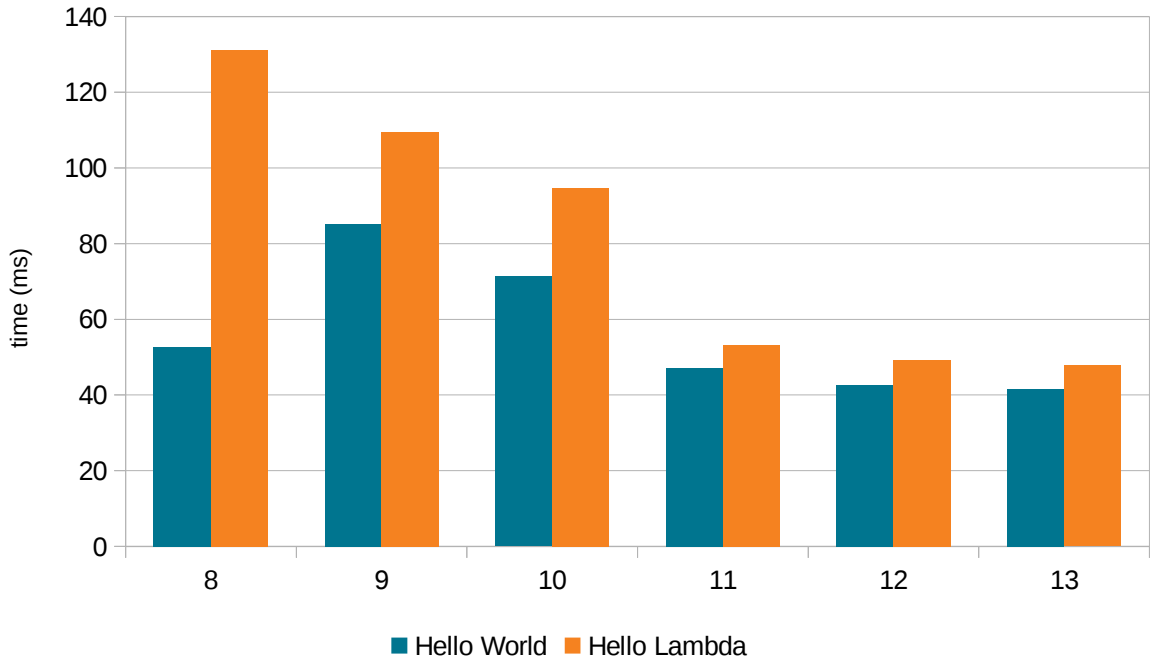
```
Hello Lambda:

  Consumer<String> println =
    System.out::println;
  println.accept("Hello World!");
```



Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

… and all the way down…

- In JDK 11 we got rid of most of the one-off overheads
- "Hello Lambda" now faster than "Hello World" was on JDK 8



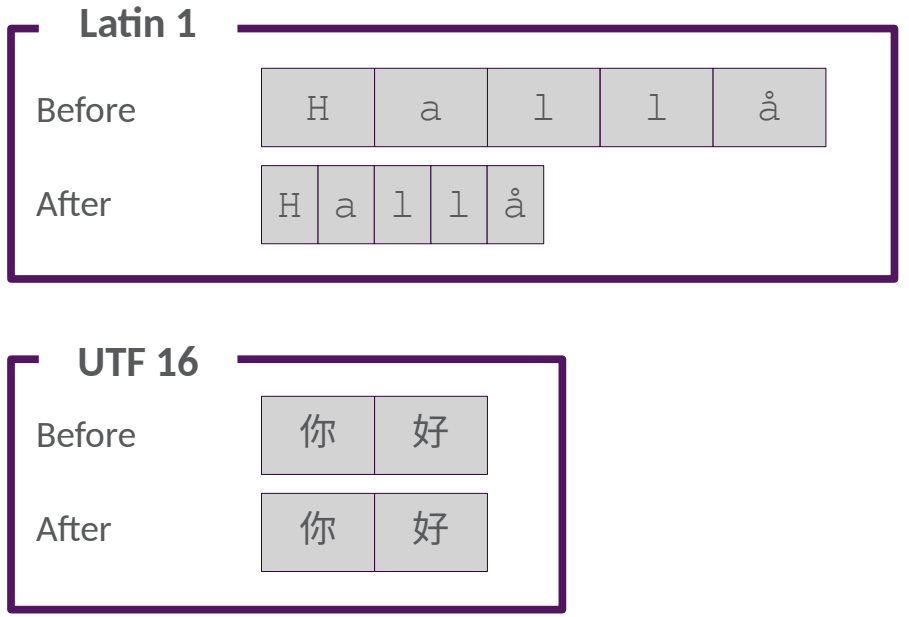
Great success! But lambdas aren't the only thing that might require expensive bootstrapping…

JDK-8198418

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

But first… Compact Strings!

- Enable denser storage of Strings
  - Internal storage changed from `char[]` to `byte[]`
  - Any string that can be encoded using Latin 1 will use one byte per character instead of two
  - Other strings will encode their chars into two bytes as before
- Obvious footprint wins
  - Most applications have a significant number of Latin 1 encodable strings
- Surprising(?) throughput improvements



https://openjdk.java.net/jeps/254

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

Indified String concatenation

- JEP 280 introduced indified String concatenation, **ISC**
  - Use **dynamic** bootstrapping of String concatenation expressions
- Large throughput and latency wins
- "Most optimal ISC strategies do 2.9x better, and 6.4x less garbage"
  - Aleksey Shipilëv, JFokus 2016

```
@Param("4711")
public int intValue;

@Benchmark
public String concat() {
  return "string" + intValue +
         "string" + intValue;
}

           time        alloc
-----------------------------
JDK 8      44.0ns/op   80B/op
JDK 13     24.5ns/op   64B/op
```

https://openjdk.java.net/jeps/280

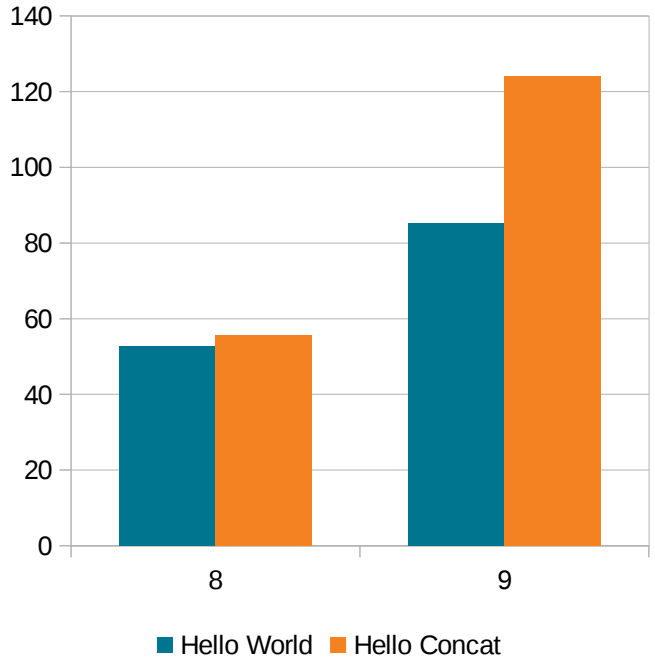
Java™ ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

