ORACLE

GraalVM for Java developers

Oleg Šelajev
Developer advocate, GraalVM team, Oracle Labs
@shelajev

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, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

GraalVM Native Image early adopter status

GraalVM Native Image technology (including SubstrateVM) is Early Adopter technology. It is available only under an early adopter license and remains subject to potentially significant further changes, compatibility testing and certification.

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

© 2020 Oracle

What is GraalVM?

A high-performance optimizing Just-in-Time (JIT) compiler

An Ahead-of-Time (AOT) “native image” compiler

Multilingual Virtual Machine

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

GraalVM™ Quick Reference

Run Java Applications

Compile a Java class:
```
javac MyApp.java
```

Run the application from a JAR file:
```
java -jar  MyApp.jar
```

Specify the class path for the app:
```
java -cp target/myapp.jar
     com.mycompany.app.MyApp
```

Run the JIT compiler as JAR or native library (default):
```
-XX:±UseJVMCINativeLibrary
```

Select the GraalVM compiler configuration:
```
-Dgraal.CompilerConfiguration=
     enterprise|community|economy
```

Print the details for the JIT compiled code:
```
-Dgraal.PrintCompilation=true
```

Produce the diagnostic data for the compilation:
```
-Dgraal.Dump
```

Load a javaagent:
```
-javaagent:<jarpath>[=<options>]
-agentlib:<libname>[=<options>]
```

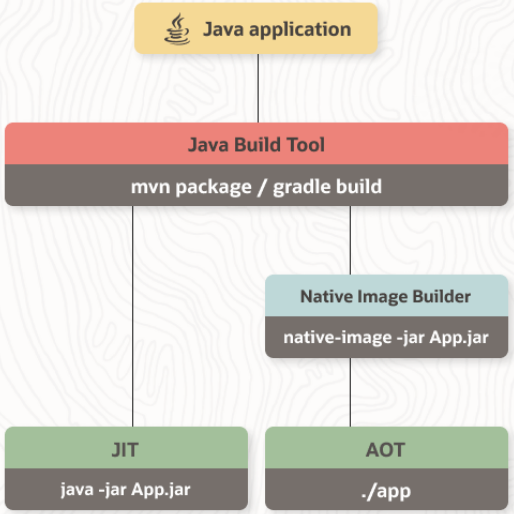
Compile to Native Executables

Install the native image builder from a local file:
```
gu install -L native-image.jar
```

Native Image command syntax:
```
native-image [options] MyClass
```

Build a native image of a JAR file:
```
native-image -jar MyApp.jar
```

Run a native image:
```
./myapp
```



Build a shared library:
```
--shared
```

Build a statically linked native image:
```
--static --libc=glibc|musl
```

Include a language runtime in the native image:
```
--language:js|python|ruby|llvm|wasm
```

Use profile-guided optimizations:
```
native-image --pgo-instrument MyApp
./myapp  #and apply load
native-image --pgo profile.iprof MyApp
```

Attach a debugger:
```
--debug-attach=[port]
```

Trace classes initialization:
```
--trace-class-initialization
```

List all image build options for experts:
```
--expert-options-all
```

Enable Polyglot Programming

Run a Node.js application:
```
node myApp.js
```

Run a JavaScript, R, Ruby, Python, LLVM application:
```
js myApp.js
graalpython myApp.py
ruby myApp.rb
R myApp.r
lli myApp
```

Run other languages in a Java application:
```
org.graalvm.polyglot
  .Context
    .createContext()
    .eval("languageId", "code");
```

Enable polyglot capabilities for an application:
```
--polyglot --jvm
```

Limit resources for the application:
```
--sandbox.MaxCPUTime=<ms>
--sandbox.MaxStatements=N
```

