GEBIT Solutions

Die Experten für Java, Anwendungsentwicklung und Requirements Engineering

# Graal oder nicht Graal - ist das hier die Frage?

Der neue Just-In-Time-Compiler und weitere Entwicklungen in der Java VM.

Stephan Frind

# Motivation

- **Performance kann wichtig sein**
- **twitter 2015**
  - 568.000.000 US$ für Rechenzentren (Hard-, Software, Strom, Löhne)
    - Annahme 200.000.000 US$ nur für Rechenleistung (Hardware + Strom)
      - 10% Verbesserung -> 20.000.000 US$ / Jahr
- **google 2014**
  - 10.000.000.000 US$ für Rechenzentren

- **Performanceverbesserung**
  - wenige % = $$$

# Motivation

- **Performance**
  - Java Interpreter
  - JIT: C1 / C2
    - verschiedene Level (Profiling, Optimization)
    - bytecode -> Maschinencode

# Motivation

- **Performance**
  - Es gibt Java Programme die schneller als C Programme laufen und semantisch das gleiche tun.
  - FreeTTS (Java) versus Flite (C++)
    - Sprachsynthese Software
    - gleiche Logik
    - Benchmarks in 2002
      - in den meisten Tasks ist FreeTTS schneller

# Motivation

- **C1 + C2**
  - sehr komplex, viele Hacks
  - Lebensende erreicht
  - kaum Weiterentwicklung

# Geschichte

- **2006 Uni Linz Forschung zu JIT Compiler**
- **2009 Thomas Wuerthinger portiert C++ Jit (C1, C2) nach Java**
- **2011 Graal Project Oracle**
  - „replace c++ parts of JVM by high qualitity java code“ (metropolis)
- **2014 JEP 243**
  - JVMCI (ab java 9)
  - JIT austauschbar
- **2017 Graal JIT bei Twitter in Produktion**

# Graal

- **JIT Compiler in Java**
- **Open-Source**

- **Ziele**
  - hohe Performance
    - eigene Laufzeit
    - des erzeugten Codes
  - polyglott
    - verschiedene Sprachen in der gleichen VM
    - ohne Overhead
    - z.B.: javascript, python, java parallel im gleichen Speicher ausführen
  - AOT

# Graal Optimierungen

- **partial evaluation**
  - Annahme, was bisher geschah gilt weiter
- **inlining**
  - Methodenaufrufe durch Body der Methode ersetzen
  - Voraussetzung für weitere Optimierungen
- **escape analyse**
  - reduziert Objekterzeugungen
  - Objekte auf dem Stack und nicht auf dem Heap erzeugen
  - optimiertes Locking
- **...**

- **Abstraktion wird zugunsten der Performance entfernt**

# Performance Benchmarks

- **schwierig**
  - JIT
  - deadcode elimination
  - data
    - z.B. Sortieren sortierter Daten
- **jmh**
  - Java Microbenchmark Harness
  - http://openjdk.java.net/projects/code-tools/jmh/

# Performance Benchmarks

falsch:

```
@Benchmark
public void measureWrong() {
        Math.log(x);
}
```

# Performance Benchmarks

falsch:

```
@Benchmark
public void measureWrong() {
        Math.log(x);
}
```

richtig:

```
@Benchmark
public double measureRight() {
        return Math.log(x);
}
```

# Graal Performance

- **im Moment im Allgemeinen nicht schneller als der alte JIT**
- **Vorteile bei verschiedenen Workloads**
  - scala code
    - scala-benchmark-suite  20% schneller
  - Highlevel Programme
    - java 8 streams (manche MicroBenchmarks >10x schneller)

- **Twitter nutzt Graal in Produktion**
  - 15% CPU Einsparung laut Christian Thalinger (Engineer Twitter)
  - $$$

# Graal Performance

- **Start eventuell ein weniger länger, wenn ohne AOT**
  - auch wenn mit freier CPU compiliert würde
- **ca. 40MB mehr Heap**
- **ca. 20MB mehr Metaspace**
- **Speicher wird nach Compilierung wieder freigegeben**
- **dafür weniger native Memory**

# Graal Performance

- **falls java >= 9**
  - mit Graal testen
- **environment JAVA_TOOL_OPTIONS**
  - --module-path org.graalvm.graal_sdk.jar
  - --upgrade-module-path jdk.internal.vm.compiler.jar
- **Linux**
  - export JAVA_TOOL_OPTIONS=--module-path=/C/temp/graalsf/data/org.graalvm.graal_sdk.jar --upgrade-module-path=/C/temp/graalsf/data/jdk.internal.vm.compiler.jar -XX:+EnableJVMCI -XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler
- **Windows**
  - set JAVA_TOOL_OPTIONS=--module-path C:\temp\graalsf\data\ --upgrade-module-path C:\temp\graalsf\data\ -XX:+UnlockExperimentalVMOptions -XX:+EnableJVMCI -XX:+UseJVMCICompiler

