ORACLE

ORACLE

Maximizing Performance with GraalVM

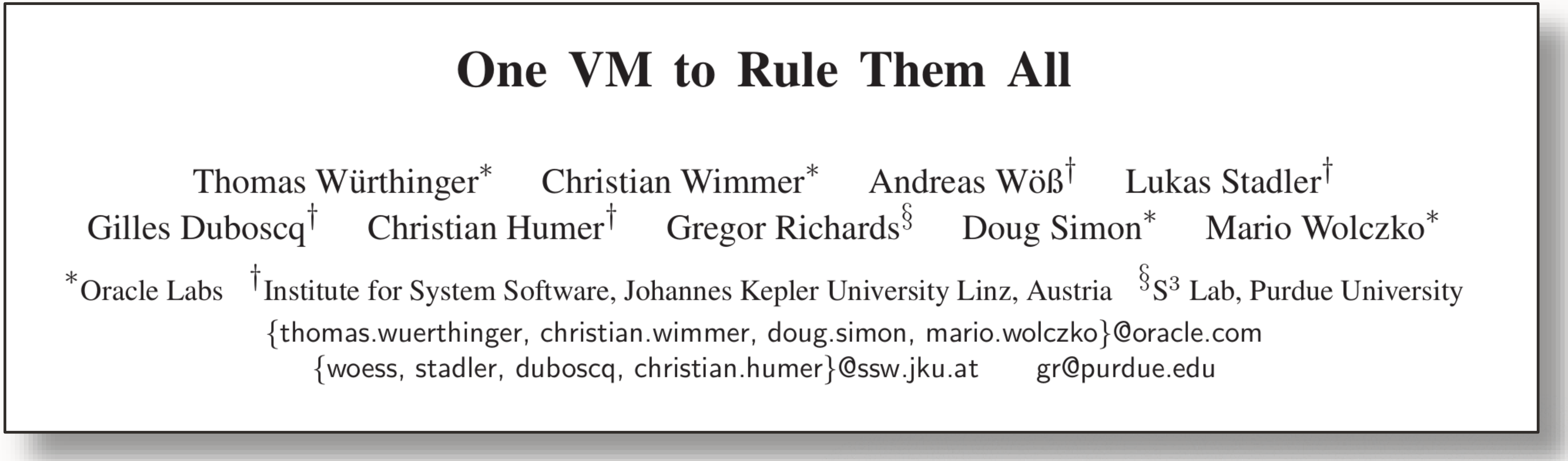
Christian Wimmer
GraalVM Native Image Project Lead
christian.wimmer@oracle.com

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 technology (including Substrate VM) 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.

~8 years ago, we started a little research project…

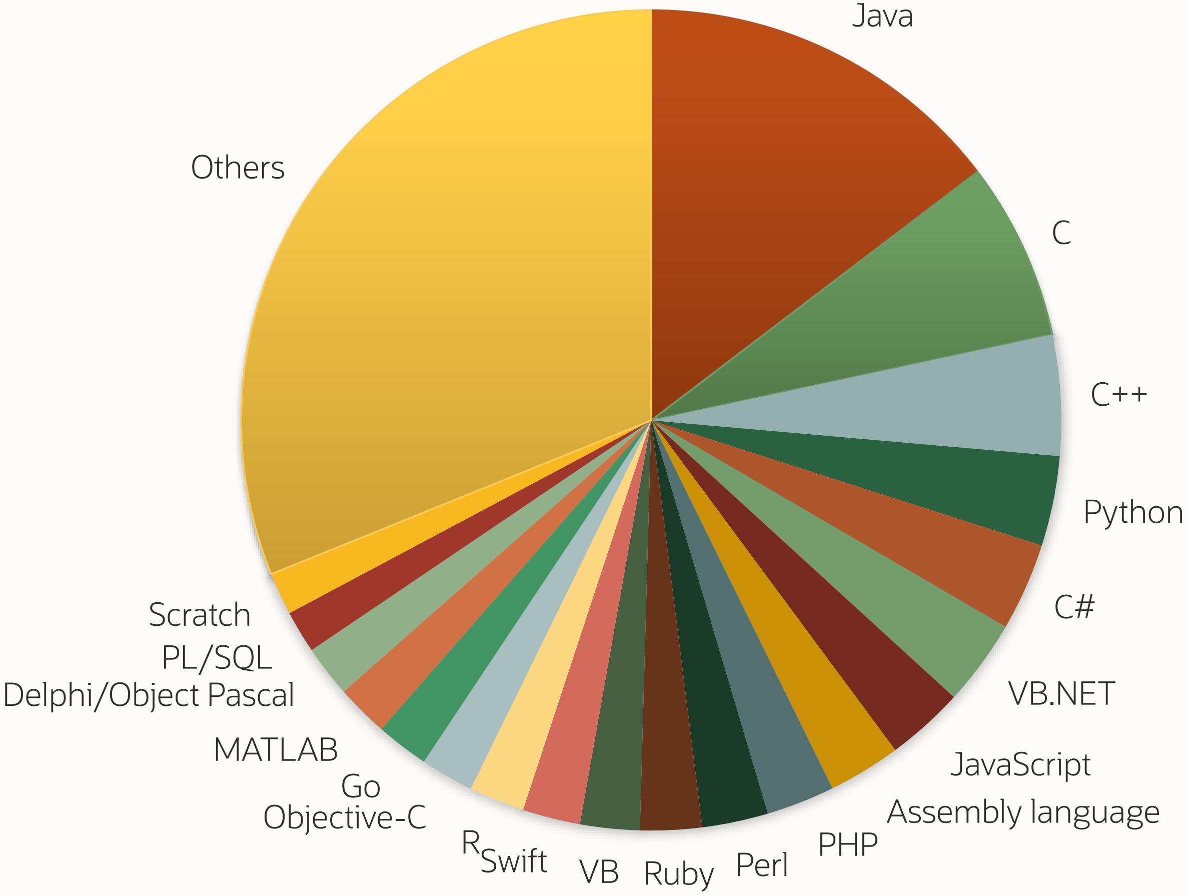


One VM to Rule Them All

Thomas Würthinger[*] Christian Wimmer[*] Andreas Wöß[†] Lukas Stadler[†]
Gilles Duboscq[†] Christian Humer[†] Gregor Richards[§] Doug Simon[*] Mario Wolczko[*]

[*]Oracle Labs [†]Institute for System Software, Johannes Kepler University Linz, Austria [§]S^3 Lab, Purdue University
{thomas.wuerthinger, christian.wimmer, doug.simon, mario.wolczko}@oracle.com
{woess, stadler, duboscq, christian.humer}@ssw.jku.at gr@purdue.edu

Programming Language Popularity (TOP 20 Languages From May 2018 Tiobe INDEX)

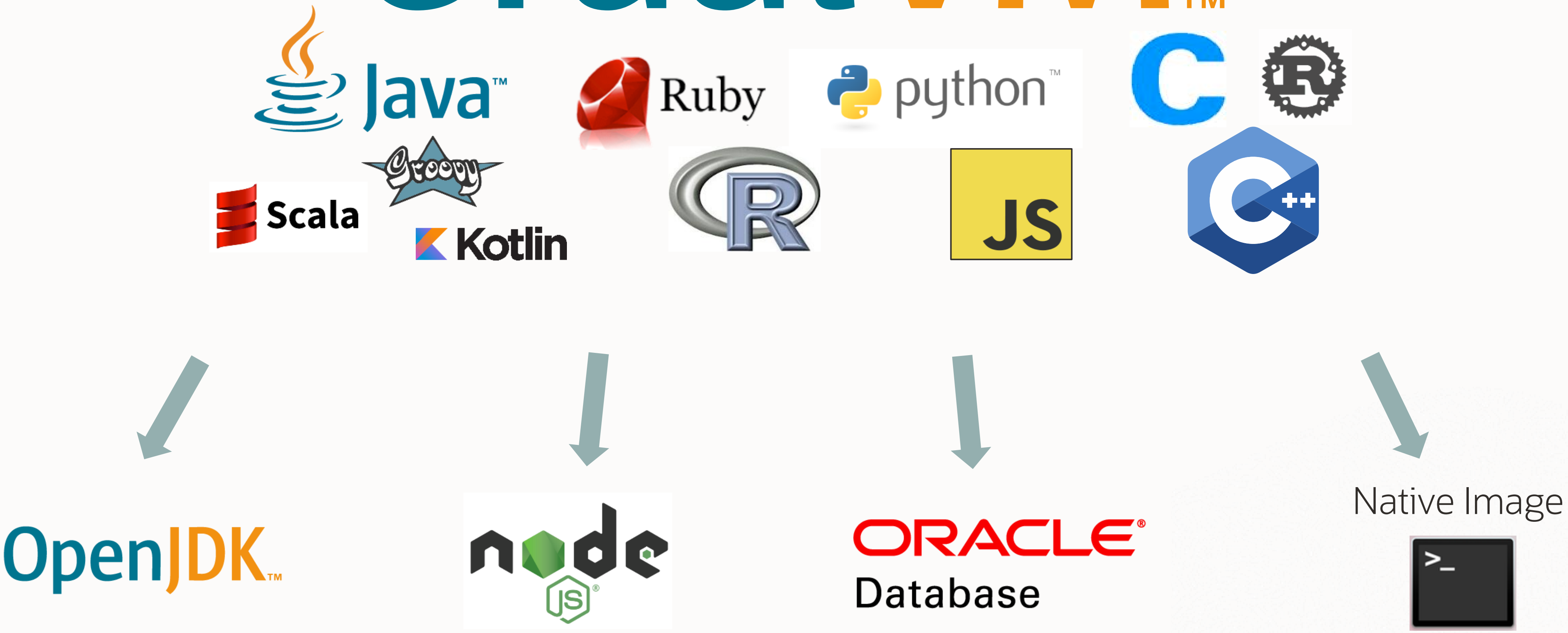


GraalVM will be the Universal VM

ubiquitous across the cloud stack

1. run programs more efficient
2. make developers more productive

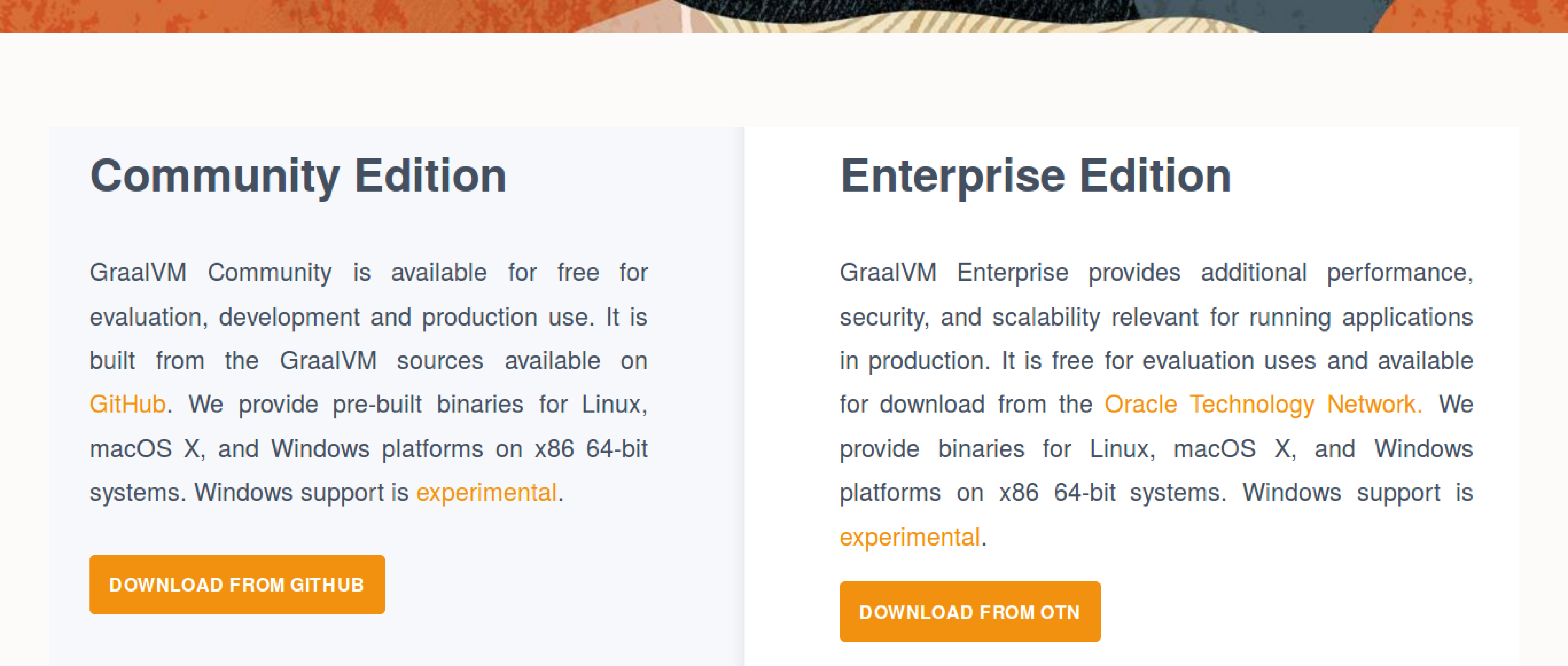
GraalVM™



Java™
Ruby
python™
C
R
Scala
Groovy
Kotlin
R
JS
C++
OpenJDK™
node JS®
ORACLE®
Database
Native Image
>_

What is Graal VM?

- Drop-in replacement for Oracle Java 8 and Java 11
  - Run your Java application faster
- Ahead-of-time compilation for Java
  - Create standalone binaries with low footprint
- High-performance JavaScript, Python, Ruby, R, …
  - The first VM for true polyglot programming
  - Implement your own language or DSL



Community Edition

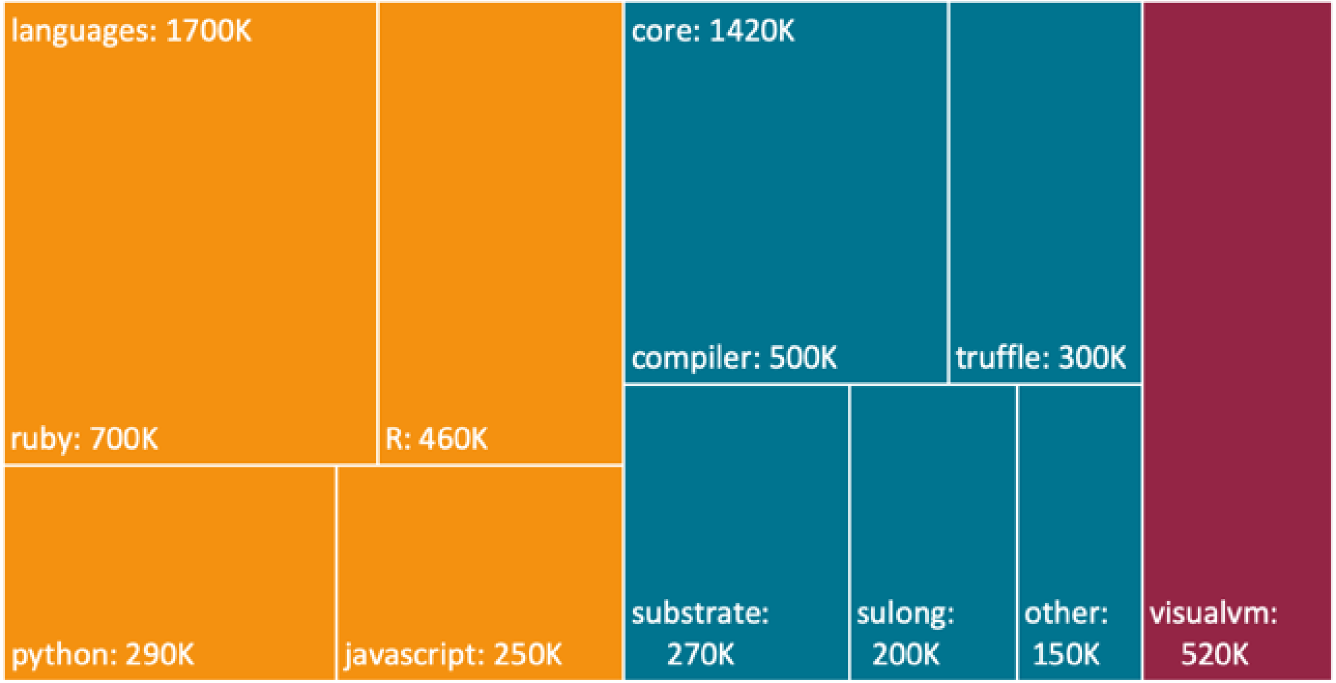
GraalVM Community is available for free for evaluation, development and production use. It is built from the GraalVM sources available on GitHub. We provide pre-built binaries for Linux, macOS X, and Windows platforms on x86 64-bit systems. Windows support is experimental.

Enterprise Edition

GraalVM Enterprise provides additional performance, security, and scalability relevant for running applications in production. It is free for evaluation uses and available for download from the Oracle Technology Network. We provide binaries for Linux, macOS X, and Windows platforms on x86 64-bit systems. Windows support is experimental.

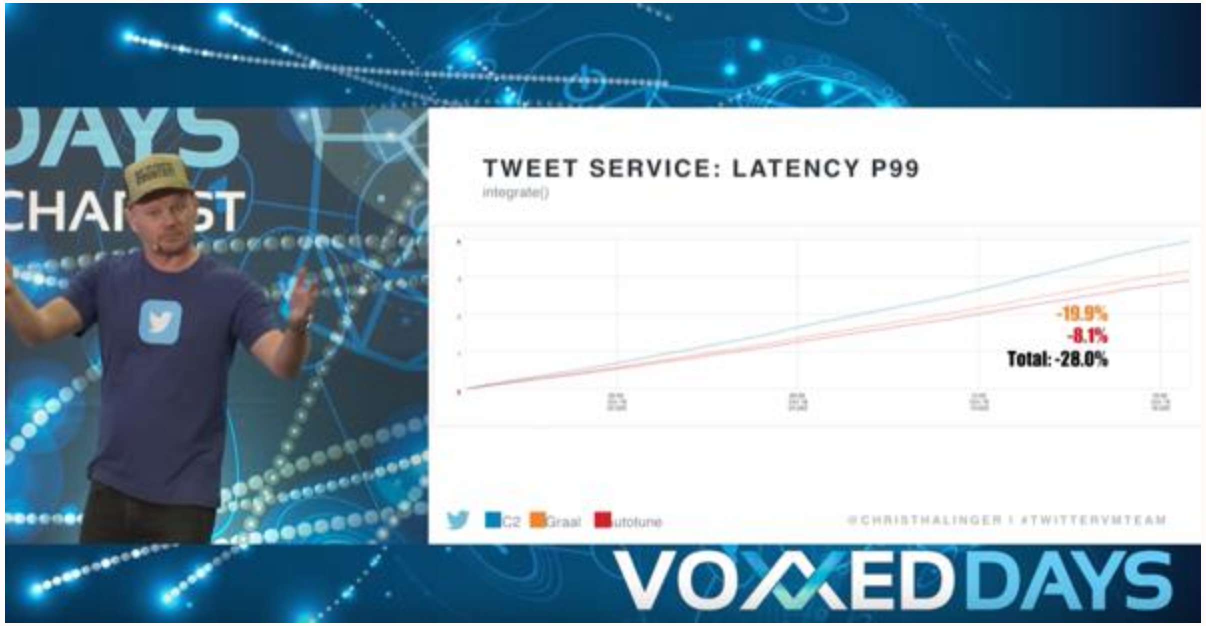
FREE on Oracle Cloud!

GraalVM Open Source

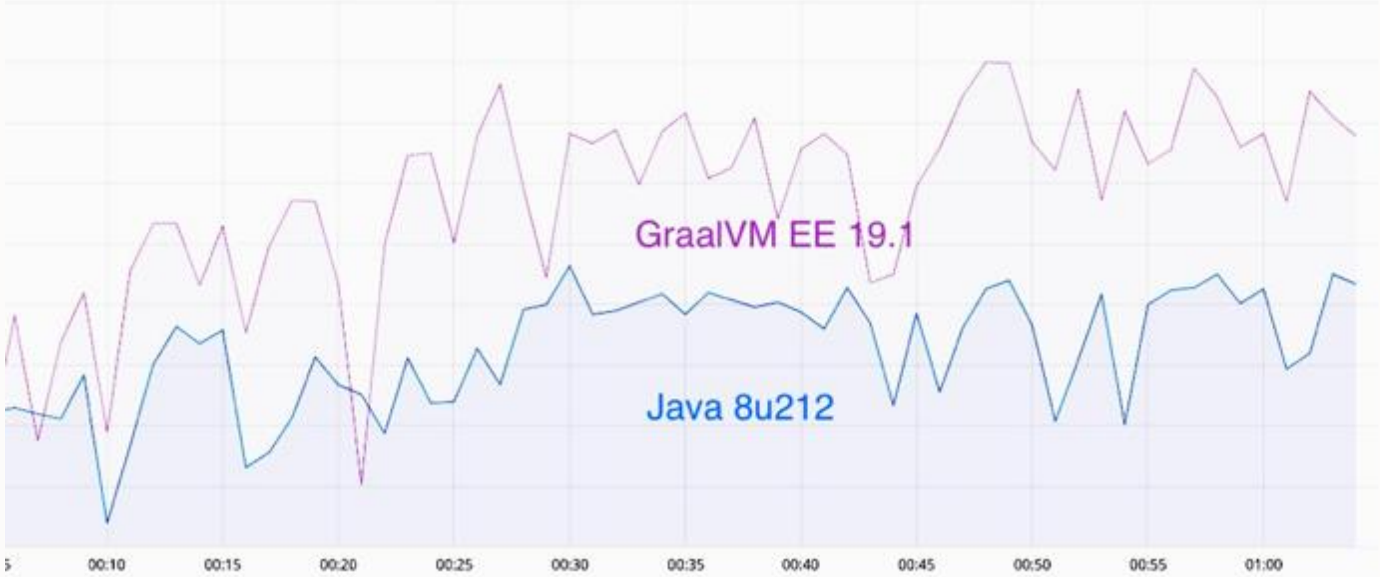


Total: 3,640,000 lines of code

Twitter uses GraalVM compiler in production to run their Scala microservices



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



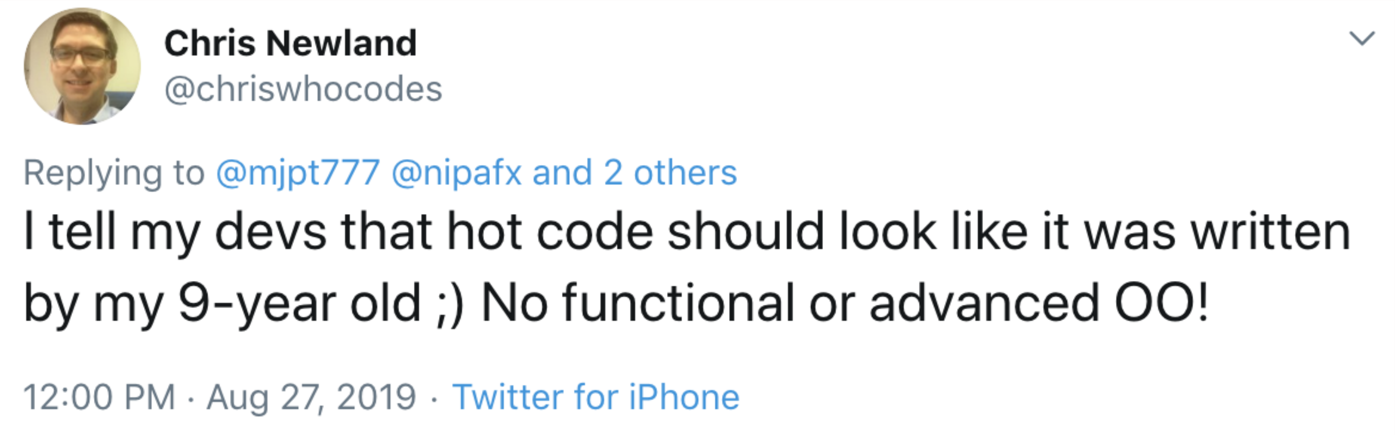
ORACLE
Cloud Infrastructure

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.

NVIDIA®

JIT Performance



Chris Newland
@chriswhocodes

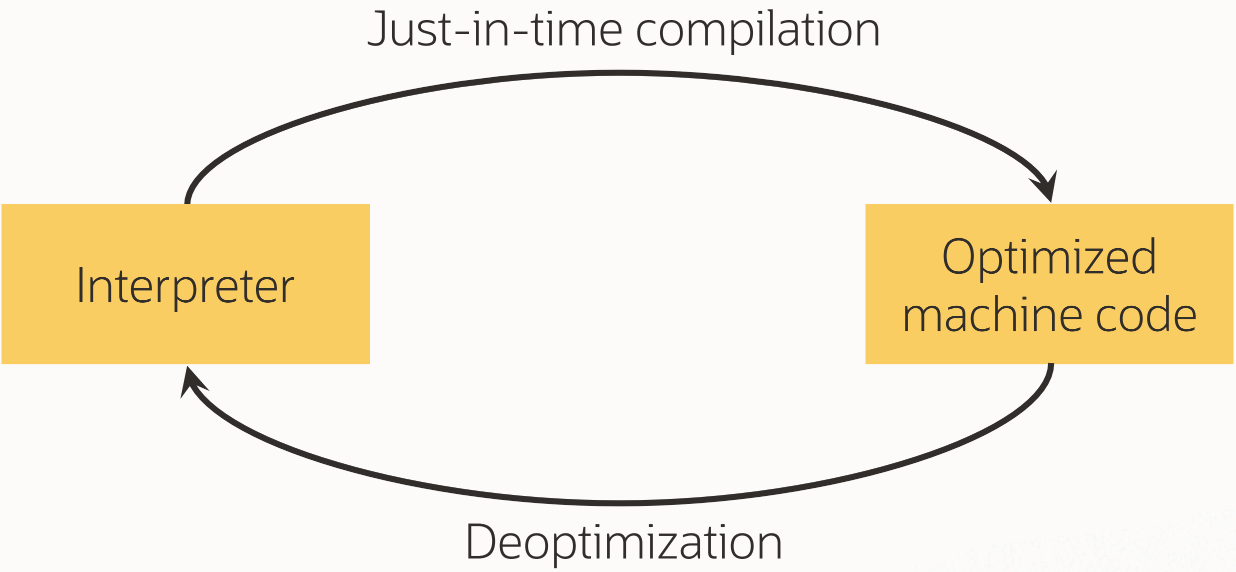
Replying to @mjpt777 @nipafx and 2 others

I tell my devs that hot code should look like it was written by my 9-year old ;) No functional or advanced OO!

12:00 PM · Aug 27, 2019 · Twitter for iPhone

GraalVM Vision:
Abstractions should be without performance regret!

Dynamic Optimization and Deoptimization



Compiler Optimization Terms to Know

- Static Single Assignment Form (SSA Form)
  - The type of intermediate representation used in modern compilers: Phi Function at control flow joints to maintain “one assignment per variable”
- Method Inlining
  - Eliminate call overhead, expand the view of the compiler
- Register Allocation
  - Use fast (but few) processor registers as much as possible
- Constant Folding / Strength Reduction
  - The basic math rules
- Dead Code Elimination
  - Remove code that can never be reached
- Common Subexpression Elimination / Value Numbering / Partial Redundancy Elimination
  - Don’t compute the same thing twice
- Loop Unrolling / Loop Peeling
  - Special focus on frequently executed code
- Escape Analysis
  - Eliminate allocation and synchronization when objects are method local or thread local
- Vectorization
  - Do several loop iterations at once
- Points-to analysis
  - Can two pointers reference the same memory location?

