ORACLE

GraalVM™

GraalVM – Neuerungen und Readiness Checklist

Concepts & Features

Wolfgang Weigend

Master Principal Solution Engineer | global Java Team

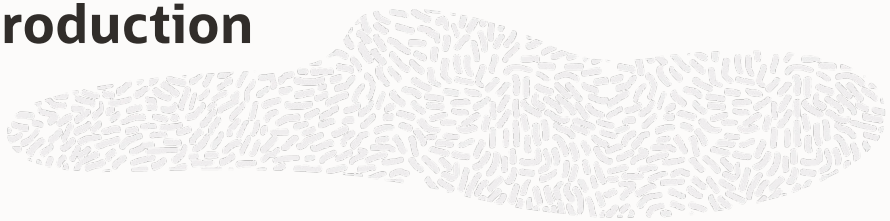
Java Technology & GraalVM and Architecture

Agenda

- GraalVM Release Update “GraalVM for JDK 23” & Technology
- GraalVM Just-in-Time Compiler
- GraalVM Native Image
- GraalVM Performance
- GraalVM & Reflection
- GraalVM Release Calendar
- GraalVM Contribution
- Summary

GraalVM Readiness Checklist — Use for Production

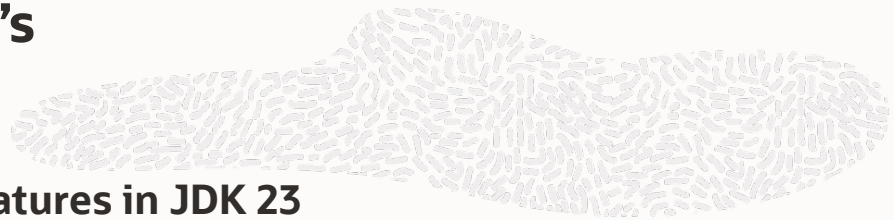
1	Performance
2	Startup Time
3	Polyglot
4	Compatibility
5	Security
6	Monitoring & Tooling
7	Development & Debug
8	Reflection
9	Frameworks
10	Fast LLM inference engine in Java
11	License & Support



Oracle GraalVM — The Release

- Oracle GraalVM available for JDK 17 and JDK 21 and released under the GraalVM Free Terms and Conditions (GFTC) license
  - https://www.oracle.com/downloads/licenses/graal-free-license.html
- This means that you can use all the greatest GraalVM features, both for development and in production, for free
  - ❖ “GraalVM for JDK 17”
    - ➢ GraalVM Free Terms and Conditions (GFTC) licensed updates to JDK 17 end in September 2025
    - ➢ GraalVM for JDK 17 updates will continue with OTN license
    - ➢ Details: https://blogs.oracle.com/java/post/jdk-17- approaches-endofpermissive-license
  - ❖ “GraalVM for JDK 21”
  - ❖ “GraalVM for JDK ..” (LTS Release)

GraalVM for JDK 23 - Delivered JDK 23 JEP's



Features in JDK 23

JEP	Description
467	Markdown Documentation Comments
471	Deprecate the Memory-Access Methods in sun.misc.Unsafe for Removal
474	ZGC: Generational Mode by Default

GraalJS EcmaScript 2025 compliant

- Replacement for Nashorn last shipped in JDK 14

Preview Features in JDK 23

JEP	Description
455	Primitive Types in Patterns, instanceof, and switch (Preview)
466	Class-File API (Second Preview)
469	Vector API (Eighth Incubator)
473	Stream Gatherers (Second Preview)
476	Module Import Declarations (Preview)
477	Implicitly Declared Classes and Instance Main Methods (Third Preview)
480	Structured Concurrency (Third Preview)
481	Scoped Values (Third Preview)
482	Flexible Constructor Bodies (Second Preview)

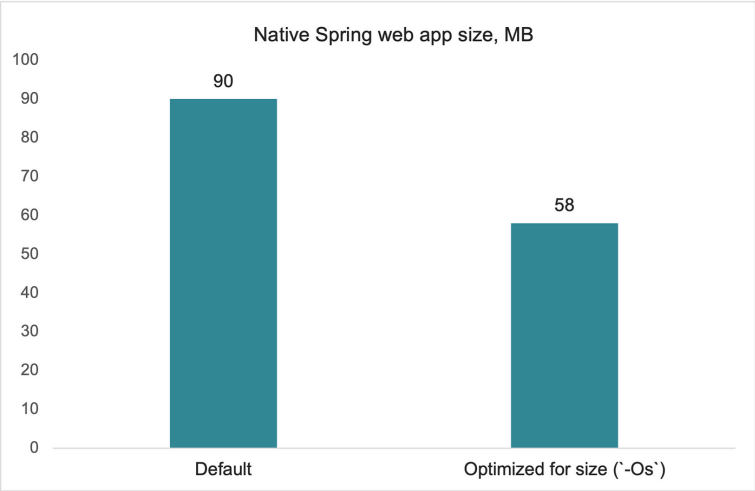
Release Notes: https://www.graalvm.org/release-notes/JDK_23/

GraalVM for JDK 23 — Overview

- Native Image
  - New optimization level, -Os (size), to minimize executable size
  - New mark and compact garbage collection mode for the Serial GC old generation
    - Reduces memory usage compared to the copying GC
    - Find more details at github.com/oracle/graal/pull/8870
    - This is an experimental feature
- Native Image can generate native executables with an embedded software bill of materials (SBOM)
- New `--emit` option to generate various additional outputs, like build reports
  - **`native-image --emit build-report=report.html HelloWorld`**
- Graal Languages
  - GraalPy and GraalWasm now GA

Oracle GraalVM — Smaller executables

- We have added a new optimization level in Native Image, -Os. We used to get requests from users who want to have smaller executables, and while it was possible before as well with other tools, it's certainly much easier now with -Os. This option will configure the build to optimize for the smallest code size, and therefore the smallest executable size. Under the hood it enables all -O2 optimizations except those that can increase the image size
- Here's an example of a basic Spring Boot web application, built with GraalVM for JDK 23. With the optimized for size mode, the native application is 35% smaller:

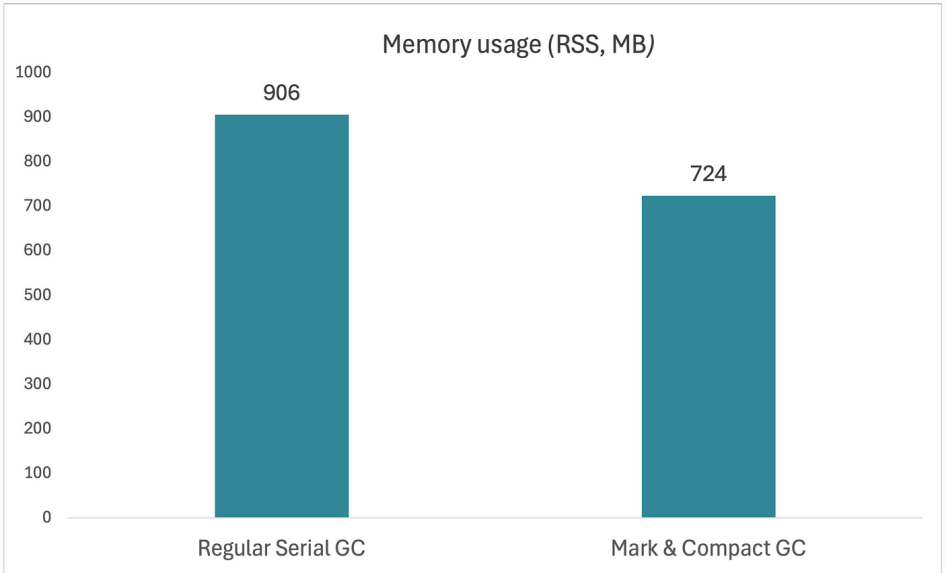


New optimization mode in Native Image: optimize for size

- Beware though that this mode is great for image size, but it might not be the best for peak throughput — benchmark and choose the optimization level based on your goals

Oracle GraalVM — New Compacting Garbage Collector

- In this release, we added a mark & compact GC for the old generation of the Serial GC (available in both Oracle GraalVM and GraalVM Community). The primary intention is to reduce the max memory usage compared to the copying GC, which can use 2x the current heap size when all objects survive. We recommend checking out this GC if you want to use small heap sizes, or are deploying to memory-constrained environments and optimize for smaller containers



Memory usage reduction with the new Compacting Garbage Collector

- As shown above on the example of the Scrabble benchmark, in this case the Compacting Garbage Collector can lead to around 20% lower memory usage. While memory usage might go down, there is also no performance penalty — peak throughput of the application stays the same

Oracle GraalVM — New SBOM Integration

- Native Image can generate native executables with an embedded software bill of materials (SBOM), that you can use for supply chain security analysis. Previously the SBOM file was always embedded in the native image. Now we also added a feature to include SBOM as a Java resource on the classpath at META-INF/native-image/sbom.json. To do this, pass the --enable-sbom=classpath to the build.
- What's great about having the SBOM on classpath is that you can then explore it with standard tools, such as Spring Actuator. Built-in support is underway, but you can already give it a try in a few simple steps:

1. Generate your Spring project and include Native Image support, Spring Web, and Spring Actuator, or clone our example
2. in **`native-maven-plugin`**, add configuration to put SBOM on the classpath via **`--enable-sbom=classpath`**
3. Build: **`mvn -Pnative native:compile`**
4. Run: **`./target/demo`**
5. Curl the Actuator endpoint to retrieve the SBOM: **`curl http://localhost:8080/actuator/sbom/native-image`**

All done - You can now explore your project's dependencies and their metadata

You can also use the SBOM to identify known CVEs in your application by running it through a vulnerability scanner of your choice

GraalVM Native Image - Build Reports provide insight into generated executable contents and metrics

Build Report - jwebserver

2025/02/18 12:05:09

Summary | Code Area | Image Heap | Resources | Software Bill of Materials (SBOM)

Download as JSON

Environment

Java Version	**23.0.1+11**
Vendor Version	**Oracle GraalVM 23.0.1+11.1**
Graal Compiler	Optimization level: 2, Target machine: armv8-a, PGO: ML-inferred
C Compiler	cc (apple, arm64, 16.0.0)
Garbage Collector	Serial GC

Analysis Results

Category	Types	in %	Fields	in %	Methods	in %
Reachable	4383	74.859%	5406	45.088%	23502	52.013%
Reflection	1486	25.380%	14	0.117%	347	0.768%
JNI	58	0.991%	56	0.467%	53	0.117%
Loaded	5855	100.000%	11990	100.000%	45185	100.000%

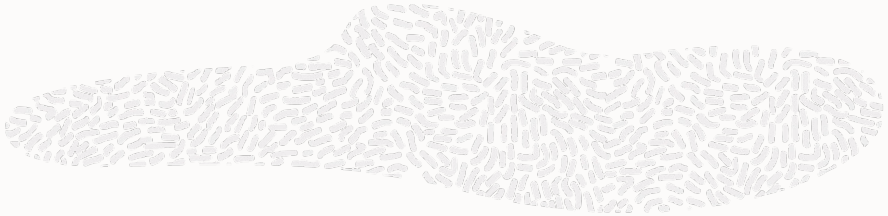
Image Details

19.19 MB in total

file:///Users/WWEIGEND/projects/hello/report_myweb.html

Graal Languages

Built on advanced GraalVM compiler technology



- Embed and run JavaScript, Python, and WebAssembly (WASM), and Java applications in Java
- Enables Java application extension/scripting with sandboxed execution
- 100% pure Java implementations (with some native libraries required by GraalPy)
- Enables:
  - Use of Python ML/Agent libraries
  - Server side rendering of ReactJS front ends
  - Leveraging Go/Rust/C libraries compiled to WASM

Oracle GraalVM — Extend your Java applications with Python and Wasm libraries

- GraalVM always offered smooth, fast, and easy interoperability between multiple languages
- Now extending your Java application with cool data science libraries from Python, or low-level packages from Wasm, got even easier
- GraalPy and GraalWasm are now considered stable and suitable for production workloads
- For GraalPy, our priority is on pure Python code and Jython use cases
- And languages are now just normal Maven dependencies

Oracle GraalVM — Foreign Function & Memory API in Native Image

- In the GraalVM for JDK 23 release, we improved the Foreign Function & Memory API (JEP 454) implementation by adding experimental support for upcalls from foreign functions (which is part of “Project Panama”)
- This is done in addition to foreign functions downcalls and foreign memory functionality, that were available in Native Image before
- Currently, foreign calls are supported on the x64 architecture. To give it a try, build your image with `-H:+ForeignAPISupport`