Debug the application:
```
--inspect[=[host:]<port number>]
--inspect-brk
```

Profile the application:
```
--cpusampler
--cputracer
--memtracer
```

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

https//: www.graalvm.org

ver: 1.000

GraalVM™

JIT

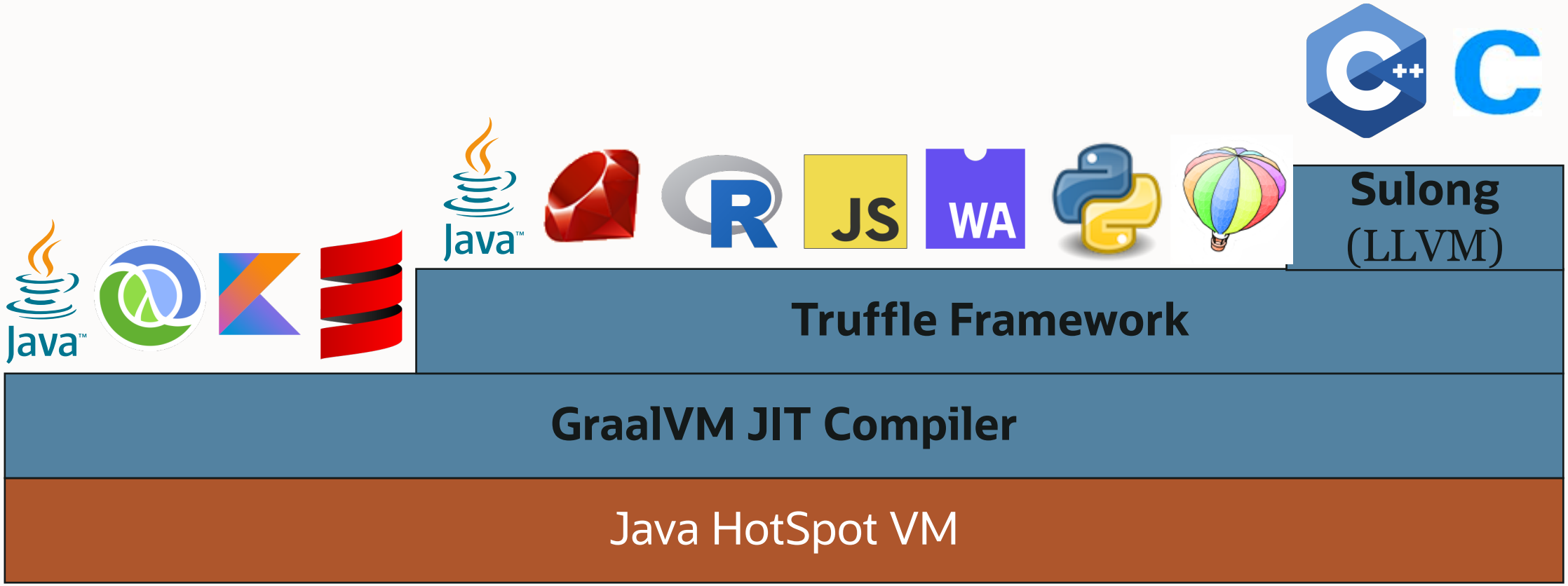
java MyMainClass

OpenJDK™

AOT

native-image MyMainClass

./mymainclass



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Getting started with GraalVM

1. Download GraalVM: https://www.oracle.com/downloads/graalvm-downloads.html
2. Unpack the archive
3. Set up and code!

```
$ export GRAALVM_HOME=/path/to/graalvm

$ export PATH=$GRAALVM_HOME/bin:$PATH

$ java –jar your-application.jar
```


© 2020 Oracle

A benchmark:

```java
public class ObjectsHash {

    static int intField;
    static long longField;
    static double doubleField;

    @Benchmark
    public int abstracted() {
        return Objects.hash(intField, longField, doubleField);
    }

    @Benchmark
    public int raw() {
        return ((intField * 31) + Long.hashCode(longField)) * 31 + Double.hashCode(doubleField);
    }
```


© 2020 Oracle

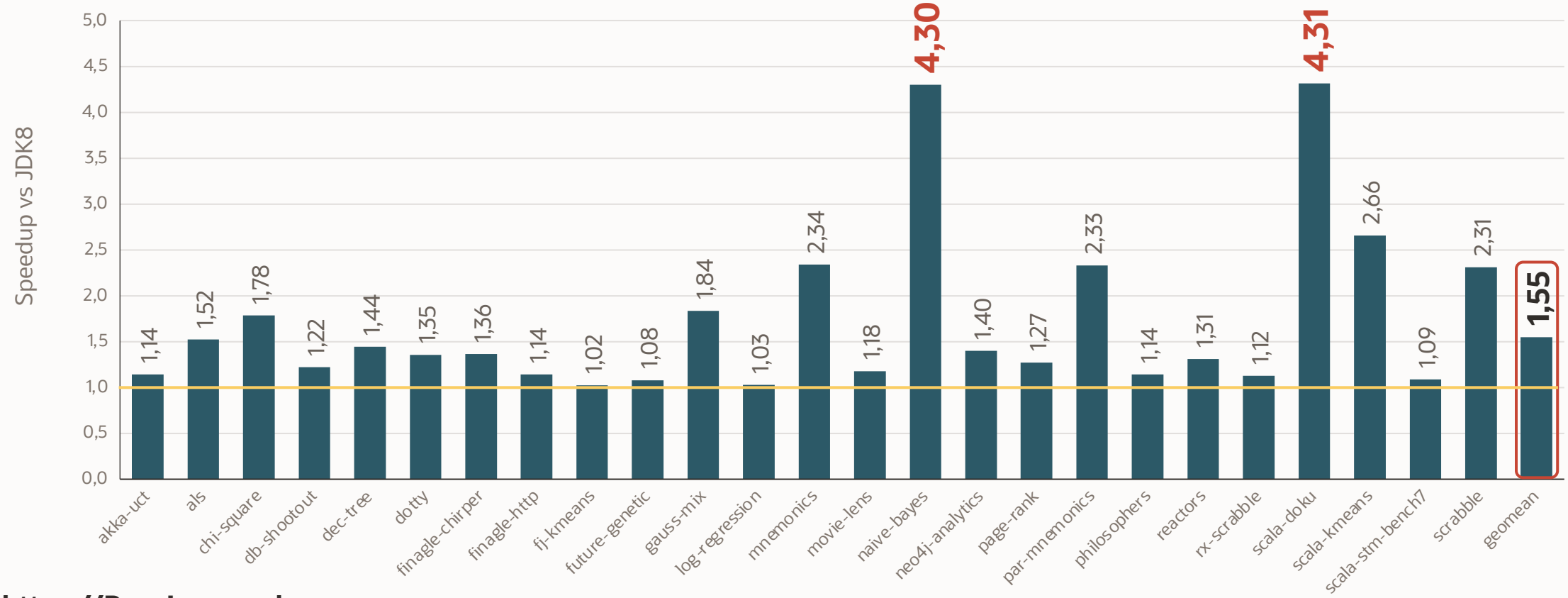
Which one do you think is closer to being true?