String Concat Redux

- ISC is cause for some bootstrap overheads of string concatenation expressions in JDK 9
- Some work done before JDK 9 release to lessen the startup blow
  - The **jlink** plugin that helped lambda bootstrapping plays a large role here

```
Hello Concat:

  String foo = ...
  System.out.println("Hello 1: " + foo);
  System.out.println("Hello 2: " + foo);
  ...
  System.out.println("Hello 10: " + foo);
```

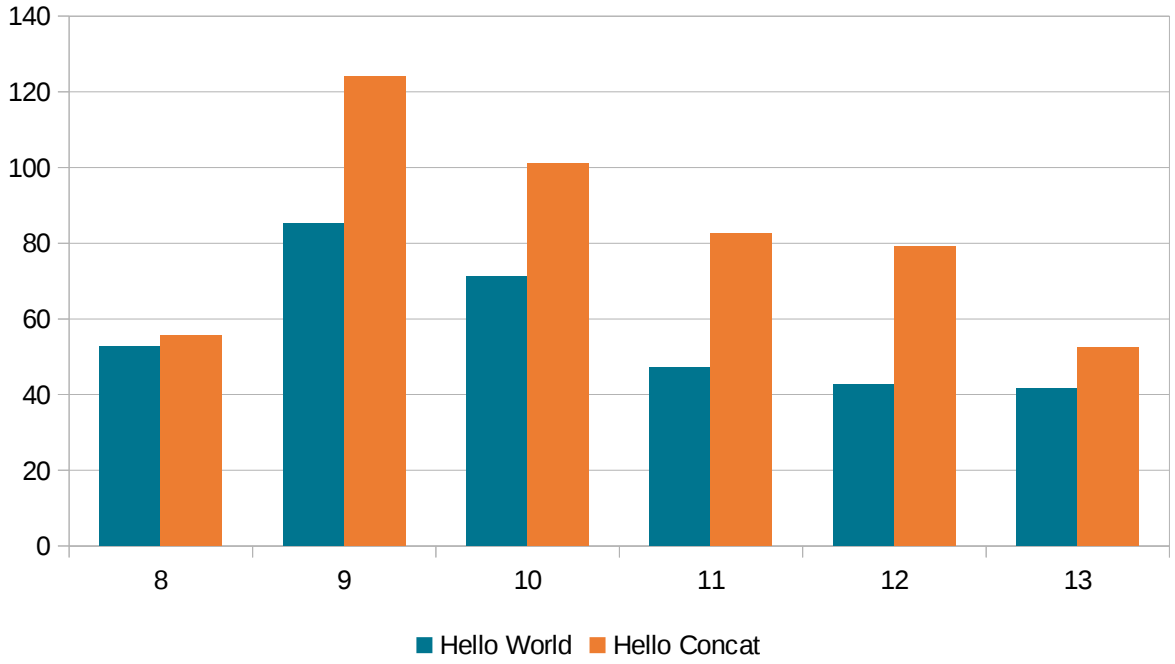


Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

String Concat Redux

- There were some improvements in JDK 10 through 12 for specific cases
- Speeding up bootstrapping of ISC expressions in general proved harder than expected
- Not until JDK 13 did we manage to cut down the overheads more generally

But we now have a robust framework for building more of these ***dynamic*** and ***performant*** things into Java (and other JVM languages) while only paying a small price for them up front



https://cl4es.github.io/2019/05/14/String-Concat-Redux.html

Java™ ORACLE

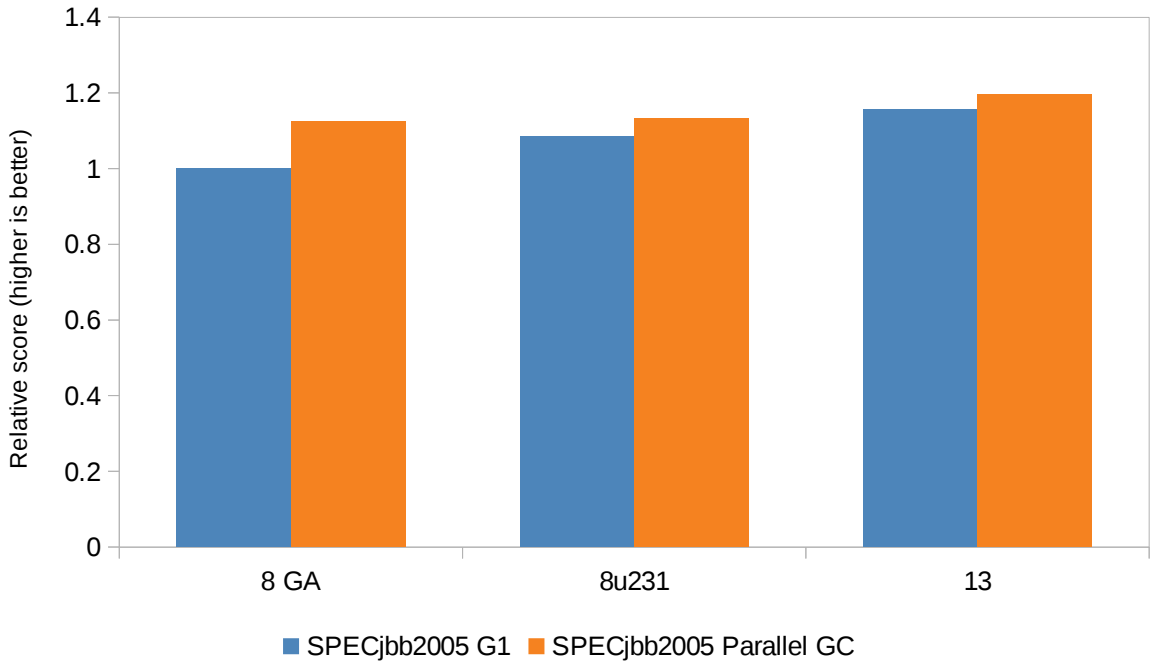
Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