# Graal Performance

- **-XX:+PrintCompilation**
  - Level 0: interpreted code
  - Level 1: simple C1 compiled code (with no profiling)
  - Level 2: limited C1 compiled code (with light profiling)
  - Level 3: full C1 compiled code (with full profiling)
  - Level 4: C2/Graal compiled code (uses profile data from the previous steps)

## Beispiel

```
244    13    3   java.lang.String::equals (65 bytes)
244    14    1   java.util.ImmutableCollections$Set0::hashCode (2 bytes)
244    15    1   java.util.Collections$EmptySet::hashCode (2 bytes)
1477  277    4   java.lang.String::equals (65 bytes)
```

# Graal Performance

- **Test, ob Graal benutzt wird**
  - -XX:+PrintCompilation

**674 537 3 org.graalvm.compiler.hotspot.HotSpotGraalCompilerFactory::adjustCompilationLevelInternal (70 bytes)**

**674 538 3 jdk.vm.ci.hotspot.HotSpotJVMCIRuntime::adjustCompilationLevel (203 bytes)**

**675 539 3 org.graalvm.compiler.hotspot.HotSpotGraalCompilerFactory::adjustCompilationLevel (9 bytes)**

**675 526 3 java.util.AbstractList$Itr::hasNext (20 bytes)**

# AOT

- **erzeugt nativen Code zur Compilezeit**
- **ab Java 9  (Java 8 Backport existiert)**
- **beschränkt auf die Platform auf der compiliert wurde**
- **setzt C-Compile Umgebung voraus**
  - nutzt Libs fürs Schreiben des Binaryformats
  - Linux z.B. apt-get -y install build-essential
  - Windows: Visual Studio Community 2017
- **Vorteile**
  - schnellere Startzeit
    - z.B. für kleine Programme wie Tools
  - ist sicher
    - Codechange Detection
    - wie JIT („Code“ Cache)

# AOT Beispiel

- **linux**
  - jaotc –-output libTest.so Test.class
  - java –XX:AOTLibrary=./libTest.so Test
- **windows**
  - jaotc --linker-path "C:\Program Files (x86)\Microsoft Visual Studio\2017\...\bin\Hostx64\x64\link.exe" –-output libTest.dll Test.class
  - java –XX:AOTLibrary=./libTest.dll Test
- **Test**
  - java -XX:+PrintAOT -XX:AOTLibrary=./libTest.dll de.frind.test.ListDir

```
238   1    loaded   ./libTest.dll  aot library
424   1    aot[ 1]  de.frind.test.ListDir.<init>()V
425   2    aot[ 1]  de.frind.test.ListDir.main([Ljava/lang/String;)V
426   3    aot[ 1]  de.frind.test.ListDir.lambda$2(Ljava/nio/file/Path;)Z
427   4    aot[ 1]  de.frind.test.ListDir.listDir(Ljava/lang/String;)V
```