1. **abstracted()** and **raw()** approximately same performance

2. **raw()** is significantly (>10%) faster than **abstracted()**

© 2020 Oracle

Real World Benchmarks - Renaissance.dev



https://Renaissance.dev

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

More Benchmarks…

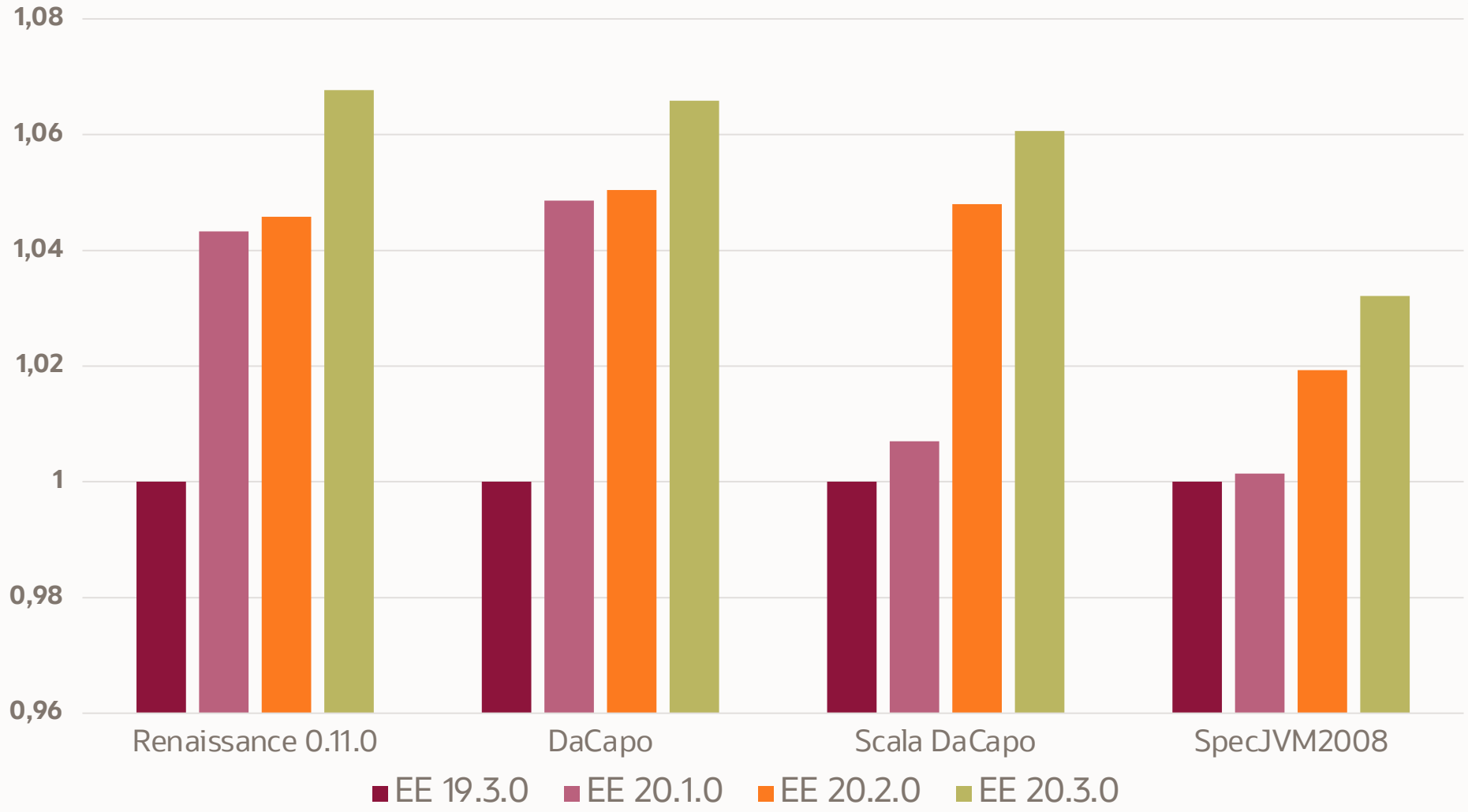
Optimizing for too few benchmarks is like overfitting a machine learning algorithm.

Therefore we started together with academic collaborators https://renaissance.dev