Oracle GraalVM — Native Memory Tracking

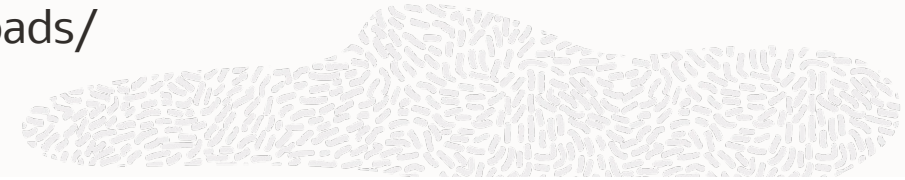
- The Native Memory Tracking (NMT) is a JVM feature that tracks internal memory usage of HotSpot
- As a first step to supporting NMT, PR #7883 added support for tracking mallocs/calloc/realloc in Linux (contribution by Red Hat).
- Virtual memory tracking support and support for NMT JFR events will be added in subsequent PRs
- To build a native executable with NMT, use option `--enable-monitoring=nmt`
- https://www.graalvm.org/jdk23/reference-manual/native-image/debugging-and-diagnostics/NMT/

Oracle GraalVM — Native Build Tools and GraalVM Reachability Repository

- GraalVM Reachability Metadata Repository enables sharing and reusing metadata of libraries in the Java ecosystem, in particular for Native Image builds
- The latest release, 0.3.8, includes configuration for Flyway 10.10, itext, Hibernate Reactive, Hibernate Core 6.5.0, and several other updates
- If you are using Native Build Tools, or one of the frameworks that leverage them under the hood, access to the Reachability metadata is enabled by default
- We've also made several changes and improvements in the recent Native Build Tools release — make sure you upgrade to 0.10.3
- **Libraries and Frameworks Tested with Native Image**
  - https://www.graalvm.org/native-image/libraries-and-frameworks/

GraalVM available on oracle.com/java downloads

https://www.oracle.com/java/technologies/downloads/



JDK 23 | JDK 21 | GraalVM for JDK 23 | GraalVM for JDK 21

GraalVM for JDK 23.0.2 downloads

GraalVM for JDK 23 binaries are free to use in production and free to redistribute, at no cost, under the GraalVM Free Terms and Conditions (GFTC).

GraalVM for JDK 23 will receive updates under these terms, until March 2025, when it will be superseded by GraalVM for JDK 24.

Oracle GraalVM uses the Graal just-in-time compiler and includes the Native Image feature as optional early adopter technology.

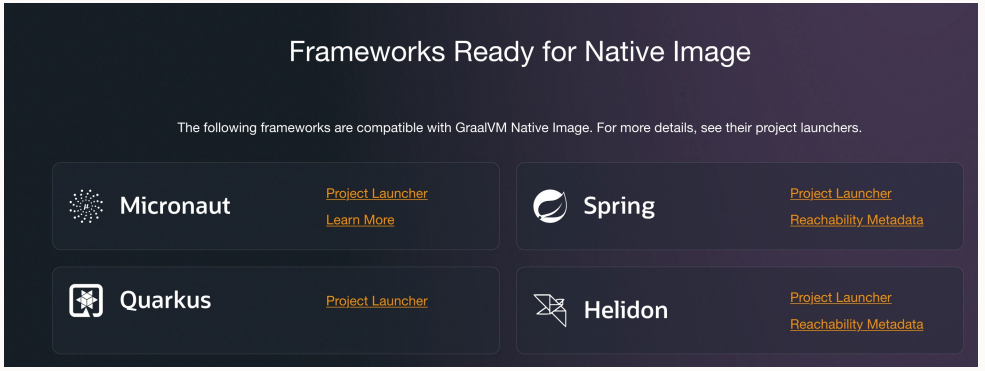
Native Image is extensively tested and supported for use in production, but is not a conformant implementation of the Java Platform. GraalVM for JDK 23 without the Native Image feature included is available for customers at My Oracle Support.

Linux | macOS | Windows

Product/file description	File size	Download
ARM64 Compressed Archive	334.15 MB	https://download.oracle.com/graalvm/23/latest/graalvm-jdk-23_linux-aarch64_bin.tar.gz (sha256)
x64 Compressed Archive	361.88 MB	https://download.oracle.com/graalvm/23/latest/graalvm-jdk-23_linux-x64_bin.tar.gz (sha256)

GraalVM Native Image with Spring Boot the Java Microservices Frameworks

- Enables compiling Java programs into standalone native executables
- Performs static analysis to identify all code reachable from the entry point
- Instant startup, low memory footprint, perfect for cloud deployments
- Integrations with Java microservices frameworks



ORACLE

Including the Graal JIT in Oracle JDK 23

Including the Graal JIT in Oracle JDK 23

It's flagged as an experimental feature

- Starting with Oracle JDK 23, the Oracle GraalVM JIT compiler (Graal JIT) is now included among the JITs available as part of the Oracle JDK. This integration offers innovations previously made available via Oracle GraalVM, such as JIT code optimization techniques, to all Oracle JDK users. This provides developers and sysadmins with more options to help fine tune and improve peak performance of their applications
- The Graal JIT is the same technology that has been part of the commercial Oracle GraalVM Enterprise Edition and Oracle GraalVM for JDK for several years and is fully supported. Because the integration is new to the Oracle JDK, it is flagged as an experimental feature, but one that is still fully commercially supported
- The Graal JIT is enabled by passing the command line options to the Java executable:

```
-XX:+UnlockExperimentalVMOptions -XX:+UseGraalJIT
```

- If you do not pass these flags at JVM startup the Oracle JDK default JIT (C2) will run as usual
- Performance benefits depend on the workload and which features are being used