Optimistic Optimizations

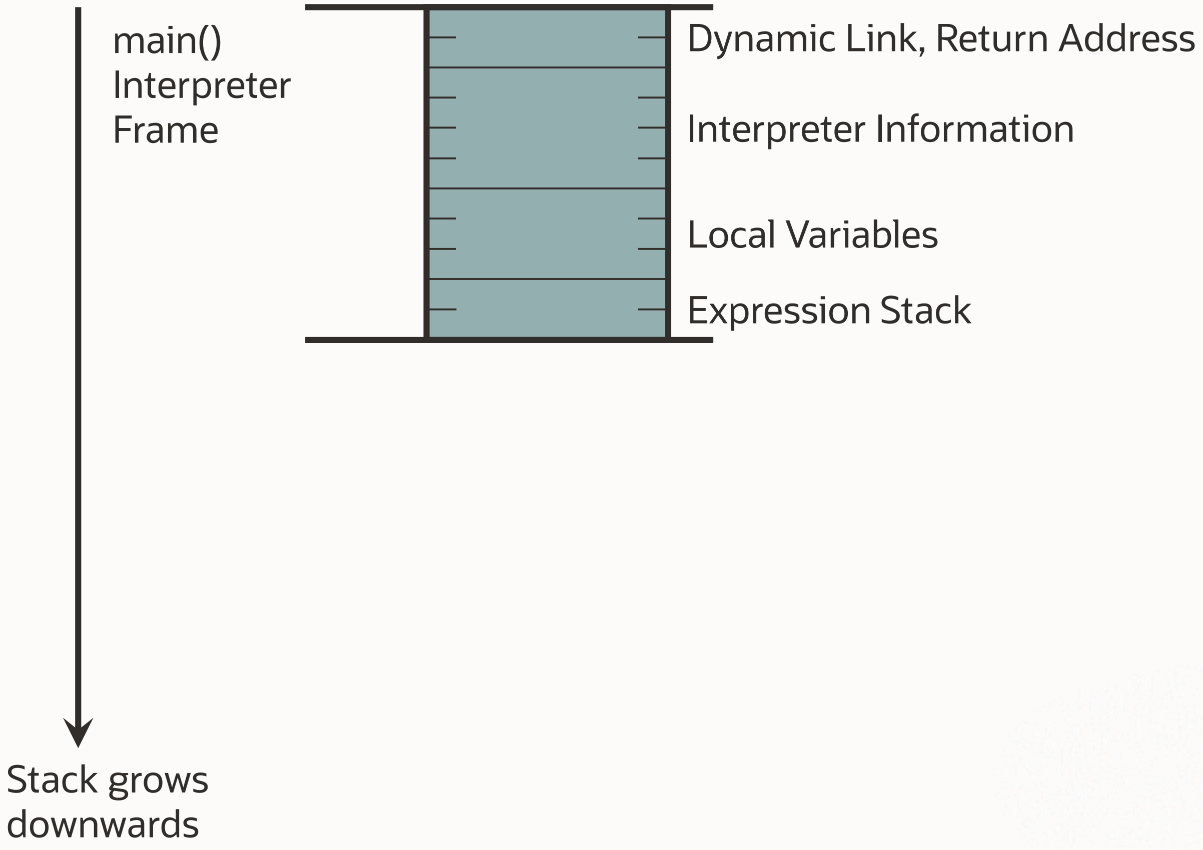
- Inlining of virtual methods
  - Most methods in Java are dynamically bound
  - Class Hierarchy Analysis
  - Inline when only one suitable method exists
- Compilation of foo() when only A loaded
  - Method getX() is inlined
  - Same machine code as direct field access
  - No dynamic type check
- Later loading of class B
  - Discard machine code of foo()
  - Recompile later without inlining
- Deoptimization
  - Switch to interpreter in the middle of foo()
  - Reconstruct interpreter stack frames
  - Expensive, but rare situation
  - Most classes already loaded at first compile

```
void foo() {
  A a = create();
  a.getX();
}
```

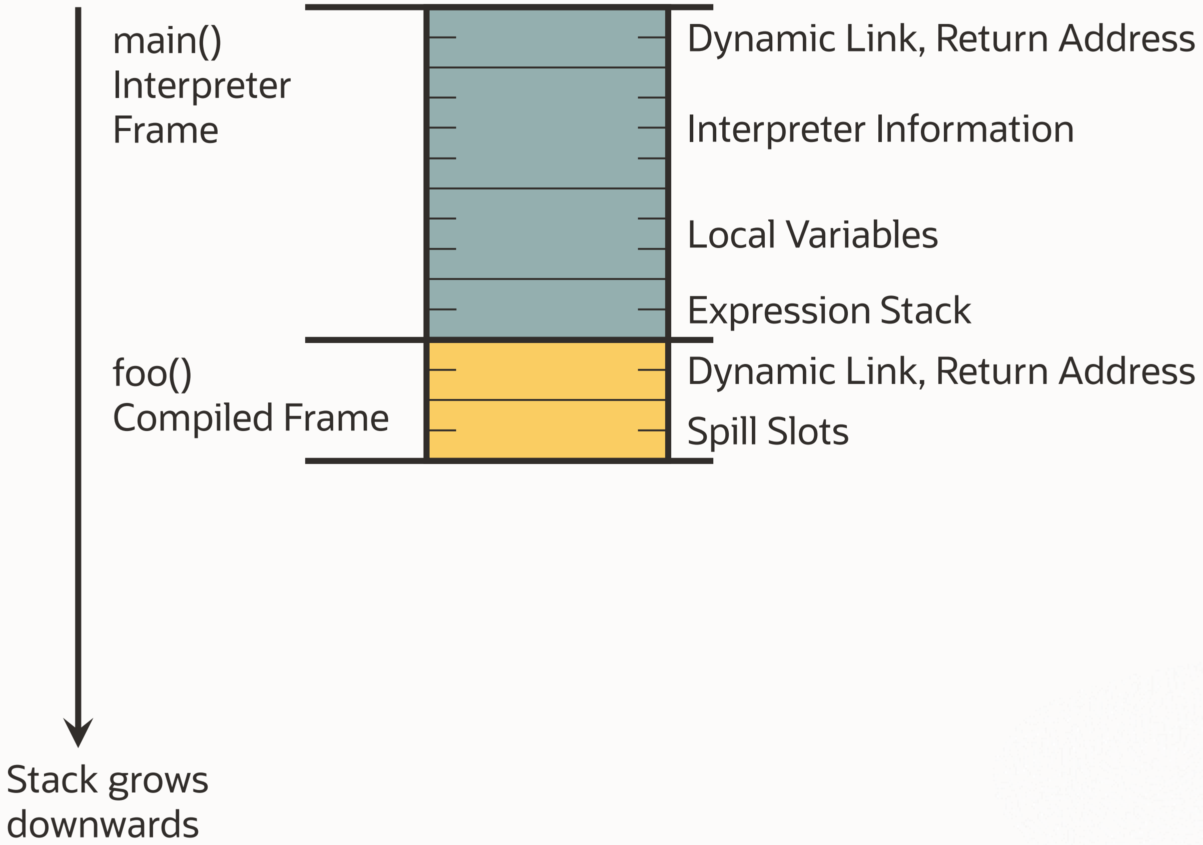
```
class A {
  int x;

  int getX() {
    return x;
  }
}
```

```
class B extends A {
  int getX() {
    return ...
  }
}
```

Deoptimization



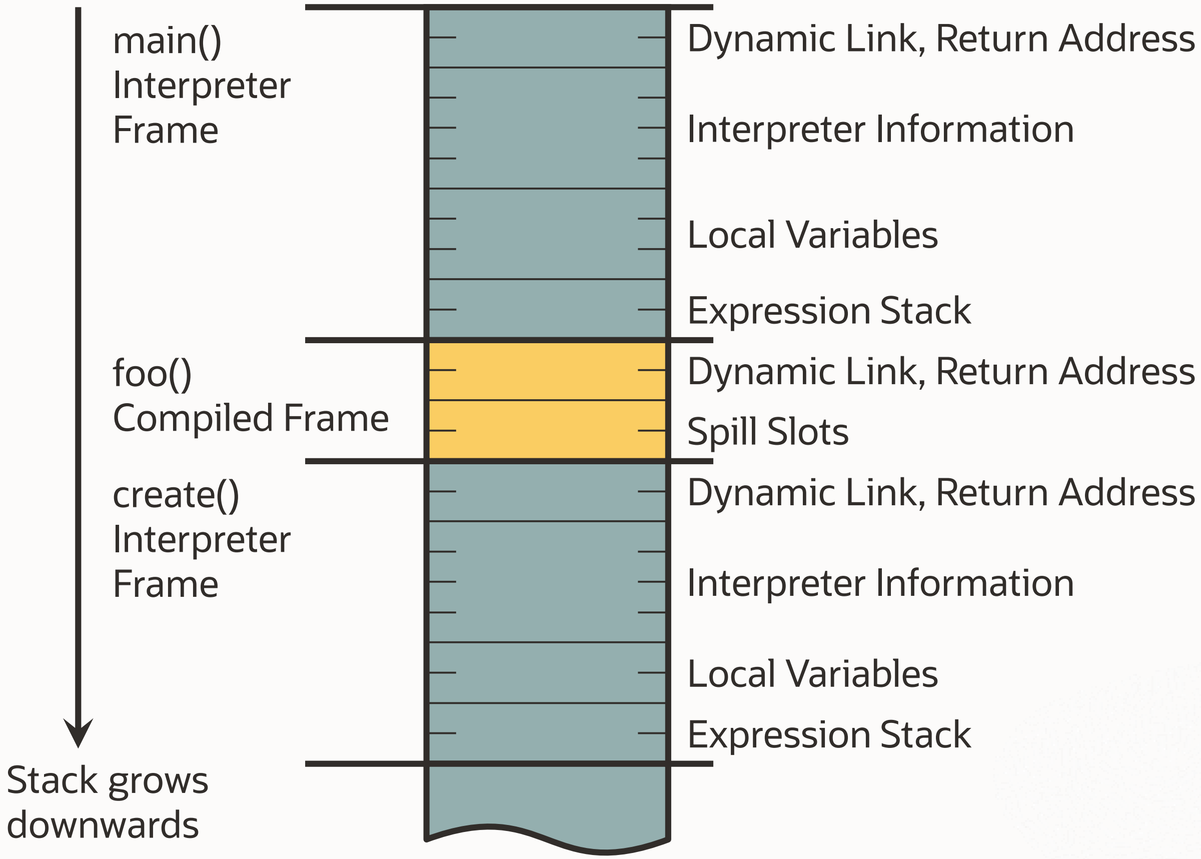
Deoptimization



Machine code for foo():

```
enter
call create
move [eax + 8] -> esi
leave
return
```

Deoptimization



main()
Interpreter
Frame

Dynamic Link, Return Address

Interpreter Information

Local Variables

Expression Stack

foo()
Compiled Frame

Dynamic Link, Return Address

Spill Slots

create()
Interpreter
Frame

Dynamic Link, Return Address

Interpreter Information

Local Variables

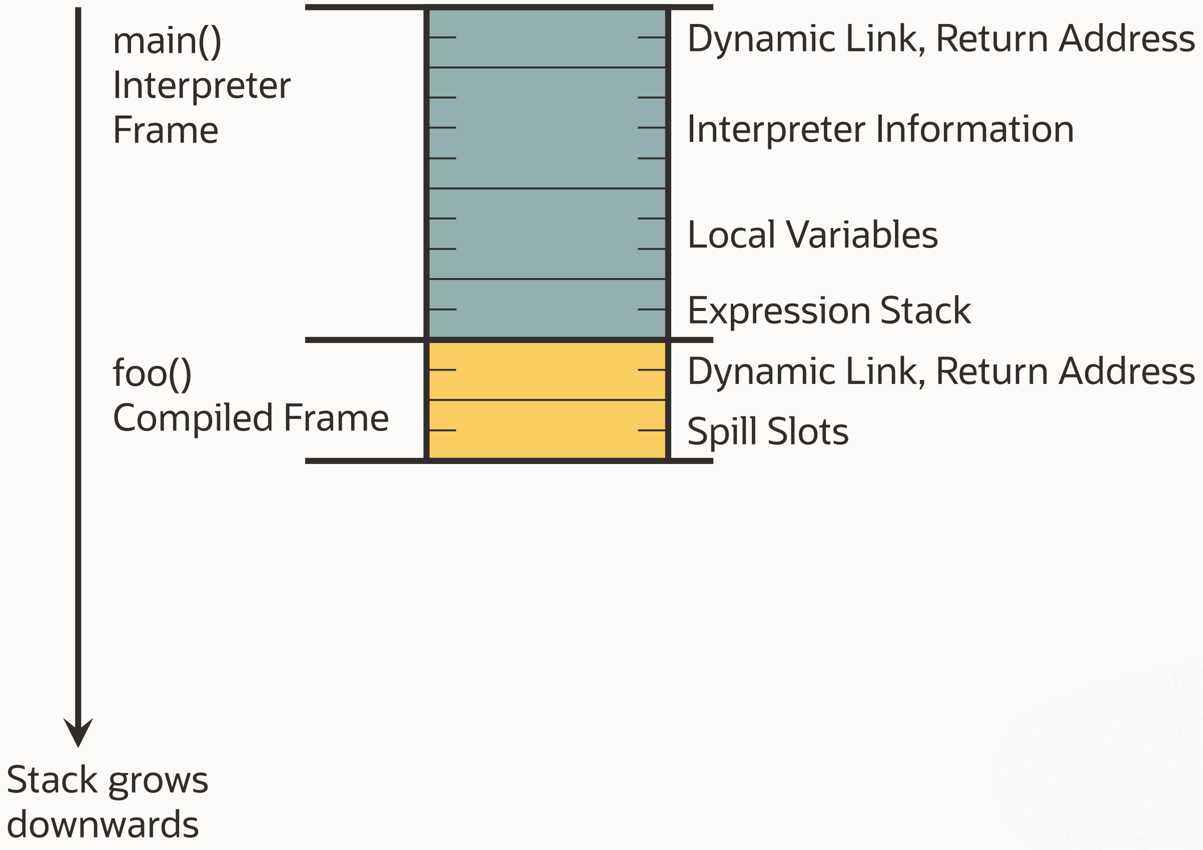
Expression Stack

Stack grows
downwards

Machine code for foo():

```
jump Interpreter
call create
call Deoptimization
leave
return
```

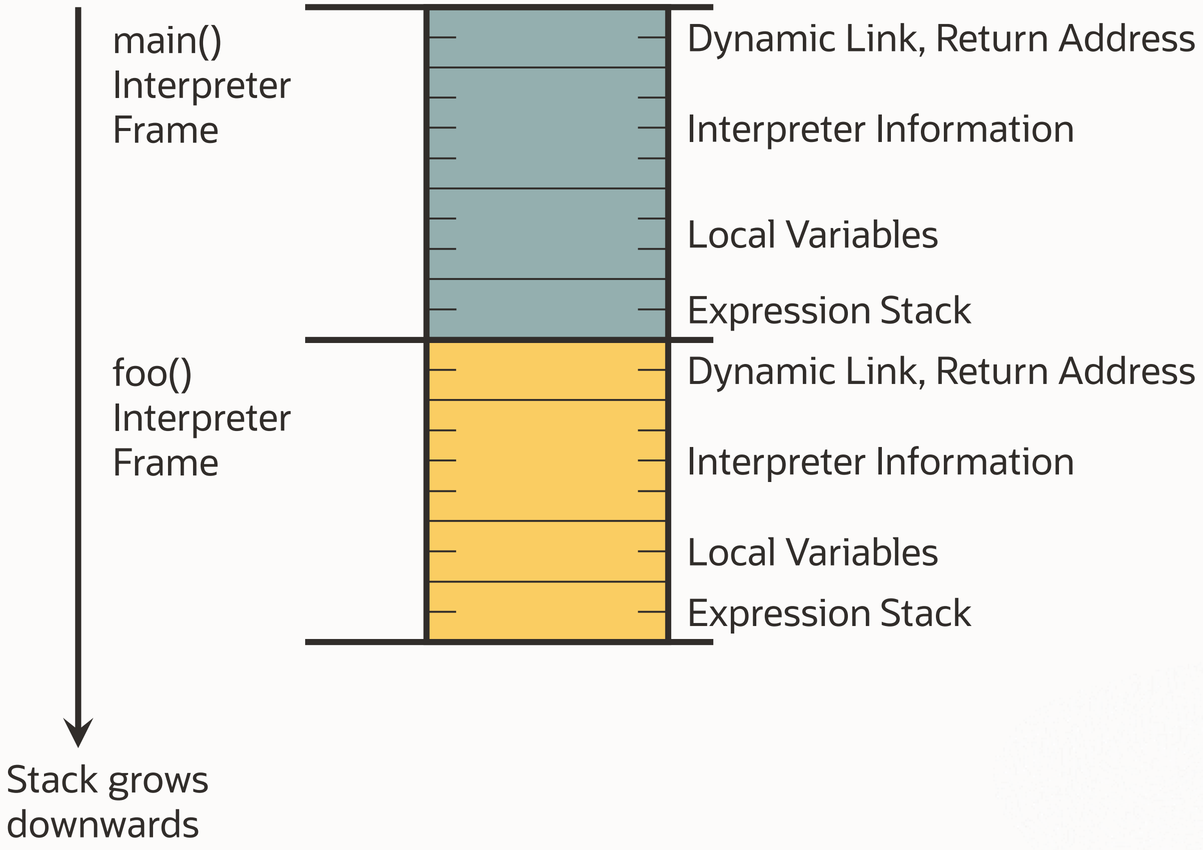
Deoptimization



Machine code for foo():

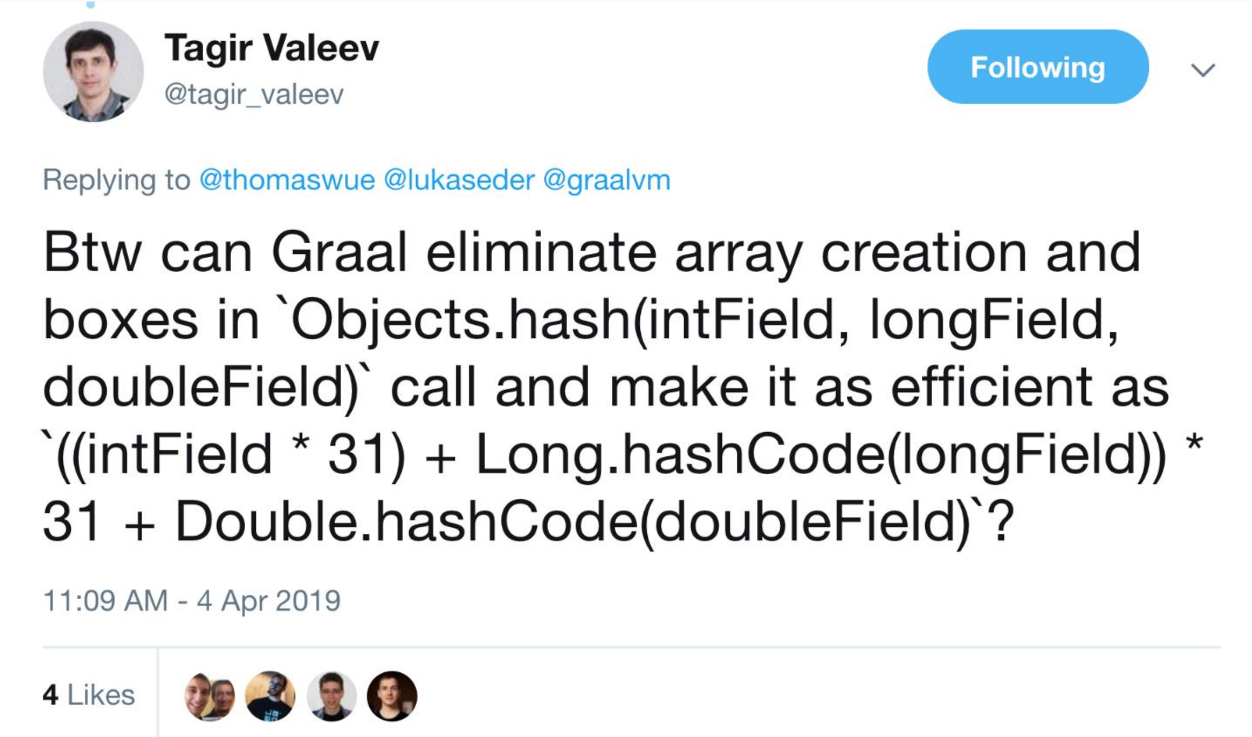
```
jump Interpreter
call create
call Deoptimization
leave
return
```

Deoptimization



Machine code for foo():

```
jump Interpreter
call create
call Deoptimization
leave
return
```



Tagir Valeev
@tagir_valeev

Replying to @thomaswue @lukaseder @graalvm

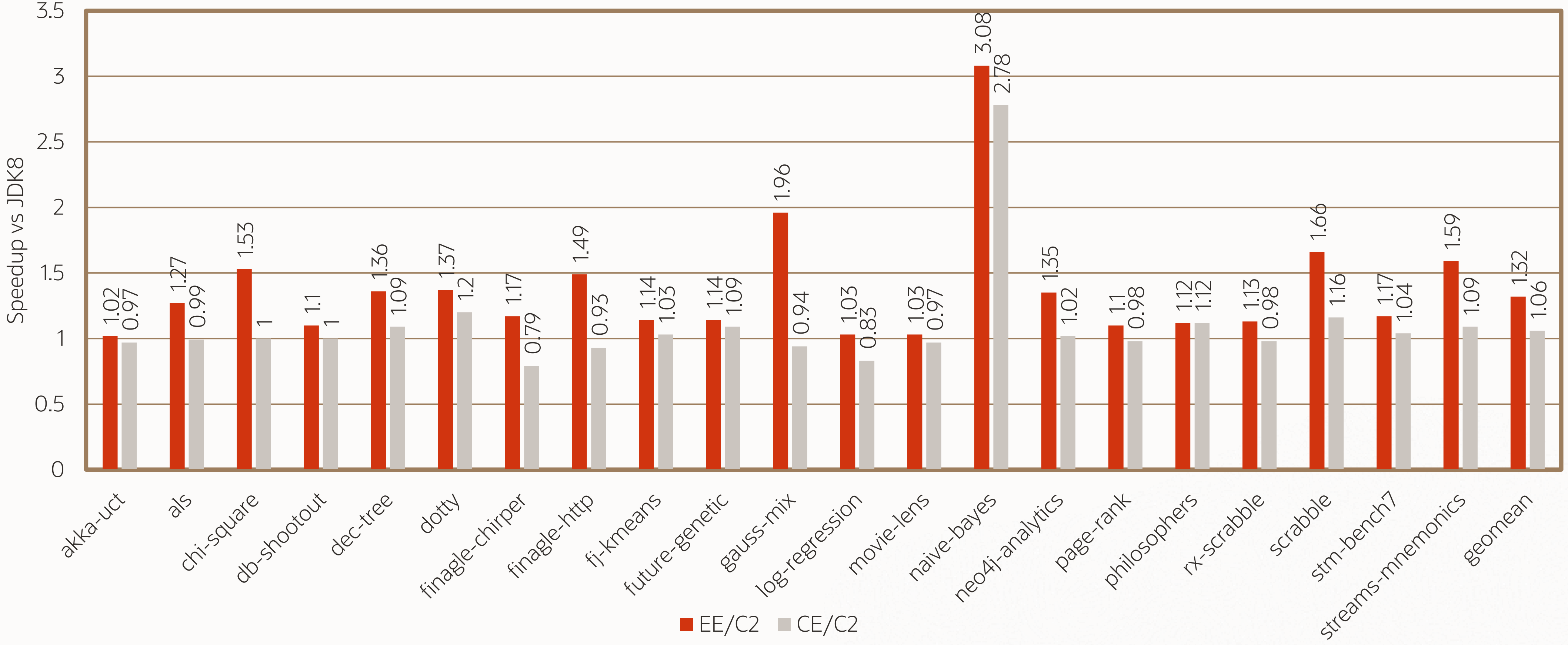
Btw can Graal eliminate array creation and boxes in `Objects.hash(intField, longField, doubleField)` call and make it as efficient as `((intField * 31) + Long.hashCode(longField)) * 31 + Double.hashCode(doubleField)`?

11:09 AM - 4 Apr 2019

4 Likes

Answer:
Of course it can!

GraalVM JIT Performance: Renaissance.dev



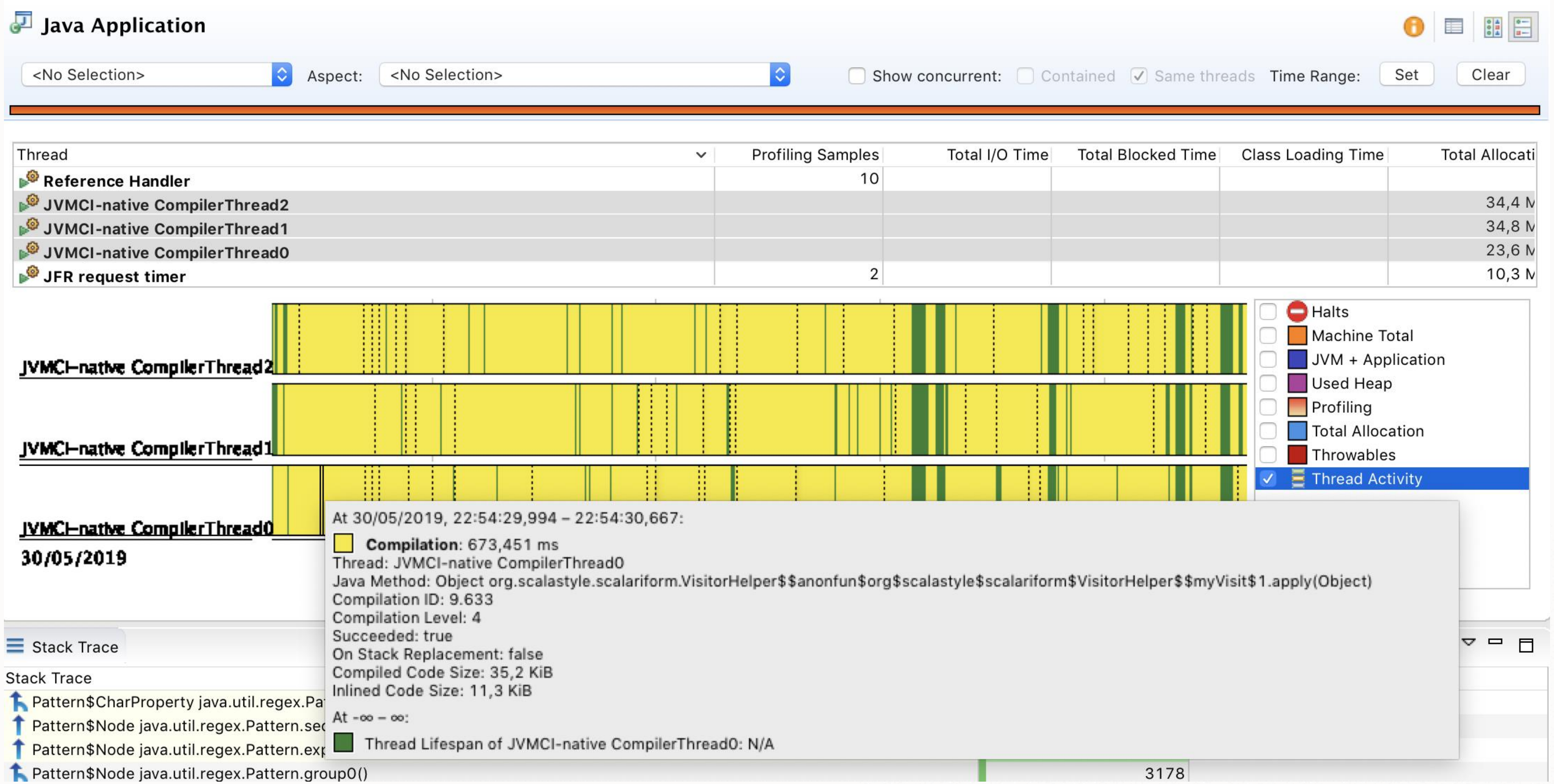
More Benchmarks…

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.