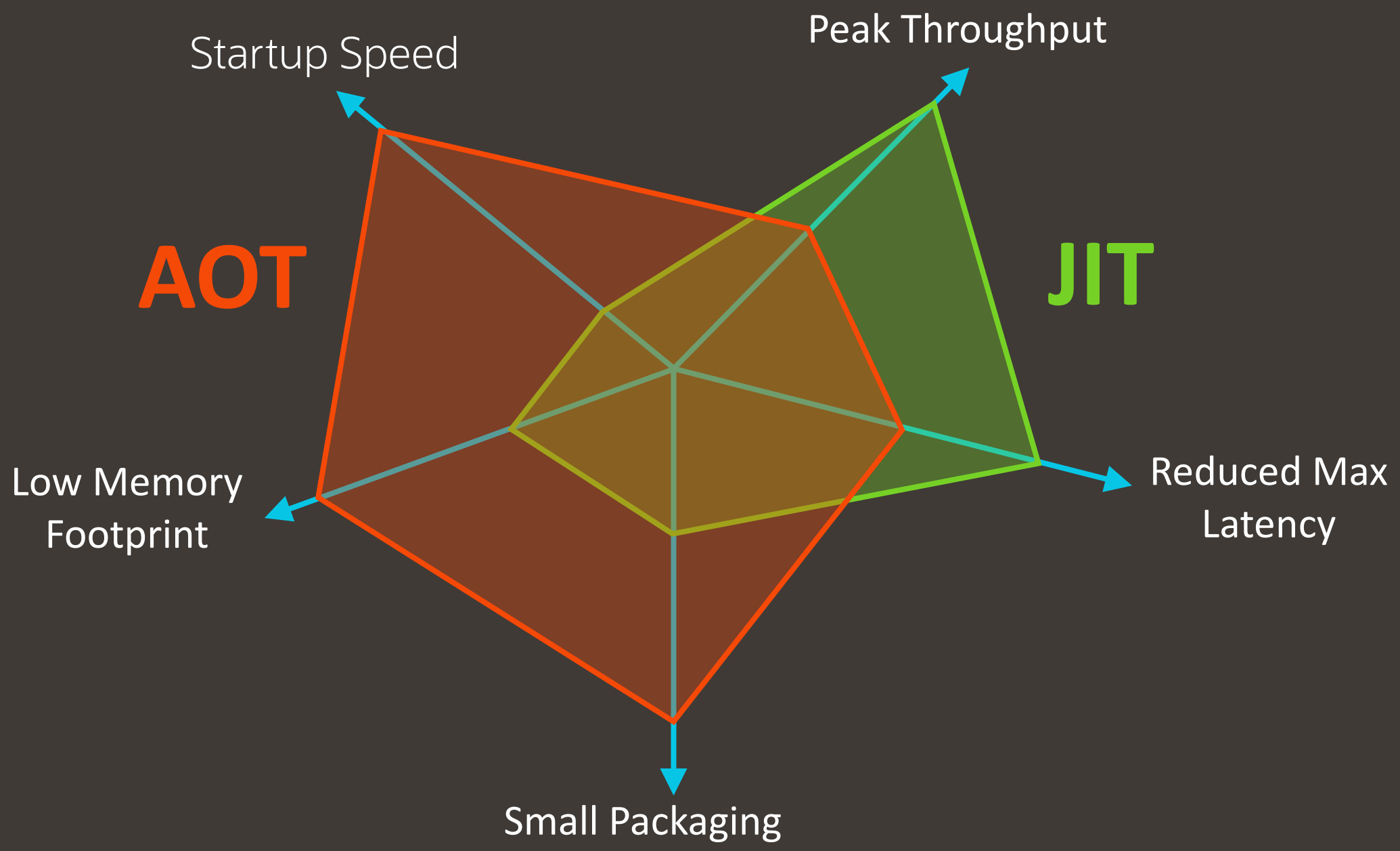
All benchmark data can be interesting; careful with conclusions though.

Speed-Up, normalized to GraalVM EE 19.3.0 JDK11

~6% improvements across all workloads



AOT vs JIT


Startup Speed
Peak Throughput
AOT
JIT
Low Memory Footprint
Reduced Max Latency
Small Packaging

GraalVM™

JIT

java MyMainClass

OpenJDK™

AOT

native-image MyMainClass

./mymainclass

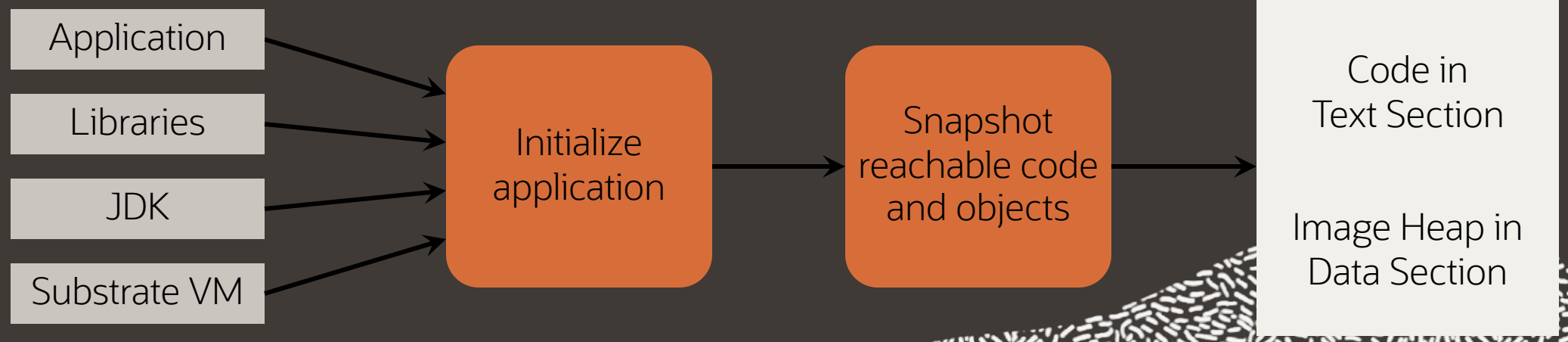
Closed World Assumption

- The points-to analysis needs to see all bytecode
  - Otherwise aggressive AOT optimizations are not possible
  - Otherwise unused classes, methods, and fields cannot be removed
  - Otherwise a class loader / bytecode interpreter is necessary at run time
- Dynamic parts of Java require configuration at build time
  - Reflection, JNI, Proxy, resources, …
- No loading of new classes at run time