ORACLE

GraalVM Technology

GraalVM

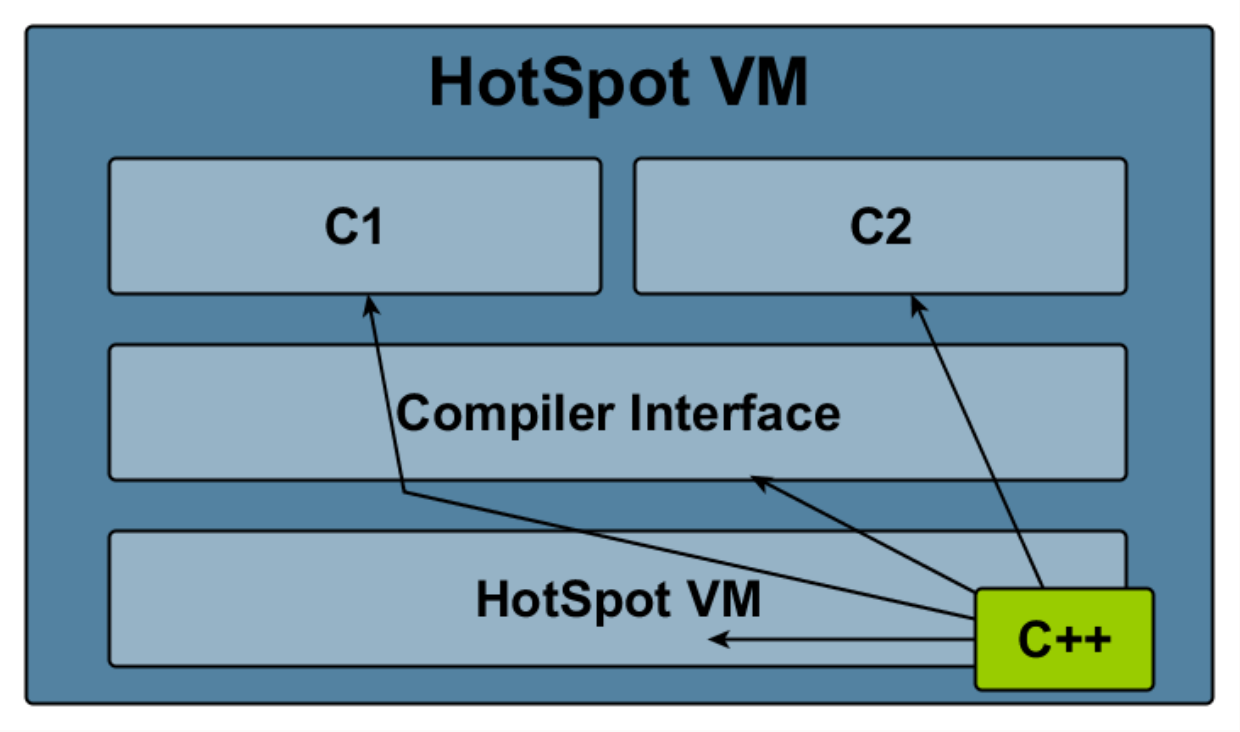
High-performance runtime that provides significant improvements in application performance and efficiency