# AOT Performance

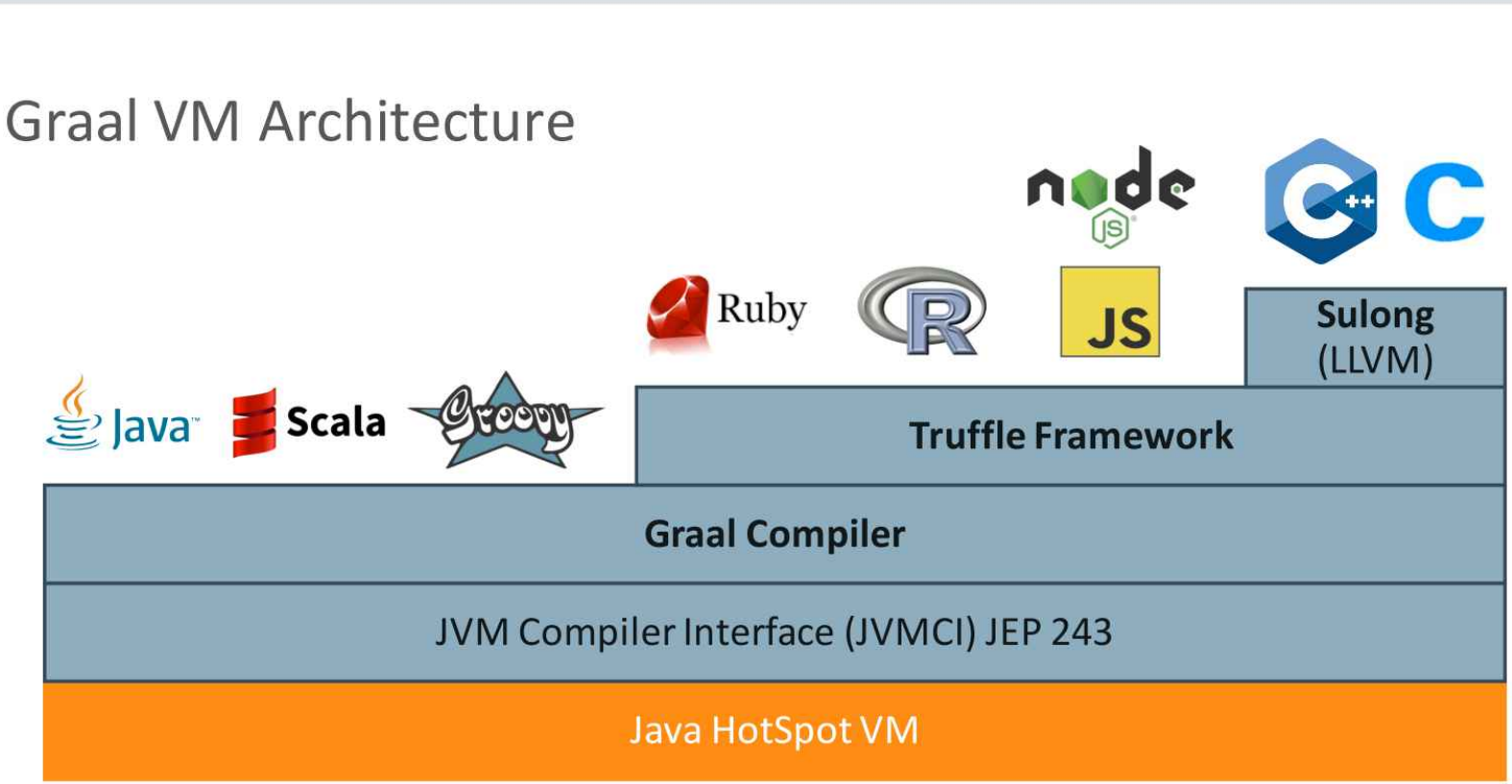
## list dir java programm

| normal | 0,130s |
|---|---|
| custom image | 0,110s |
| custom image + AOT | 0,103s |

## fibonnaci von 1_000_000

| normal | 0,800s |
|---|---|
| custom image | 0,840s |
| custom image + AOT | 0,460s |

# GraalVM




ORACLE

# GraalVM

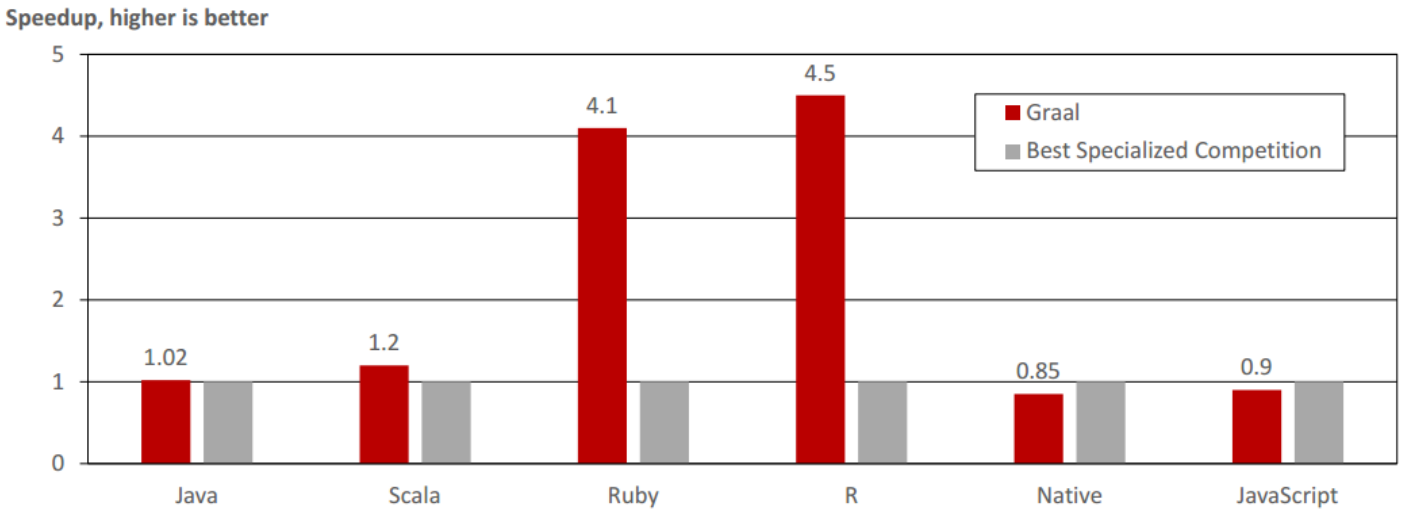
- **enthält**
  - Graal Compiler
  - Truffle
  - Languages
    - JavaScript
    - R
    - Ruby
    - LLVM (C, C++, Fortran, Cobol)
      - Input: LLVM Bitcode
    - ...
  - Native Image Generation Tool
- **9.05.2019: Version 19.0 “first production release“**

# GraalVM

- **OpenJDK**
- **standalone (native)**
- **embedded (StoredProcedures ...)**
  - Oracle Database MLE
  - MySQL 8
  - PostgreSQL