© 2019 Oracle

Native Image: Principle

Input:
All classes from application, libraries, and VM

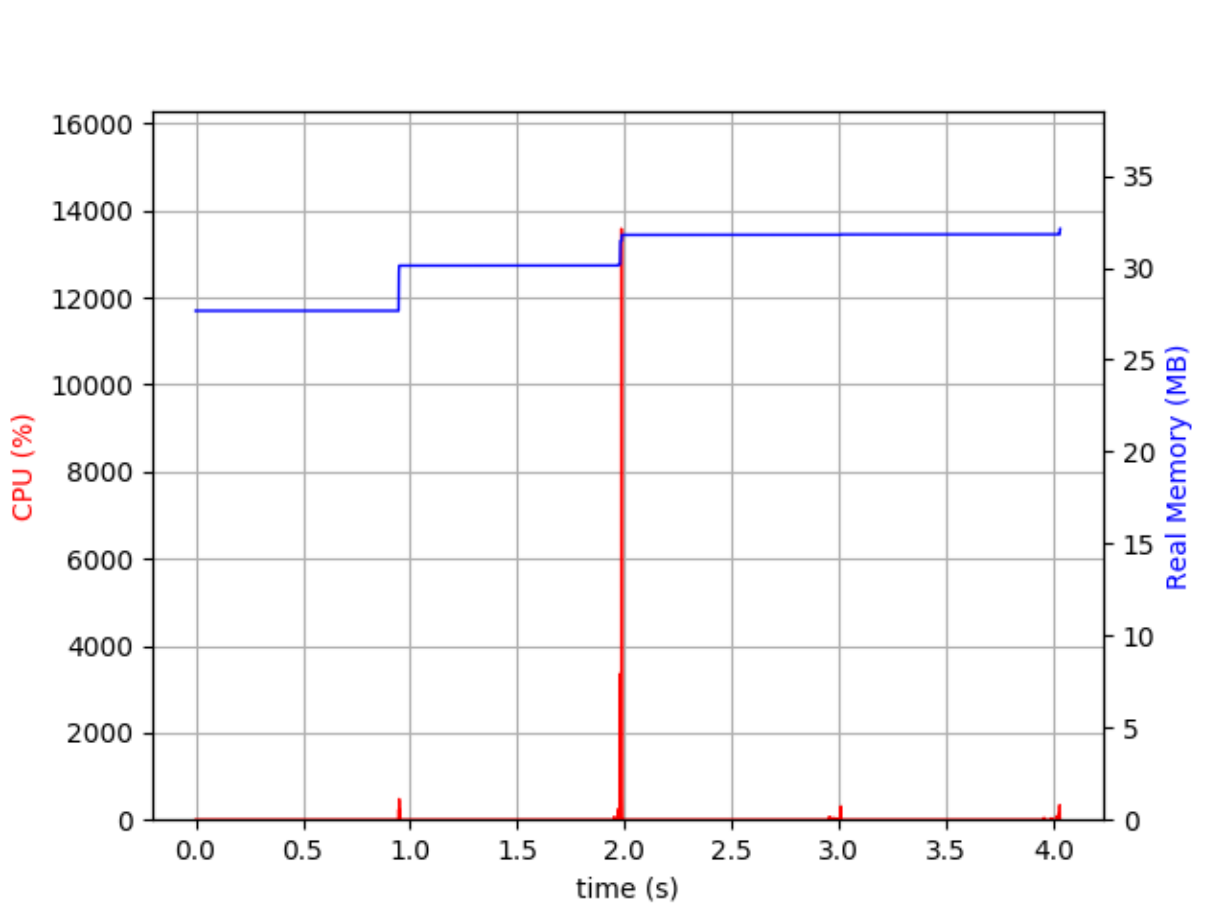


Output:
Native executable

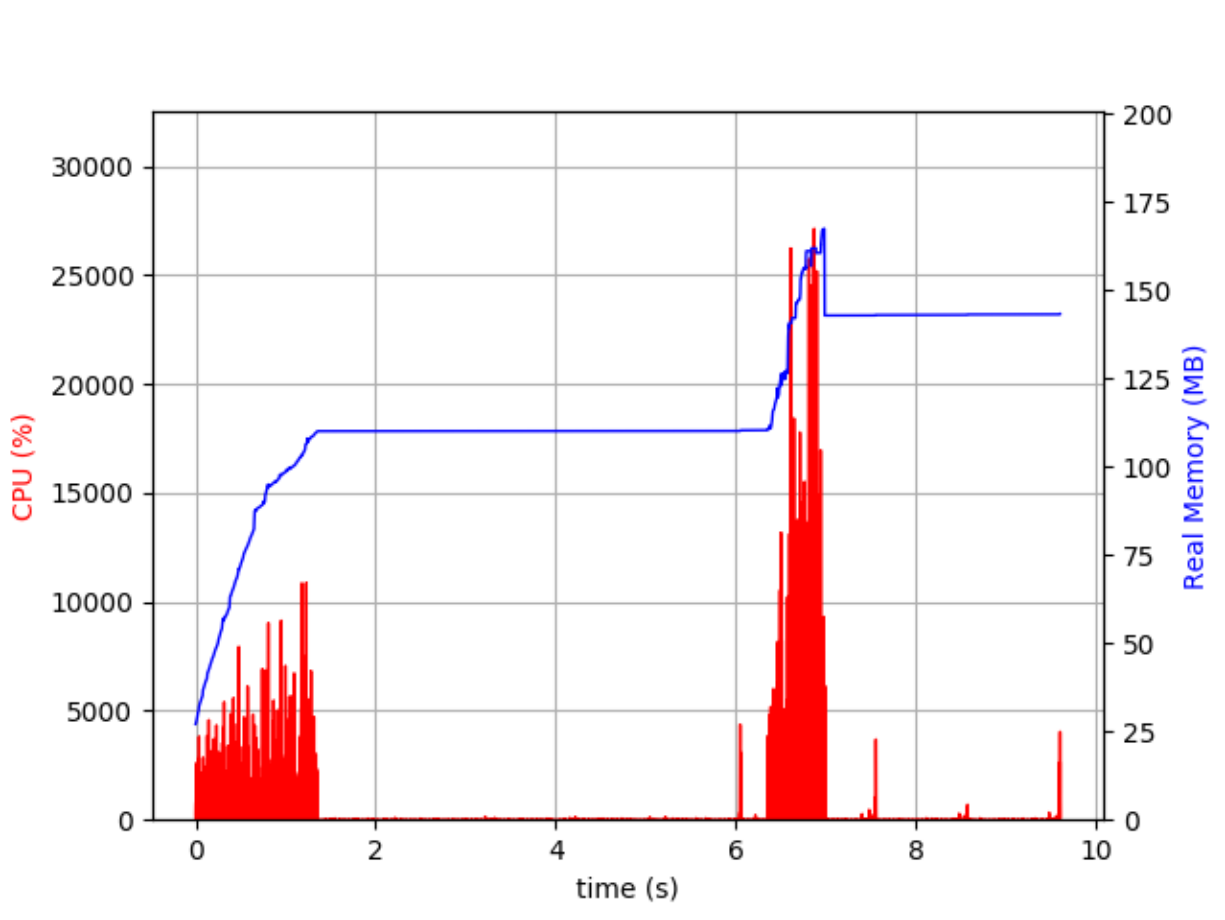
© 2019 Oracle

Starting up and serving 3 requests (Micronaut example)