High-performance optimizing Just-in-Time (JIT) compiler

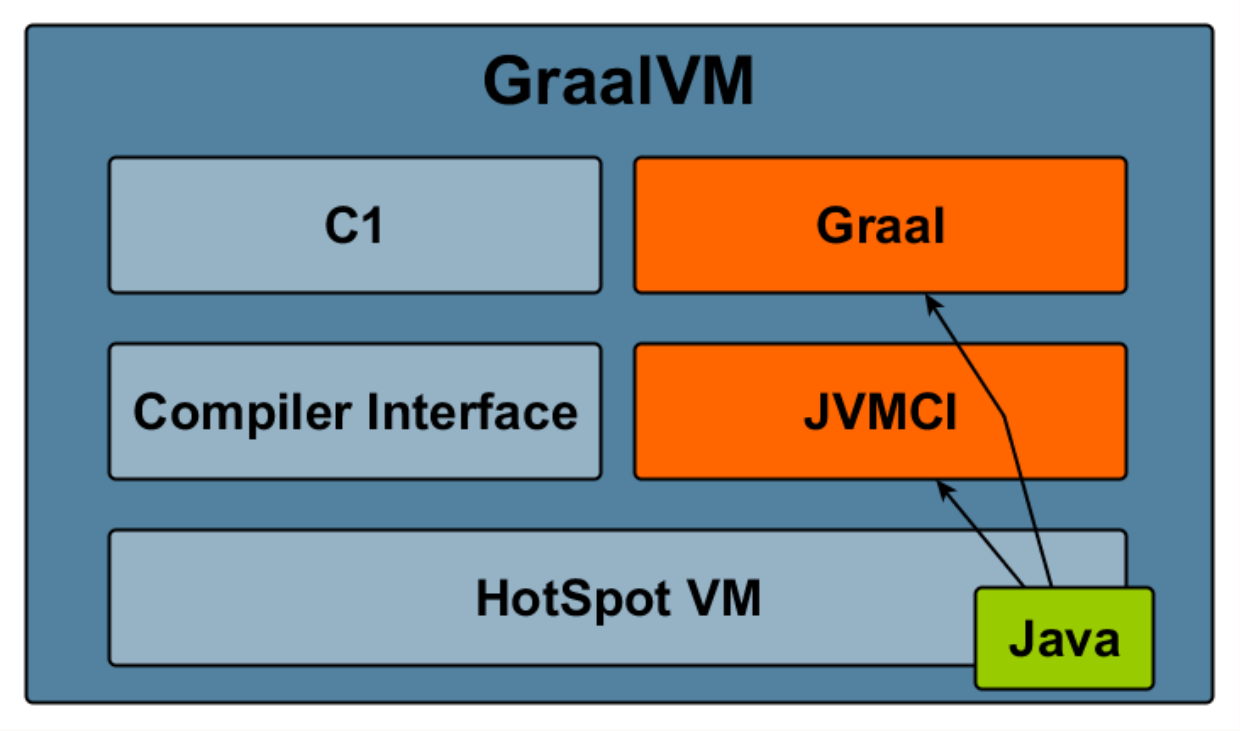
Multi-language support for the JVM

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

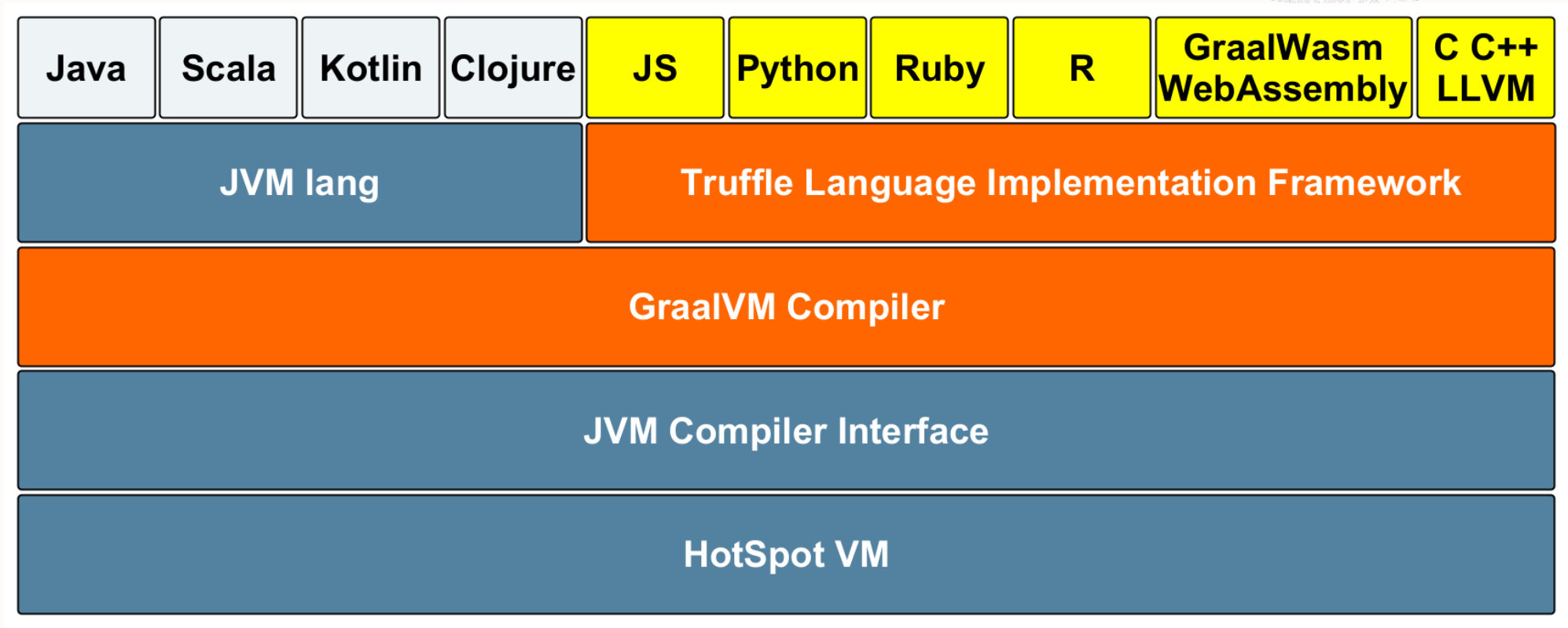
JIT Compiler written in C++



JIT Compiler written in Java



GraalVM — Polyglot

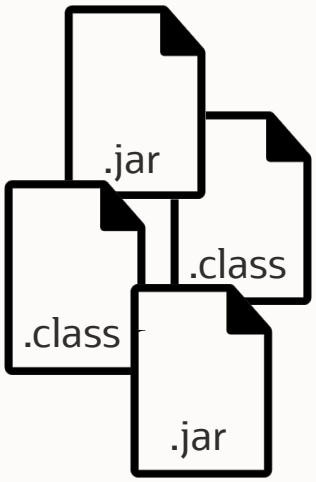


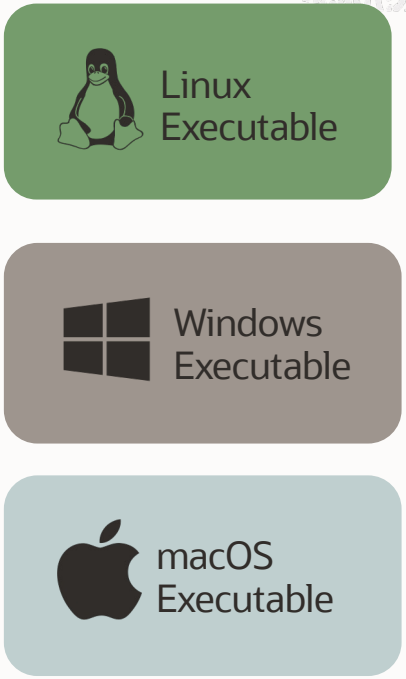
ORACLE

GraalVM Native Image

GraalVM Native Image—Ahead-of-time compiler & runtime

Microservices and Containers

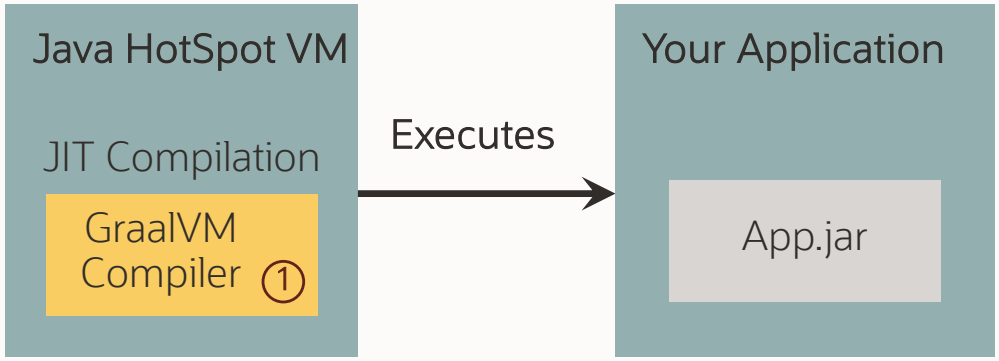




Up to 5x less memory
100x faster startup

One Compiler, Many Configurations

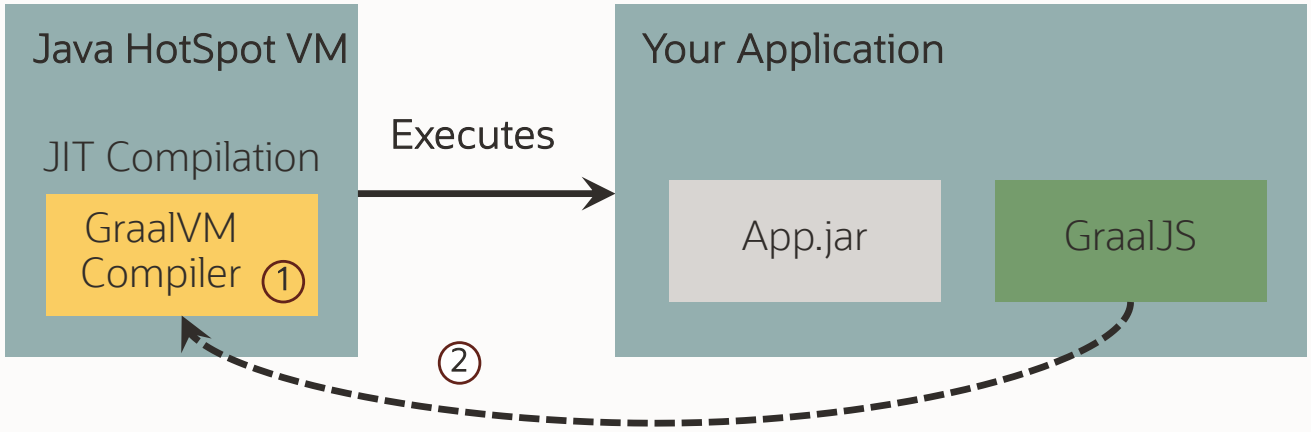
GraalVM



① Compiler configured for just-in-time compilation inside the Java HotSpot VM

One Compiler, Many Configurations