And now for something completely different…

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

The G1 garbage collector saw a lot of improvements

- G1 was still lagging behind ParallelGC
  - At least on throughput-oriented benchmarks!
- Over the course of 8 updates and more recent releases, the gap has been shrinking
- Just being a few percent behind on *throughput* is great for a GC that is meant to optimize more for *latency!*



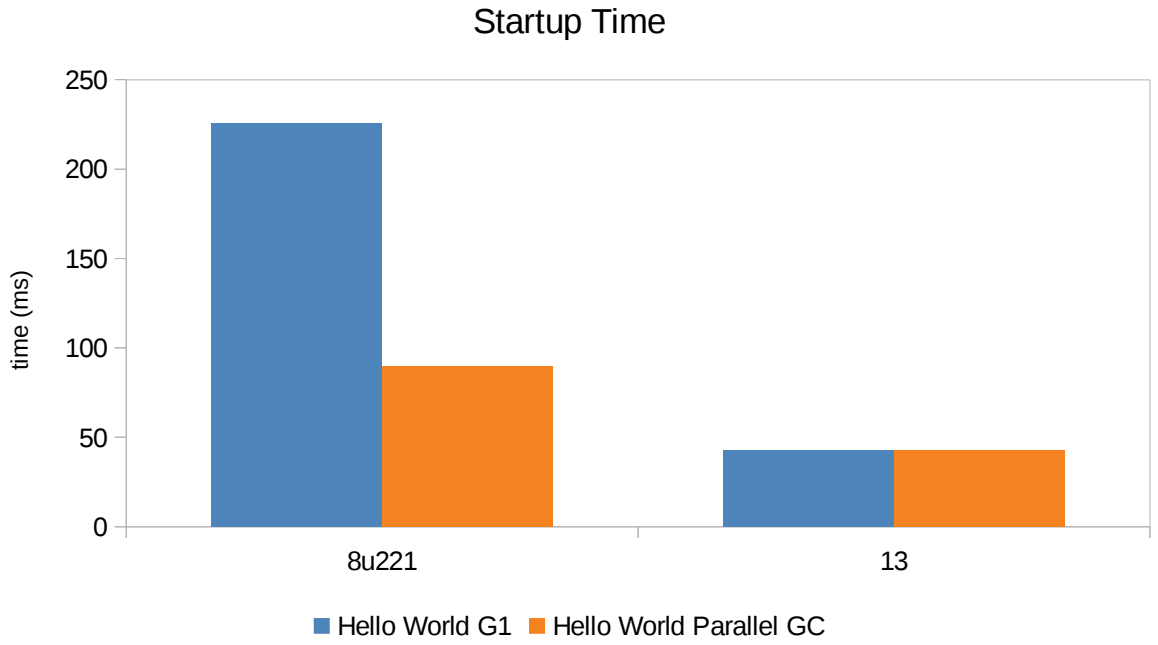
SPECjbb® 2005 is a registered trademark of the Standard Performance Evaluation Corporation (spec.org). The actual results are not represented as compliant because the SUT may not meet SPEC's requirements for general availability.

Java™ ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

The G1 garbage collector saw a lot of improvements

- A great number of issues affecting startup time in G1 was addressed
  - Getting parity on out-of-the-box with simpler GCs
  - Cutting minutes off of startup in extremer cases (> 1TB heap)



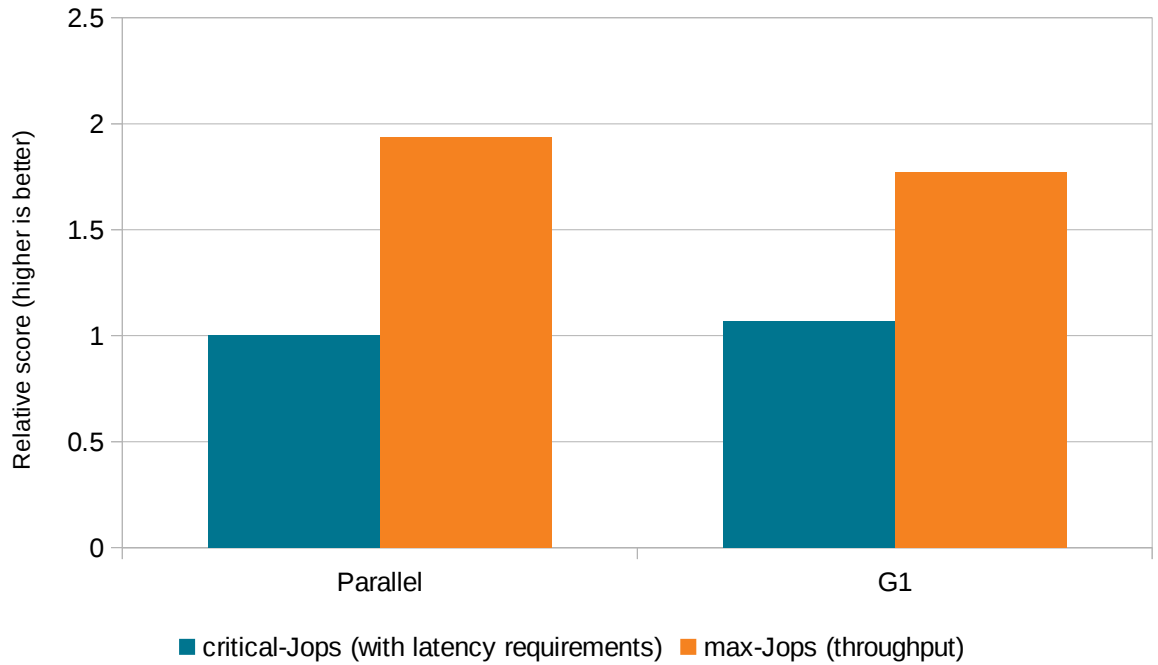
The numbers shown may somewhat exaggerate the *startup* overhead when running G1 on 8, since some of it was related to an issue with unnecessary delays when shutting down the JVM

Java™ ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

… and was made default in JDK 9

- Throughput penalties around 3-10% are common
- More extreme corner cases exist
- Still a good trade-off
  - Trade some raw throughput to reduce risk of really long pauses



Mode: Composite **Heap Size:** 128G **OS:** Oracle Linux 7.5
HW: Intel Xeon E5-2690 2.9GHz 2 sockets, 16 cores (32 hw-threads)

SPECjbb® 2015 is a registered trademark of the Standard Performance Evaluation Corporation (spec.org). The actual results are not represented as compliant because the SUT may not meet SPEC's requirements for general availability.