AOT



JIT



GraalVM support in the ecosystem

helidon.io

MICRONAUT

picocli

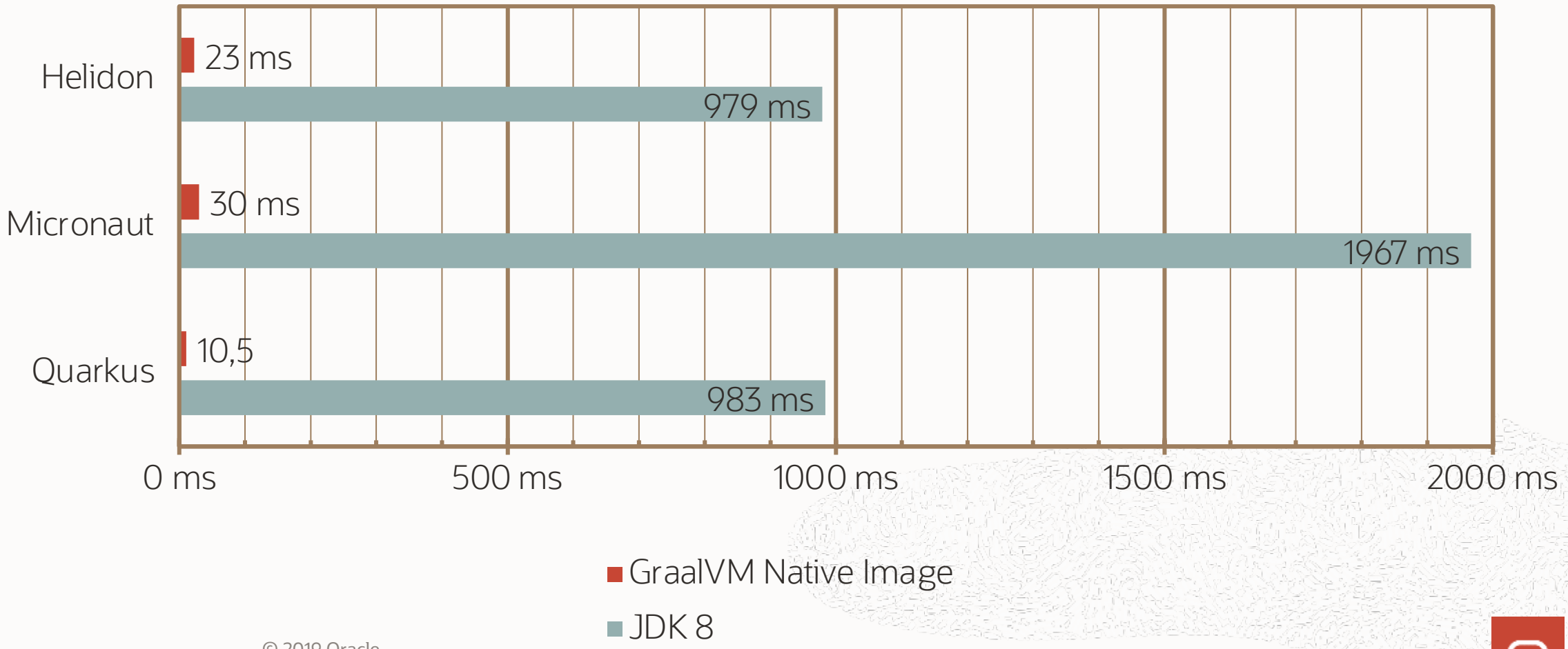
ORACLE
GraalVM

spring
by Pivotal

QUARKUS

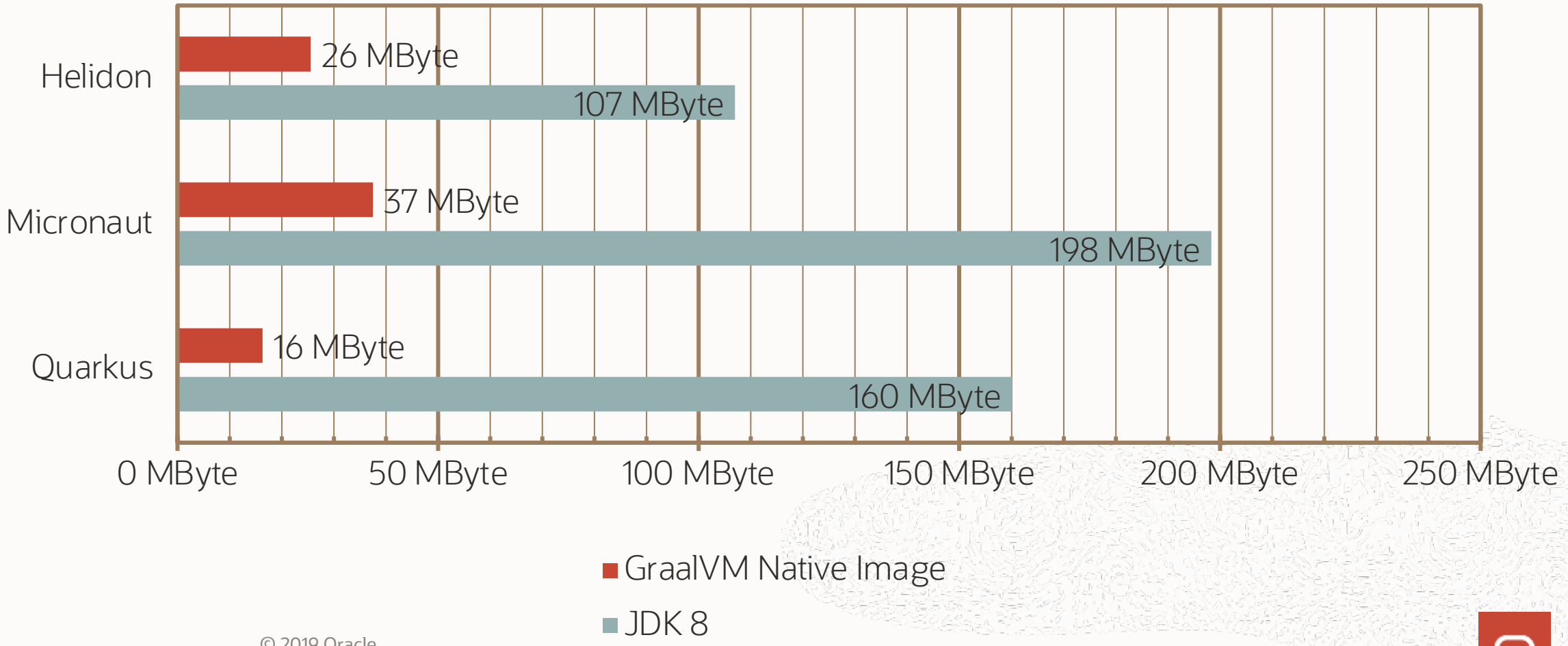
Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Startup Time