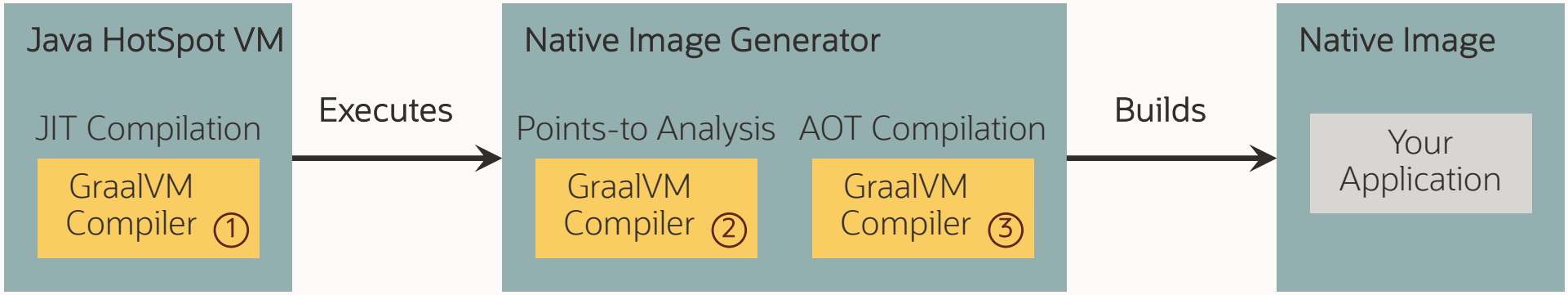
GraalVM



1. Compiler configured for just-in-time compilation inside the Java HotSpot VM
2. Compiler also used for just-in-time compilation of JavaScript code

One Compiler, Many Configurations

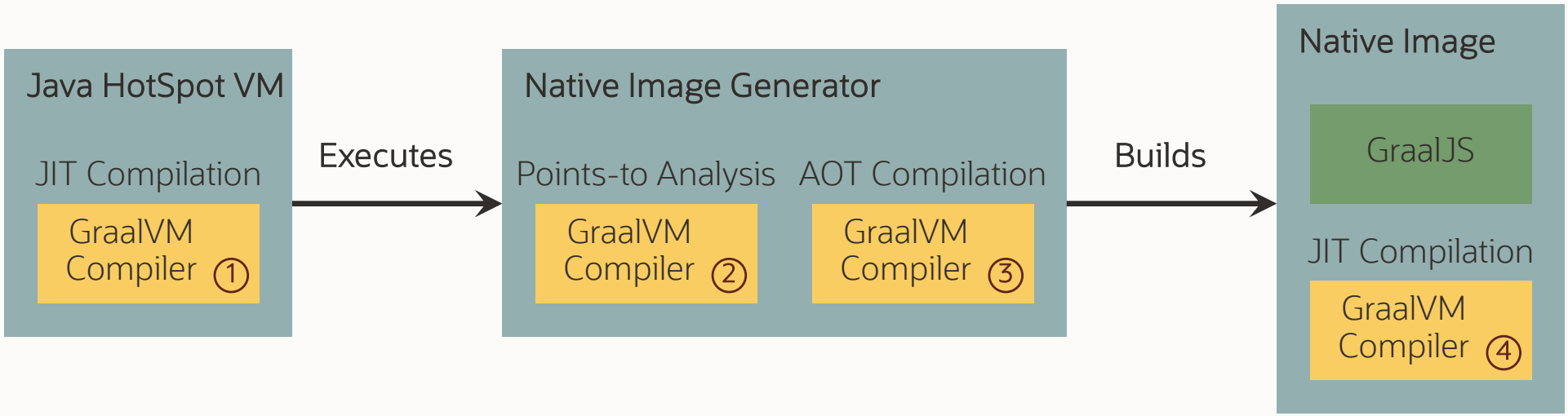
GraalVM



① Compiler configured for just-in-time compilation inside the Java HotSpot VM
② Compiler configured for static points-to analysis
③ Compiler configured for ahead-of-time compilation

One Compiler, Many Configurations

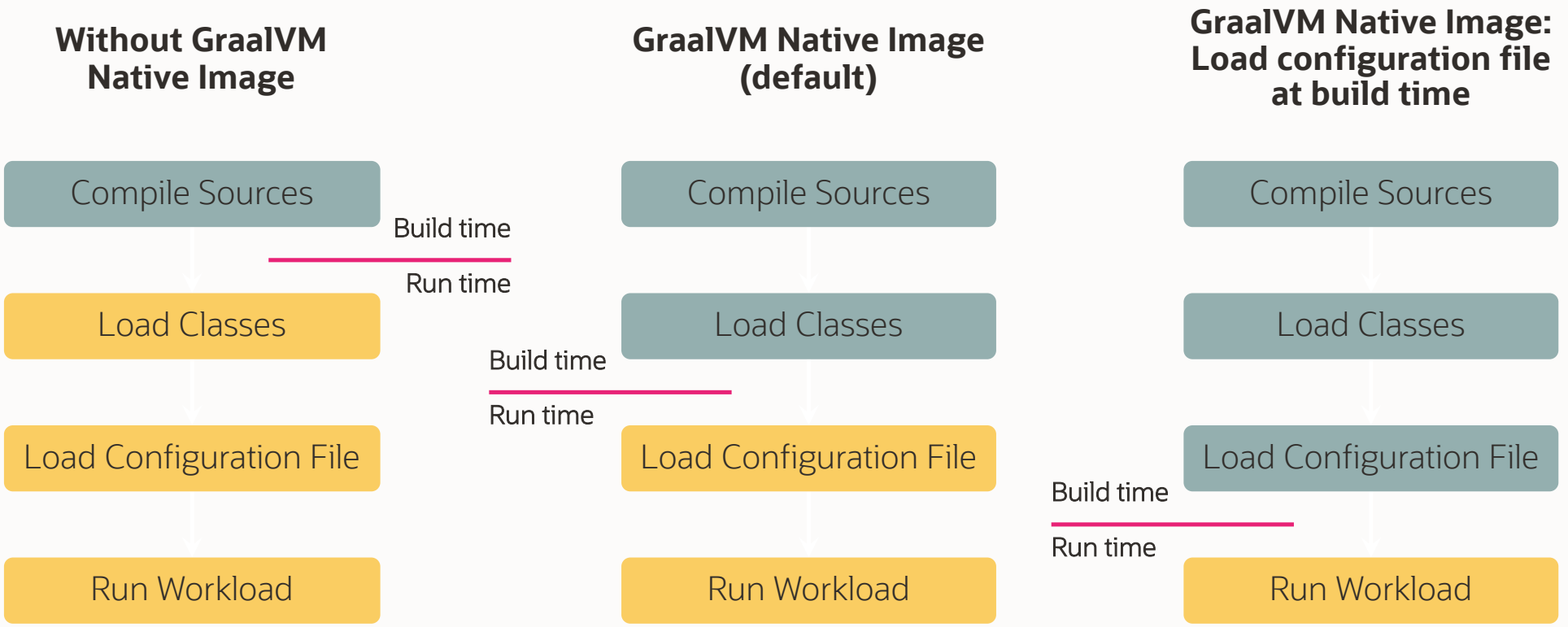
GraalVM



① Compiler configured for just-in-time compilation inside the Java HotSpot VM
② Compiler configured for static points-to analysis
③ Compiler configured for ahead-of-time compilation
④ Compiler configured for just-in-time compilation inside a Native Image