Java Flight Recorder Compilation Information



Thread	Profiling Samples	Total I/O Time	Total Blocked Time	Class Loading Time	Total Allocati
Reference Handler	10				
JVMCI-native CompilerThread2					34,4 M
JVMCI-native CompilerThread1					34,8 M
JVMCI-native CompilerThread0					23,6 M
JFR request timer	2				10,3 M

At 30/05/2019, 22:54:29,994 – 22:54:30,667:

🟨 **Compilation**: 673,451 ms

Thread: JVMCI-native CompilerThread0

Java Method: Object org.scalastyle.scalariform.VisitorHelper$$anonfunorgscalastyle$scalariform$VisitorHelper$$myVisit$1.apply(Object)

Compilation ID: 9.633

Compilation Level: 4

Succeeded: true

On Stack Replacement: false

Compiled Code Size: 35,2 KiB

Inlined Code Size: 11,3 KiB

At -∞ – ∞:

🟩 Thread Lifespan of JVMCI-native CompilerThread0: N/A

Stack Trace

Stack Trace

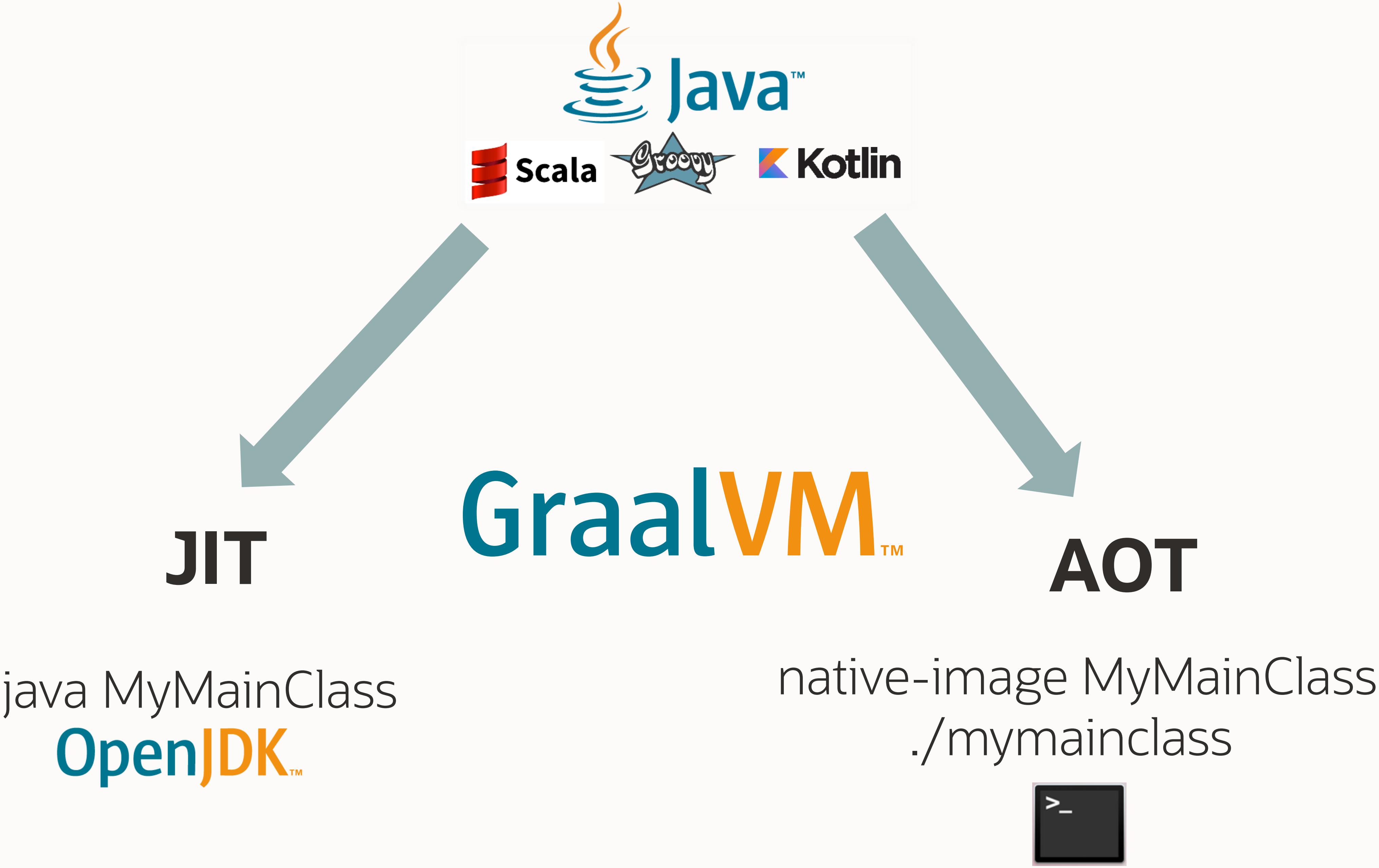
Pattern$CharProperty java.util.regex.Pa

Pattern$Node java.util.regex.Pattern.seq

Pattern$Node java.util.regex.Pattern.exp

Pattern$Node java.util.regex.Pattern.group0() 3178

GraalVM Native Image



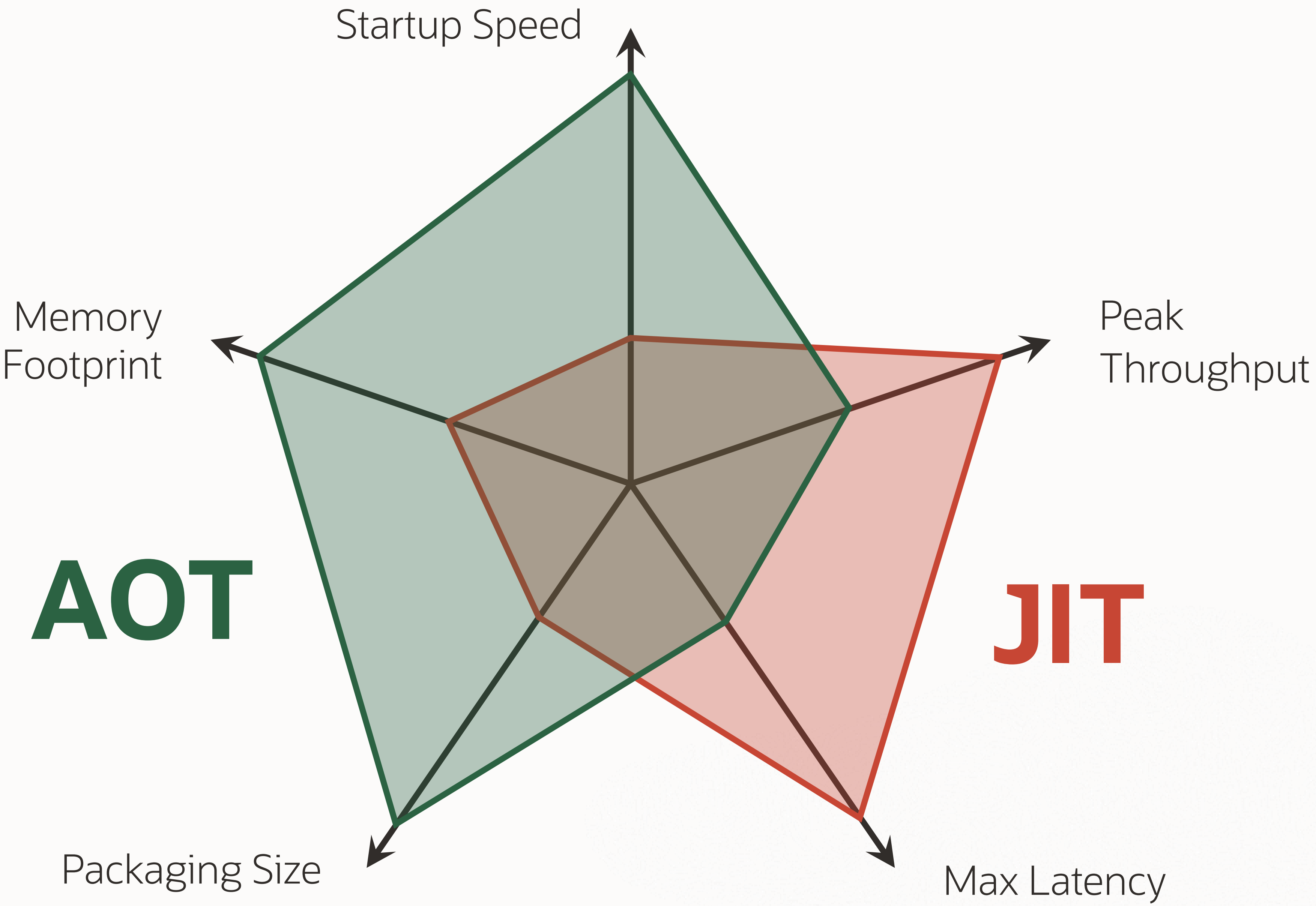
JIT

```
java MyMainClass
```

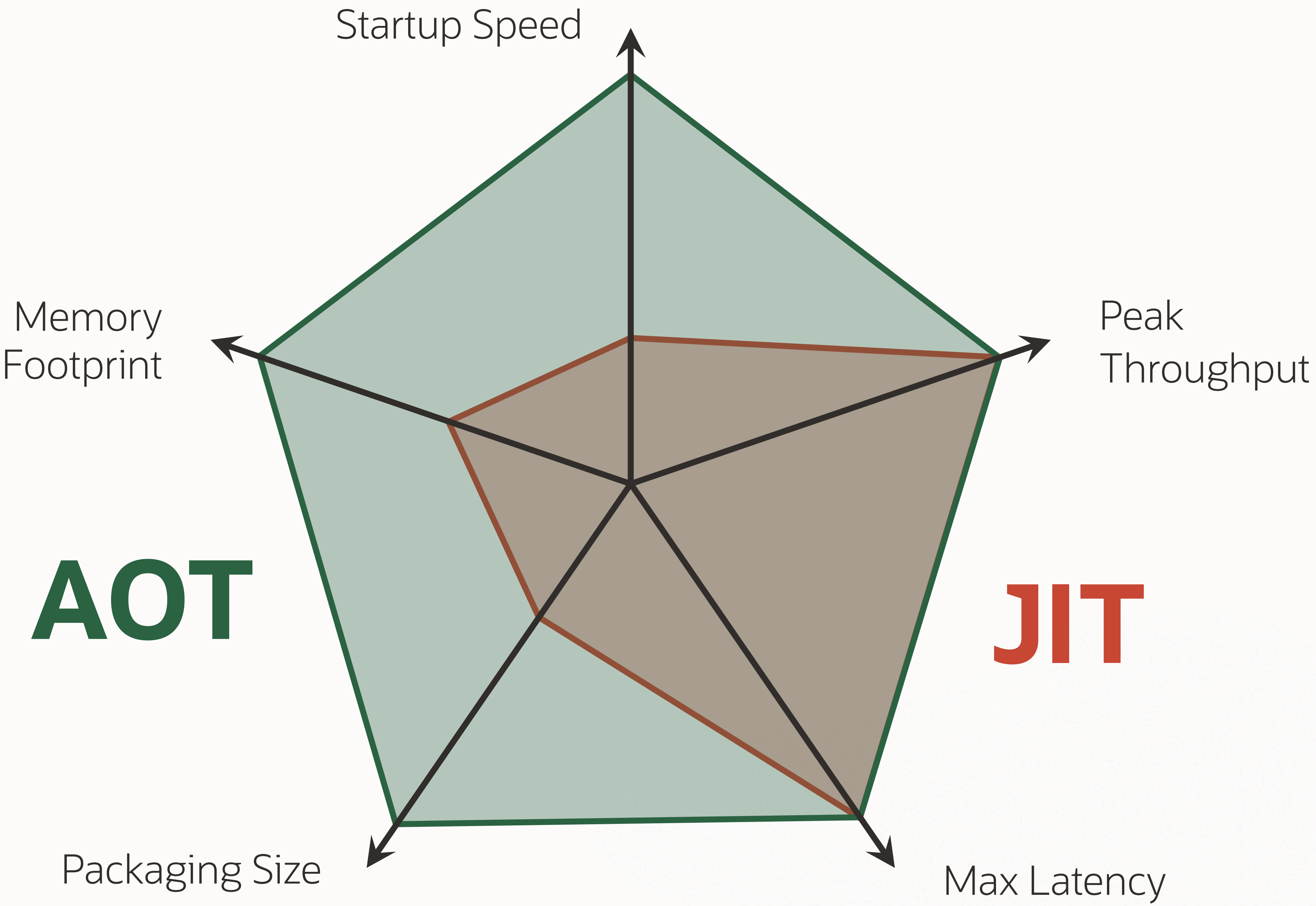
AOT

```
native-image MyMainClass
./mymainclass
```

Currently


Startup Speed
Memory
Footprint
Peak
Throughput
AOT
JIT
Packaging Size
Max Latency

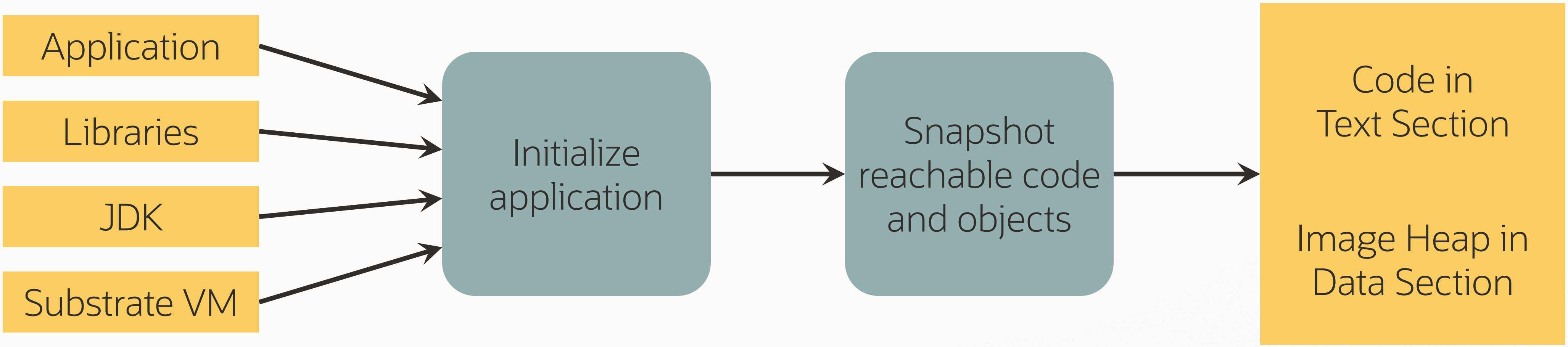
Goal


Startup Speed
Memory Footprint
Peak Throughput
AOT
JIT
Packaging Size
Max Latency

Native Image: Principle

Input:
All classes from application, libraries, and VM

Output:
Native executable



Application

Libraries

JDK

Substrate VM

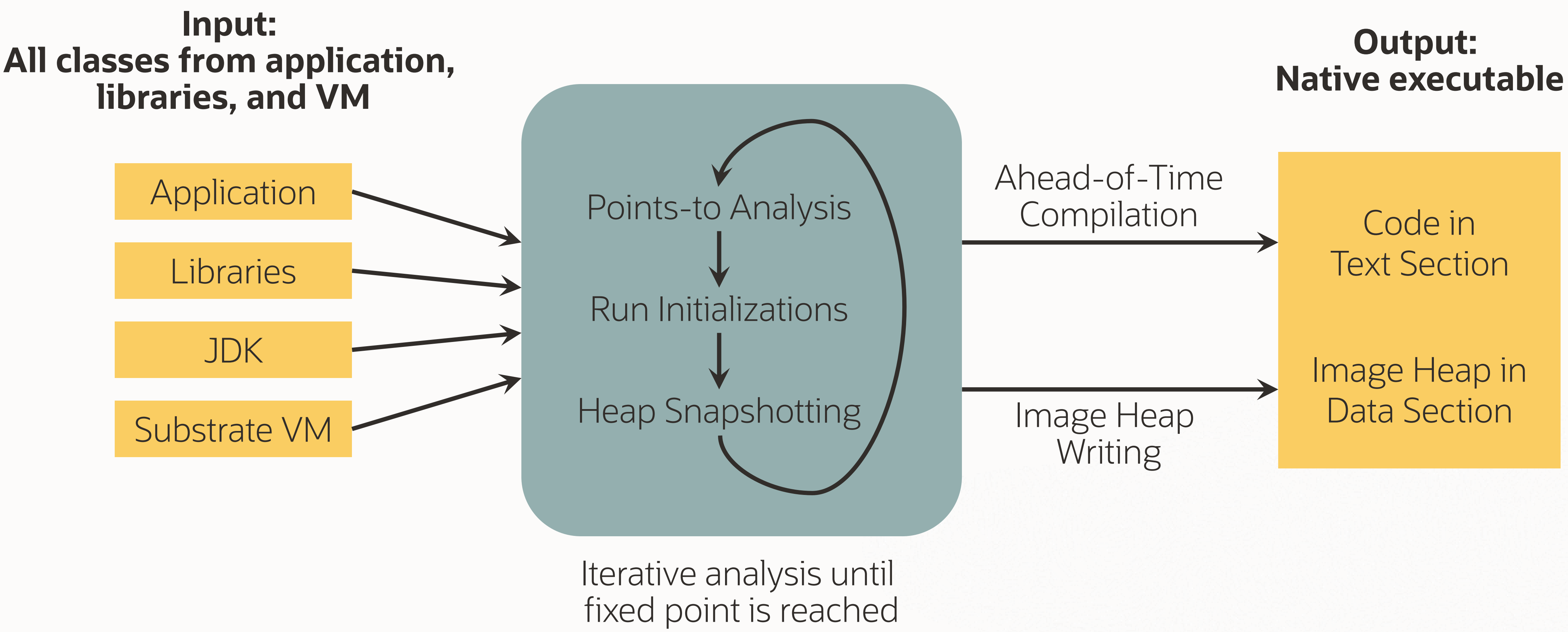
Initialize application

Snapshot reachable code and objects

Code in Text Section

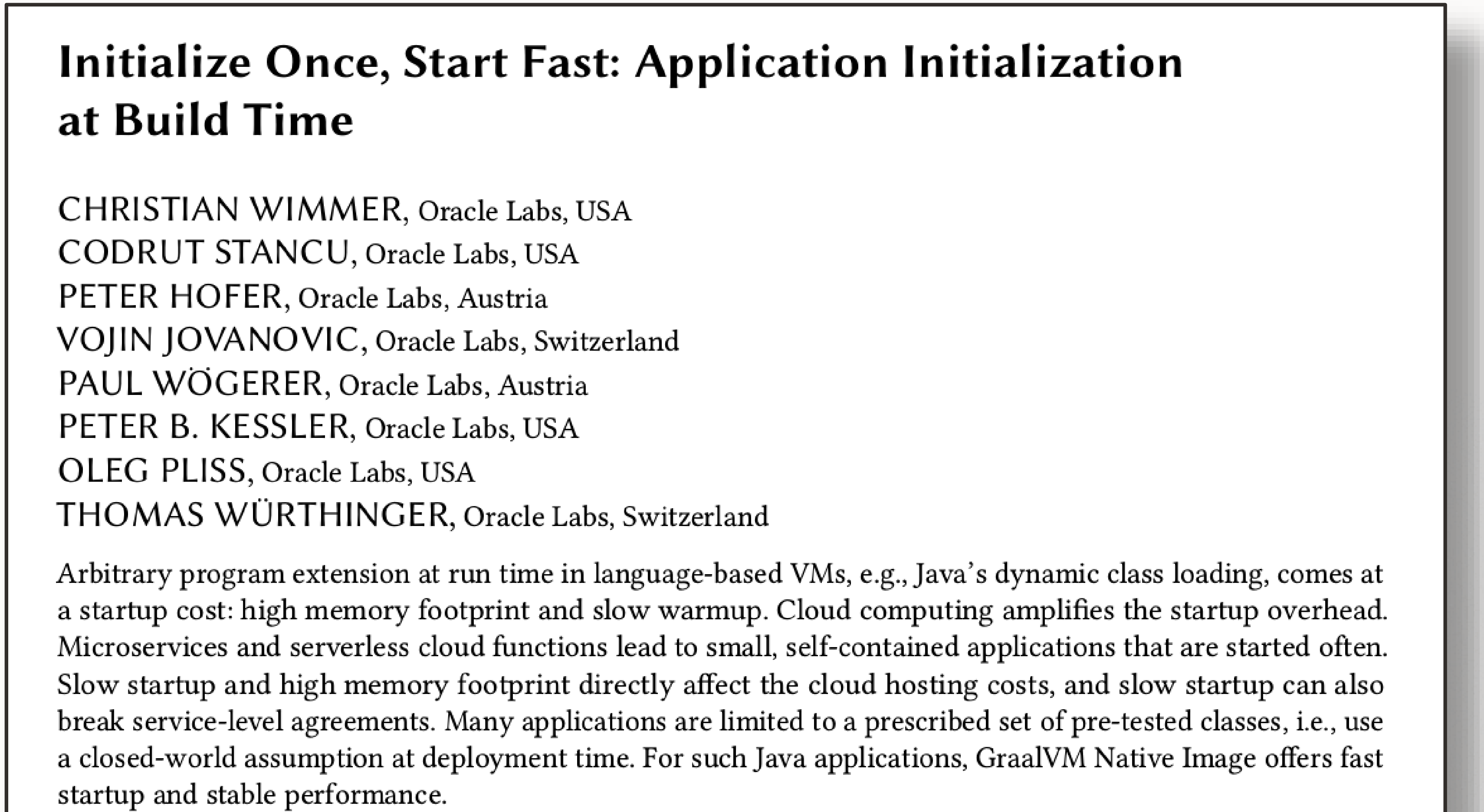
Image Heap in Data Section

Native Image: Details



Paper with Details, Examples, Benchmarks

http://www.christianwimmer.at/Publications/Wimmer19a/Wimmer19a.pdf



Initialize Once, Start Fast: Application Initialization at Build Time

CHRISTIAN WIMMER, Oracle Labs, USA
CODRUT STANCU, Oracle Labs, USA
PETER HOFER, Oracle Labs, Austria
VOJIN JOVANOVIC, Oracle Labs, Switzerland
PAUL WÖGERER, Oracle Labs, Austria
PETER B. KESSLER, Oracle Labs, USA
OLEG PLISS, Oracle Labs, USA
THOMAS WÜRTHINGER, Oracle Labs, Switzerland

Arbitrary program extension at run time in language-based VMs, e.g., Java's dynamic class loading, comes at a startup cost: high memory footprint and slow warmup. Cloud computing amplifies the startup overhead. Microservices and serverless cloud functions lead to small, self-contained applications that are started often. Slow startup and high memory footprint directly affect the cloud hosting costs, and slow startup can also break service-level agreements. Many applications are limited to a prescribed set of pre-tested classes, i.e., use a closed-world assumption at deployment time. For such Java applications, GraalVM Native Image offers fast startup and stable performance.

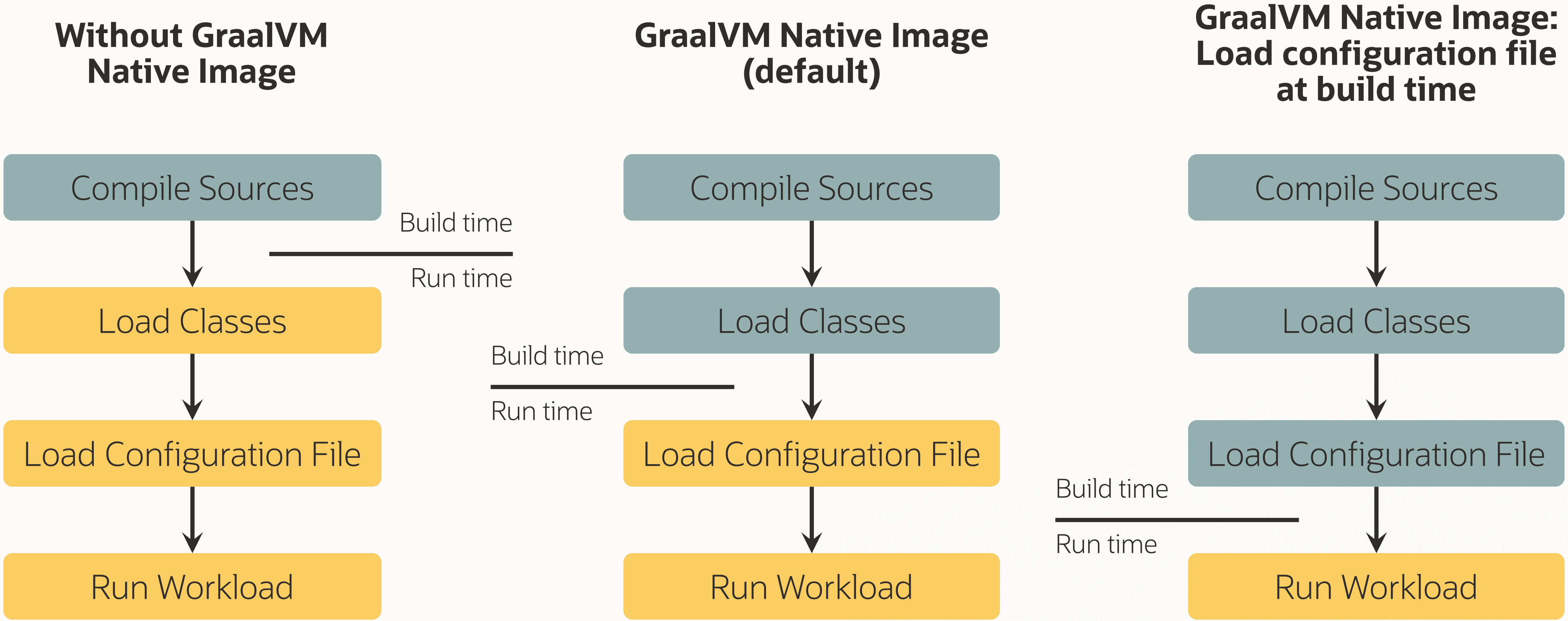
Closed World Assumption

- 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, …
  - That’s what this talk is about
- No loading of new classes at run time

Image Heap

- Execution at run time starts with an initial heap: the “image heap”
  - Objects are allocated in the Java VM that runs the image generator
  - Heap snapshotting gathers all objects that are reachable at run time
- Do things once at build time instead at every application startup
  - Class initializers, initializers for static and static final fields
  - Explicit code that is part of a so-called “Feature”
- Examples for objects in the image heap
  - java.lang.Class objects, Enum constants

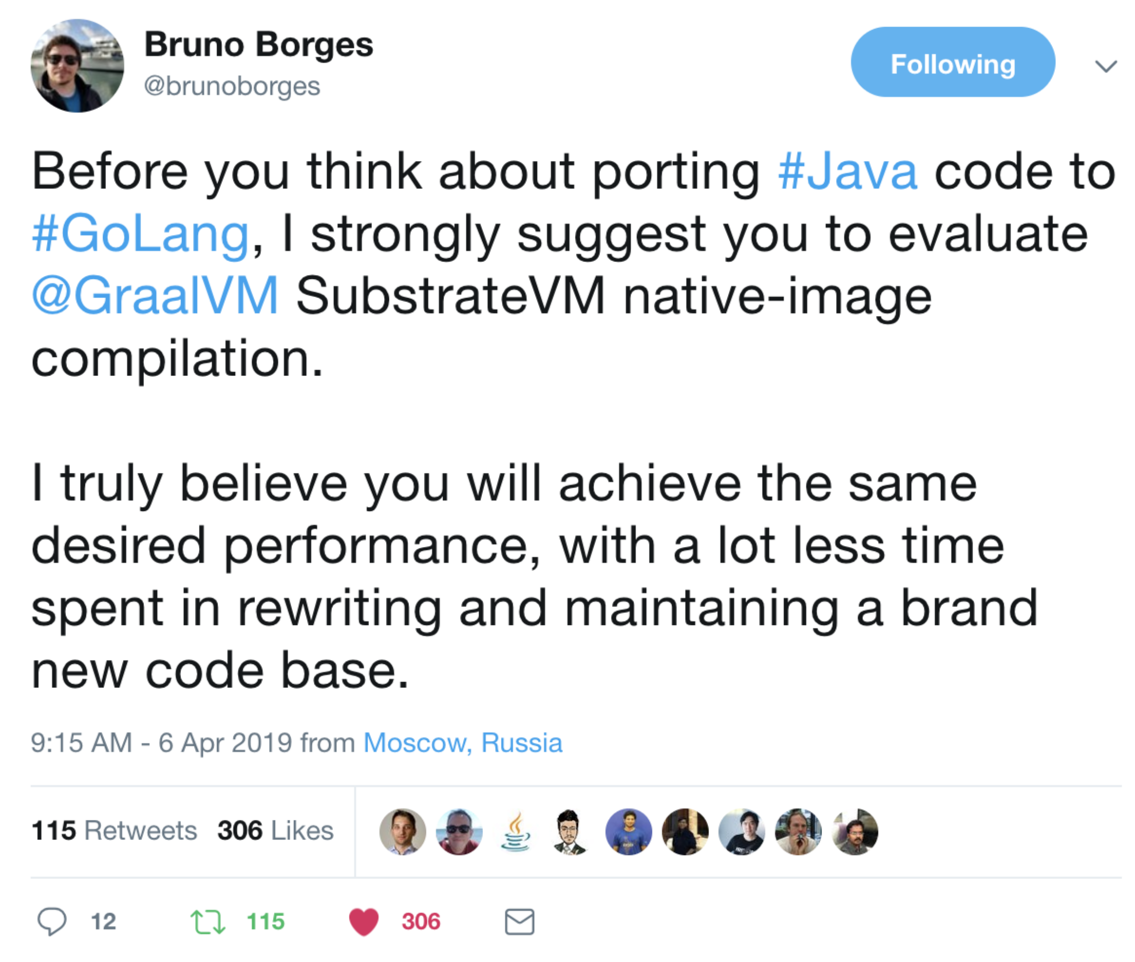
Benefits of the Image Heap



Get VMs Ready for the Cloud and Microservices

Important evaluation metrics:

- Startup time
- Memory footprint
- Peak requests per MByte-second



Bruno Borges
@brunoborges

Before you think about porting #Java code to #GoLang, I strongly suggest you to evaluate @GraalVM SubstrateVM native-image compilation.

I truly believe you will achieve the same desired performance, with a lot less time spent in rewriting and maintaining a brand new code base.