© 2019 Oracle

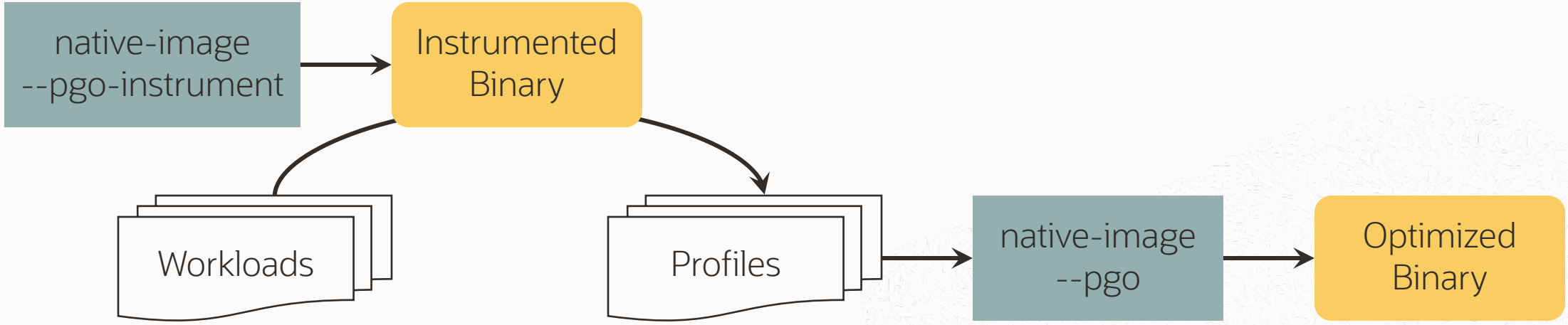
Memory Footprint



© 2019 Oracle

Addressing Peak Thoughput & Max Latency

- G1 Garbage Collector option for native image for better max latency
- Profile-guided optimizations (PGO) for better peak performance