Benefits of the Image Heap

GraalVM

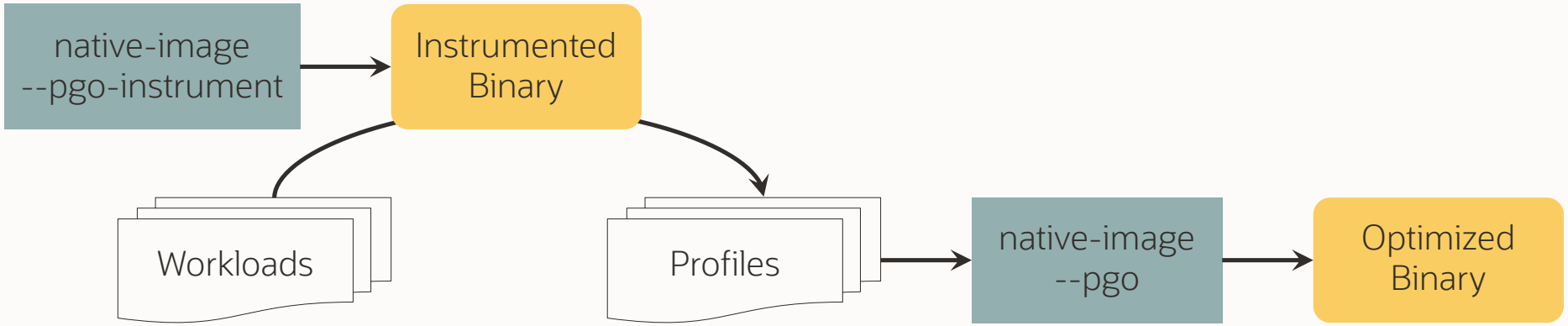


Profile-Guided Optimizations (PGO)

GraalVM

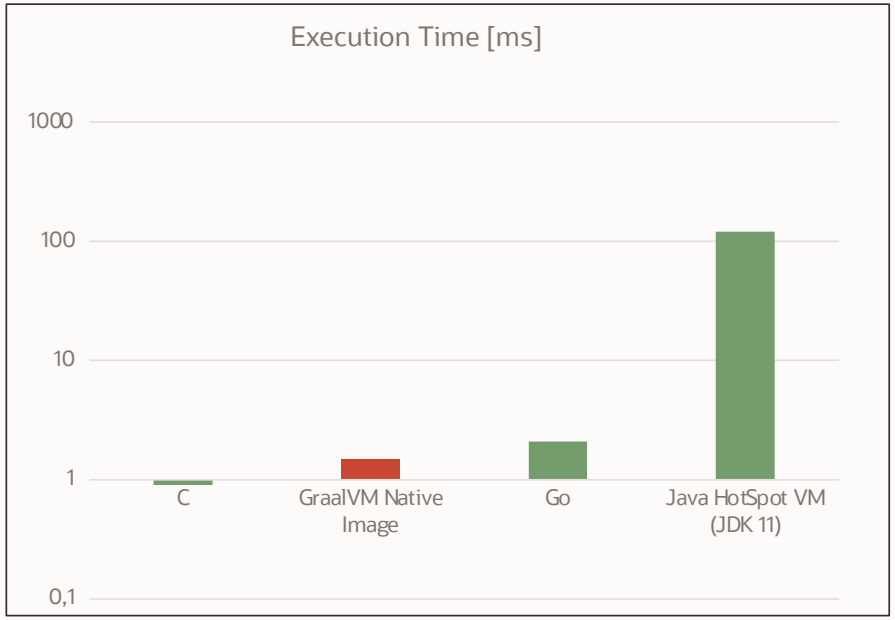
Out of Band Optimization

- AOT compiled code cannot optimize itself at run time (no “hot spot” compilation)
- PGO requires representative workloads
- Optimized code runs immediately at startup, no “warmup” curve