9:15 AM - 6 Apr 2019 from Moscow, Russia

115 Retweets 306 Likes

Bruno Borges,
Microsoft Azure Advocate

Java Microservice Frameworks with GraalVM Native Image Support

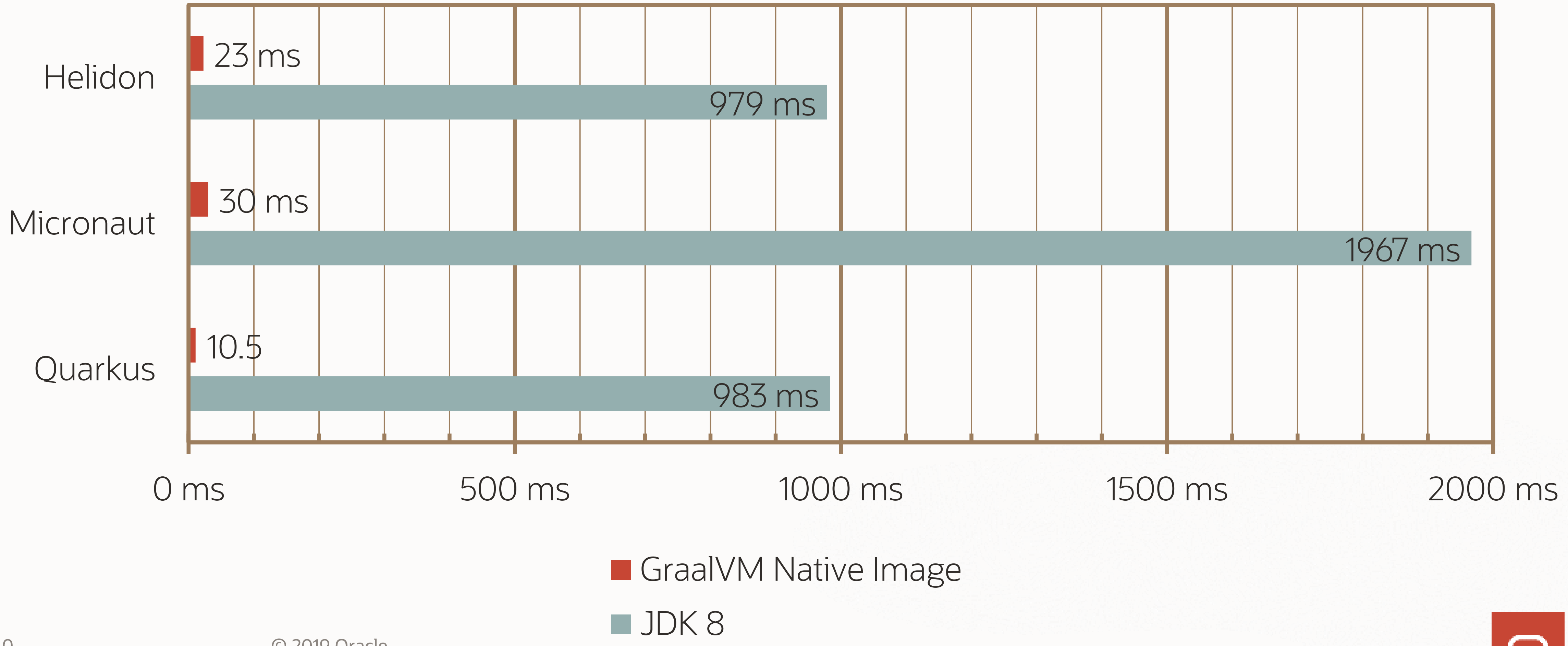
https://micronaut.io

https://helidon.io

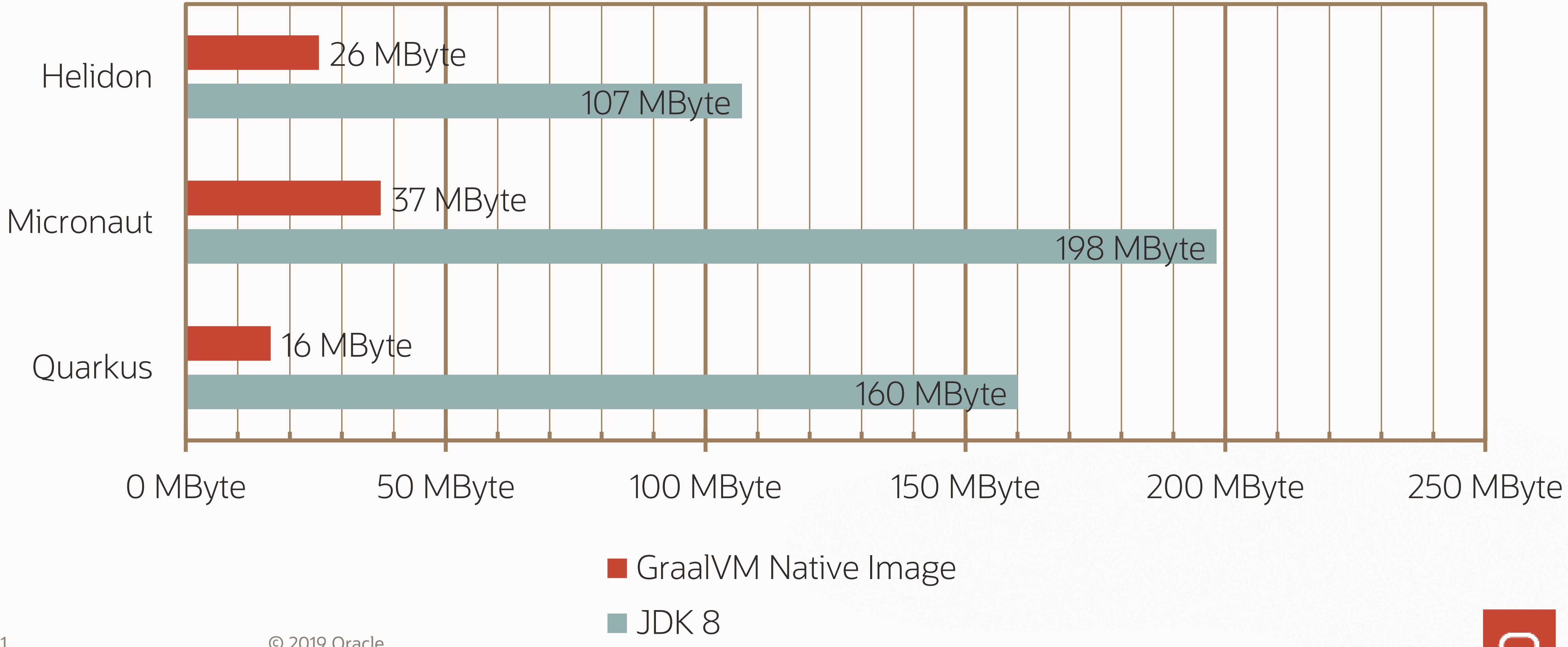
https://quarkus.io

Soon also Spring support
https://github.com/spring-projects-experimental/spring-graal-native

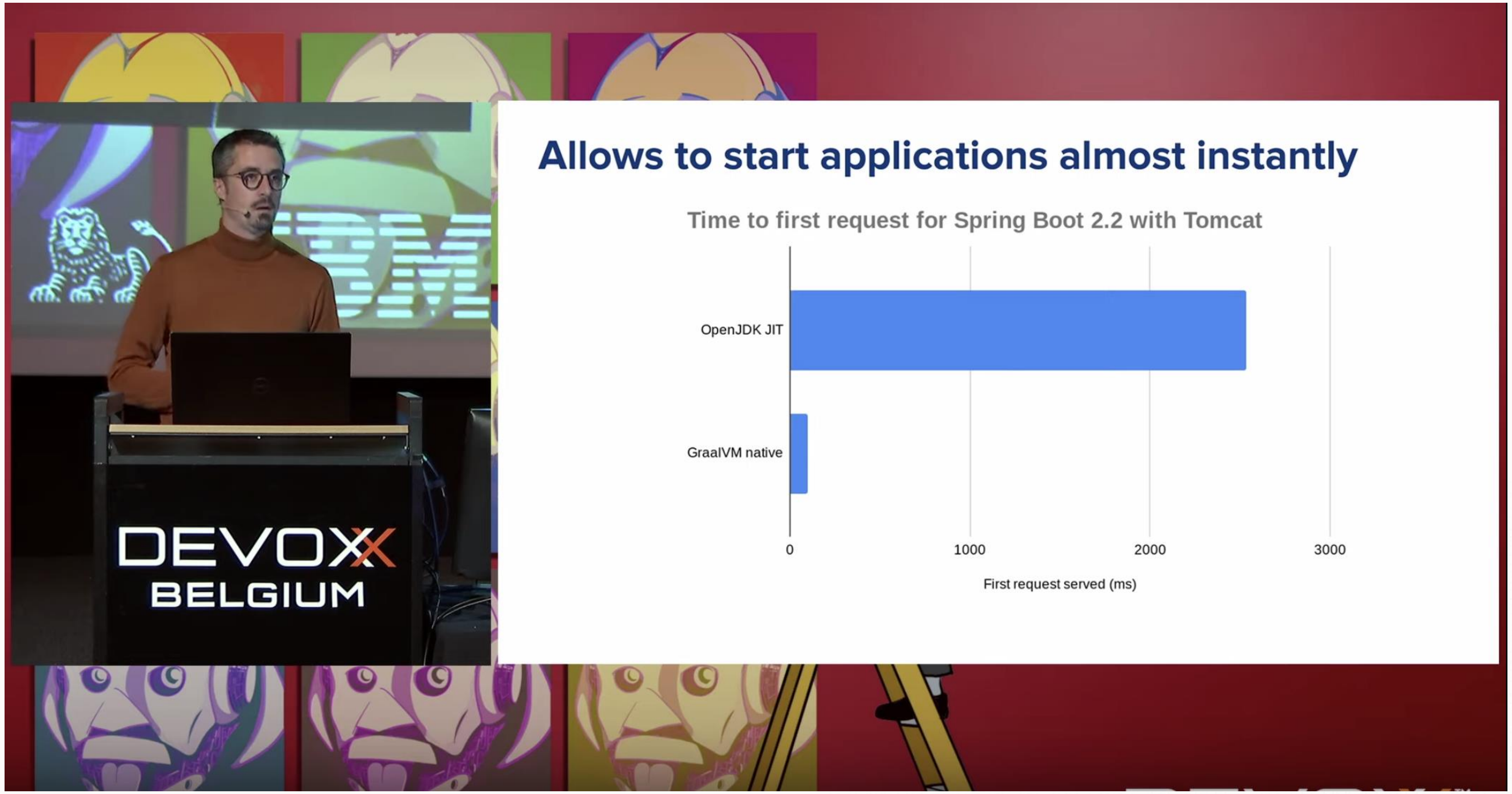
Startup Time



Memory Footprint



Spring Boot Applications as GraalVM Native Images



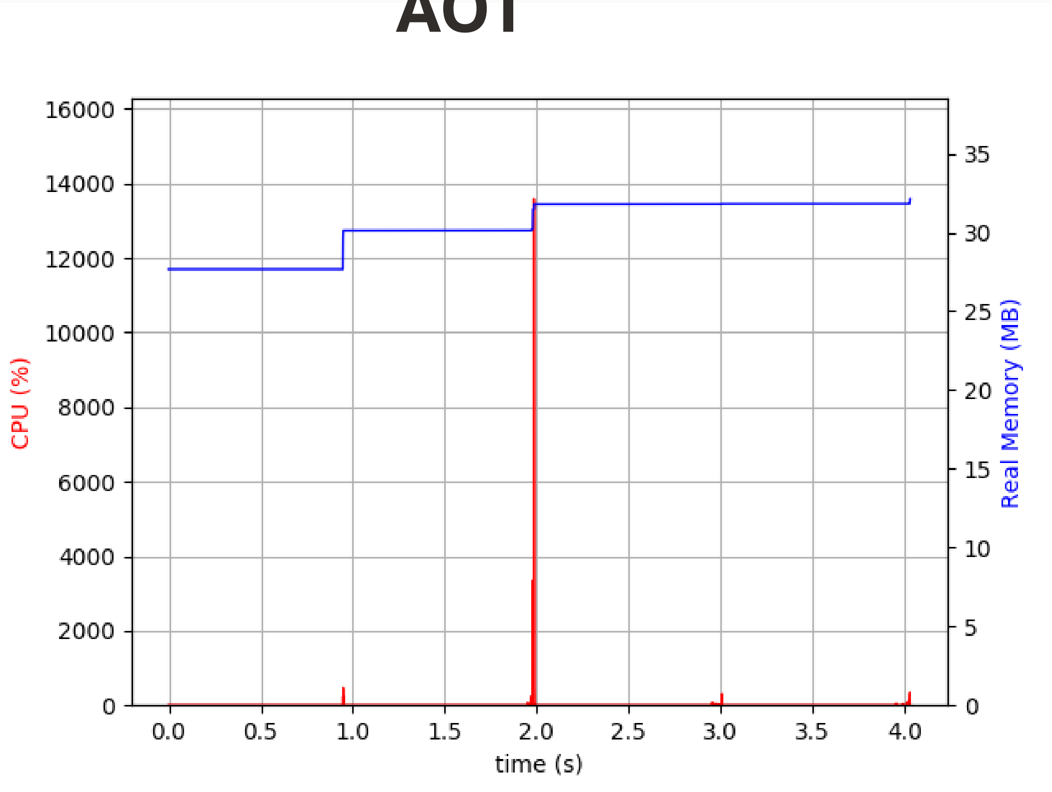
Allows to start applications almost instantly

Time to first request for Spring Boot 2.2 with Tomcat
OpenJDK JIT
GraalVM native
0
1000
2000
3000
First request served (ms)
DEVOXX
BELGIUM

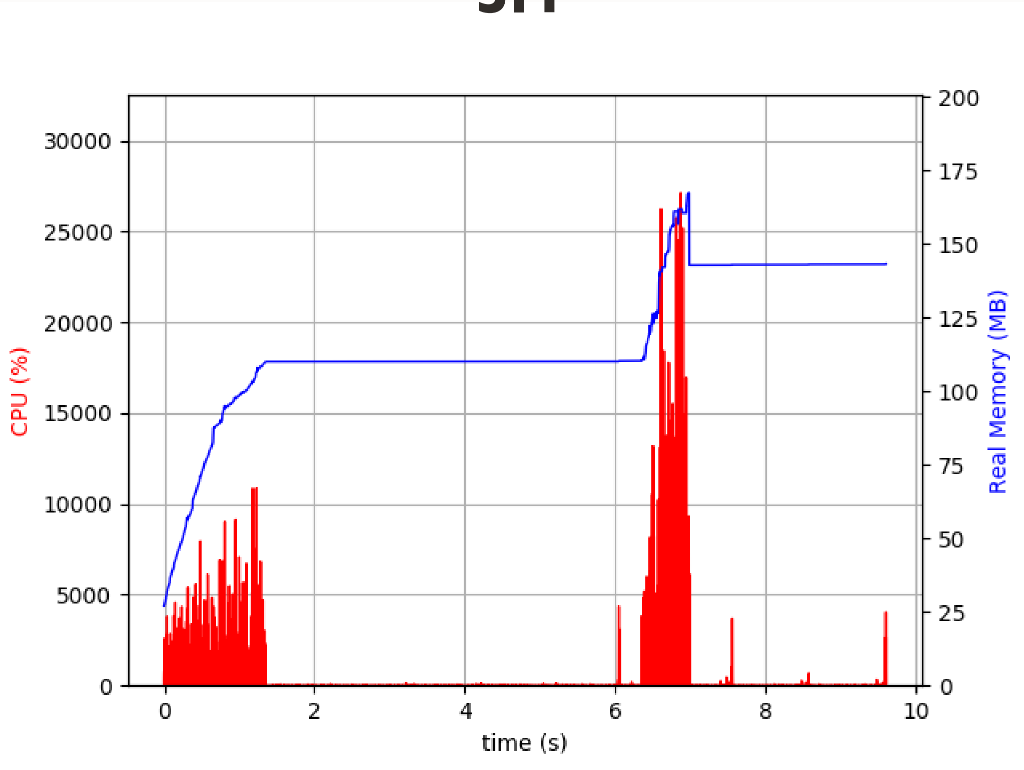
https://www.youtube.com/watch?v=3eoAxphAUIg

Starting up and serving 3 requests (Micronaut example)

AOT

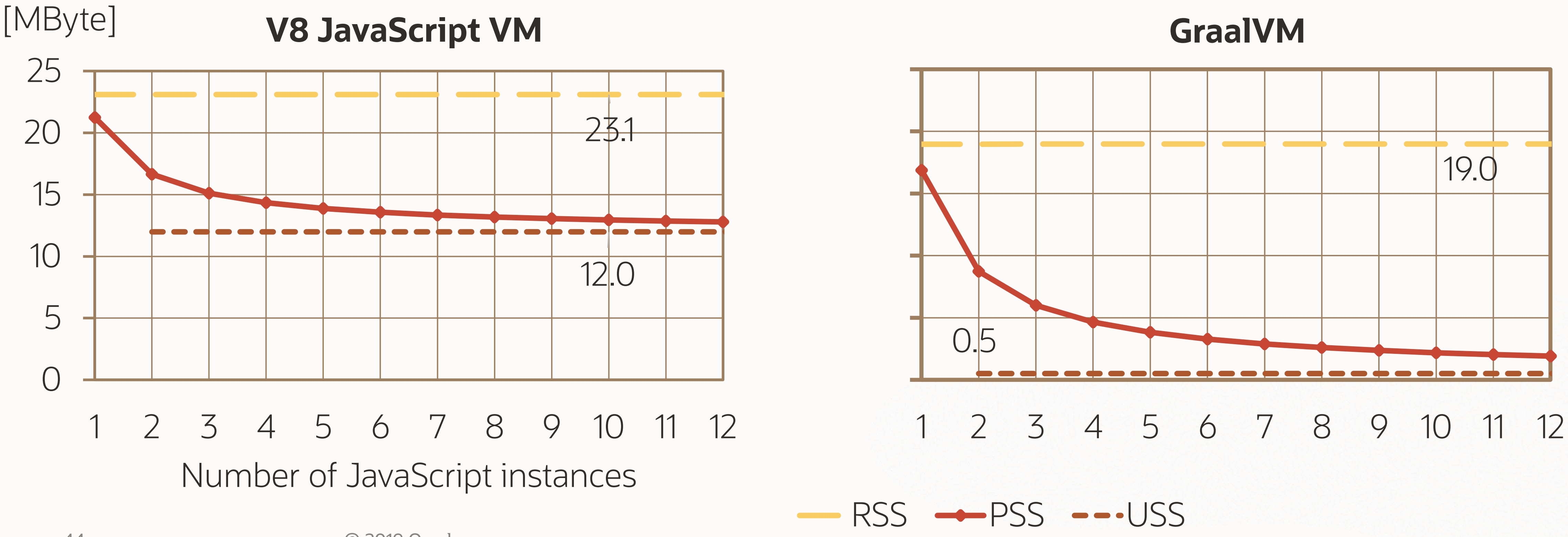


JIT



JavaScript Memory Footprint: V8 vs. GraalVM

- Memory for the first JavaScript instance: **23 MByte vs. 19 MByte**
- Memory for each additional JavaScript instance: **12 MByte vs. 0.5 MByte**



Native Image vs. Java HotSpot VM

- Use GraalVM Native Image when
  - Startup time matters
  - Memory footprint matters
    - Small to medium-sized heaps (100 MByte – a few GByte)
  - All code is known ahead of time

- Use Java HotSpot VM when
  - Heaps size is large
    - Multiple GByte – TByte heap size
  - Classes are only known at run time

Jump Start Your Project

- How do I know quickly if my application will run as a native image?
- Disable fallback image generation
  - --no-fallback
- Report unsupported features at run time
  - --report-unsupported-elements-at-runtime
- Allow incomplete class path: throw linking errors at run time
  - --allow-incomplete-classpath
- Trace reflection, JNI, resource, … usage on Java HotSpot VM
  - java -agentlib:native-image-agent=config-output-dir=META-INF/native-image …
- Initialize all application classes at run time: default since GraalVM 19.0

Incomplete Classpath

- Default: image build fails when a reachable class is missing
  - Guarantees no linking errors at run time
  - We believe that is a desirable goal
- But some applications have not all dependencies on the class path
  - Missing optional dependencies of libraries
  - Static analysis not always precise enough to exclude such code
- Solution: support incomplete class path
  - Throw linking errors at run time
  - Command line option: --allow-incomplete-classpath

Reflection and JNI

- Need configuration at image build time
  - Classes, methods, and fields that are reflectively visible
  - Necessary to keep the metadata small and to avoid too conservative points-to analysis

Example: Gson library to serialize / deserialize Java objects

Data class:

```java
class Element {
  String value;
}
```

Serialize Java object to JSON:

```java
Element element = new Element();
element.value = "Hello World";
String json = new Gson().toJson(element);
```

Deserialize JSON to Java object:

```java
String json = "{\"value\":\"Hello World\"}";
Element element = new Gson().fromJson(json, Element.class);
```

Reflection configuration:

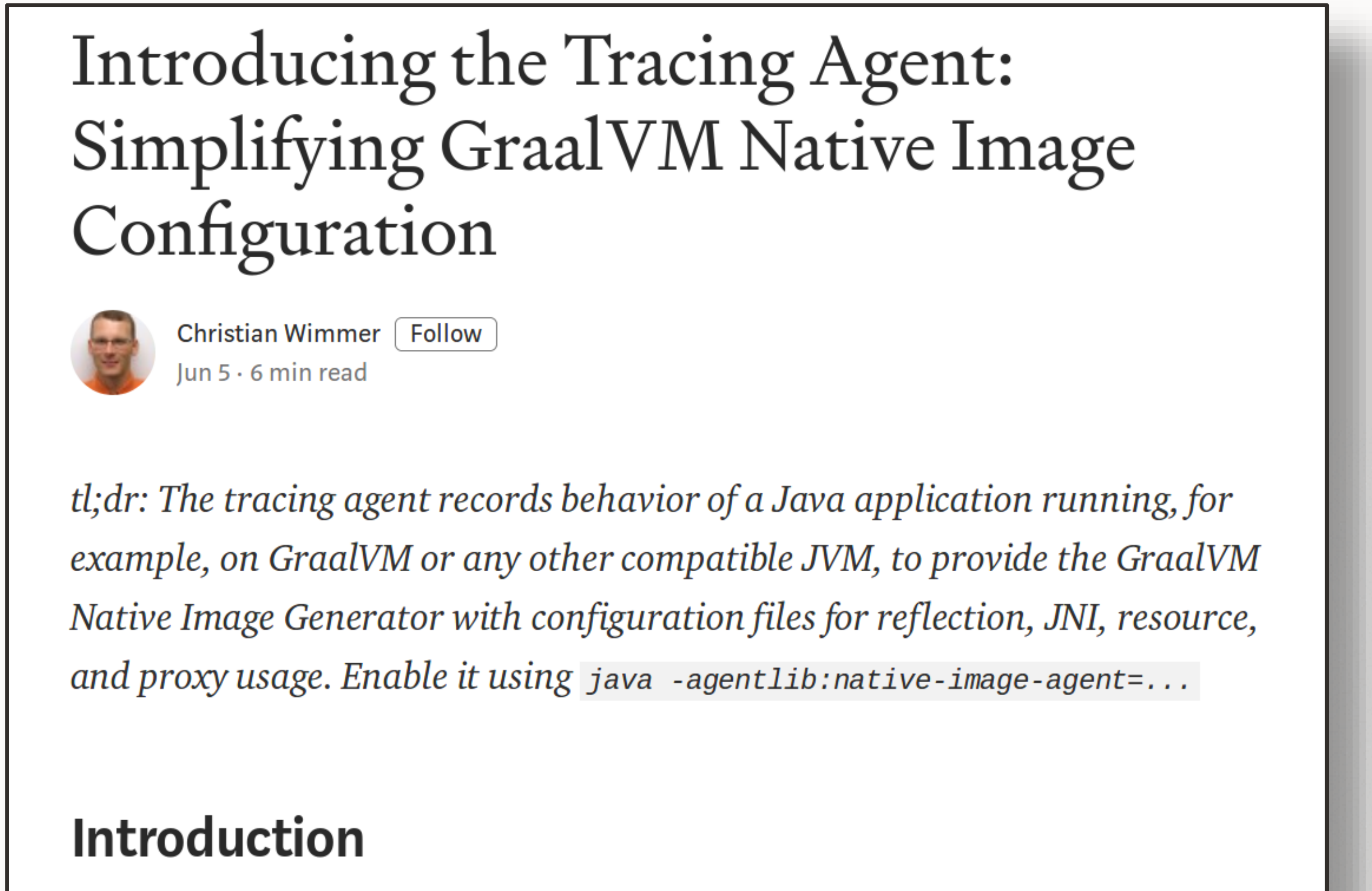
```json
[
  {
    "name" : "com.oracle.test.Element",
    "fields" : [
      { "name" : "value" }
    ],
    "methods" : [
      { "name" : "<init>" }
    ]
  }
]
```

Tracing Agent

- Trace reflection, JNI, resource usage on Java HotSpot VM
  - Agent to record usage and produce configuration files for native images
    - java -agentlib:native-image-agent=config-output-dir=META-INF/native-image ...
  - Simplify the getting-started process
    - Everything that was executed on the Java HotSpot VM also works in the native image
  - Manual adjustment / addition will still be necessary
    - Unless you have an excellent test suite for your application
- Fun fact: Agent is a Java Native Image
  - JVMTI interface implemented using the low-level C interface of Native Image

Blog Article with Details and Examples

https://medium.com/graalvm/c3b56c486271



Introducing the Tracing Agent: Simplifying GraalVM Native Image Configuration

Christian Wimmer

Jun 5 · 6 min read

tl;dr: The tracing agent records behavior of a Java application running, for example, on GraalVM or any other compatible JVM, to provide the GraalVM Native Image Generator with configuration files for reflection, JNI, resource, and proxy usage. Enable it using `java -agentlib:native-image-agent=...`

Introduction

Class Initialization

- Class initialization at image build time improves application startup
- Configurable per class / package / package prefix
  - --initialize-at-build-time=... --initialize-at-run-time=...
  - By default, application classes are initialized at run time
  - Most JDK classes are initialized at image build time
  - Static analysis finds class initializers that can run at image build time
- Performance implications of class initialization at run time
  - If a class is initialized at run time, no instances can be in the image heap
  - Runtime checks before static method calls, static field accesses, and allocations

Tracing Class Initialization

- Class initialization is tricky
  - Some reasons for class initialization are not obvious
    - Adding a default method in an interface changes class initialization behavior
  - Class initialization order can be non-deterministic
    - When class dependencies are cyclic

- Common usability problem: A class that is marked for initialization at run time gets initialized at build time

```
-H:+TraceClassInitialization
```

Class Initialization: Corner Cases

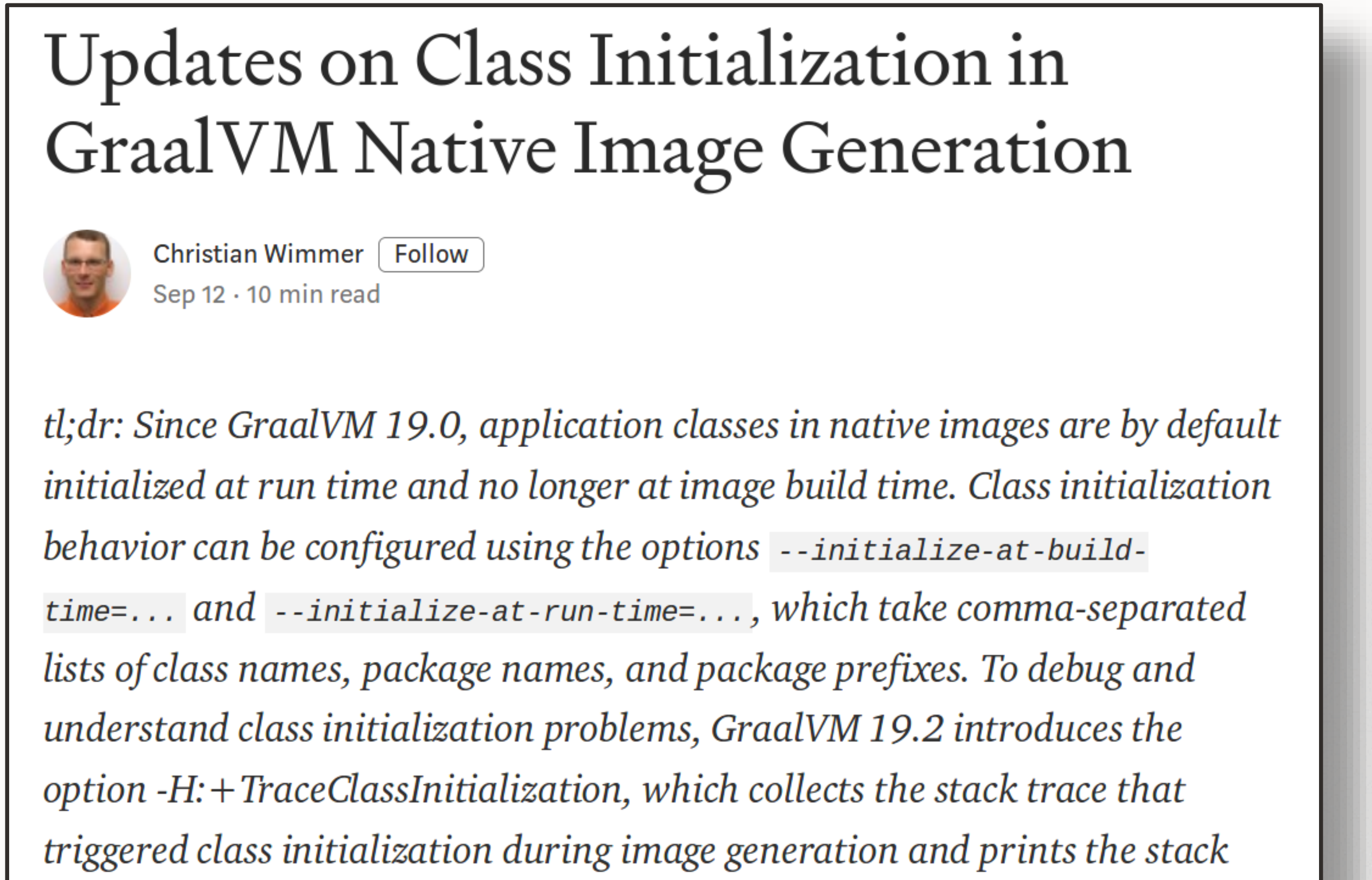
- Order of class initialization at image build time
  A static analysis cannot determine the order that would be used at run time
  Possible solution: track class initialization order in the Tracing Agent