© 2020 Oracle

Java on Truffle

GraalVM™

Java™

Scala Groovy Kotlin

JIT

OpenJDK™

java MyMainClass

Java on Truffle

java -truffle MyMainClass

AOT

native-image MyMainClass
./mymainclass

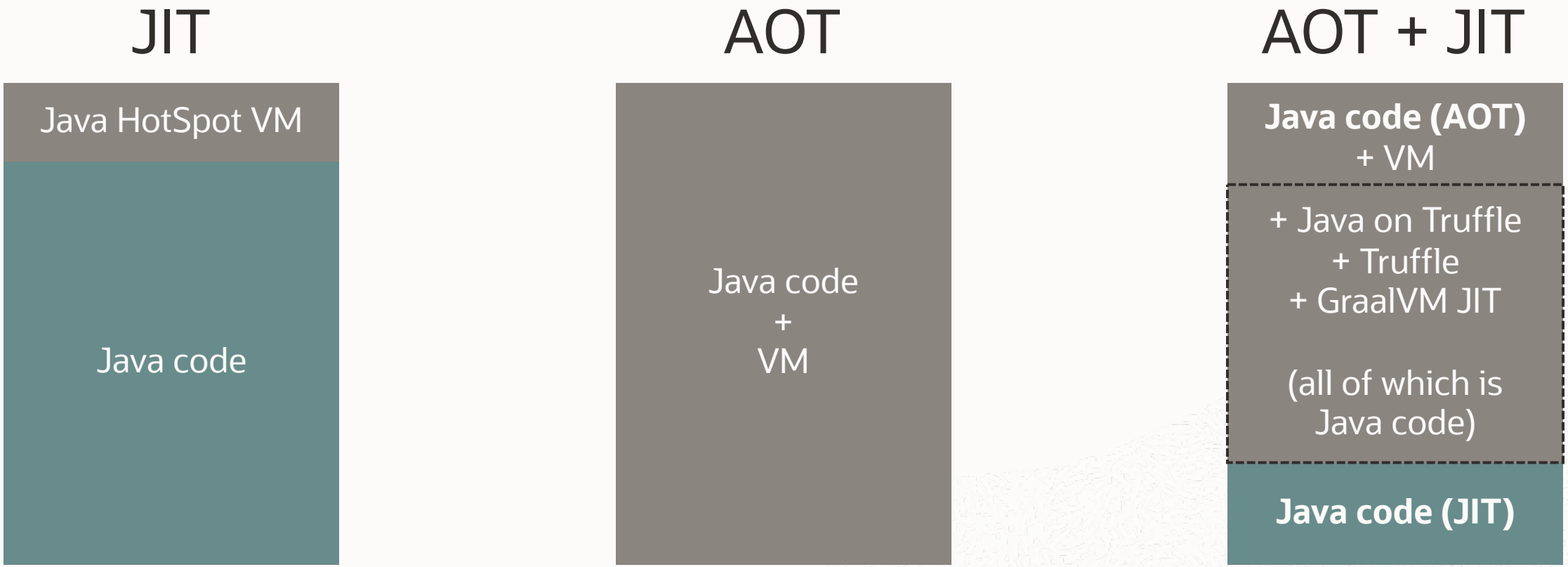
© 2021 Oracle

Advanced class redefinition

Supported	Not supported currently
Adding and Removing methods	Changes to fields
Adding and Removing constructors	Changes to class access modifiers
Adding and Removing methods from interfaces	Changing the superclass
Changing access modifiers of methods	Changing the list of implemented interfaces
Changing access modifiers of constructors	Changes to enums
Changings to lambdas	
Adding new anonymous inner classes	
Removing anonymous inner classes	

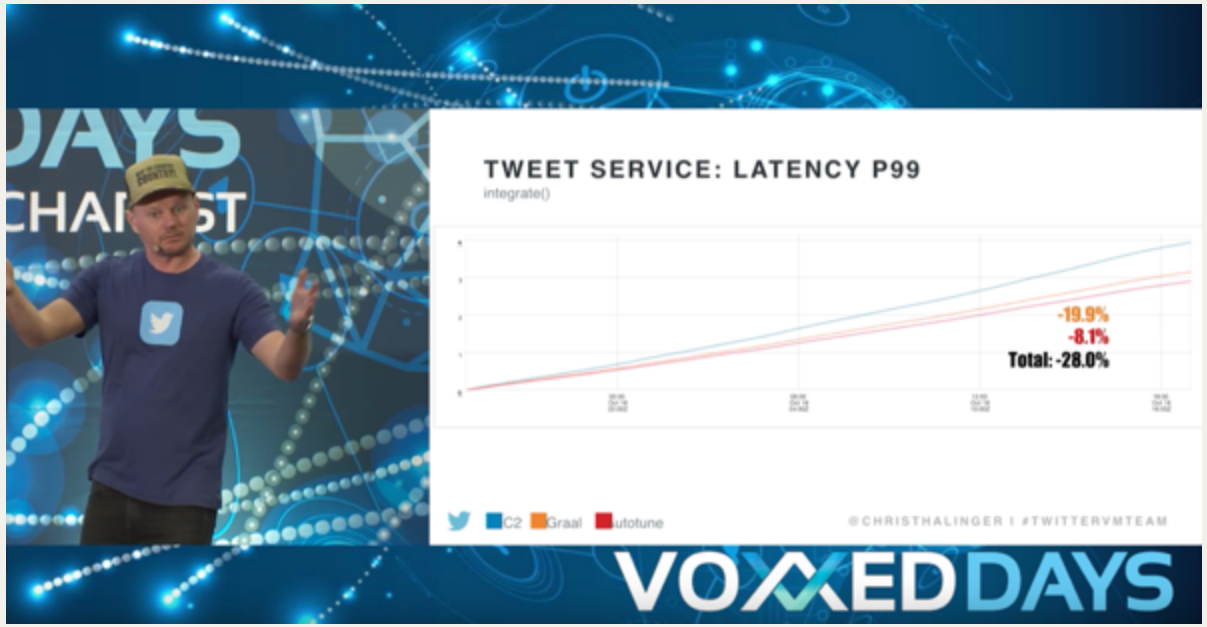
© 2021 Oracle

Mixing AOT and JIT in Java programs



© 2021 Oracle

Twitter uses GraalVM compiler in production to run their Scala microservices



© 2019 Oracle

- Peak performance: +10%
- Garbage collection time: -25%
- Seamless migration

ORACLE
Cloud Infrastructure

https://blogs.oracle.com/cloud-infrastructure/graalvm-powers-oracle-cloud-infrastructure

© 2019 Oracle

The rich ecosystem of CUDA-X libraries is now available for GraalVM applications.

GPU kernels can be directly launched from GraalVM languages such as R, JavaScript, Scala and other JVM-based languages.

https://devblogs.nvidia.com/grcuda-a-polyglot-language-binding-for-cuda-in-graalvm/

© 2019 Oracle

Oracle GraalVM Enterprise Edition

- Higher performance
- Smaller footprint
- Enhanced security for native code
- Oracle Enterprise Support 7x24x365
  - Support directly from the GraalVM Team

ORACLE®
GraalVM

© 2019 Oracle