# Graal Performance

## Performance: GraalVM Summary



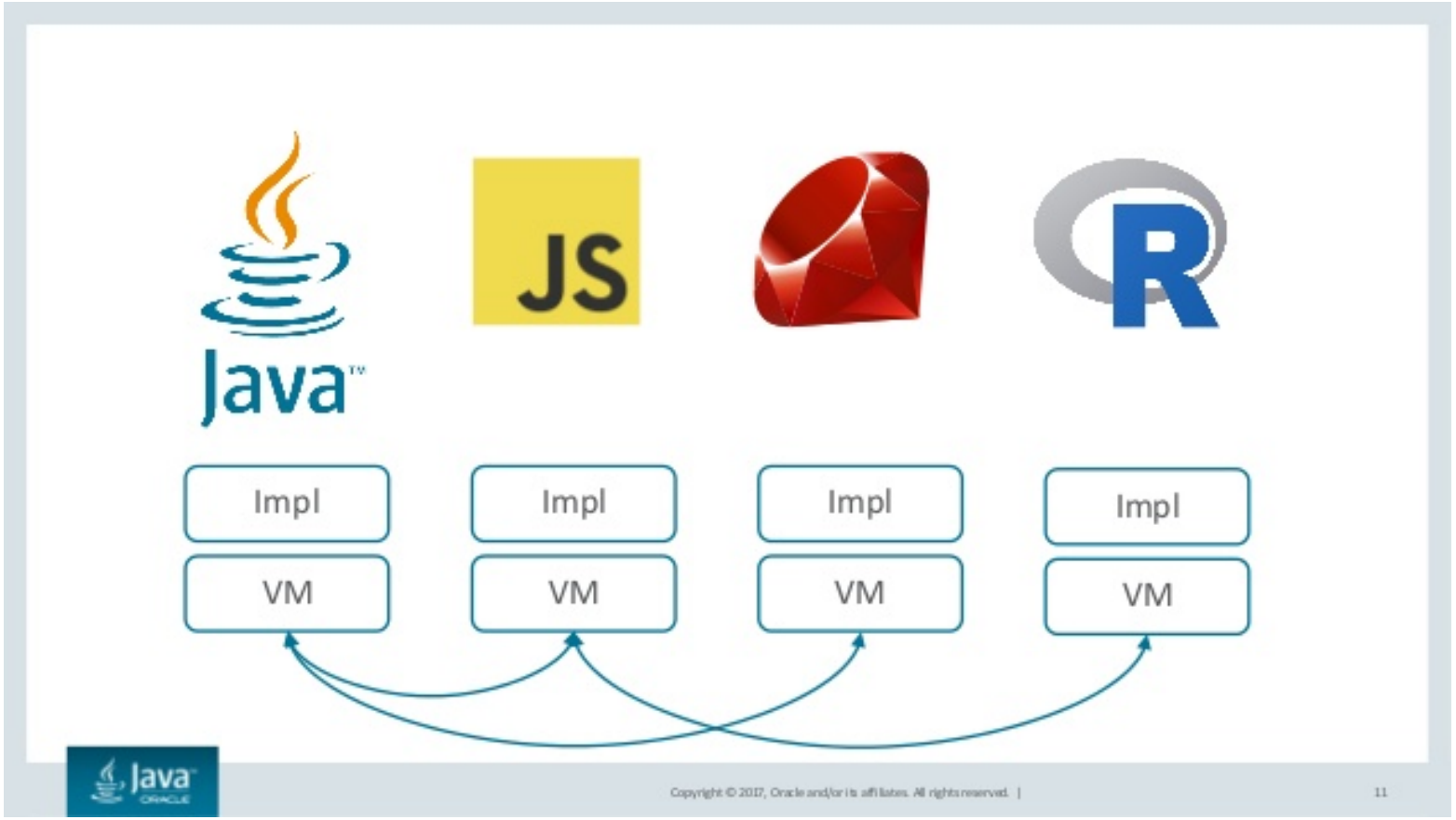

**Performance relative to:**
**HotSpot/Server, HotSpot/Server running JRuby, GNU R, LLVM AOT compiled, V8**

ORACLE