- Java lambdas trigger class initialization at image build time
  JDK forces initialization
  Fixed in GraalVM 19.3: patch JDK

- Annotation classes are always initialized at image build time
  Not a problem because annotation classes cannot do much
  But they can trigger the initialization of enum classes
  No reasonable solution, needs application changes

Blog Article with Details and Examples

https://medium.com/graalvm/c61faca461f7



Updates on Class Initialization in GraalVM Native Image Generation

Christian Wimmer Follow

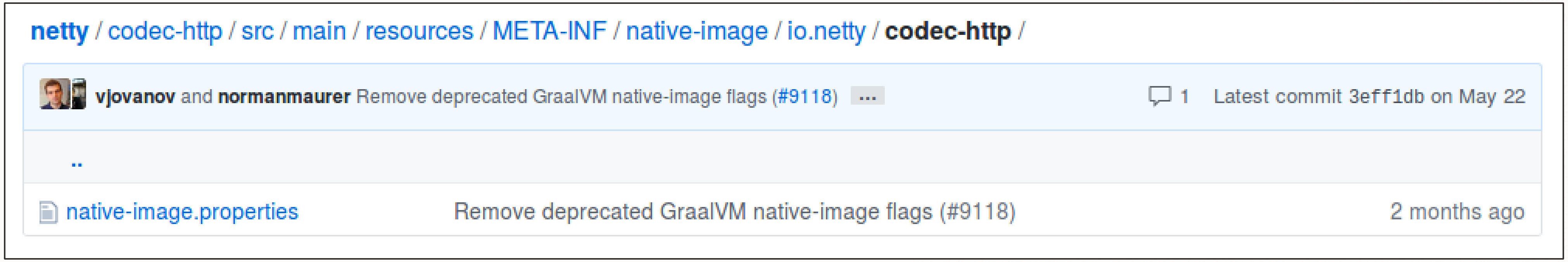
Sep 12 · 10 min read

tl;dr: Since GraalVM 19.0, application classes in native images are by default initialized at run time and no longer at image build time. Class initialization behavior can be configured using the options `--initialize-at-build-time=...` and `--initialize-at-run-time=...`, which take comma-separated lists of class names, package names, and package prefixes. To debug and understand class initialization problems, GraalVM 19.2 introduces the option -H:+TraceClassInitialization, which collects the stack trace that triggered class initialization during image generation and prints the stack

Native Image Support in Libraries

- Configuration options should be provided by libraries
  - Library and framework developers know best what their code needs

- Configuration files in META-INF/native-image are automatically picked up
  - native-image.properties for command line options like class initialization options
  - reflect-config.json, jni-config.json, ... for configuration files created by tracing agent
  - Use subdirectories to make files composeable (inspired by Maven)
    - META-INF/native-image/your.group.id/artifactId/



netty / codec-http / src / main / resources / META-INF / native-image / io.netty / codec-http /

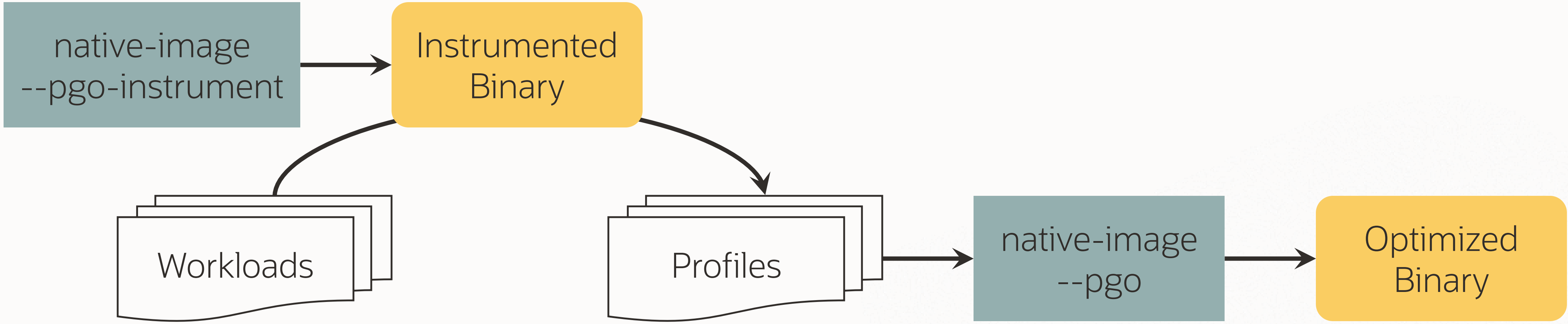
vjovanov and normanmaurer Remove deprecated GraalVM native-image flags (#9118) ... 1 Latest commit 3eff1db on May 22

..

native-image.properties	Remove deprecated GraalVM native-image flags (#9118)	2 months ago

Profile-Guided Optimizations (PGO)

- AOT compiled code cannot optimize itself at run time
  - No dynamic "hot spot" compilation
- PGO requires relevant workloads at build time
- Optimized code runs immediately at startup, no "warmup" curve



Service Loader

- Allows dynamic configuration of Java applications
  - Look up classes implementing a service interface
  - Requires a combination of resources and reflection
- Native image automatically detects used service interfaces
  - All resource and reflection registration done automatically
  - It might add a little bit too much that is unused in your application
    - But simplifies many common use cases

Native Image Specific Code

- Ideally, application code is not aware of native image
- In practice, sometimes different code paths are necessary
  - Work around limitations of native image
  - Benefit from initialization at build time

- API to query how code is running

```
ImageInfo.inImageCode()
ImageInfo.inImageRuntimeCode()
ImageInfo.inImageBuildtimeCode()
```

Operating System and Java 11 Support

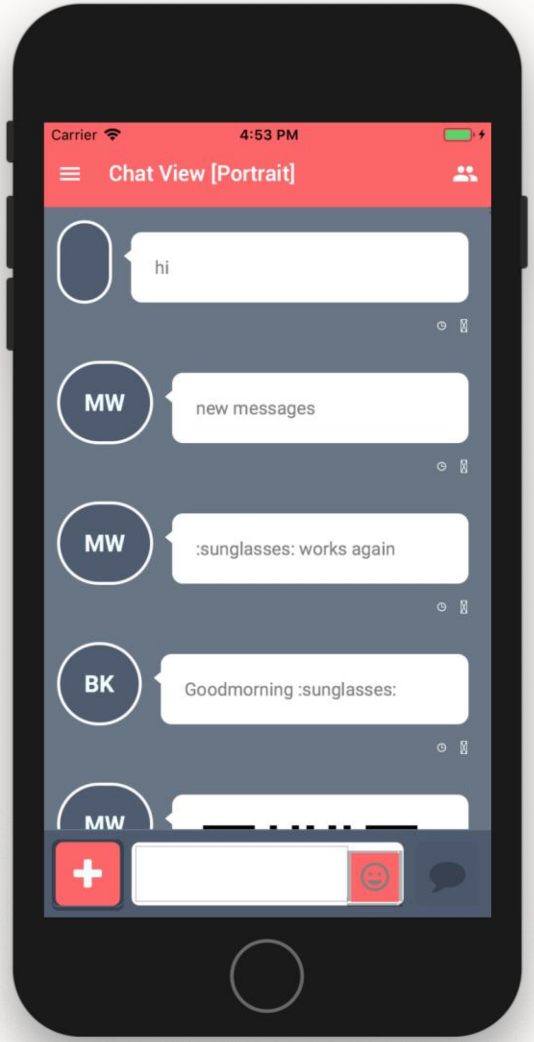
- Original design: C code of the JDK manually translated to low-level Java code
  - Necessary because there was no JNI support
  - Feasible because only part of JDK was needed
- Problems
  - Even translations for Linux are not complete yet
  - Maintenance problem: all bugfixes need to be ported from C to Java
  - Windows support: as much work as Linux support because little C code is shared
  - Java version support: differences between Java 8, 11, 13, and future versions
- New approach (under development)
  - Statically link in the C code of the JDK
  - Invocation of C code using JNI

What’s Next for GraalVM Native Image?

- Low-latency, high-throughput, and parallel GC
  - Inspired by G1 of Java HotSpot VM
- Peak performance improvements
  - Same or better peak performance compared to Java HotSpot VM
- More supported platforms
  - JDK 11, AArch64, Windows
- Work with the community to support important libraries

Architecture Support

- Graal compiler supports x64, AArch64, and Sparc
  - Native image currently limited to x64
  - Native Image Support for AArch64 under development
- How to support all architectures?
  - LLVM compiler infrastructure has broad architecture support
  - Slow compile time of LLVM does not matter for ahead-of-time compilation
- Using LLVM comes with a price
  - Moving GC support in LLVM is still experimental
  - Currently slower code due to constraints of the LLVM IR
    - For example no fixed register for threads
  - Not suitable for JIT compilation



hi

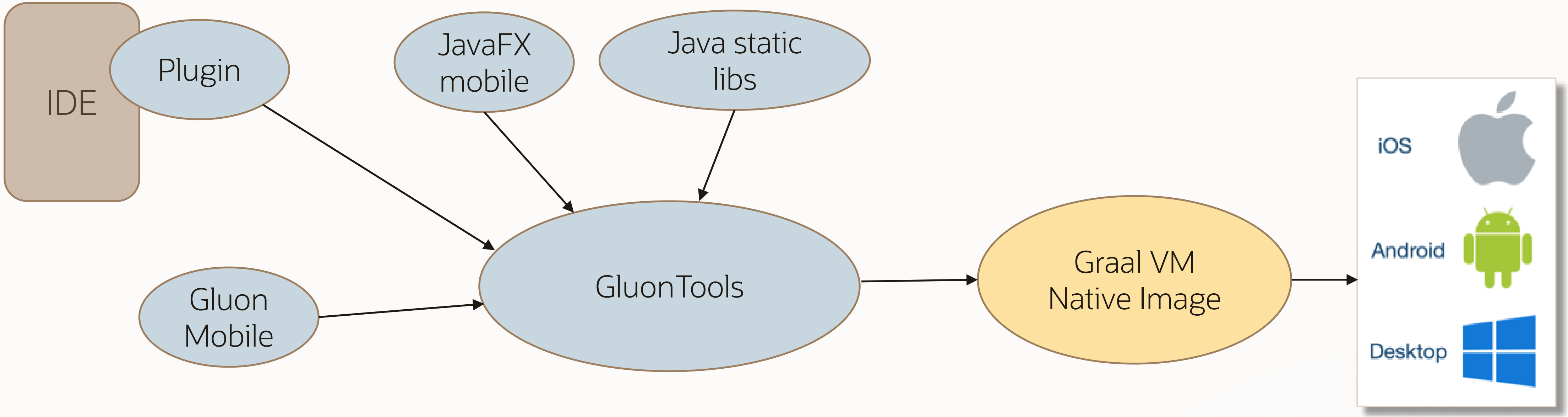
MW new messages

MW :sunglasses: works again

BK Goodmorning :sunglasses:

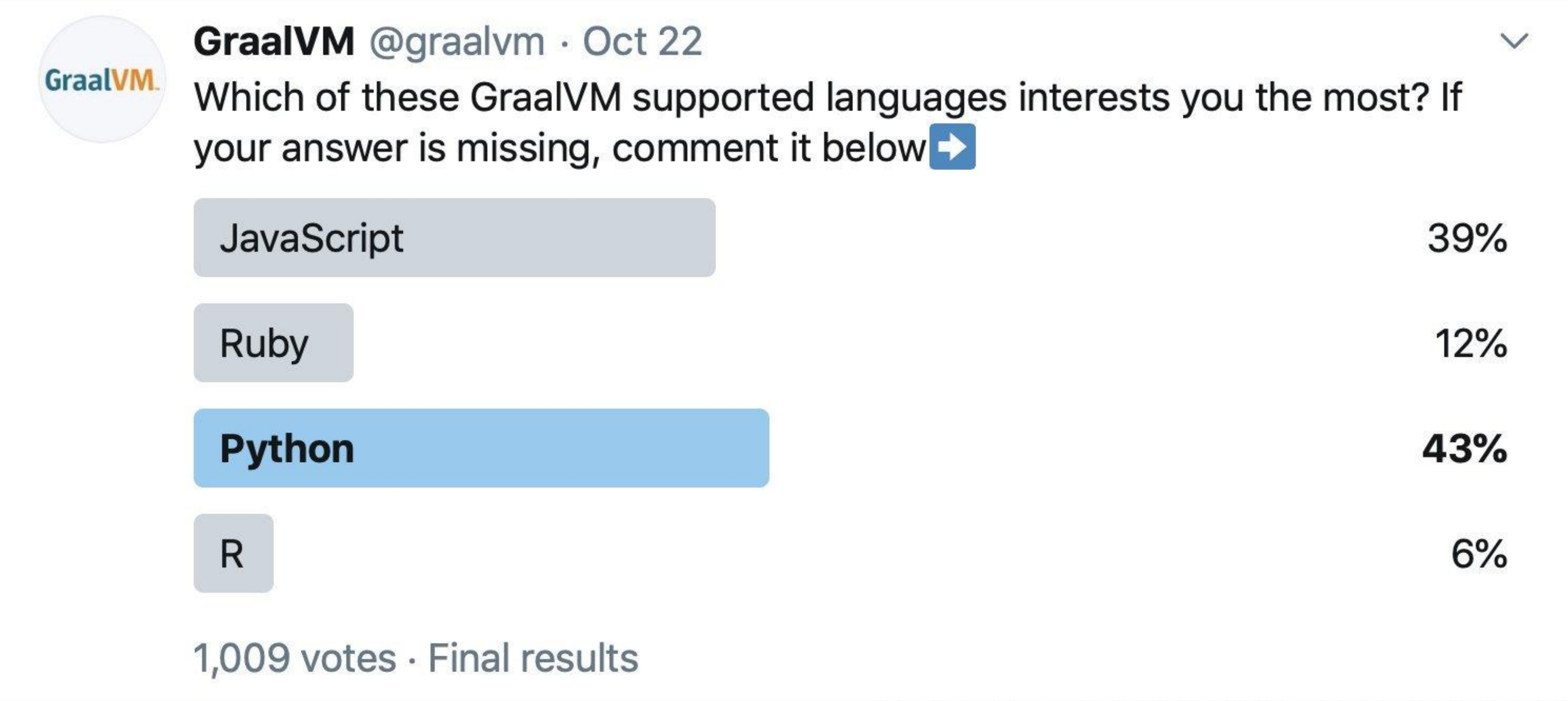
iPhone 6 — 12.2

Cross-Platform GraalVM



- JavaFX extensions for mobile
- Integration with mobile functionality (e.g. GPS/camera)
- Mobile-specific connectivity

Polyglot and Embeddability



GraalVM @graalvm · Oct 22

Which of these GraalVM supported languages interests you the most? If your answer is missing, comment it below➡️

JavaScript 39%

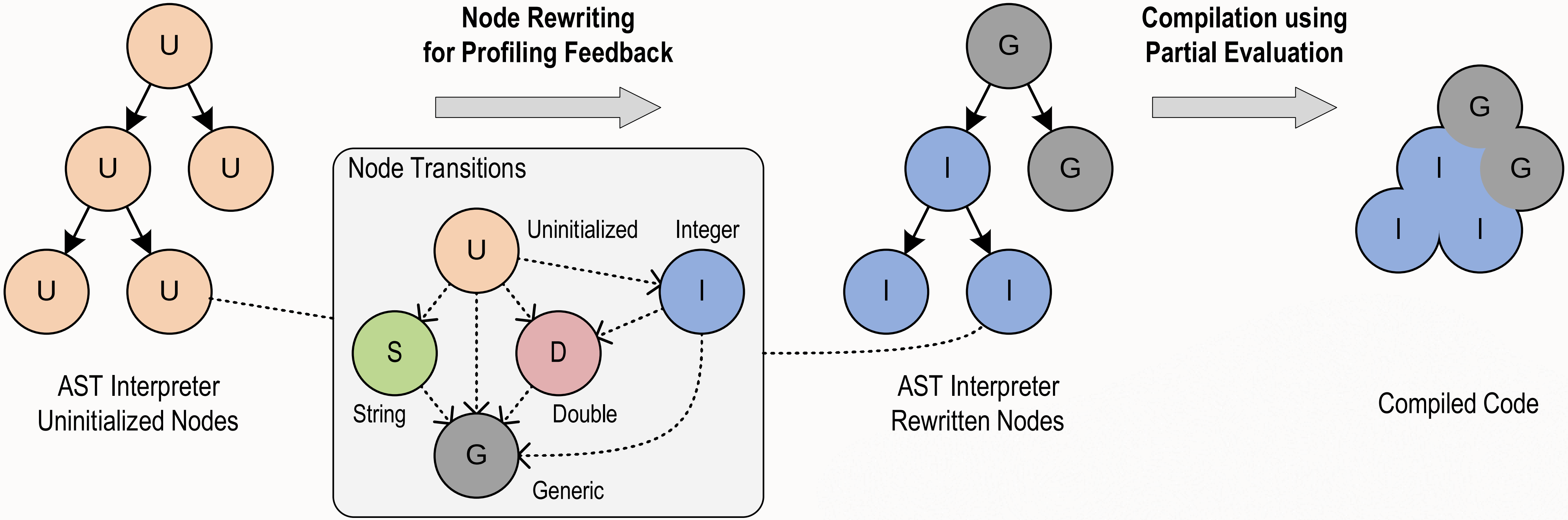
Ruby 12%

Python **43%**

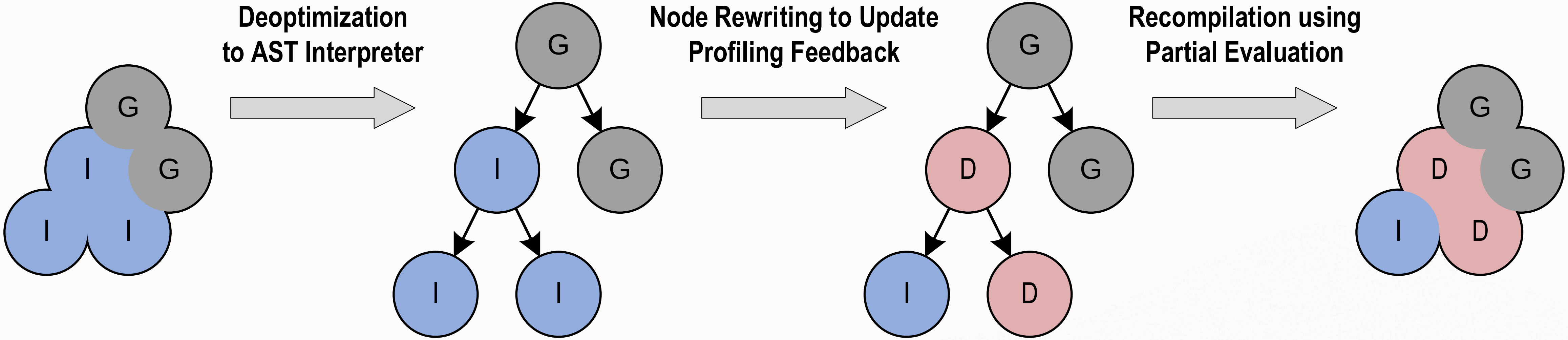
R 6%

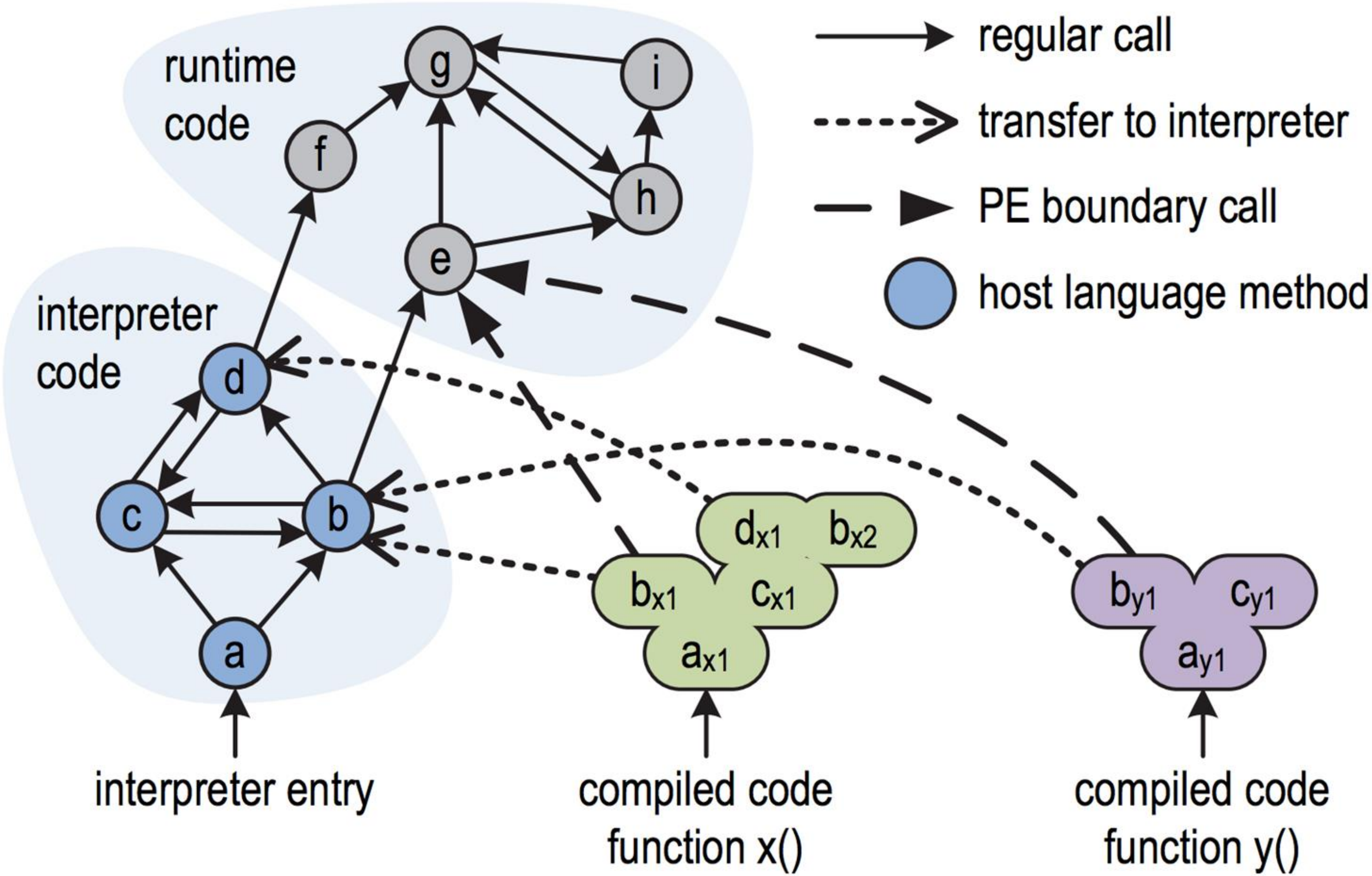
1,009 votes · Final results

Optimization and Speculation...

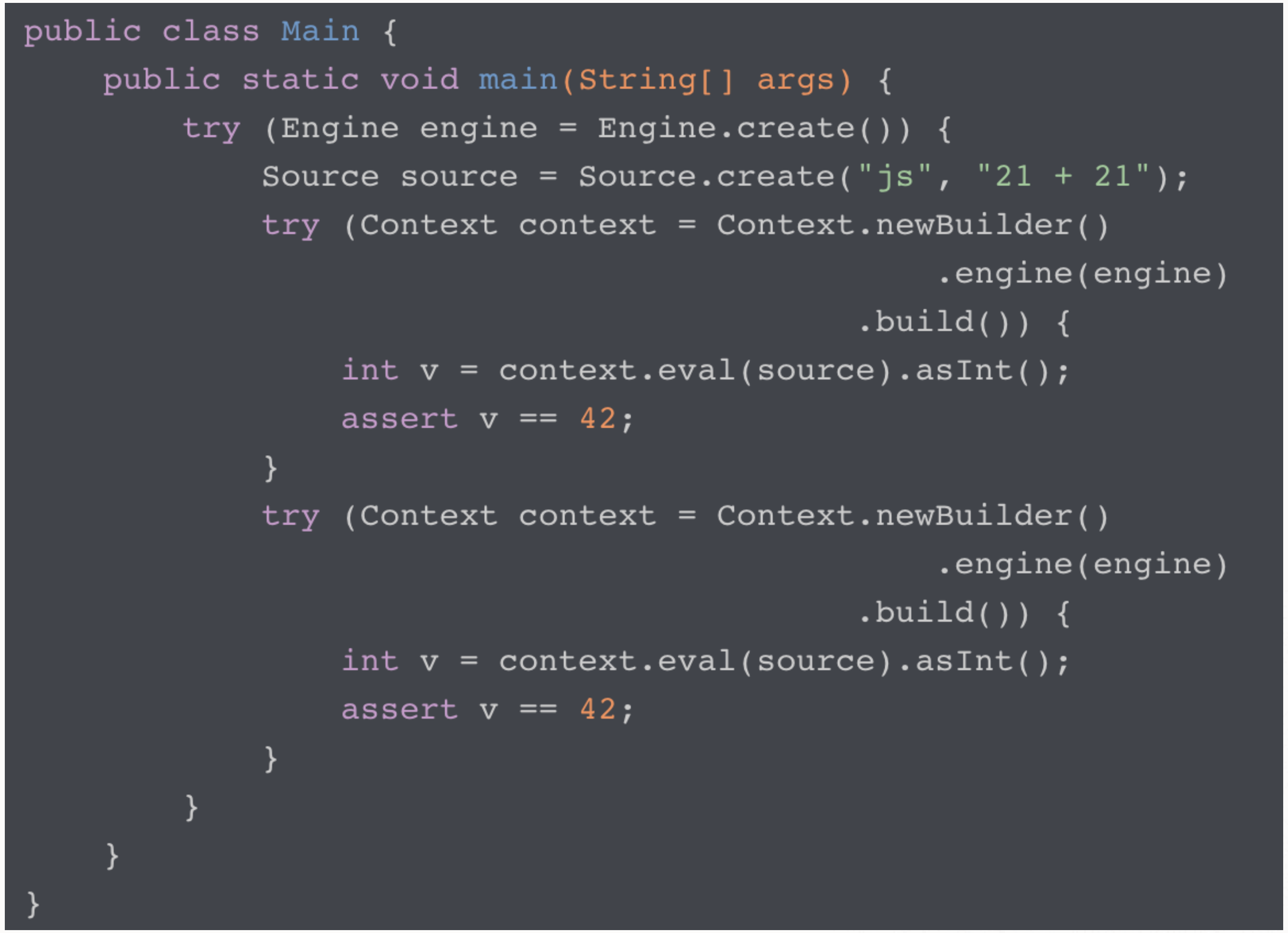


And Deoptimization!