https://openjdk.java.net/jeps/248

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

The Z Garbage Collector

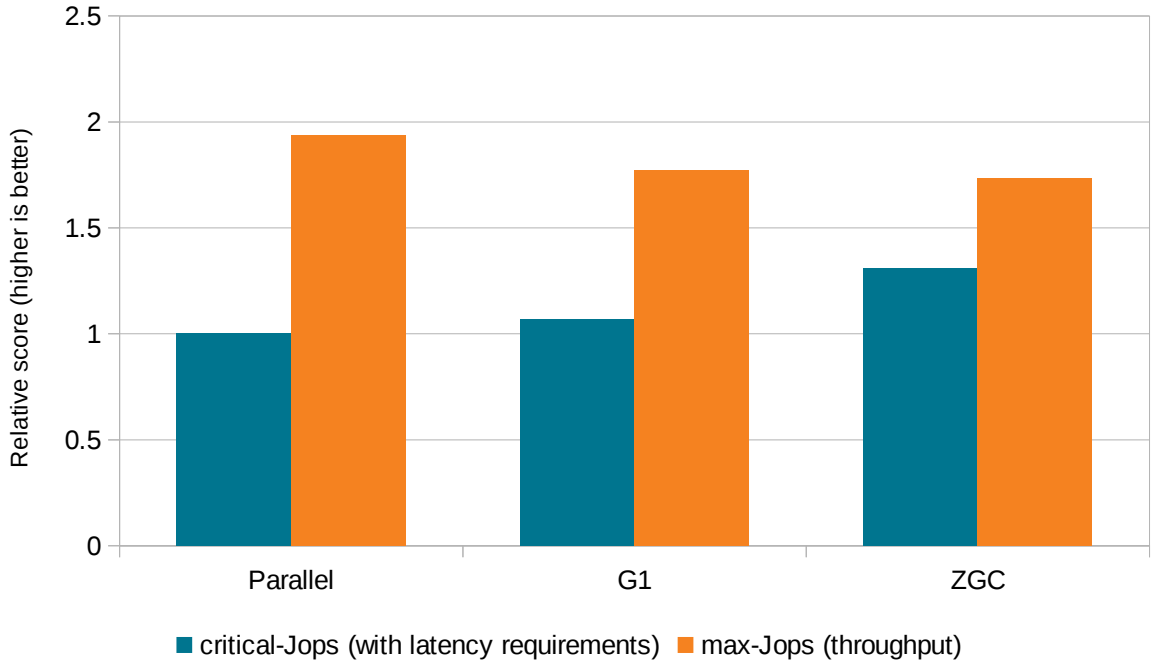
- Scalable, concurrent low-latency GC
- Pause times **below** 10ms - often below **1** ms
- Scale from hundreds of Mb to **16*** terabytes (* from JDK 13)
- Experimental: `-XX:+UnlockExperimentalVMOptions -XX:+UseZGC`
- Goal is to complement the other GCs:
  - **Parallel GC** optimizes for *throughput*
  - **G1** strives for a balance between *throughput* and *low pause* times
  - **ZGC** spares no expense to attain as *low pause times* as possible

https://wiki.openjdk.java.net/display/zgc

Java™ ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

The Z Garbage Collector

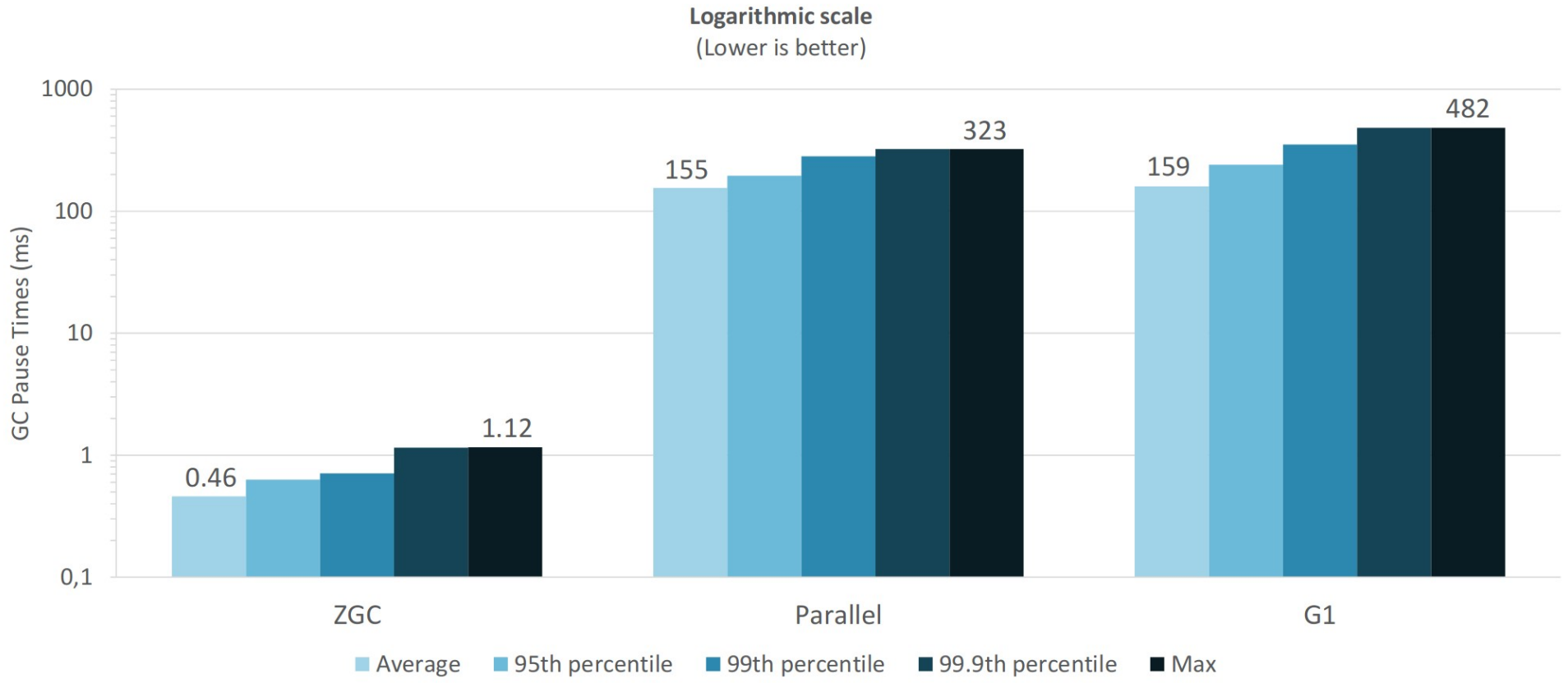


Mode: Composite **Heap Size:** 128G **OS:** Oracle Linux 7.5
HW: Intel Xeon E5-2690 2.9GHz 2 sockets, 16 cores (32 hw-threads)

SPECjbb® 2015 is a registered trademark of the Standard Performance Evaluation Corporation (spec.org). The actual results are not represented as compliant because the SUT may not meet SPEC's requirements for general availability.