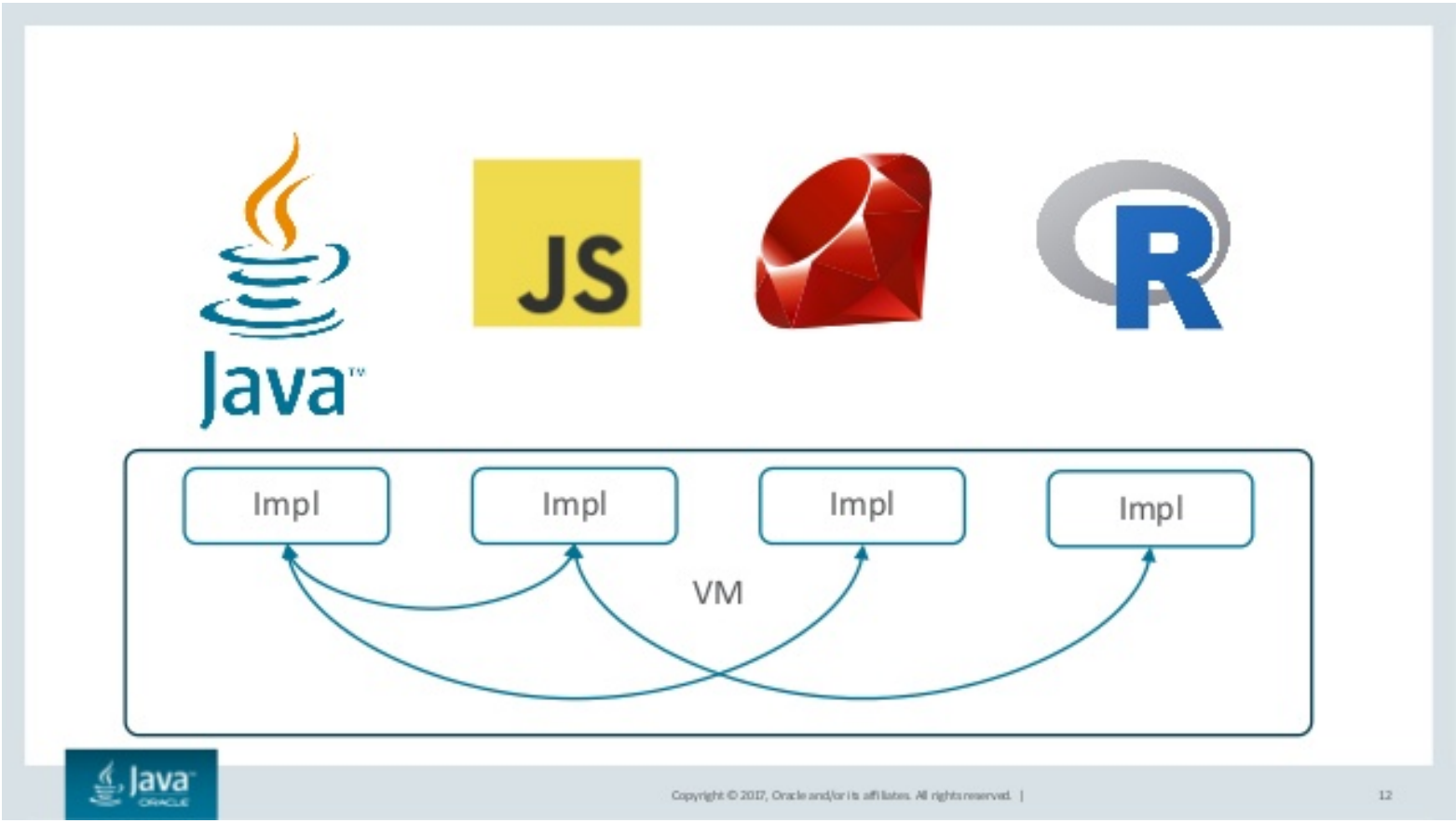
# Truffle

- **Framework für Interpreter**
  - baut AST
- **einfach(er) zu implementieren als JIT**
- **Graal macht es dann schnell**
  - Partial Evaluation
  - Escape Analyse
  - ...