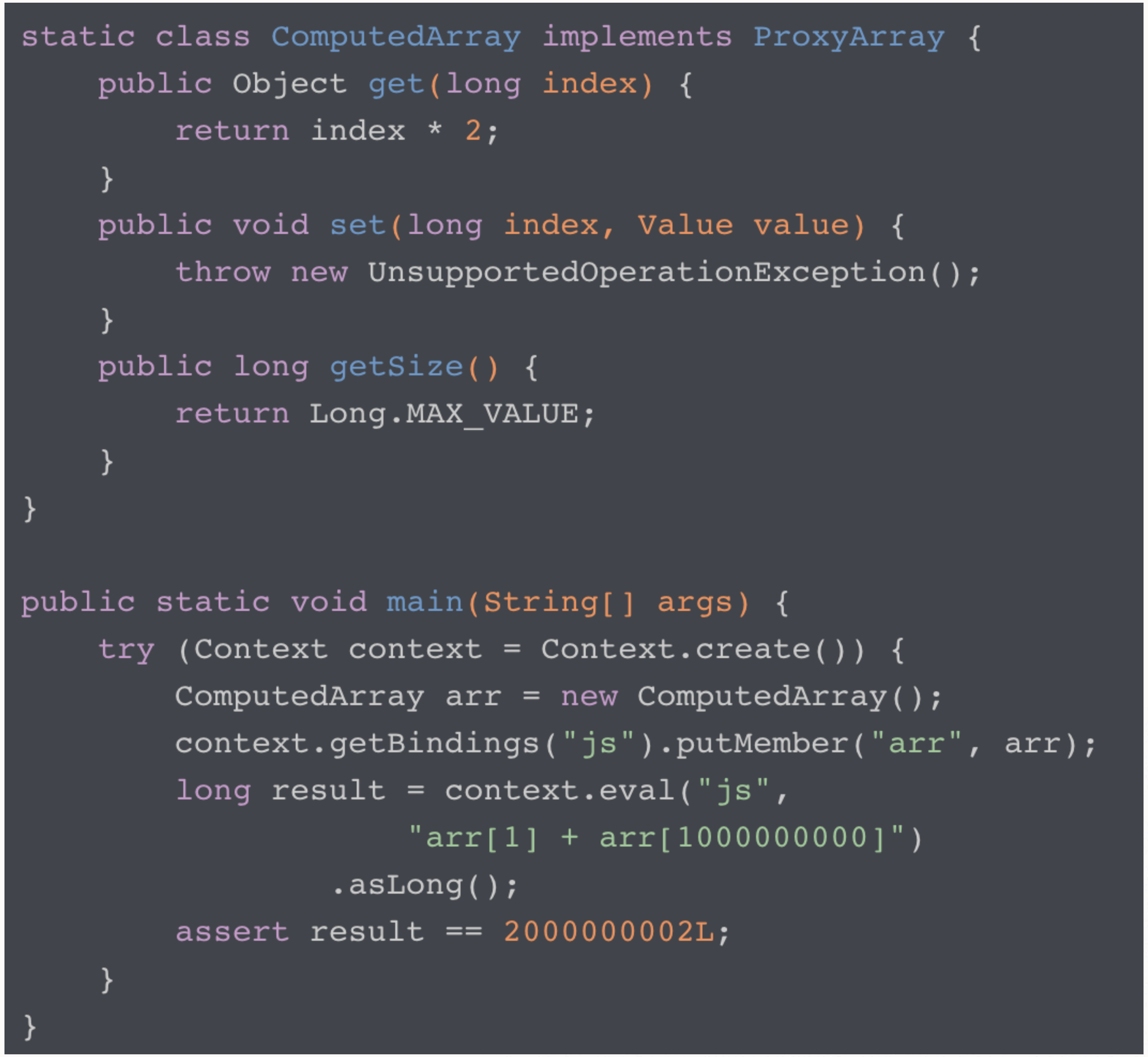

runtime code
g
i
f
h
e
interpreter code
d
c
b
a
interpreter entry
d_{x1}
b_{x2}
b_{x1}
c_{x1}
a_{x1}
compiled code function x()
b_{y1}
c_{y1}
a_{y1}
compiled code function y()
regular call
transfer to interpreter
PE boundary call
host language method

Simple Embeddability



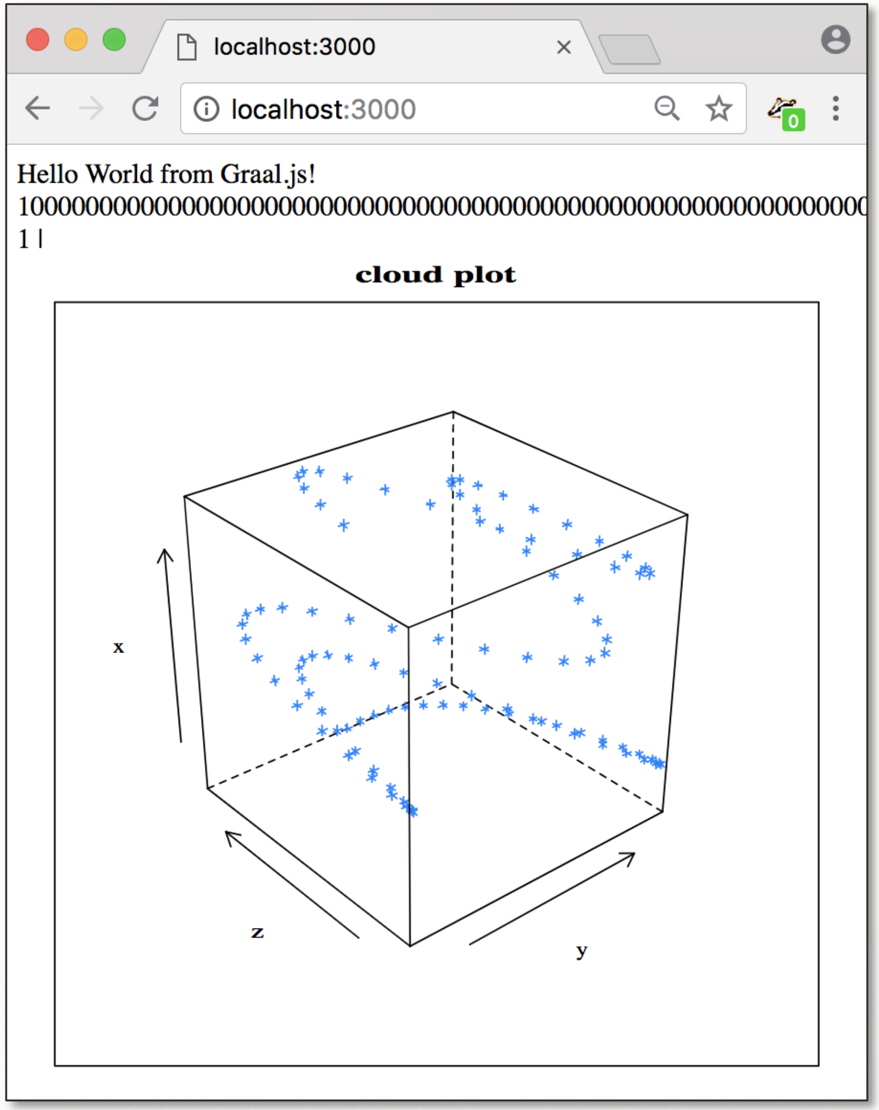
```
public class Main {
    public static void main(String[] args) {
        try (Engine engine = Engine.create()) {
            Source source = Source.create("js", "21 + 21");
            try (Context context = Context.newBuilder()
                                              .engine(engine)
                                          .build()) {
                int v = context.eval(source).asInt();
                assert v == 42;
            }
            try (Context context = Context.newBuilder()
                                              .engine(engine)
                                          .build()) {
                int v = context.eval(source).asInt();
                assert v == 42;
            }
        }
    }
}
```

Full separation of logical and physical data layout, enabling virtual data structures



```
static class ComputedArray implements ProxyArray {
    public Object get(long index) {
        return index * 2;
    }
    public void set(long index, Value value) {
        throw new UnsupportedOperationException();
    }
    public long getSize() {
        return Long.MAX_VALUE;
    }
}

public static void main(String[] args) {
    try (Context context = Context.create()) {
        ComputedArray arr = new ComputedArray();
        context.getBindings("js").putMember("arr", arr);
        long result = context.eval("js",
                    "arr[1] + arr[1000000000]")
                .asLong();
        assert result == 2000000002L;
    }
}
```

```
const express = require('express');
const app = express();
app.listen(3000);
app.get('/', function(req, res) {
  var text = 'Hello World!';
  const BigInteger = Java.type(
    'java.math.BigInteger');
  text += BigInteger.valueOf(2)
    .pow(100).toString(16);
  text += Polyglot.eval(
    'R', 'runif(100)')[0];
  res.send(text);
})
```

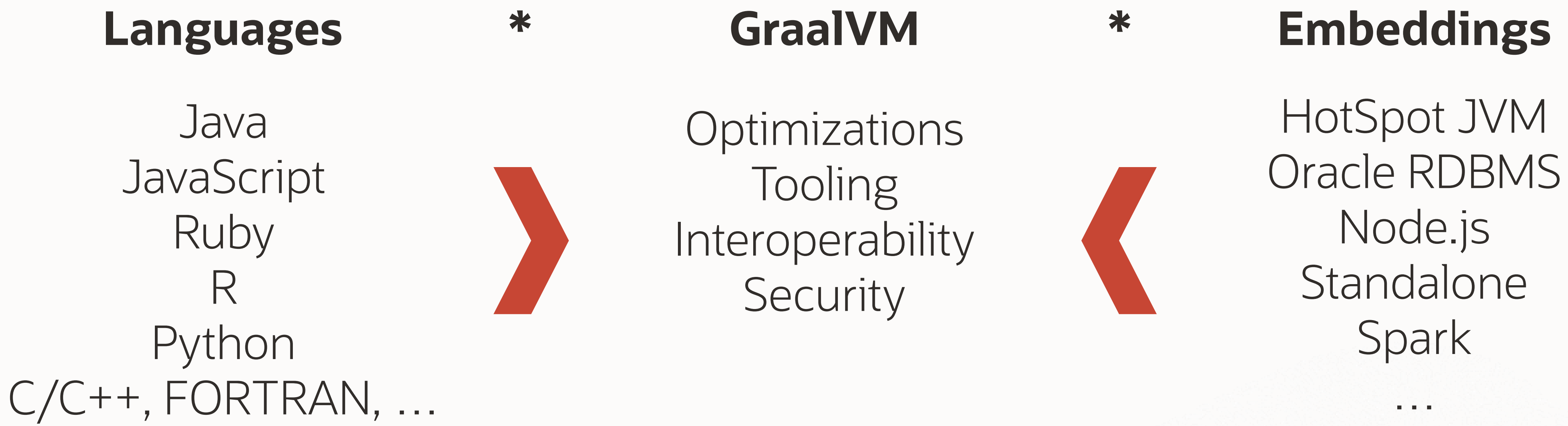


Hello World from Graal.js!
100000000000000000000000000000000000000000000000000000000000000000000
1 |

Enable Polyglot interoperability:

node --polyglot

Multiplicative Value-Add of GraalVM Ecosystem



Languages * **GraalVM** * **Embeddings**

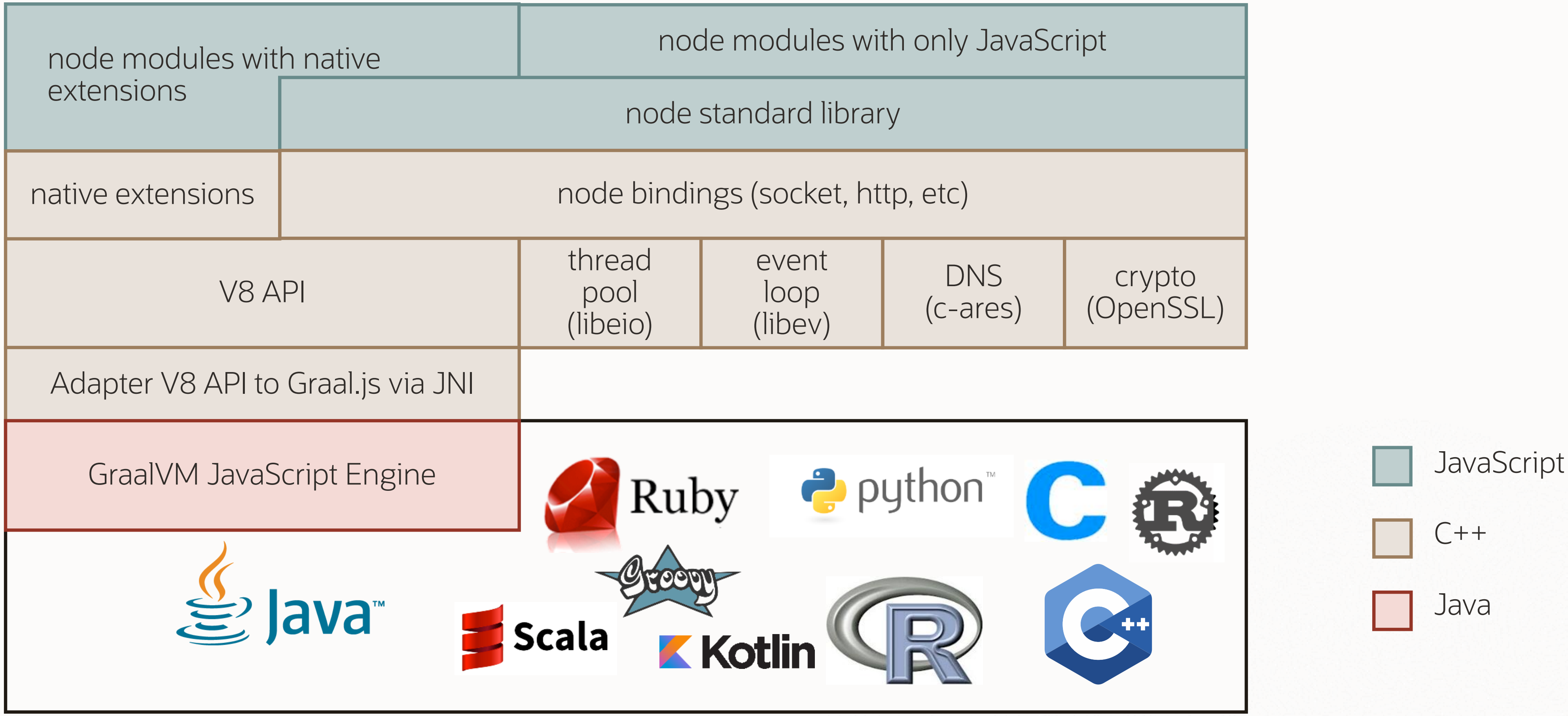
Java
JavaScript
Ruby
R
Python
C/C++, FORTRAN, …

Optimizations
Tooling
Interoperability
Security

HotSpot JVM
Oracle RDBMS
Node.js
Standalone
Spark
…

Add your own language or embedding or language-agnostic tools!

Architecture of Node.js running via GraalVM



node modules with native extensions

node modules with only JavaScript

node standard library

native extensions

node bindings (socket, http, etc)

V8 API

thread pool (libeio)

event loop (libev)

DNS (c-ares)

crypto (OpenSSL)

Adapter V8 API to Graal.js via JNI

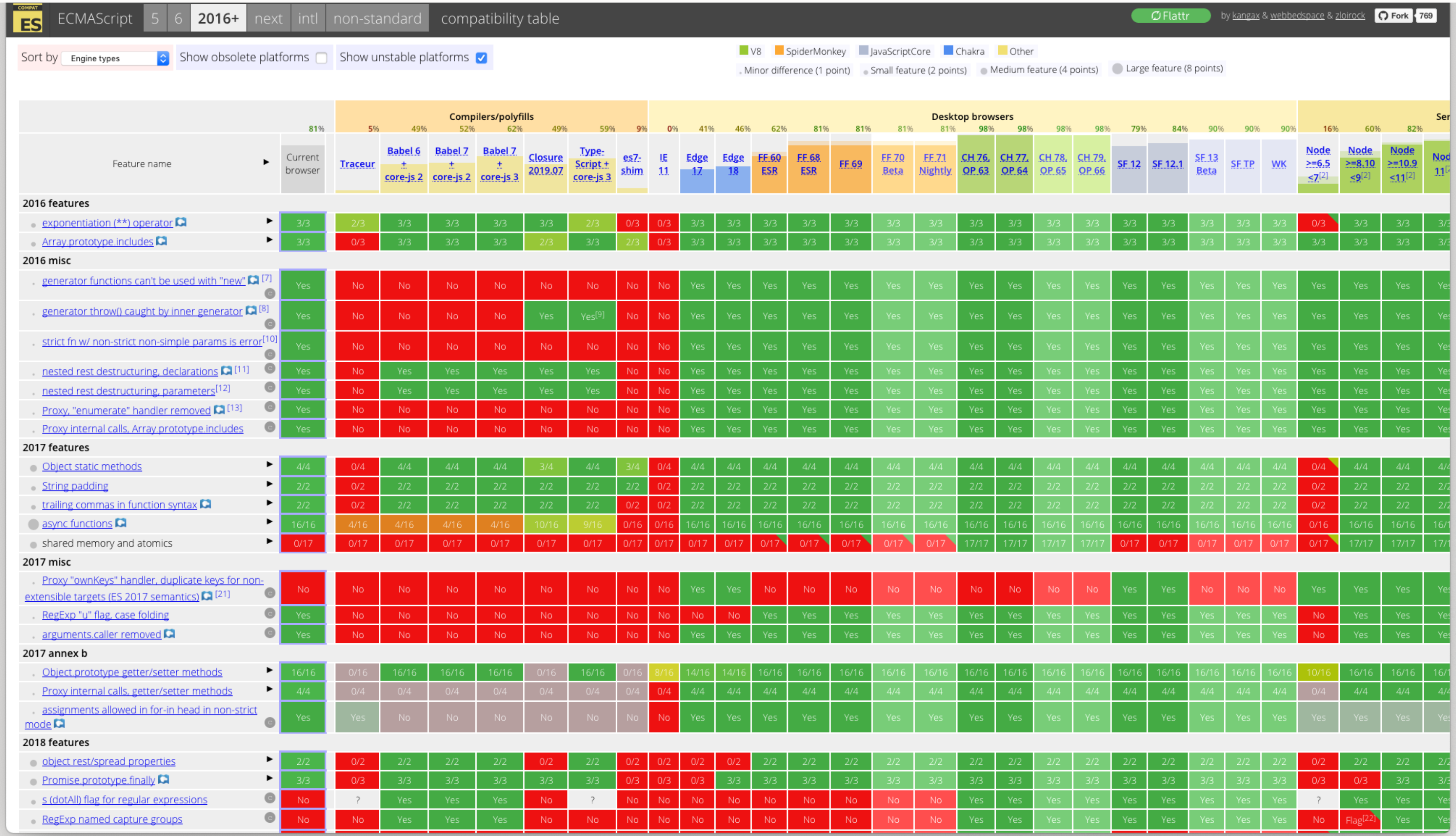
GraalVM JavaScript Engine

JavaScript

C++

Java

https://kangax.github.io



COMPAT ES ECMAScript 5 6 **2016+** next intl non-standard compatibility table

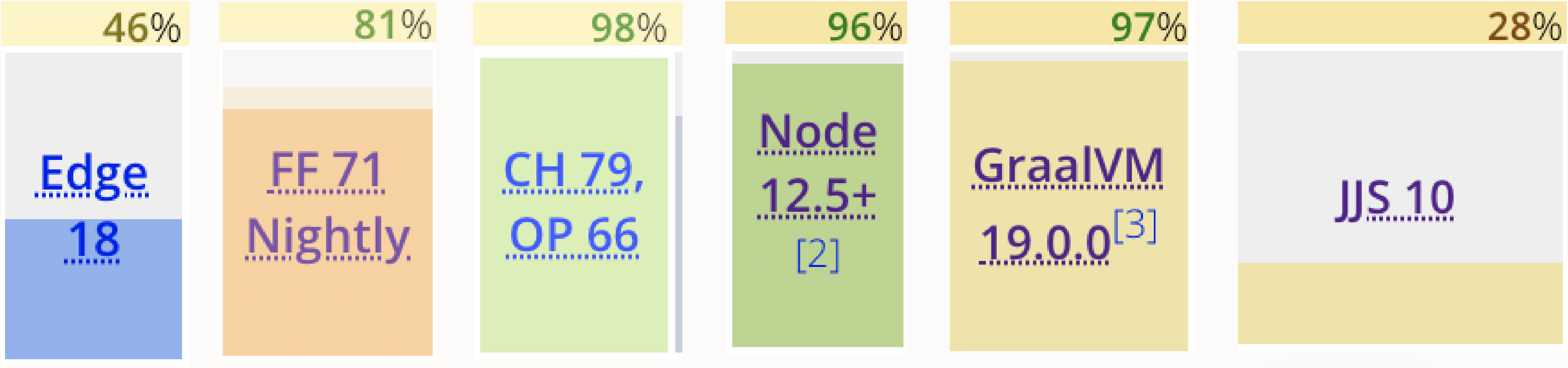
Flattr by kangax & webbedspace & zloirock Fork 769

Sort by Engine types Show obsolete platforms ☐ Show unstable platforms ☑

🟩 V8 🟧 SpiderMonkey ▨ JavaScriptCore 🟦 Chakra 🟨 Other

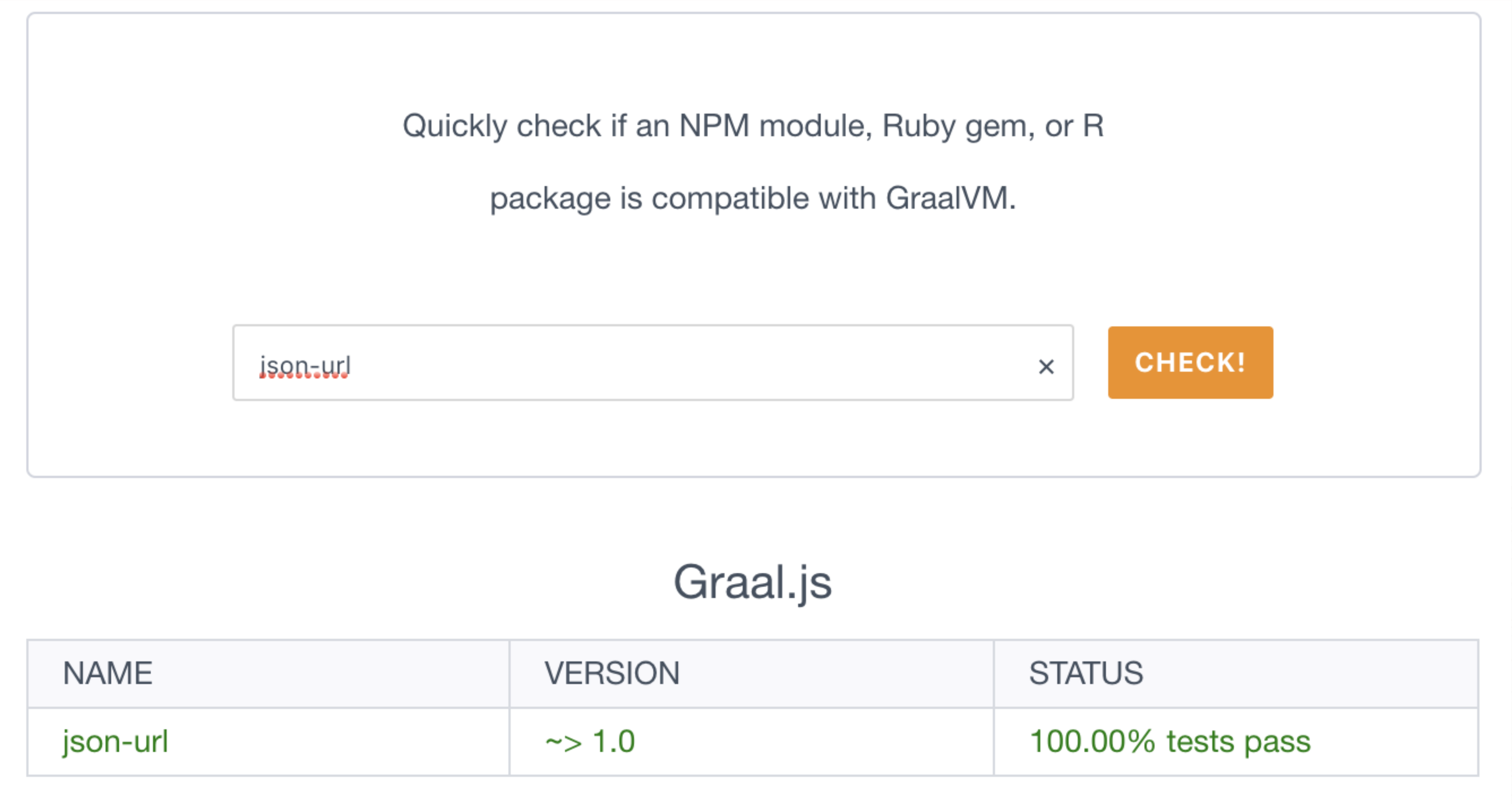
. Minor difference (1 point) • Small feature (2 points) ● Medium feature (4 points) ⬤ Large feature (8 points)

		Compilers/polyfills							Desktop browsers																			Ser			
	81%	5%	49%	52%	62%	49%	59%	9%	0%	41%	46%	62%	81%	81%	81%	81%	98%	98%	98%	98%	79%	84%	90%	90%	90%	16%	60%	82%	
Feature name ▶	Current browser	Traceur	Babel 6 + core-js 2	Babel 7 + core-js 2	Babel 7 + core-js 3	Closure 2019.07	Type-Script + core-js 3	es7-shim	IE 11	Edge 17	Edge 18	FF 60 ESR	FF 68 ESR	FF 69	FF 70 Beta	FF 71 Nightly	CH 76, OP 63	CH 77, OP 64	CH 78, OP 65	CH 79, OP 66	SF 12	SF 12.1	SF 13 Beta	SF TP	WK	Node >=6.5 <7[2]	Node >=8.10 <9[2]	Node >=10.9 <11[2]	Nod 11[2
2016 features																															
exponentiation (**) operator ▶	3/3	2/3	3/3	3/3	3/3	3/3	2/3	0/3	0/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	0/3	3/3	3/3	3/
Array.prototype.includes ▶	3/3	0/3	3/3	3/3	3/3	2/3	3/3	2/3	0/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/
2016 misc																															
generator functions can't be used with "new" [7]	Yes	No	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Ye
generator throw() caught by inner generator [8]	Yes	No	No	No	No	Yes	Yes[9]	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Ye
strict fn w/ non-strict non-simple params is error[10]	Yes	No	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Ye
nested rest destructuring, declarations [11]	Yes	No	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Ye
nested rest destructuring, parameters[12]	Yes	No	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Ye
Proxy, "enumerate" handler removed [13]	Yes	No	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Ye
Proxy internal calls, Array.prototype.includes	Yes	No	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Ye
2017 features																															
Object static methods ▶	4/4	0/4	4/4	4/4	4/4	3/4	4/4	3/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	0/4	4/4	4/4	4/
String padding ▶	2/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2	2/2	2/2	2/
trailing commas in function syntax ▶	2/2	0/2	2/2	2/2	2/2	2/2	2/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2	2/2	2/2	2/
async functions ▶	16/16	4/16	4/16	4/16	4/16	10/16	9/16	0/16	0/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	0/16	16/16	16/16	16/
shared memory and atomics ▶	0/17	0/17	0/17	0/17	0/17	0/17	0/17	0/17	0/17	0/17	0/17	0/17	0/17	0/17	0/17	0/17	17/17	17/17	17/17	17/17	0/17	0/17	0/17	0/17	0/17	0/17	17/17	17/17	17/
2017 misc																															
Proxy "ownKeys" handler, duplicate keys for non-extensible targets (ES 2017 semantics) [21]	No	No	No	No	No	No	No	No	No	Yes	Yes	No	No	No	No	No	No	No	No	No	Yes	Yes	No	No	No	Yes	Yes	Yes	Ye
RegExp "u" flag, case folding	Yes	No	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Ye
arguments.caller removed	Yes	No	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Ye
2017 annex b																															
Object.prototype getter/setter methods ▶	16/16	0/16	16/16	16/16	16/16	0/16	16/16	0/16	8/16	14/16	14/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	10/16	16/16	16/16	16/
Proxy internal calls, getter/setter methods ▶	4/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	0/4	4/4	4/4	4/
assignments allowed in for-in head in non-strict mode	Yes	Yes	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Ye
2018 features																															
object rest/spread properties ▶	2/2	0/2	2/2	2/2	2/2	0/2	2/2	0/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2	2/2	2/2	2/
Promise.prototype.finally ▶	3/3	0/3	3/3	3/3	3/3	3/3	3/3	0/3	0/3	0/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	0/3	0/3	3/3	3/
s (dotAll) flag for regular expressions	No	?	Yes	Yes	Yes	No	?	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	?	Yes	Yes	Ye
RegExp named capture groups	No	No	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Flag[22]	Yes	Ye



46%	81%	98%	96%	97%	28%
Edge 18	FF 71 Nightly	CH 79, OP 66	Node 12.5+ [2]	GraalVM 19.0.0[3]	JJS 10

https://www.graalvm.org/docs/reference-manual/compatibility/



Quickly check if an NPM module, Ruby gem, or R package is compatible with GraalVM.