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

The Z Garbage Collector



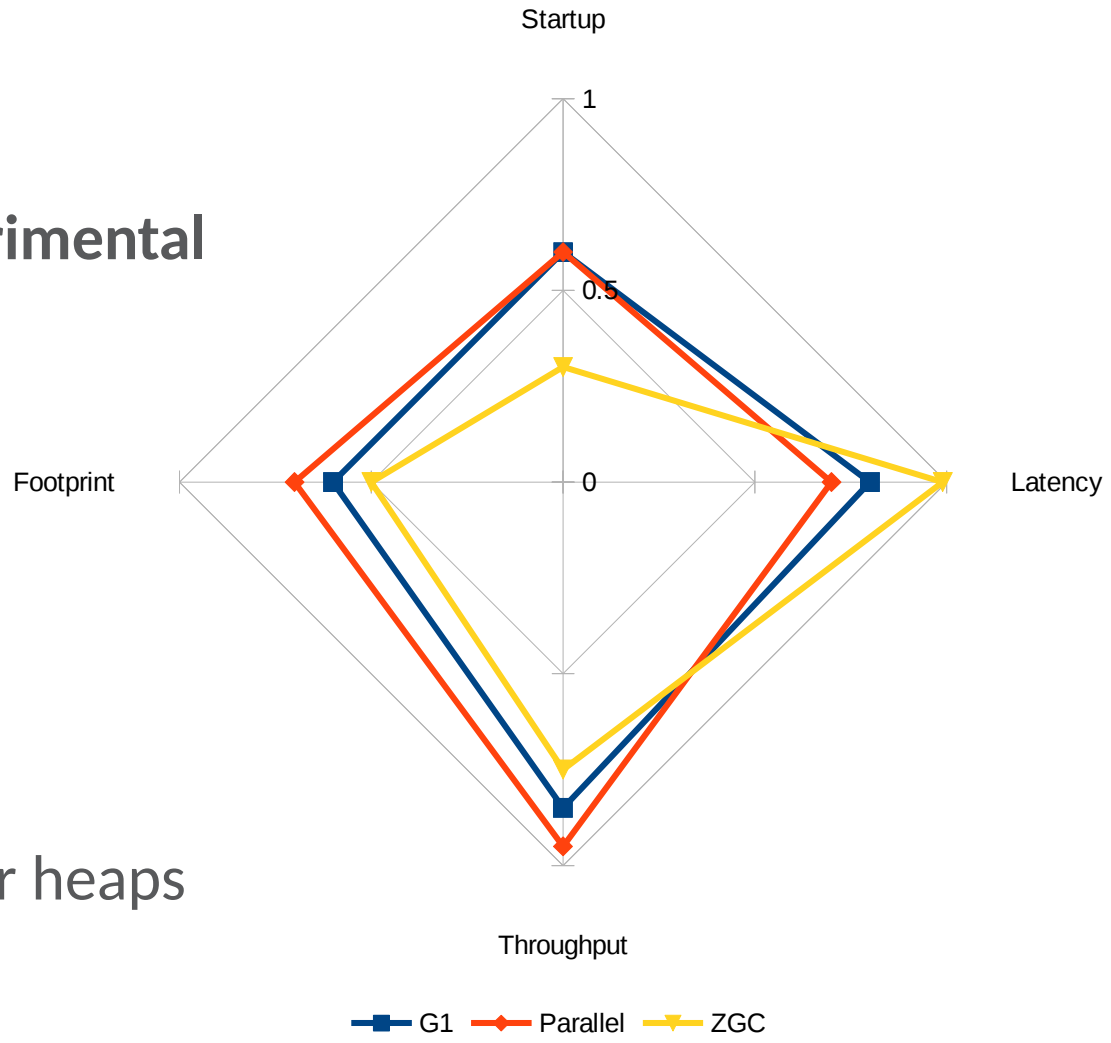
https://cr.openjdk.java.net/~pliden/slides/ZGC-PLMeetup-2019.pdf

Java™ ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

Some ZGC caveats

- Support: Linux only, x64 and AArch64
- Mostly feature complete in 13, but still **experimental**
- No compressed pointers
  - Higher footprint on smaller heaps
- Potentially heavy startup cost
  - No ***CDS*** support
  - Memory initialization overheads on larger heaps

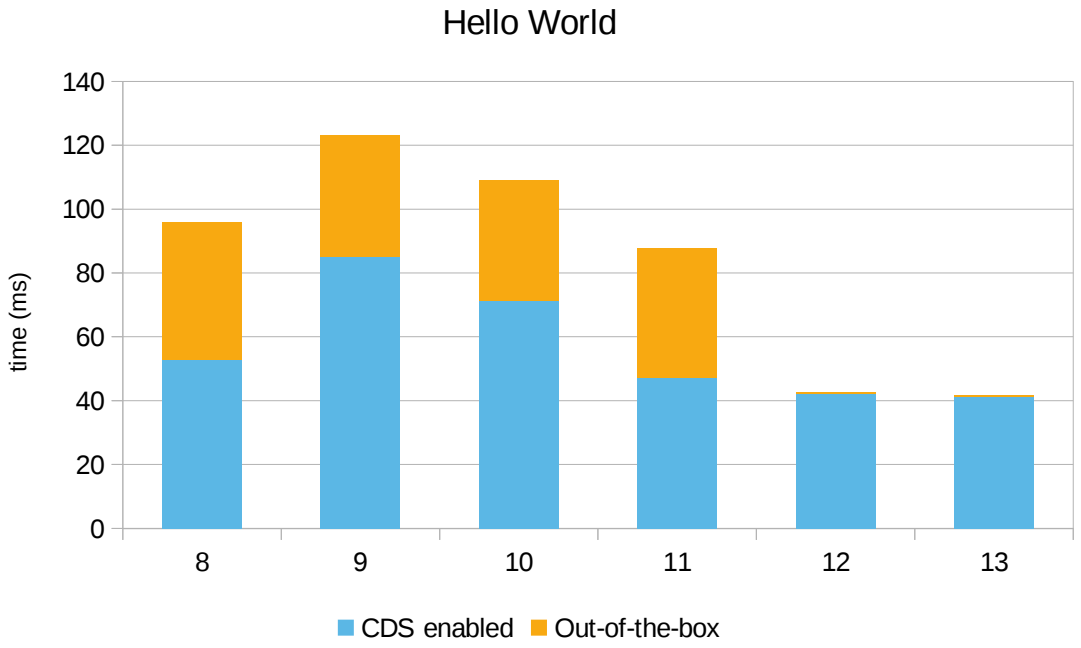


Java ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

"CDS?"