# Polyglot

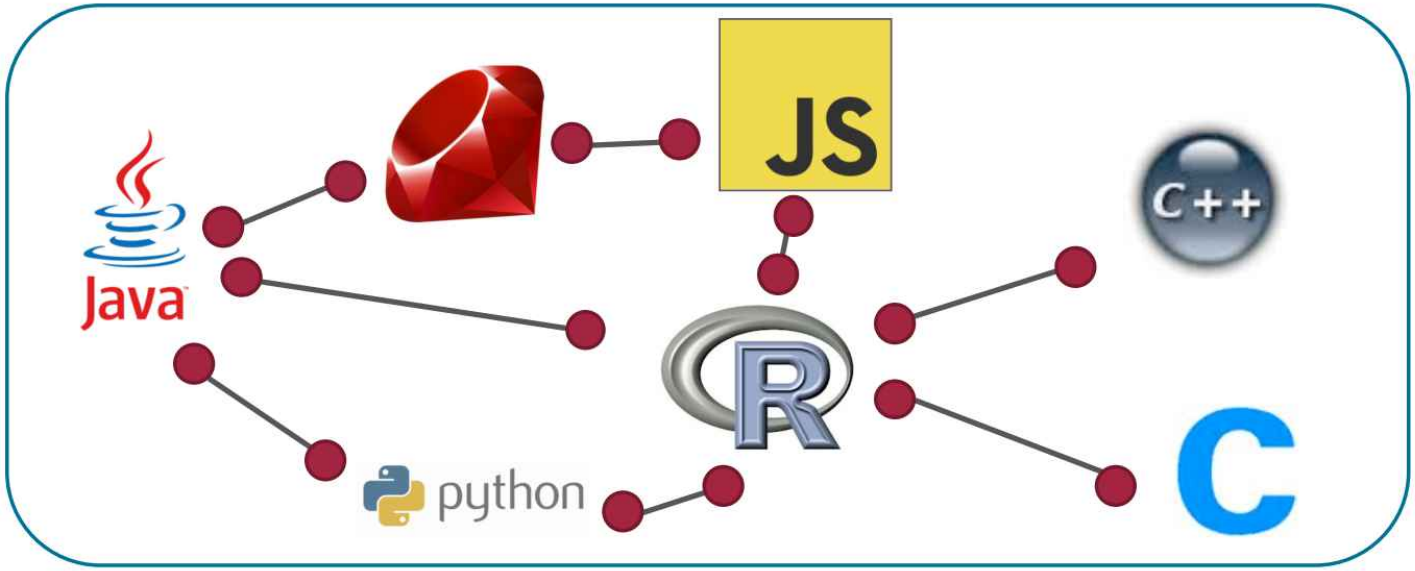


# Polyglot



# Polyglot

## Zero Overhead Interoperability




ORACLE

# Polyglot

```
import org.graalvm.polyglot.Context;

public class Test {
  public static void main(String[] args) {
    System.out.println("Hello Java!");
    Context context = Context.create();
    context.eval("js", "print('Hello JavaScript!');");
  }
}
```

getestet mit graalvm-1.0.0.-rc1 unter linux

# Polyglot

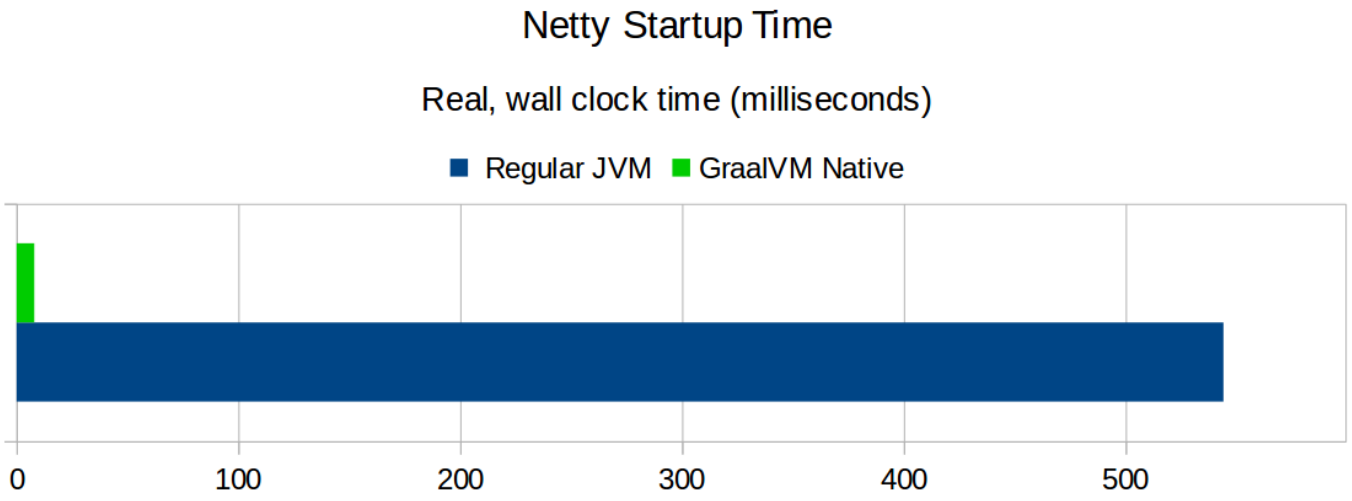
- **Ökosysteme der Sprachen**
  - große Communities
    - JS, R, Phyton
  - gut gepflegte Libraries
  - vernünftige IDEs

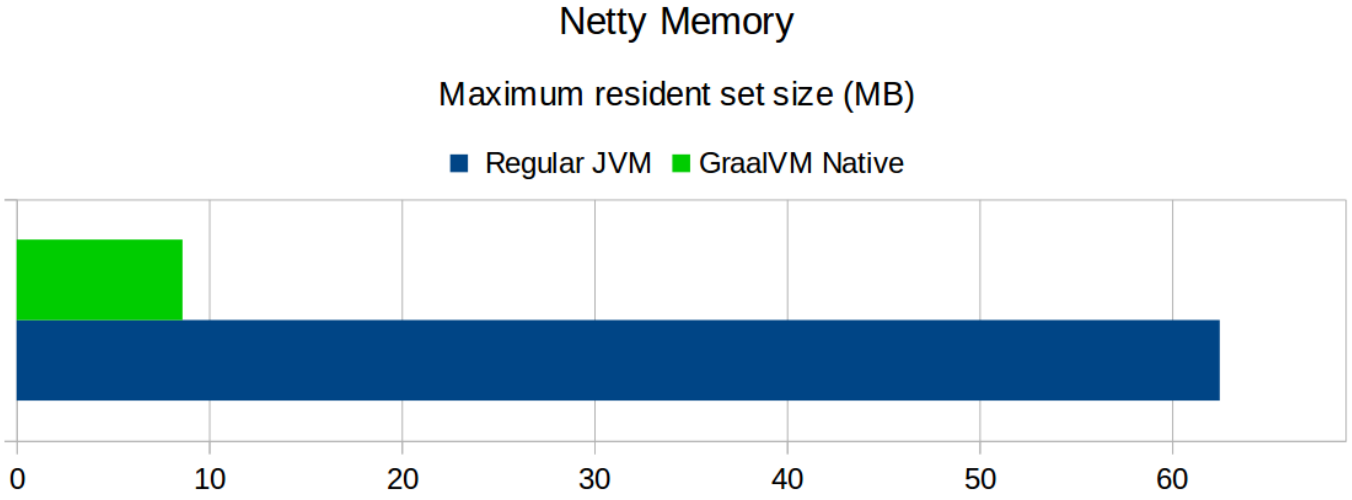
- **Beispiele**
  - javascript libraries
    - Email Adress Check
    - Levenshtein distance
  - R
    - viele Funktionen um Daten darzustellen