Graal.js

NAME	VERSION	STATUS
json-url	~> 1.0	100.00% tests pass

Trade-Offs

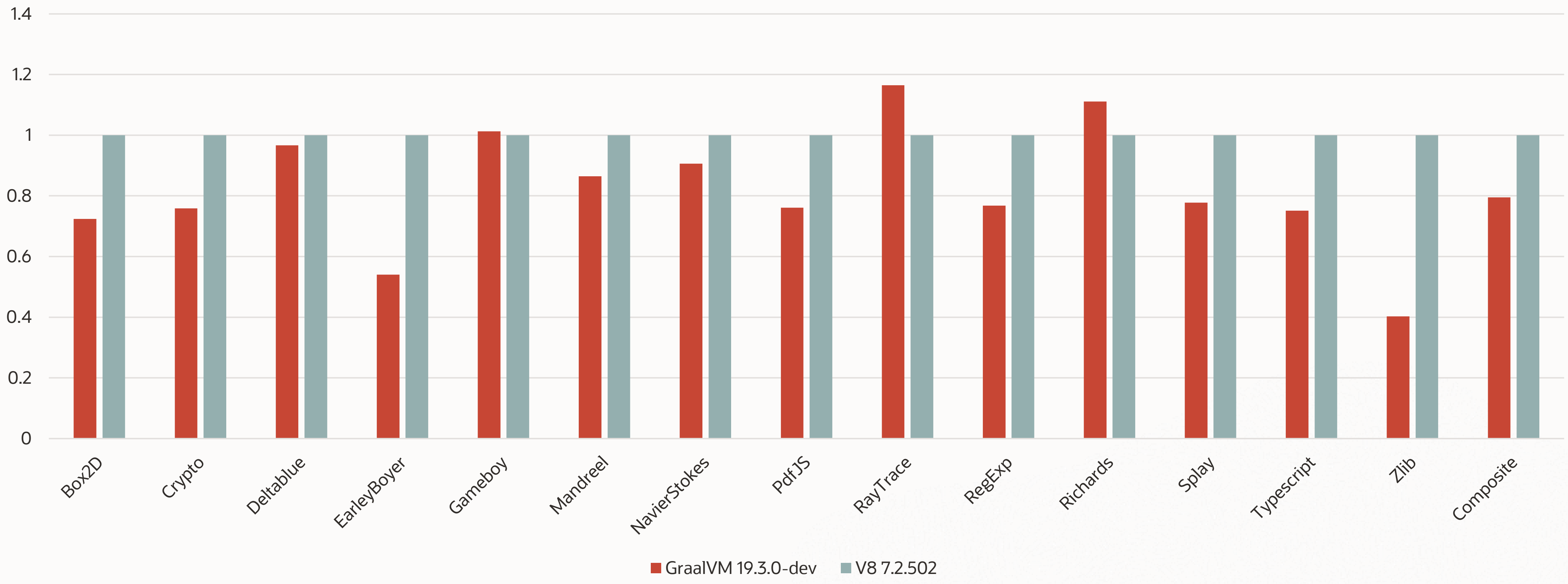
Advantages

- Java interoperability: Use any Java library or framework
- Polyglot capabilities (Python, Ruby, R)
- Run with large heaps and JVM garbage collectors

Disadvantages

- Longer start-up time
- Higher memory footprint

Graal.js Performance (versus V8)



FastR

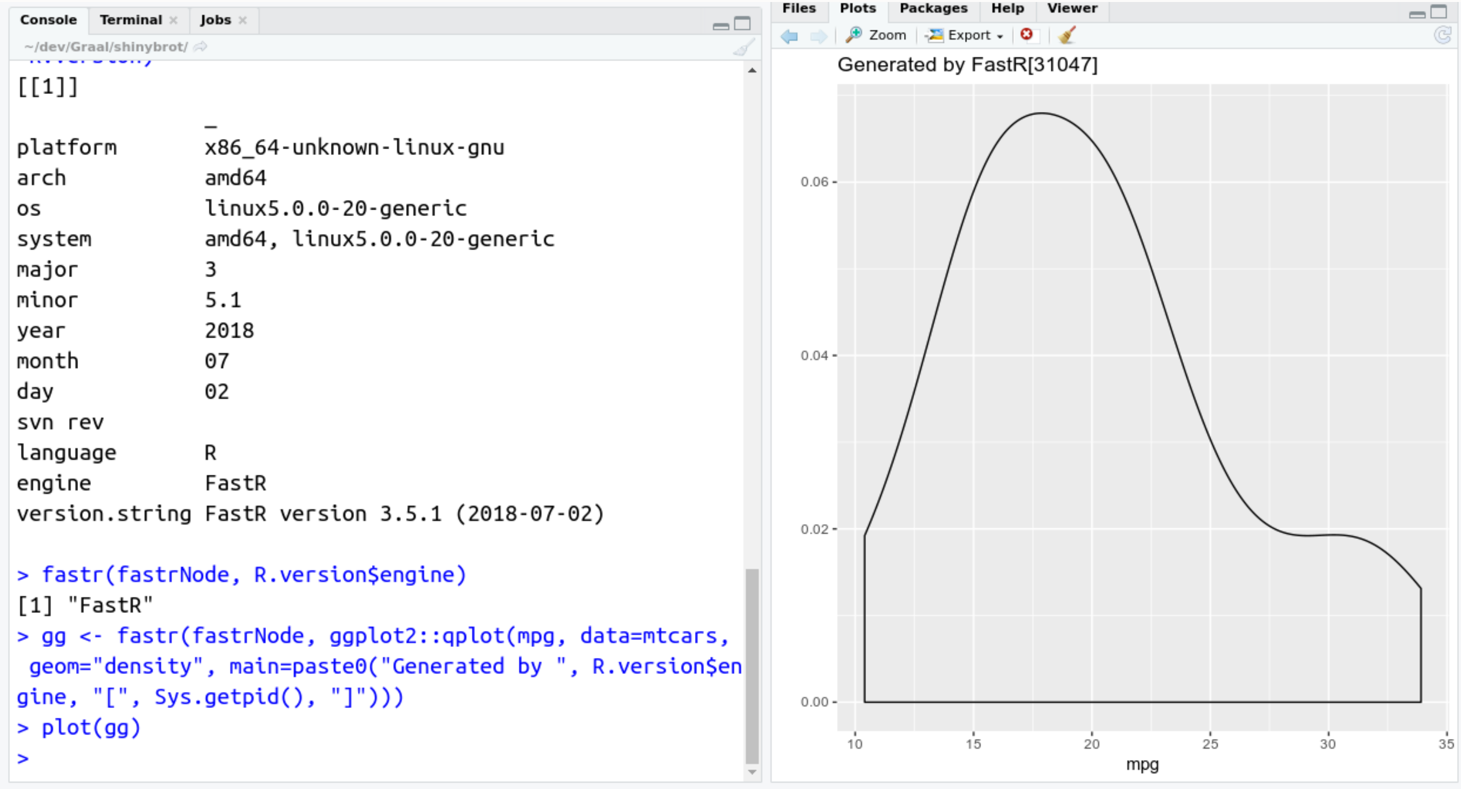
- GNU-R compatible R implementation
  - Including the C/Fortran interface

- Built on top of the GraalVM platform
  - Leverages GraalVM optimizing compiler
  - Integration with GraalVM dev tools
  - Zero overhead interop with other GraalVM languages

FastR Cluster Package

`fastRCluster` package to use FastR as a “cluster” from GNU-R

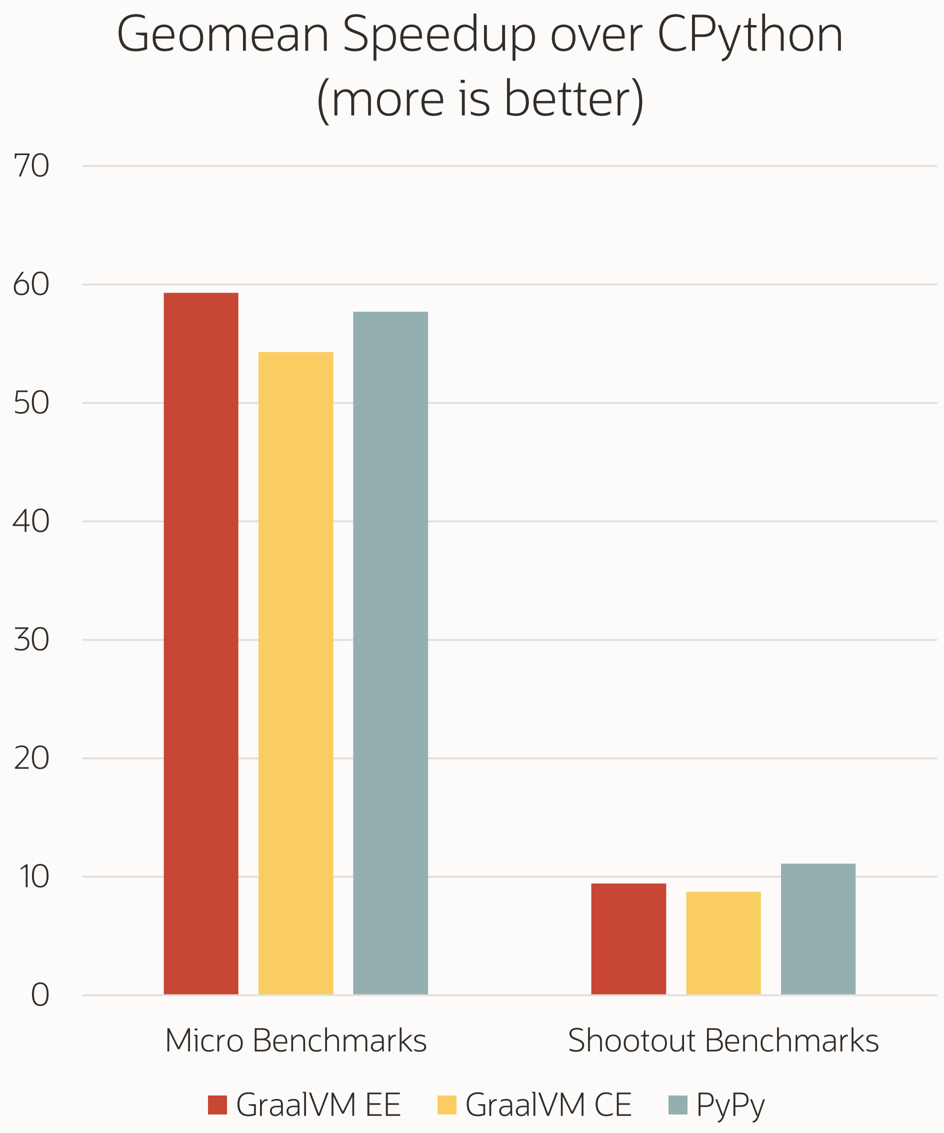
- Works like other “cluster” providers packages
- Offload to efficient R runtime similar to offloading to cluster



```
[[1]]

               _
platform       x86_64-unknown-linux-gnu
arch           amd64
os             linux5.0.0-20-generic
system         amd64, linux5.0.0-20-generic
major          3
minor          5.1
year           2018
month          07
day            02
svn rev
language       R
engine         FastR
version.string FastR version 3.5.1 (2018-07-02)

> fastr(fastrNode, R.version$engine)
[1] "FastR"
> gg <- fastr(fastrNode, ggplot2::qplot(mpg, data=mtcars,
 geom="density", main=paste0("Generated by ", R.version$en
gine, "[", Sys.getpid(), "]")))
> plot(gg)
>
```

Python Performance

Comparable to PyPy, the fastest alternative



Using Numpy from Java – Calling into Numpy

```
try (Context context = Context.newBuilder()
   .option("python.PythonPath", "/path/to/numpy-1.16.4-py3.7-macosx-10.14-x86_64.egg")
   .allowAllAccess(true).build()) {
 Value geomean = context.eval("python", "import numpy\n" + "import math\n" +
   "lambda x: math.pow(numpy.array(x).prod(), 1/len(x))");
```

anonymous function that calculates the geometric mean using numpy and the math module

```
 double[] values = new double[] { 1, 5, 8, 3, 5, 8, 8, 7, 5, 6 };
 double mean = geomean.execute(values).asDouble();
 System.out.println(mean);
 // 4.905181164183902

}
```

Using Numpy from Java – Arrays, Methods

```
Value arrayFunction = context.eval("python", "import numpy\n" + "numpy.array");
```

retrieve numpy’s array class

call “cumulative product” function

```
Value cumprod = arrayFunction.execute(values).invokeMember("cumprod");
System.out.println(cumprod);
// array([1.000e+00, 5.000e+00, 4.000e+01, 1.200e+02, 6.000e+02, 4.800e+03,...])
```

access array-like data structures (from numpy or any other language / library)

```
System.out.println(cumprod.getArraySize() + " / ” + cumprod.getArrayElement(6));
// 10 / 38400.0
```

Using Numpy from Java – Matrix, Operations

```
double[][] matrixValues = new double[][]
  { { 1,2,3,4 }, { 5,6,7,8 }, { 9,10,11,12 } };

Value matrix = arrayFunction.execute(matrixValues);
System.out.println(matrix);
// array([[ 1., 2., 3., 4.], [ 5., 6., 7., 8.], ...]
System.out.println(matrix.getArrayElement(2).getArrayElement(1));
// 11.0
```

also works
for matrixes

```
System.out.println(matrix.invokeMember("__add__", 1));
// array([[ 2., 3., 4., 5.], [ 6., 7., 8., 9.], ...]
```

addition via Python
“magic function”

```
Value plus = context.eval("python", "lambda x, y: x + y");
System.out.println(plus.execute(matrix, 1));
// array([[ 2., 3., 4., 5.], [ 6., 7., 8., 9.], ...]
```

addition via
Python snippet

Using Numpy from Java – Code in Files

```
class ALSModel():
  def __init__(self, iterations):
    self.iterations = iterations

  def fit(self, data):
    pass

  def infer(self, data):
    return len(data)

ALSModel
```

```
Value clazz = context.eval(Source.newBuilder("python", new File("mycode.py")).build());
assert clazz.canInstantiate();
```

create class instance

```
Value instance = clazz.newInstance(1234);
System.out.println(instance);
// <__main__.ALSModel object at 0x6b3b4f37>
System.out.println(instance.invokeMember("infer", new int[] { 1, 2, 3 }));
// 3
```

Using Numpy from Java – Code in Files

```
interface Inference {
    Object fit(int[] data);
    Object infer(int[] data);
}
```

```
Inference instance = clazz.newInstance(1234).as(Inference.class);

System.out.println(instance.infer(new int[] { 1, 2, 3 }));
// 3
```

call into Python class via interface functions

Using Numpy from Java – Modules

```
Value numpy = context.eval("python", "import numpy\n" + "numpy");
```

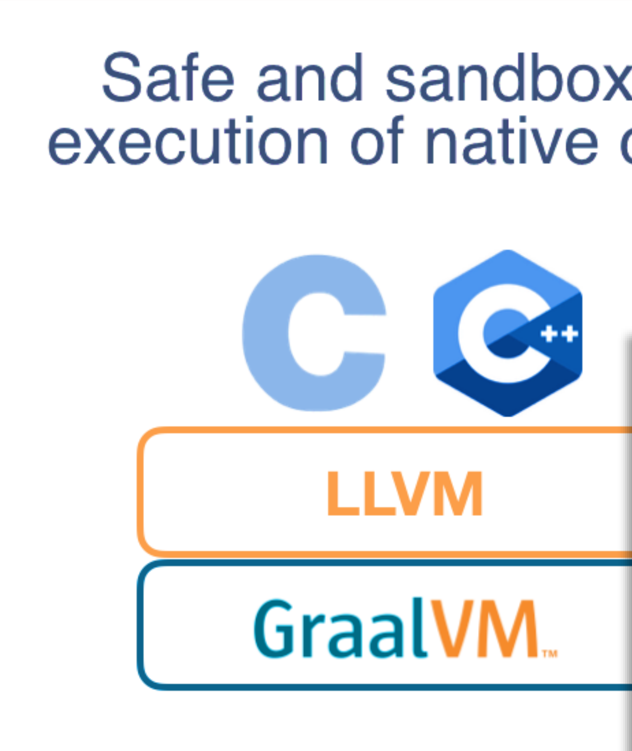
retrieve the numpy module

```
Value intArray = numpy.invokeMember("array", matrixValues, "int32");
```

use members of the module, e.g., the array constructor

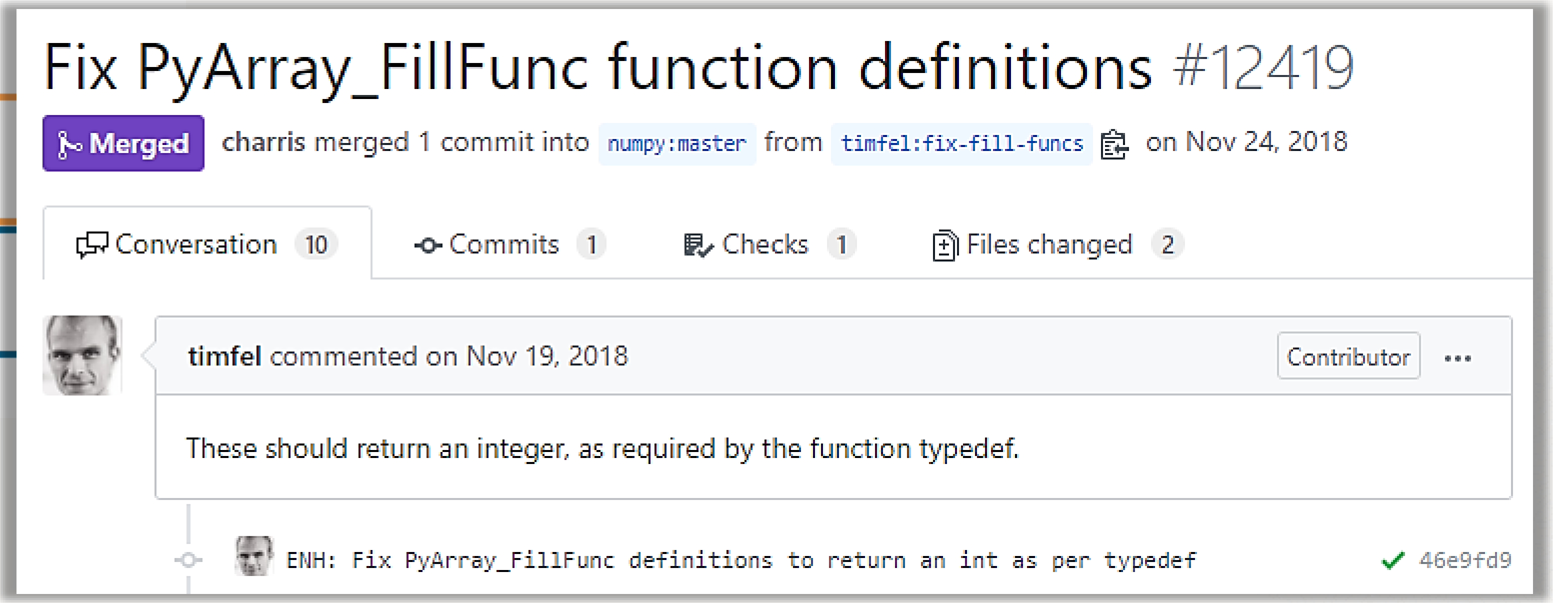
```
System.out.println(intArray.getMember("dtype"));
// dtype('int32')
```

“Managed Mode” Execution



Safe and sandboxed execution of native code

- LLVM Interpreter on GraalVM
- Run native extensions safely
- Catch memory errors as exceptions



Fix PyArray_FillFunc function definitions #12419

Merged charris merged 1 commit into `numpy:master` from `timfel:fix-fill-funcs` on Nov 24, 2018

timfel commented on Nov 19, 2018

These should return an integer, as required by the function typedef.

```
ENH: Fix PyArray_FillFunc definitions to return an int as per typedef
```

✓ 46e9fd9

Package Support

- Pip installer is not available, yet (c.f.: no sockets)
  - We ship our own installer ginstall
- Pandas and NumPy can be installed work for a wide range of code in the latest builds
  - Performance of native extensions is under active development
- SciPy support is currently work-in-progress
- More and more pure Python packages “just work” and compatibility is improving

Jython Replacement

- A “Jython-mode” gives Jython features in Python code
  - Direct import of Java packages and classes, exception handling with Python and Java exceptions, seamless interaction with Java objects
- Language-agnostic GraalVM embedder API replaces Jython-specific APIs
- Are you using Jython? We’re interested in supporting your use-case!

Using CUDA to Access Nvidia GPUs

- Different binding libraries / APIs for CUDA in different programming languages
- Varying set of supported features
- Translation to/from unmanaged environment (in Java, C#, Python, etc.)

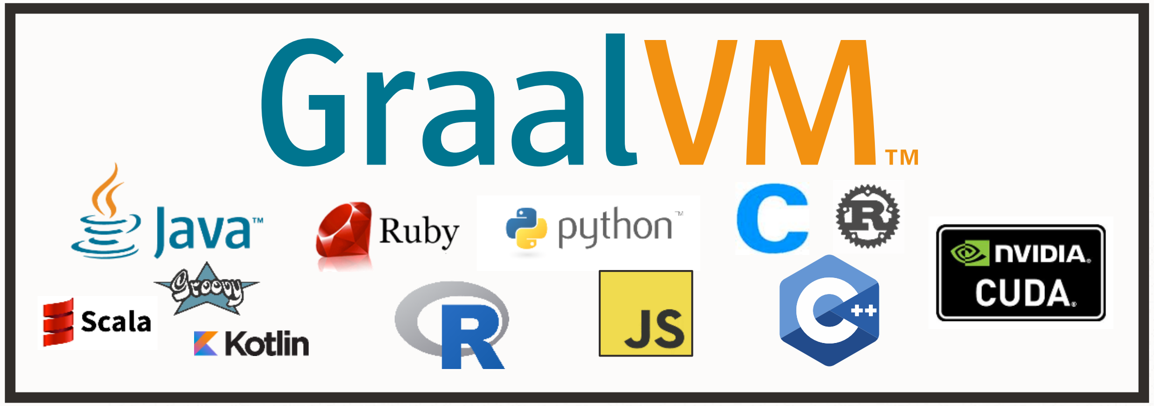
Python	Numba, cuPy, PyCUDA
Java	JCuda, jCUDA, CUDA4J
C / C++	CUDA C/C++ (language extension)
R	gpuR, indirectly through Rcpp
JS	gpu.js (WebGL), node-cuda, cuda-ts
C#	Hybridizer, ManagedCUDA, Alea GPU, ILGPU
Ruby	RbCUDA

Using grCUDA to Access Nvidia GPUs

- Efficient exchange of data between host language and GPU without burdening the programmer
- Expose GPU resources in ways that are native in the host language, e.g., as arrays
- Allow programmers to invoke existing GPU code from their host language
- Allow programmers to define new GPU kernels on the fly
- Polyglot interface: uniform bindings across several programming languages

- Implemented as a “Truffle Language” (although “CUDA” is a platform, not a language)

- Developed by NVIDIA in collaboration with Oracle Labs
- BSD 3-clause license



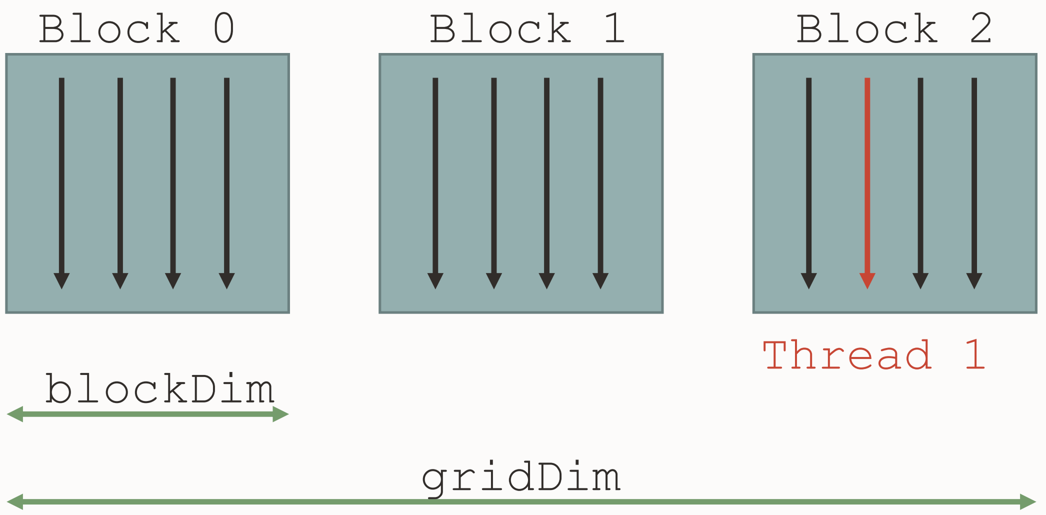
Creating and Using Device Arrays (Python)

```
import polyglot

# Get constructor function as callable
DeviceArray = polyglot.eval(language=‘grcuda’, string=‘DeviceArray’)

# Create 1D device array that can hold 1000 int values
dev_int_arr = DeviceArray(‘int’, 1000)

# Create 2D device array that can hold 1000 x 100 float values
dev_float_2d = DeviceArray(‘float’, 1000, 100)

# Setting array elements
for i in range(len(dev_int_array)):
    dev_int_arr[i] = i
for i in range(len(dev_float_2d)):
    for j in range(len(dev_float_2d[0])):
        dev_float_2d[i][j] = i + j
```

As fast or faster than native language constructs (e.g., R vectors) because of simpler semantics (e.g., no NA values)

GPU Kernels in CUDA C++

```
__global__ void inc_kernel(float *out_arr, const float *in_arr, size_t num_elements) {
  for (auto idx = blockIdx.x * blockDim.x + threadIdx.x; idx < num_elements;
           idx += gridDim.x * blockDim.x) {
      out_arr[idx] = in_arr[idx] + 1;
  }
}
```



Number of blocks and threads can be configured in 3 dimensions (x, y, z)

Launching GPU Kernels (JS)

```
const DeviceArray = Polyglot.eval('grcuda', string='DeviceArray')
const N = 1000
const in_arr = DeviceArray('float', N)
const out_arr = DeviceArray('float', N)
for (let i = 0; i < N; i++)
  in_arr[i] = i
const code = '__global__ void inc_kernel(...) ...'
const buildkernel = Polyglot.eval('grcuda', string='buildkernel')
const incKernel = buildkernel(code, 'inc_kernel', 'pointer, pointer, uint64')

// Launch kernel in grid consisting of 160 blocks with 256 threads each
incKernel(160, 256)(out_arr, in_arr, N)

for (let i = 0; i < 10; i++) {
  console.log(out_arr[i]);
}
```

Device arrays `in_arr` and `out_arr` can be passed to GPU kernel

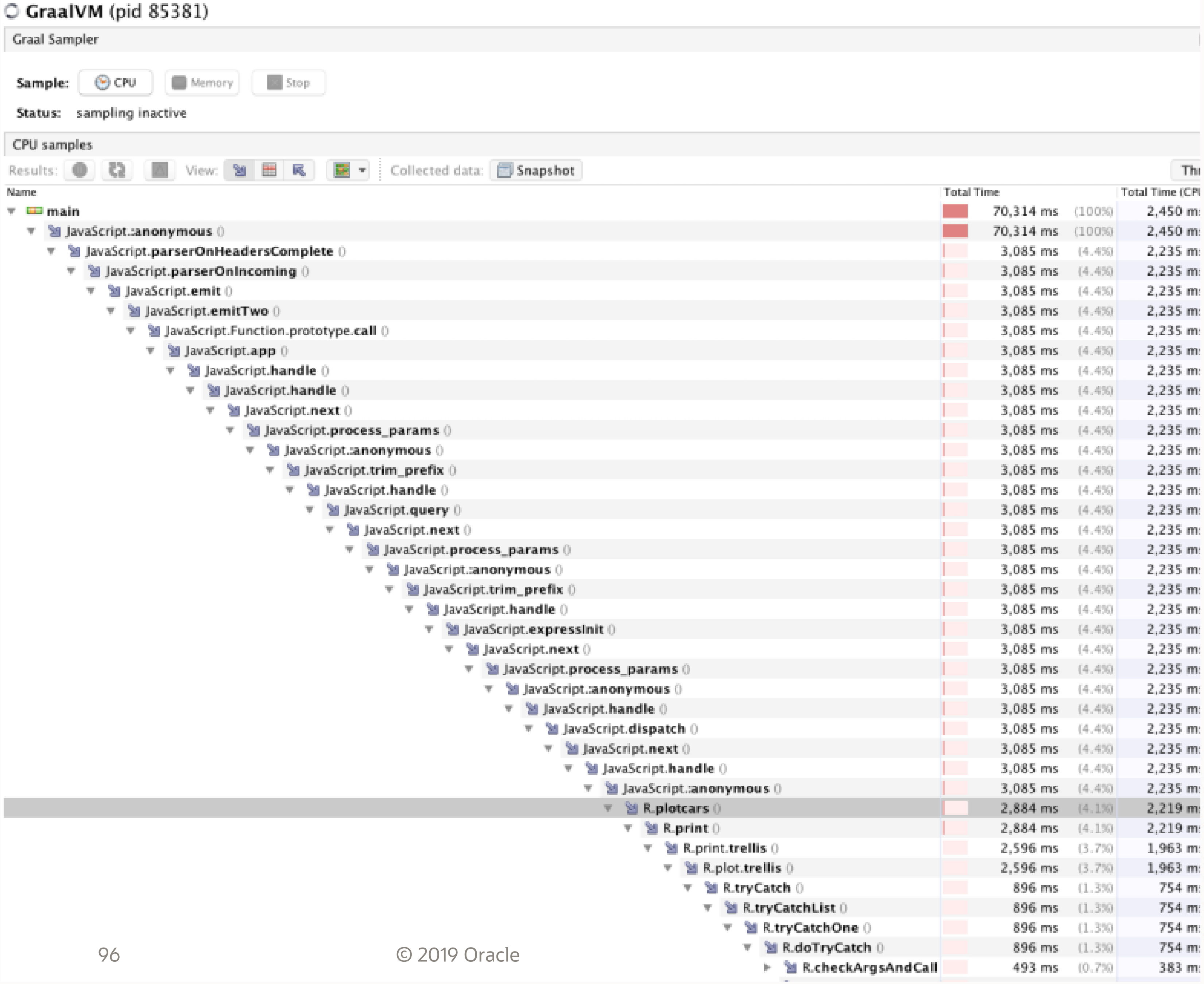
grCUDA

- … and a lot more, e.g.:
    - Use existing GPU-accelerated libraries with DeviceArray objects
    - Load pre-compiled kernels from .ptx and .cubin files
    - Query GPU device properties, e.g., grid and memory sizes
- More features planned, e.g.:
    - Device Memory managed by grCUDA using explicit transfers.
    - Asynchronous execution of copy and kernel launches.
    - Conversion and shredding of host-language objects (array-of-structs vs. struct-of-arrays)