- Class-Data Sharing
  - Turns class loading from a time-consuming task into a simple matter of mapping in memory
  - Support for archiving part of the heap
- Run `java -Xshare:dump` once to enable
- Since JDK 12 CDS enabled and prepared *out of the box*



https://openjdk.java.net/jeps/341

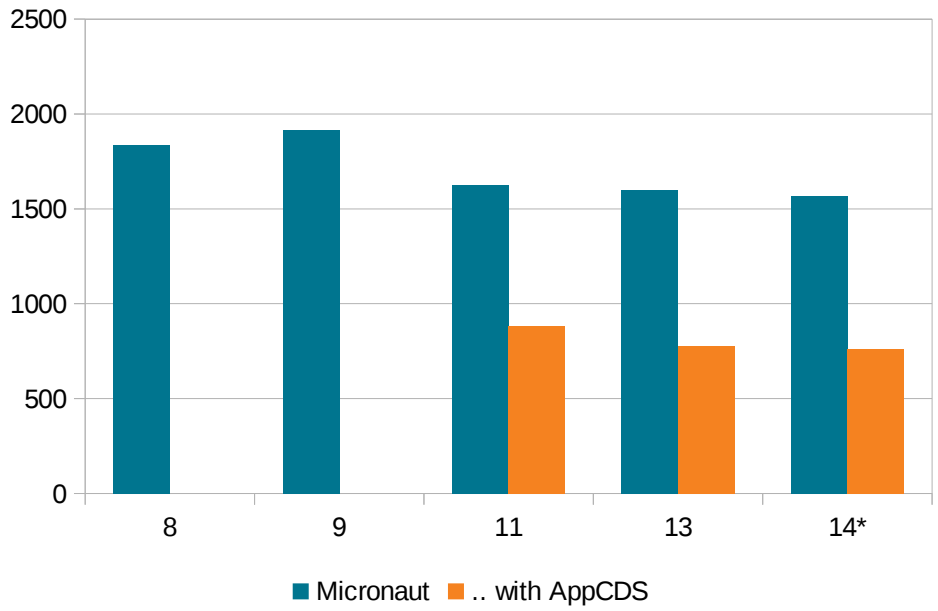
Java™ ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

Bring your own SharedArchiveFile

- Application Class-Data Sharing - AppCDS - was contributed to the OpenJDK in JDK 10
- Typically cuts 20-50% off startup numbers
- Gradually improved since inception
- Dynamic CDS (JDK 13) makes it easy to use:

```
# Generate archive with a training run
java -XX:ArchiveClassesOnExit=MyApp.jsa MyApp

# Ship it!
java -XX:SharedArchiveFile=MyApp.jsa MyApp
```



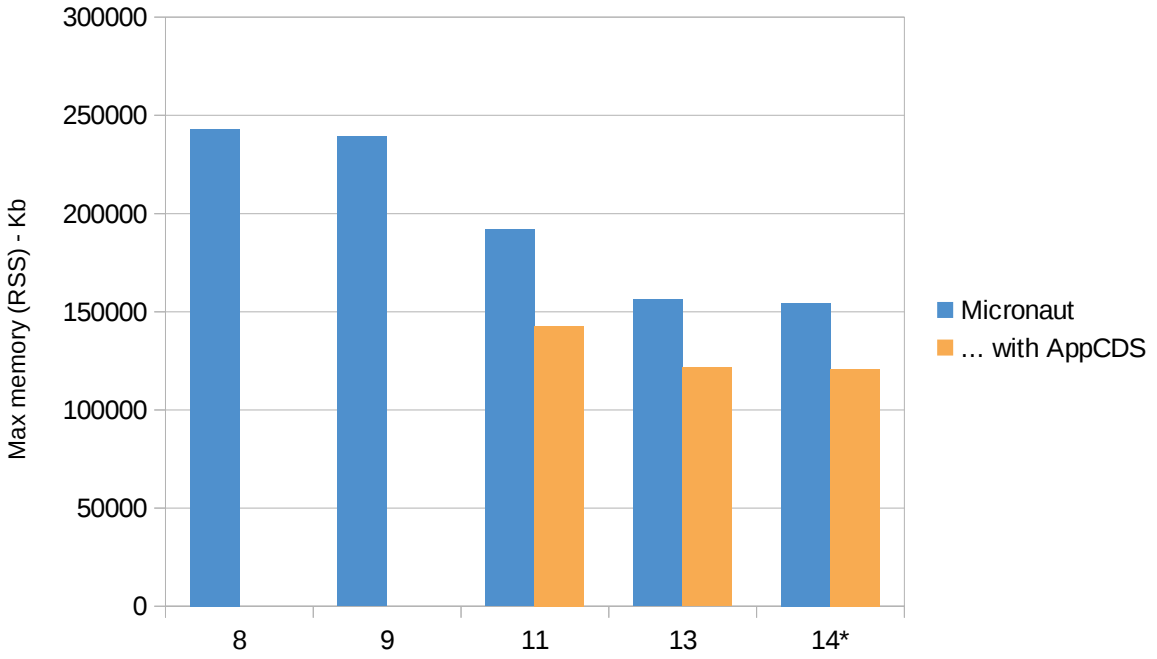
https://openjdk.java.net/jeps/310
https://openjdk.java.net/jeps/350

Java™ ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

Footprint improvements to boot!

- Startup improvements/features often go hand in hand with footprint improvements
- AppCDS helps, too, partly by removing the need to do bytecode verification at runtime