# Native Image

- **AOT extrem (nativeimage != openJDK AOT)**
- **nur noch MaschinenCode + schmale VM**
- **keine ClassLoading etc.**
- **Ziel**
  - vergleichbare Startzeit zu nativen Programmen
  - weniger Speicherverbrauch
- **Einsatzgebiete**
  - Tools
  - serverless Java
- **Compilierung eventuell aufwendig**
  - diverse Hilfestellungen eventuell nötig
    - welche Klassen müssen enthalten sein
  - static initializers laufen zur Buildzeit (default, einstellbar)

# Native Image



# Native Image ListDir

**Beispiel: ListDir.java**

```
Path tempPath = Paths.get(aPath);
Files.list(tempPath).filter(Files::isRegularFile).forEach(System.out::println);
```

# Native Image

- **ListDir Startup**
  - jdk: 130 ms

# Native Image

- **ListDir Startup**
  - jdk: 130 ms
  - native image: 2 ms

# Native Image

- **Micronaut**
  - Framework ähnlich zu SpringBoot und Microprofile
  - HelloWorld  Startup
    - jdk:  650 ms

# Native Image

- **Micronaut**
  - Framework ähnlich zu SpringBoot und Microprofile
  - HelloWorld  Startup
    - jdk: 650 ms
    - native image: 25 ms

# Ideal graph visualizer

- **Graal bei der Arbeit zuschauen**

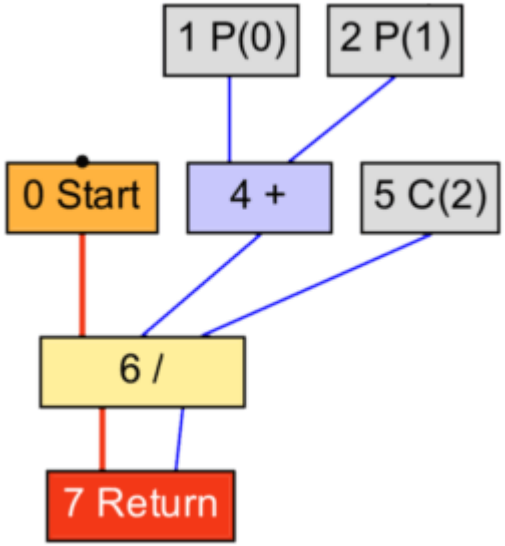
# Ideal graph visualizer Phasen

- **Byte code parsing**
- **High tier**
  - Optimierungen wie
    - partial escape analysis
    - inlining
- **Mid tier**
  - Speicher Optimierungen
  - Lock coarsening (Locks mergen etc.)
- **Low tier**
  - NullChecks durch trap signals ersetzen
  - aufräumen