Get it from https://github.com/NVIDIA/grcuda
More info on Nvidia's Blog: https://blogs.nvidia.com/

VisualVM

Polyglot Stack Trace



GraalVM (pid 85381)

Graal Sampler

Sample: CPU Memory Stop

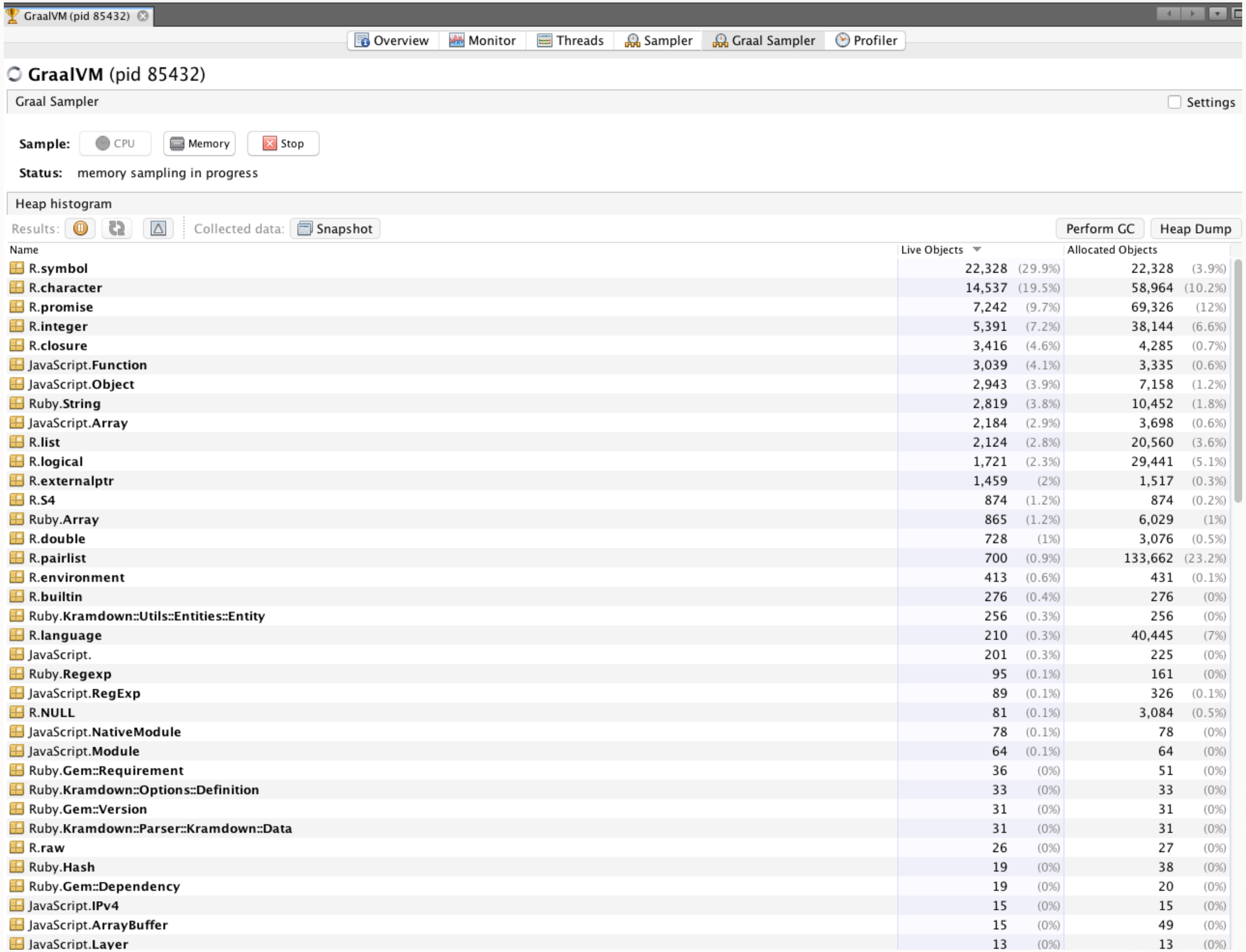
Status: sampling inactive

CPU samples

Name	Total Time		Total Time (CP
main	70,314 ms	(100%)	2,450 m
JavaScript.:anonymous ()	70,314 ms	(100%)	2,450 m
JavaScript.parserOnHeadersComplete ()	3,085 ms	(4.4%)	2,235 m
JavaScript.parserOnIncoming ()	3,085 ms	(4.4%)	2,235 m
JavaScript.emit ()	3,085 ms	(4.4%)	2,235 m
JavaScript.emitTwo ()	3,085 ms	(4.4%)	2,235 m
JavaScript.Function.prototype.call ()	3,085 ms	(4.4%)	2,235 m
JavaScript.app ()	3,085 ms	(4.4%)	2,235 m
JavaScript.handle ()	3,085 ms	(4.4%)	2,235 m
JavaScript.handle ()	3,085 ms	(4.4%)	2,235 m
JavaScript.next ()	3,085 ms	(4.4%)	2,235 m
JavaScript.process_params ()	3,085 ms	(4.4%)	2,235 m
JavaScript.:anonymous ()	3,085 ms	(4.4%)	2,235 m
JavaScript.trim_prefix ()	3,085 ms	(4.4%)	2,235 m
JavaScript.handle ()	3,085 ms	(4.4%)	2,235 m
JavaScript.query ()	3,085 ms	(4.4%)	2,235 m
JavaScript.next ()	3,085 ms	(4.4%)	2,235 m
JavaScript.process_params ()	3,085 ms	(4.4%)	2,235 m
JavaScript.:anonymous ()	3,085 ms	(4.4%)	2,235 m
JavaScript.trim_prefix ()	3,085 ms	(4.4%)	2,235 m
JavaScript.handle ()	3,085 ms	(4.4%)	2,235 m
JavaScript.expressInit ()	3,085 ms	(4.4%)	2,235 m
JavaScript.next ()	3,085 ms	(4.4%)	2,235 m
JavaScript.process_params ()	3,085 ms	(4.4%)	2,235 m
JavaScript.:anonymous ()	3,085 ms	(4.4%)	2,235 m
JavaScript.handle ()	3,085 ms	(4.4%)	2,235 m
JavaScript.dispatch ()	3,085 ms	(4.4%)	2,235 m
JavaScript.next ()	3,085 ms	(4.4%)	2,235 m
JavaScript.handle ()	3,085 ms	(4.4%)	2,235 m
JavaScript.:anonymous ()	3,085 ms	(4.4%)	2,235 m
R.plotcars ()	2,884 ms	(4.1%)	2,219 m
R.print ()	2,884 ms	(4.1%)	2,219 m
R.print.trellis ()	2,596 ms	(3.7%)	1,963 m
R.plot.trellis ()	2,596 ms	(3.7%)	1,963 m
R.tryCatch ()	896 ms	(1.3%)	754 m
R.tryCatchList ()	896 ms	(1.3%)	754 m
R.tryCatchOne ()	896 ms	(1.3%)	754 m
R.doTryCatch ()	896 ms	(1.3%)	754 m
R.checkArgsAndCall	493 ms	(0.7%)	383 m

VisualVM

Polyglot Heap Dump



GraalVM (pid 85432)

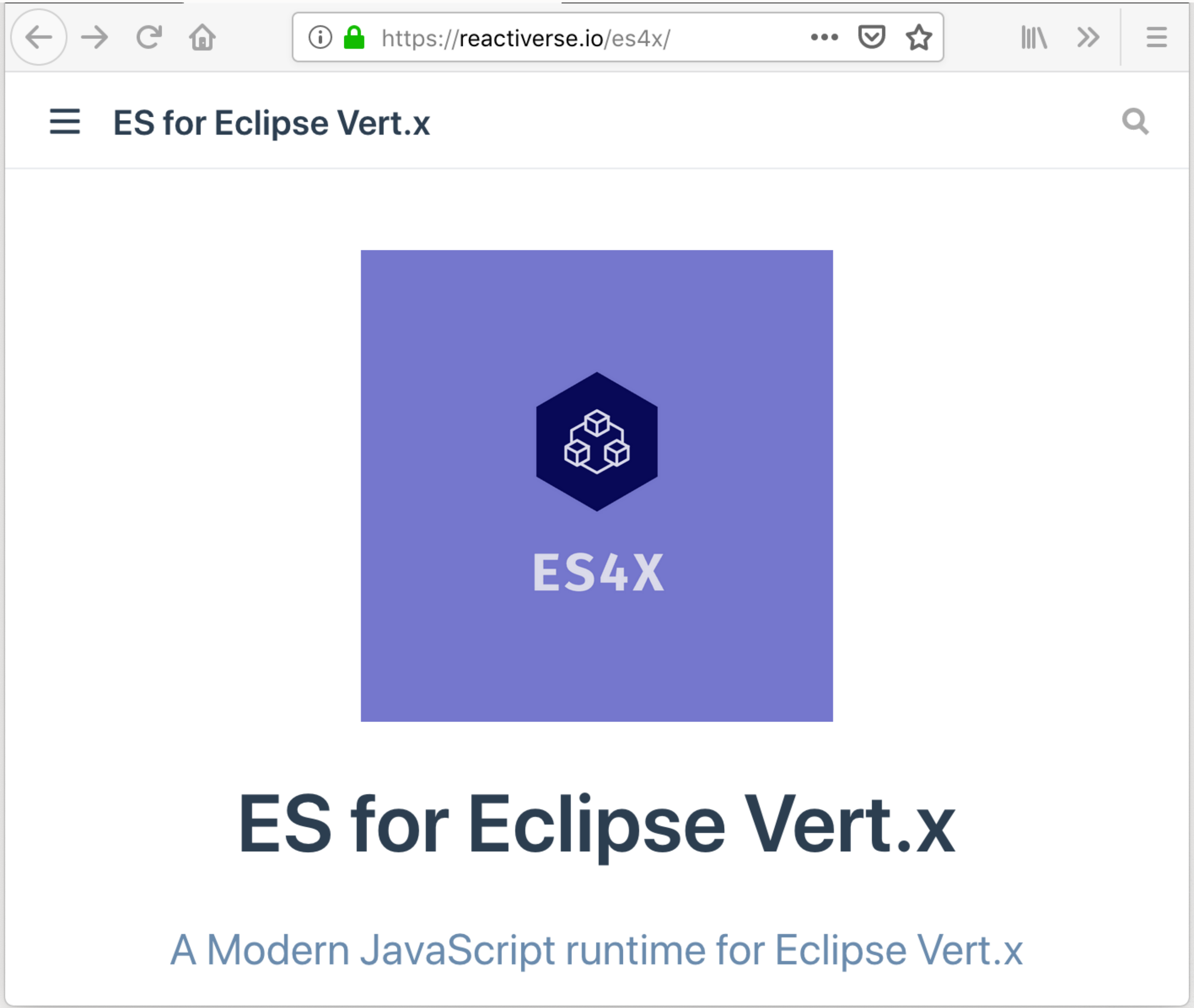
Graal Sampler

Sample: CPU Memory Stop

Status: memory sampling in progress

Heap histogram

Name	Live Objects		Allocated Objects	
R.symbol	22,328	(29.9%)	22,328	(3.9%)
R.character	14,537	(19.5%)	58,964	(10.2%)
R.promise	7,242	(9.7%)	69,326	(12%)
R.integer	5,391	(7.2%)	38,144	(6.6%)
R.closure	3,416	(4.6%)	4,285	(0.7%)
JavaScript.Function	3,039	(4.1%)	3,335	(0.6%)
JavaScript.Object	2,943	(3.9%)	7,158	(1.2%)
Ruby.String	2,819	(3.8%)	10,452	(1.8%)
JavaScript.Array	2,184	(2.9%)	3,698	(0.6%)
R.list	2,124	(2.8%)	20,560	(3.6%)
R.logical	1,721	(2.3%)	29,441	(5.1%)
R.externalptr	1,459	(2%)	1,517	(0.3%)
R.S4	874	(1.2%)	874	(0.2%)
Ruby.Array	865	(1.2%)	6,029	(1%)
R.double	728	(1%)	3,076	(0.5%)
R.pairlist	700	(0.9%)	133,662	(23.2%)
R.environment	413	(0.6%)	431	(0.1%)
R.builtin	276	(0.4%)	276	(0%)
Ruby.Kramdown::Utils::Entities::Entity	256	(0.3%)	256	(0%)
R.language	210	(0.3%)	40,445	(7%)
JavaScript.	201	(0.3%)	225	(0%)
Ruby.Regexp	95	(0.1%)	161	(0%)
JavaScript.RegExp	89	(0.1%)	326	(0.1%)
R.NULL	81	(0.1%)	3,084	(0.5%)
JavaScript.NativeModule	78	(0.1%)	78	(0%)
JavaScript.Module	64	(0.1%)	64	(0%)
Ruby.Gem::Requirement	36	(0%)	51	(0%)
Ruby.Kramdown::Options::Definition	33	(0%)	33	(0%)
Ruby.Gem::Version	31	(0%)	31	(0%)
Ruby.Kramdown::Parser::Kramdown::Data	31	(0%)	31	(0%)
R.raw	26	(0%)	27	(0%)
Ruby.Hash	19	(0%)	38	(0%)
Ruby.Gem::Dependency	19	(0%)	20	(0%)
JavaScript.IPv4	15	(0%)	15	(0%)
JavaScript.ArrayBuffer	15	(0%)	49	(0%)
JavaScript.Layer	13	(0%)	13	(0%)

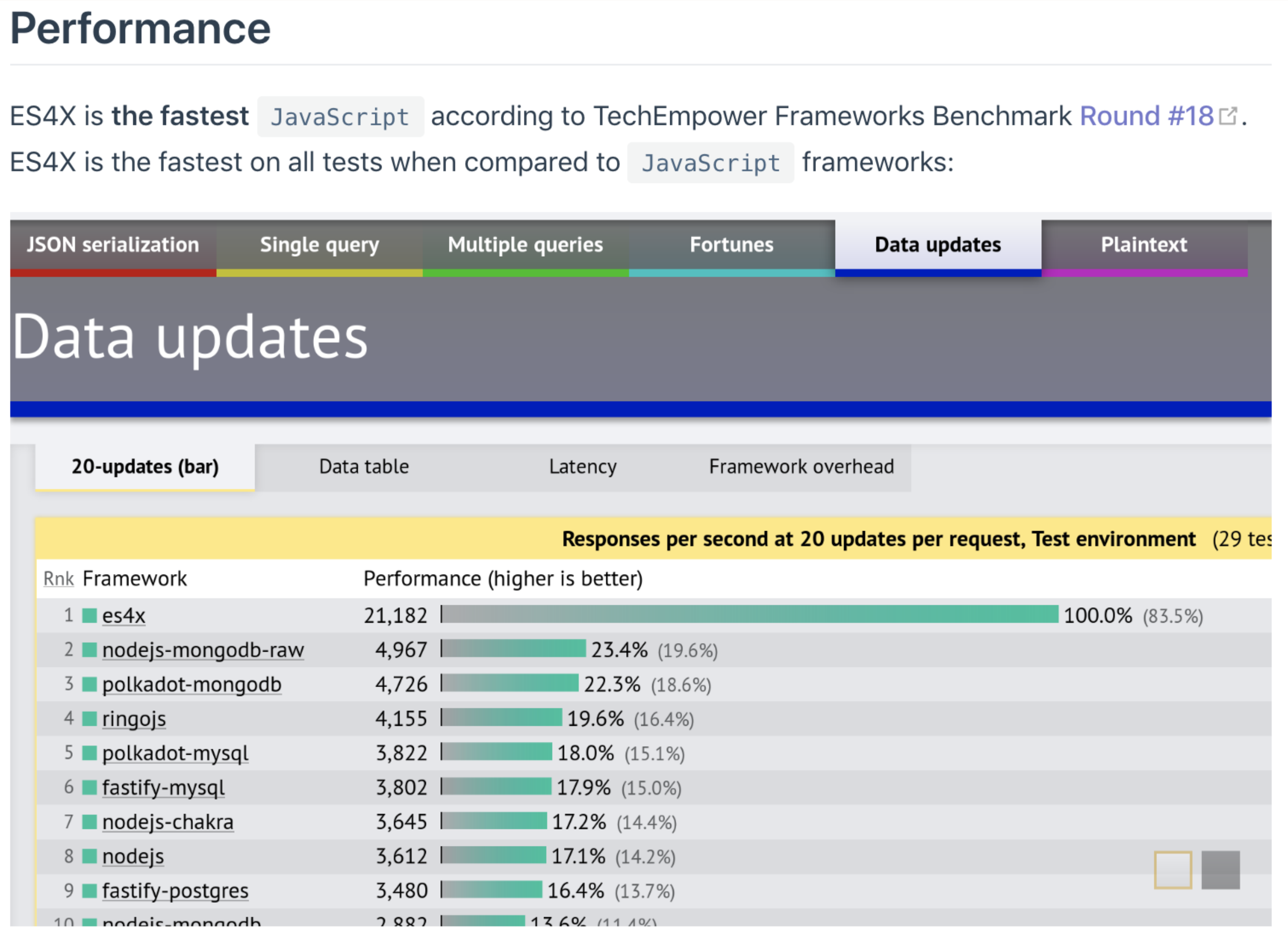


ES for Eclipse Vert.x

A Modern JavaScript runtime for Eclipse Vert.x

Performant

ES4X runs on top of GraalVM offering a great performance for JavaScript



Performance

ES4X is **the fastest** `JavaScript` according to TechEmpower Frameworks Benchmark Round #18.
ES4X is the fastest on all tests when compared to `JavaScript` frameworks:

Data updates

Responses per second at 20 updates per request, Test environment (29 tes

Rnk	Framework	Performance (higher is better)		
1	🟩 es4x	21,182	100.0%	(83.5%)
2	🟩 nodejs-mongodb-raw	4,967	23.4%	(19.6%)
3	🟩 polkadot-mongodb	4,726	22.3%	(18.6%)
4	🟩 ringojs	4,155	19.6%	(16.4%)
5	🟩 polkadot-mysql	3,822	18.0%	(15.1%)
6	🟩 fastify-mysql	3,802	17.9%	(15.0%)
7	🟩 nodejs-chakra	3,645	17.2%	(14.4%)
8	🟩 nodejs	3,612	17.1%	(14.2%)
9	🟩 fastify-postgres	3,480	16.4%	(13.7%)
10	🟩 nodejs-mongodb	2,882	13.6%	(11.4%)

Summary

Key Performance Takeaways

- Write small methods
- Local allocations are free, global data structures expensive
- Don’t hand optimize, unless you have studied the compiler graph
  - Make a pull request (or at least an issue) for the GraalVM project!

- For best throughput use GraalVM JIT,
  for best startup & footprint use GraalVM AOT (native images)

Top 3 Misconceptions about GraalVM

assert(“GraalVM does not support reflection and can run only a subset of Java”)
=> false

assert(“GraalVM is a competition to the HotSpot JVM technology”)
=> false

assert(“GraalVM is just a research project and not ready for production”)
=> false

Production-Ready

Java

Scala, Groovy, Kotlin

JavaScript

Node.js

Native Image

VisualVM

Experimental

Ruby

R

LLVM Toolchain

Visionary

Python

VSCode Plugin

GPU Integration

Webassembly

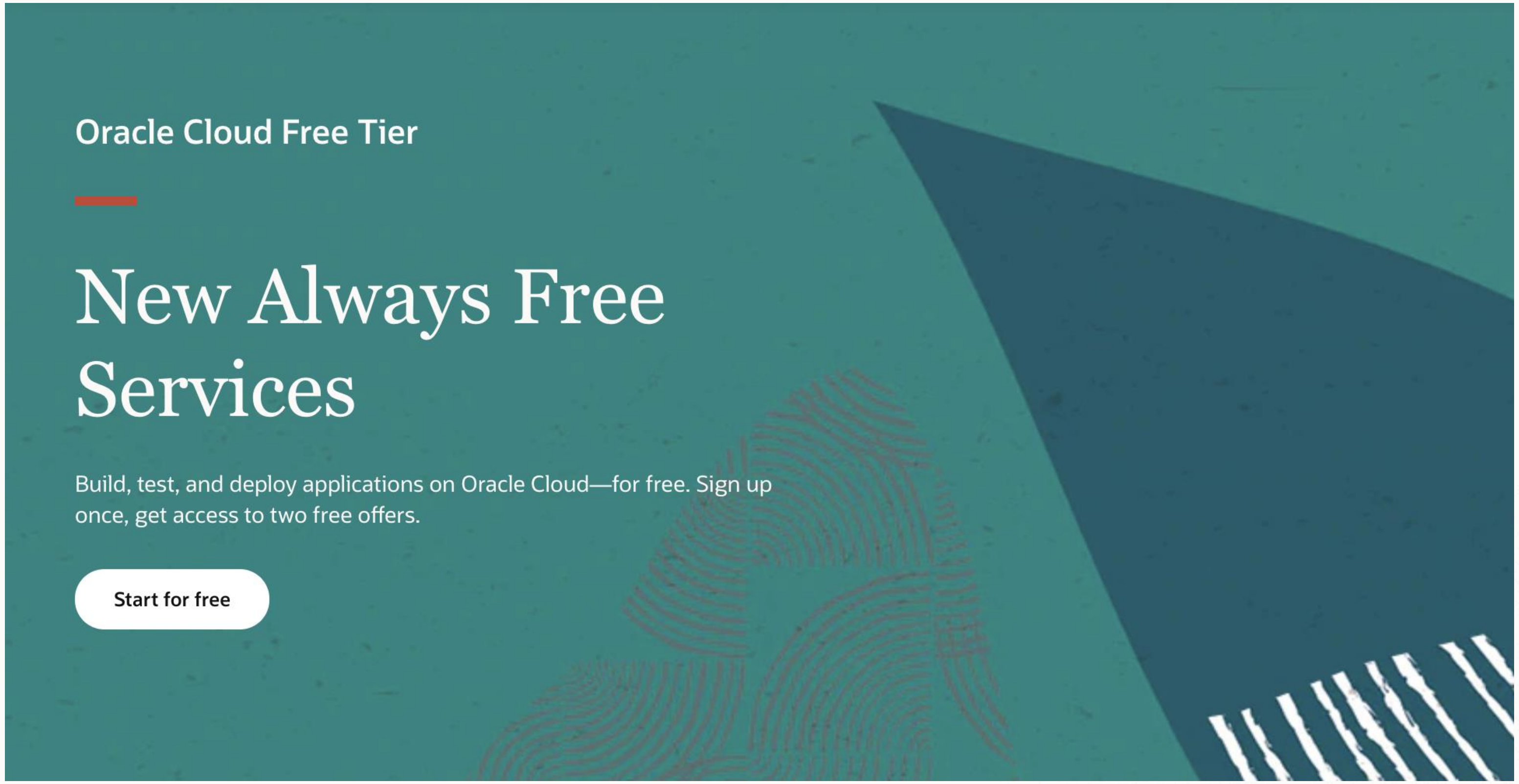
LLVM Backend

Oracle GraalVM Enterprise Edition

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

www.oracle.com/cloud/free/

2 GByte instance = 60+ GraalVM native images ;)



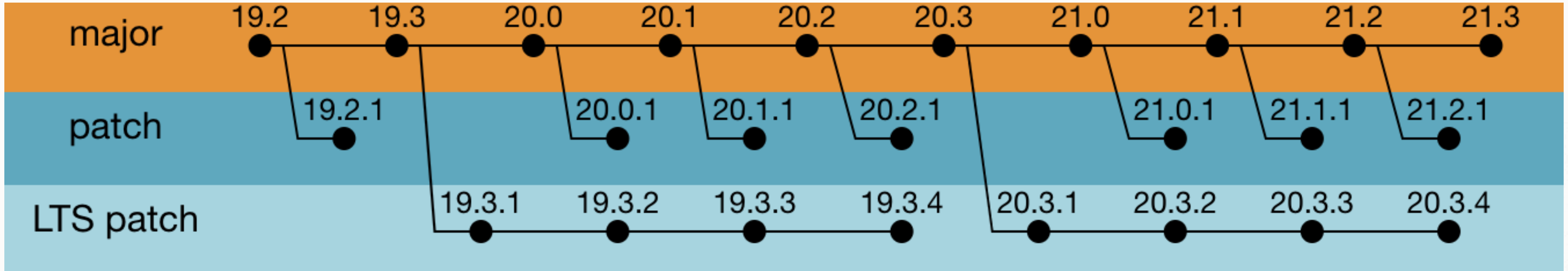
Oracle Cloud Free Tier

New Always Free Services

Build, test, and deploy applications on Oracle Cloud—for free. Sign up once, get access to two free offers.

Versioning

- Major release every 3 months named YEAR.x
- Both major and CPU releases on predictable dates
- Last major release of a year receives patches for the following year



More Platform Support

- JDK11 support will be in GraalVM 19.3 (November 19, 2019)
- Working on:
    - ARM64 support
    - Windows support
- Considering to add also JDK-latest version

GraalVM Value Proposition

1. High performance for abstractions of any language

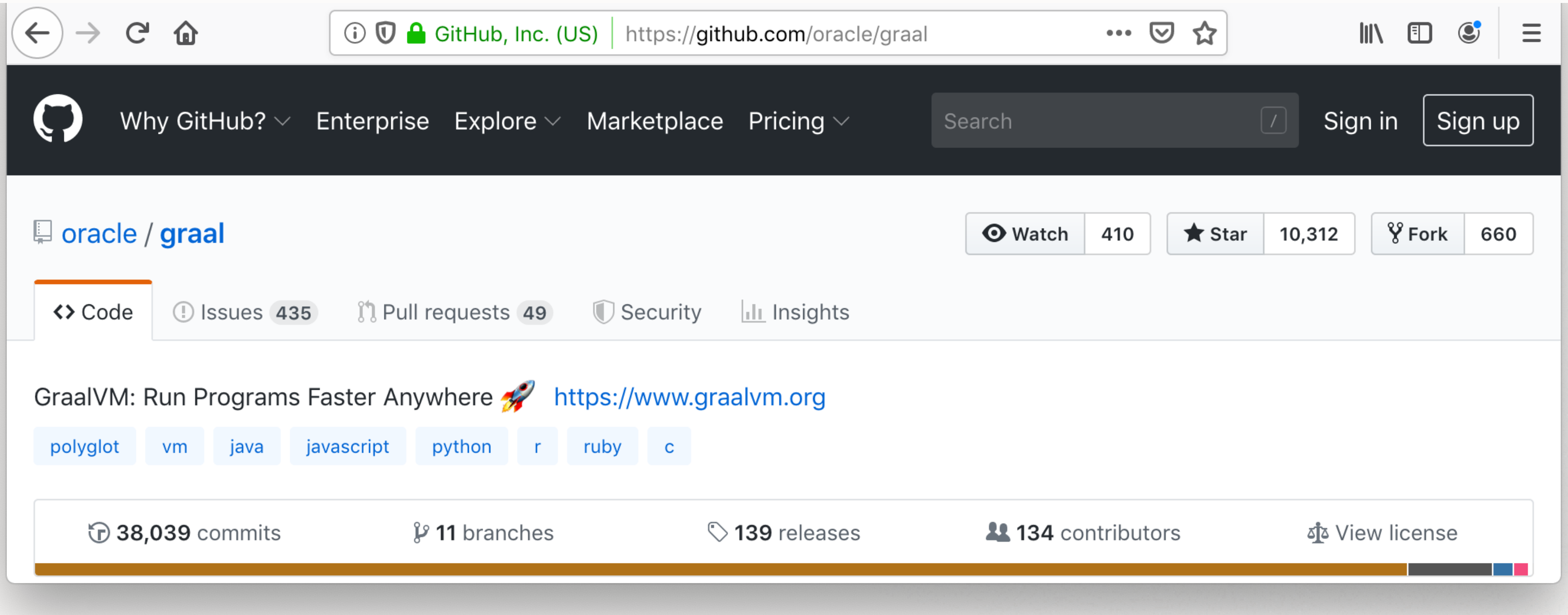
2. Low footprint ahead-of-time mode for JVM-based languages

3. Convenient language interoperability and polyglot tooling

4. Simple embeddability in native and managed programs

Download and use it for your application!

- 100% production ready
- https://www.graalvm.org
- https://github.com/graalvm/
- @graalvm



GraalVM: Run Programs Faster Anywhere 🚀 https://www.graalvm.org

38,039 commits | 11 branches | 139 releases | 134 contributors | View license

Thank you