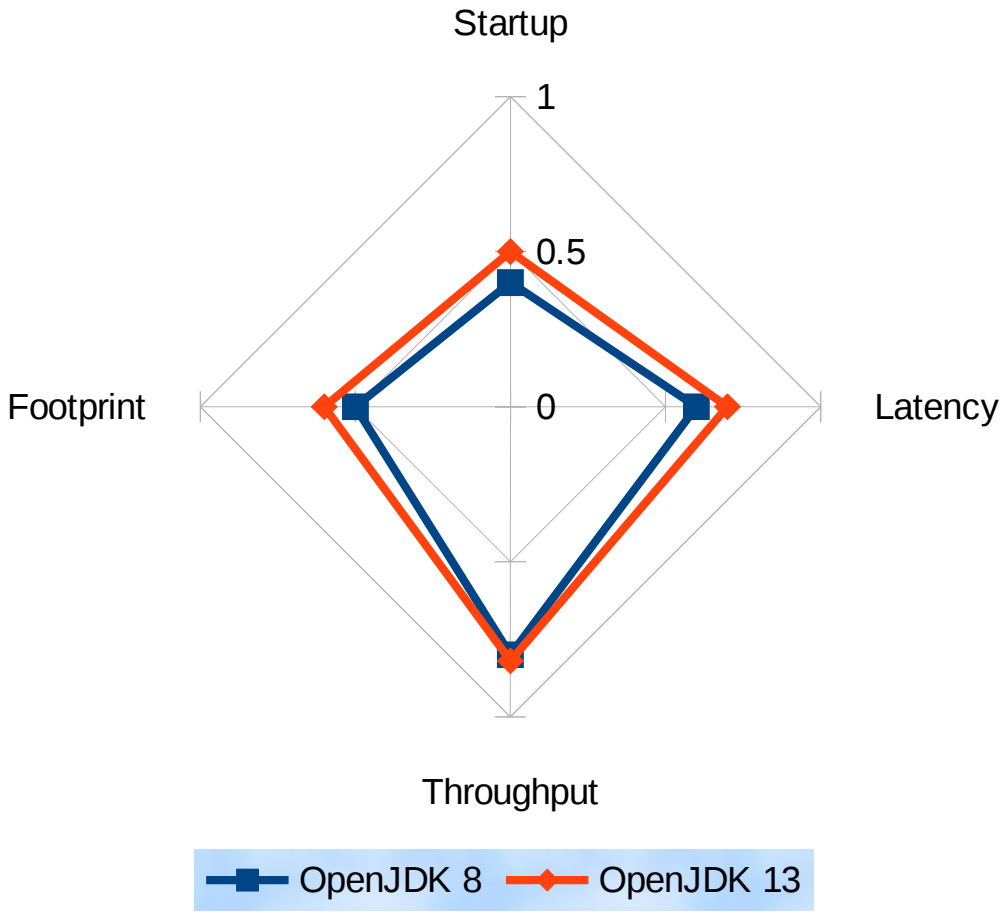
Java™ ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

The Story So Far… from JDK 8 to 13

- Numerous startup and footprint improvements to the out-of-the-box experience
- Performance features like Compact Strings have potential to improve performance in general
- While still experimental, work on ZGC already benefit **production** GCs like G1 and Parallel GC

YMMV



Java™ ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

"The best feature pipeline ever!"

- Project Valhalla
  - Value types for the JVM
  - Enable "flattening" objects, which improves density, which speeds up throughput
- Project Loom
  - Make it simple to write highly concurrent applications
- Project Amber
  - Umbrella for adding smaller productivity-oriented features to the java language
  - `var` delivered in 11
  - Switch expressions, text blocks being previewed in 13 - much more coming!
- Project Panama
  - Better (faster) and simpler native code interaction

Java™ ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

Potential startup plays

- JWarmup: https://openjdk.java.net/jeps/8203832
  - Record profiling information in one execution
  - Feed profile information into JVM during subsequent executions
  - Shortens the "warmup" phase by enabling JITs to do the Right Thing up front
- Constant folding, lazy finals, etc…
  - Language level support could enable (javac) compile time constant folding

    https://www.youtube.com/watch?v=iSEjlLFCS3E
  - VM support for dynamic creation of constants could enable lazy finals

    https://openjdk.java.net/jeps/8209964
- CRIU
- AOT?

Java™ ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

AOT vs. JIT

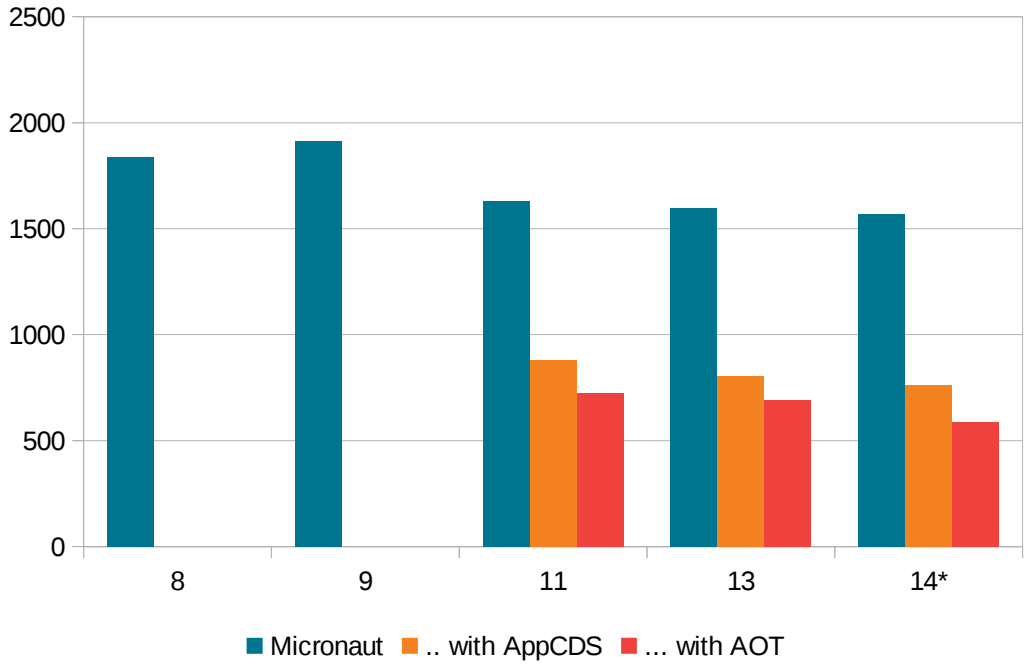
- An *Ahead-of-Time* compiler compiles source code into some target binary form
- The OpenJDK primarily uses *Just-In-Time compilers* to optimize the bytecode it executes *at runtime*
- JIT compilers solve three problems:
  - Not knowing *exactly* what hardware you're going to run on
  - Not knowing *exactly* what OS you're going to run on
  - Not knowing *how* your code is going to run
- JIT compilation can consume a lot of memory and CPU

Java™ ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

Experiments in AOT

- JDK 9 added the `jaotc` tool to enable AOT compilation
- Startup improvements…?
  - Main gain is reducing CPU and memory overhead of early JIT activity
  - Noticeable improvements for sufficiently complex applications
- Rough edges
  - Need to `--compile-for-tiered` to not cause substantial throughput penalty
  - Hard to fine-tune what to AOT to get good results
  - Relatively large binary sizes



Java ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

The GraalVM compiler - your next JIT?