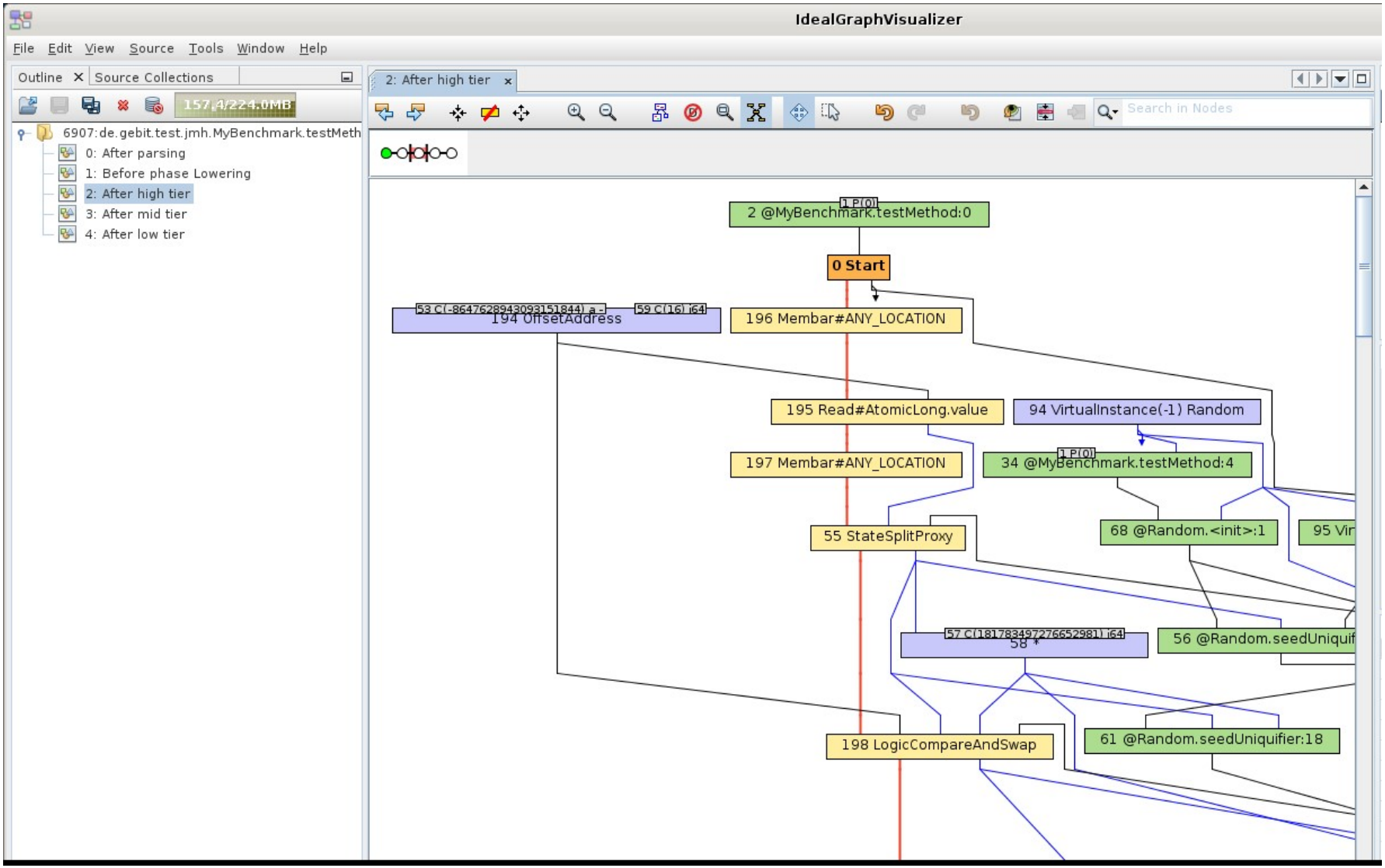
# Ideal graph visualizer Phasen

```
1  int average(int a, int b) {
2      return (a + b) / 2;
3  }
```

- **Kanten**
  - control flow: rot
  - data flow: blau
- **Knoten**
  - Daten (Parameter, Konstanten, ...)
  - Operationen
  - Barrieren
  - usw.



# Ideal graph visualizer



# Ideal graph visualizer

- **starten**
  - /usr/graalvm-ee-1.0.0-rc9/bin/idealgraphvisualizer
- **test, ob die Verbindung klappt**
  - java -XX:+UseJVMCICompiler -XX:+BootstrapJVMCI -XX:-TieredCompilation -Dgraal.Dump= -Dgraal.MethodFilter=Node.updateUsages –version

  Bootstrapping JVMCI....Connected to the IGV on 127.0.0.1:4445
- **Achtung**
  - JIT muss zur Ausführung gebracht werden
    - JMH nutzen
    - TieredCompilation ausschalten
- **eigenen Code ausführen**
  - java -XX:+UseJVMCICompiler -XX:+BootstrapJVMCI -XX:-TieredCompilation -Dgraal.Dump= -Dgraal.MethodFilter=MyBenchmark.testMethod -jar benchmarks.jar

# GraalVM Flavours

- **CE Community Edition**
  - open source
  - free for production use
- **EE Enterprise Edition**
  - free for evaluation
  - soll immer ein paar % schneller als CE sein
    - enthält z.B. im Moment auto vectorization
  - Zielgruppe
    - Anwender die pay per use in der Cloud nutzen
    - besonders performance-kritische Anwendungen

# Fazit

- **JVM Entwicklung geht weiter !**

# Fazit

- **JVM Entwicklung geht weiter !**
- **Grundlage für schnellere Weiterentwicklung**
  - JDK Projekt metropolis
    - C++ Teile der VM werden durch Java ersetzt

# Fazit

- **JVM Entwicklung geht weiter !**
- **Grundlage für schnellere Weiterentwicklung**
  - JDK Projekt metropolis
    - C++ Teile der VM werden durch Java ersetzt
  - Wird vielleicht der Default JIT
- **flexible Optimierungsmöglichkeiten**
  - große Anzahl an Libraries verschiedener Sprachen nutzbar
    - native Libs ohne JNI
  - Performance erhöhen
  - Startup verkürzen
  - Speicherverbrauch reduzieren

# Empfehlung

- **Testet !**
  - graal einschalten ist sehr einfach
  - graalvm etwas schwieriger
    - aktuell nur linux, macOs

# Vielen Dank für Ihr Interesse !

stephan.frind@gebit.de