- Since JDK 9, OpenJDK contains a version of the GraalVM compiler
- Used to implement the `jaotc` tool
- Can be used as a replacement for the C2 compiler *today:*

```
-XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler
```

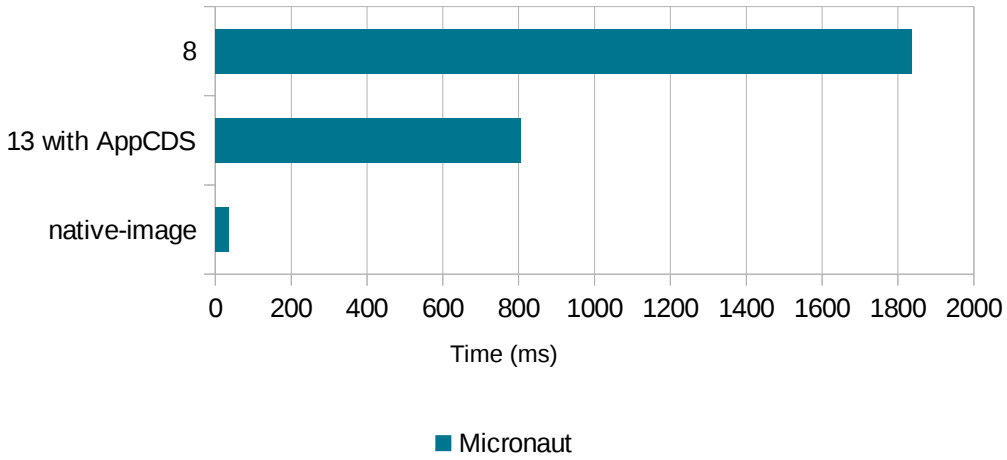
- Outperforms C2 on some workloads
- Written in Java

Java™ ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

GraalVM native-image

- "Substrate VM is a framework that allows ahead-of-time (AOT) compilation of Java applications under **closed-world assumption** into executable images or shared objects"
- In short: programs/shared libraries that start/load really fast
  - Not having a JIT also means tiny footprint compared to a HotSpot JVM



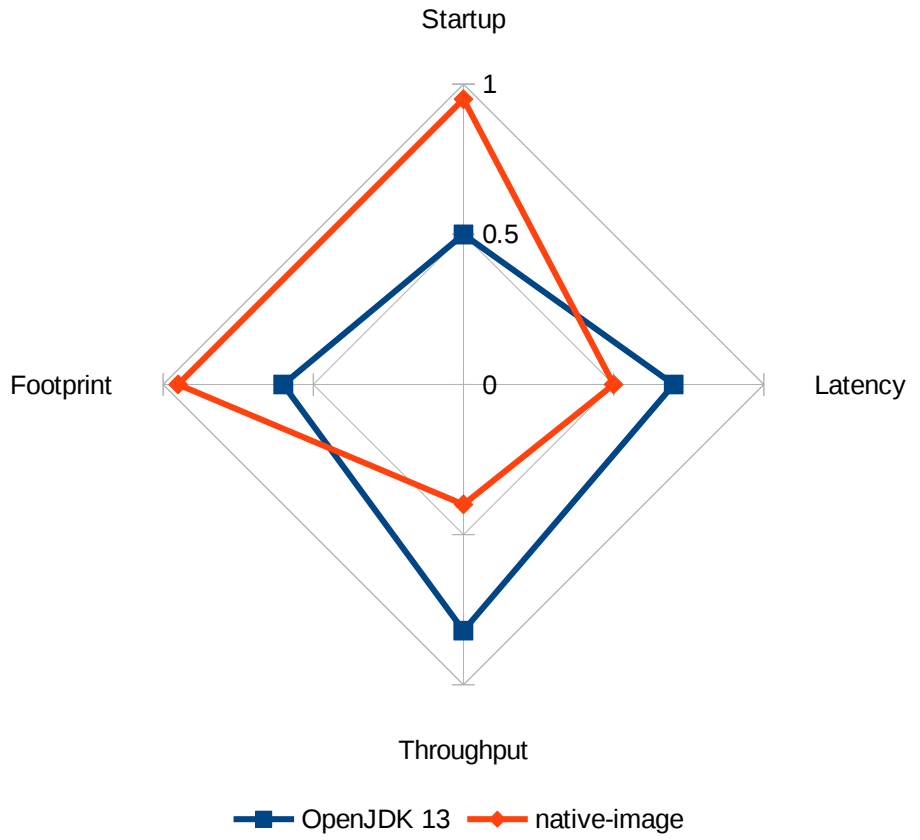
https://github.com/oracle/graal/tree/master/substratevm

Java™ ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

GraalVM native-image

- **Closed-world** assumption means that everything must be given up front to the compiler
  - Makes reflection, indy, condy… *complicated*
  - Native binaries, not a JVM
    - No JIT, limited GC, debugging and monitoring options, …
- Compiling the GraalVM compiler itself as a shared library resolve most startup and footprint issues when used as a JIT by HotSpot
  - Implemented in GraalVM, but not yet in OpenJDK mainline

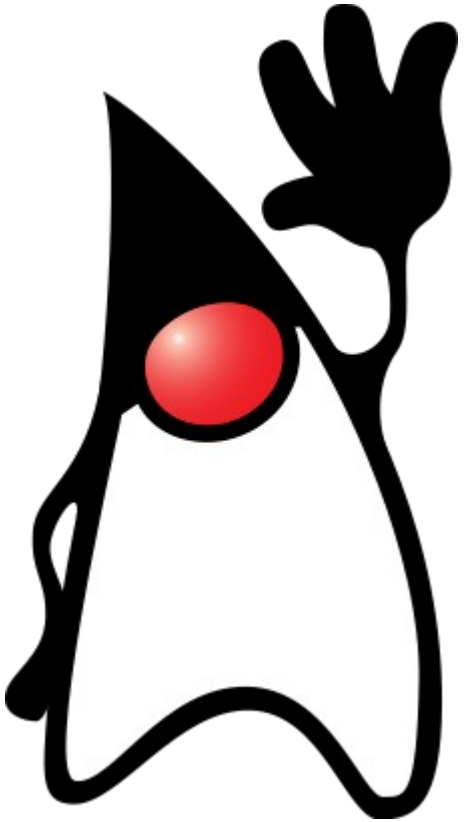


https://www.youtube.com/watch?v=RMtukctD220

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

In conclusion

- OpenJDK 13 is great!
  - … but it's just a bit better than OpenJDK 12
  - … which in turn is just a bit better than OpenJDK 11
- It will keep getting better!
- (Opinion) Releasing a new feature release every six months has revitalized the OpenJDK project
  - Projects delivered when done, and in smaller increments
  - Minimal risk of something holding up the release
  - This means less stress
  - More opportunities for smaller enhancements to actually get done



Java™ ORACLE

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

Q & A

Copyright © 2018, Oracle and/or its affiliates. All rights reserved. |