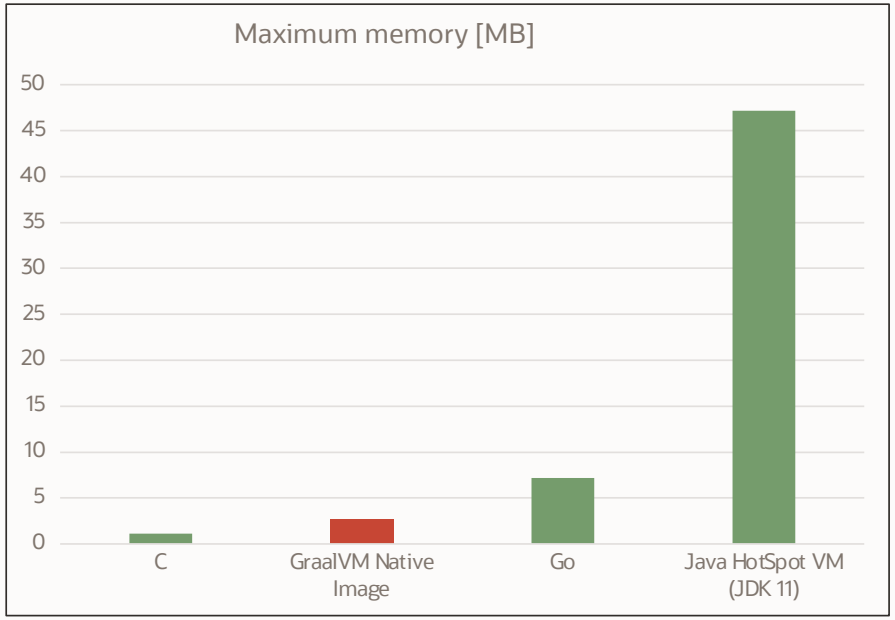


GraalVM Native Image

Lower cloud costs for containerized workloads and microservices

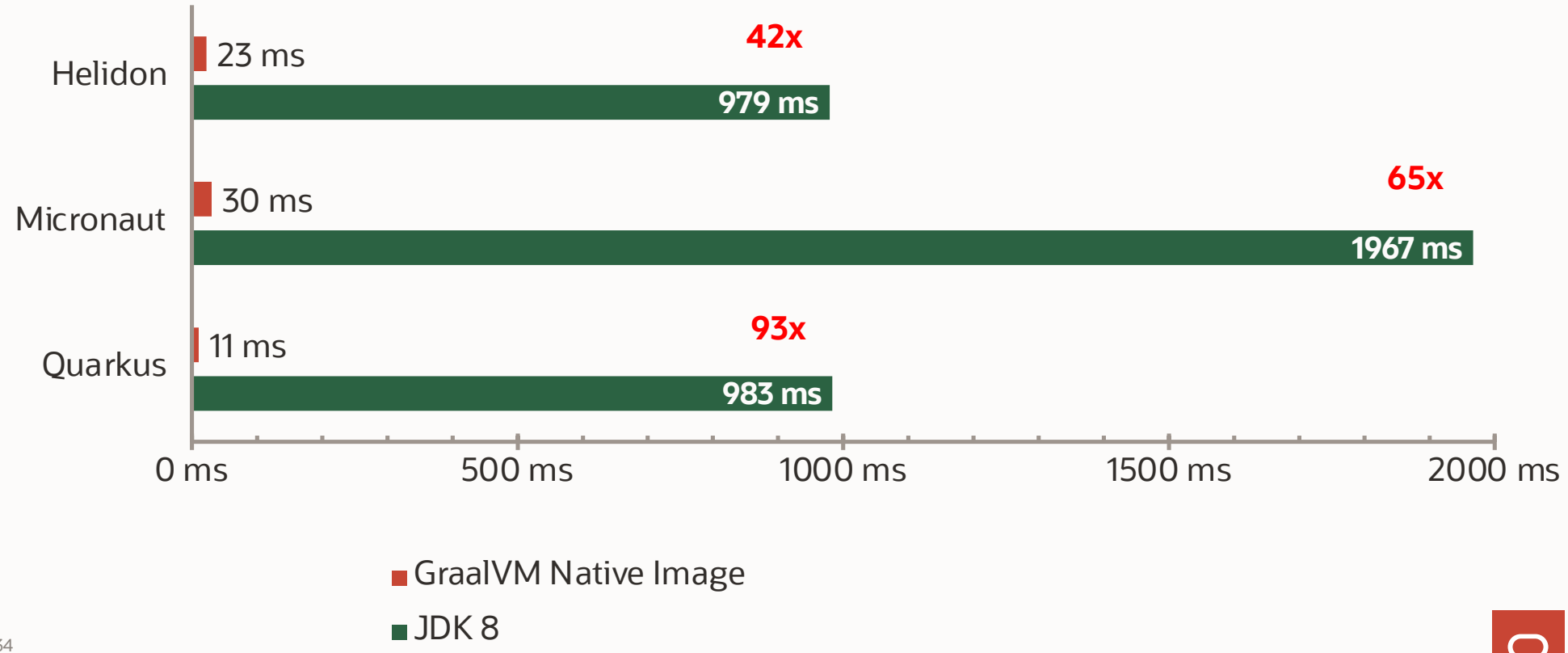


Competitive startup time

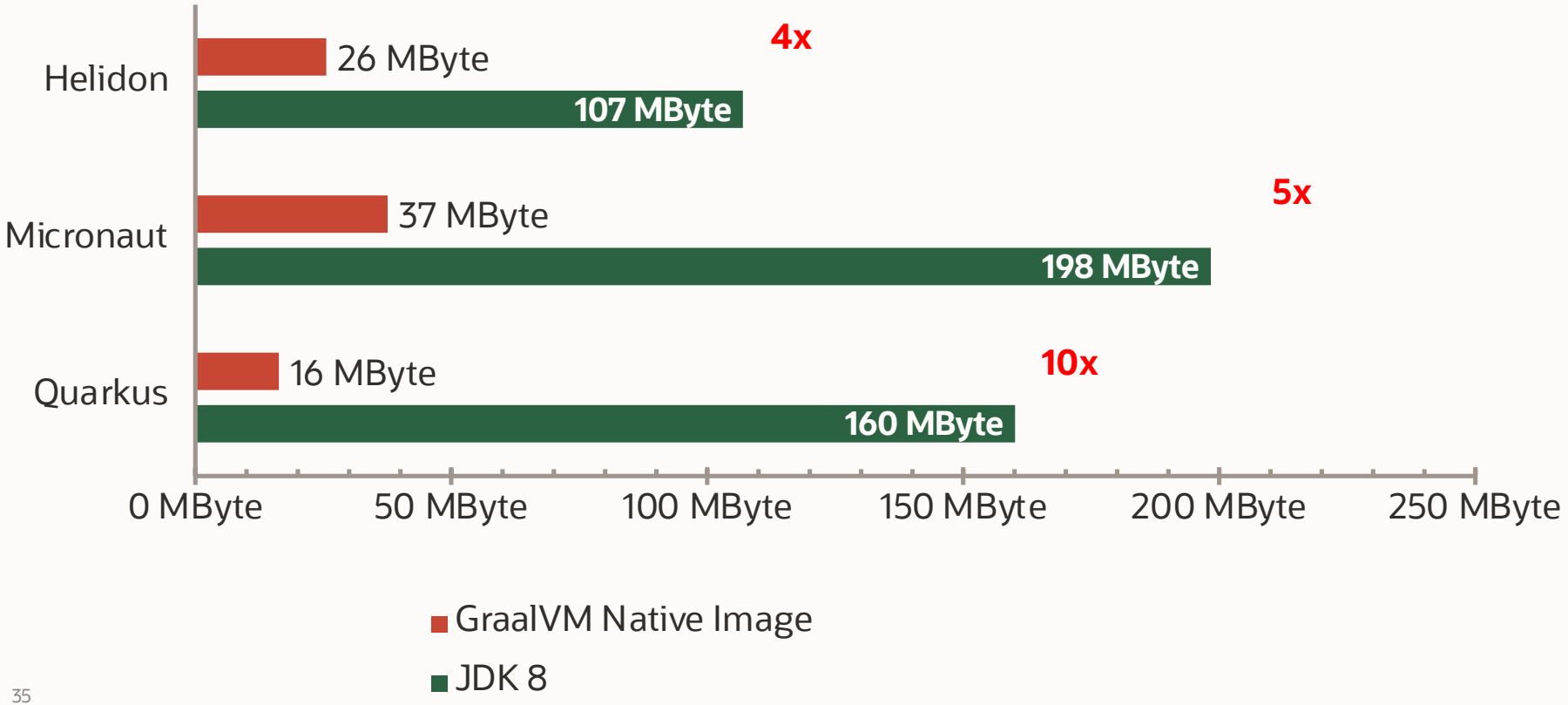


Significantly reduced memory requirements

Cloud Services — Startup Time



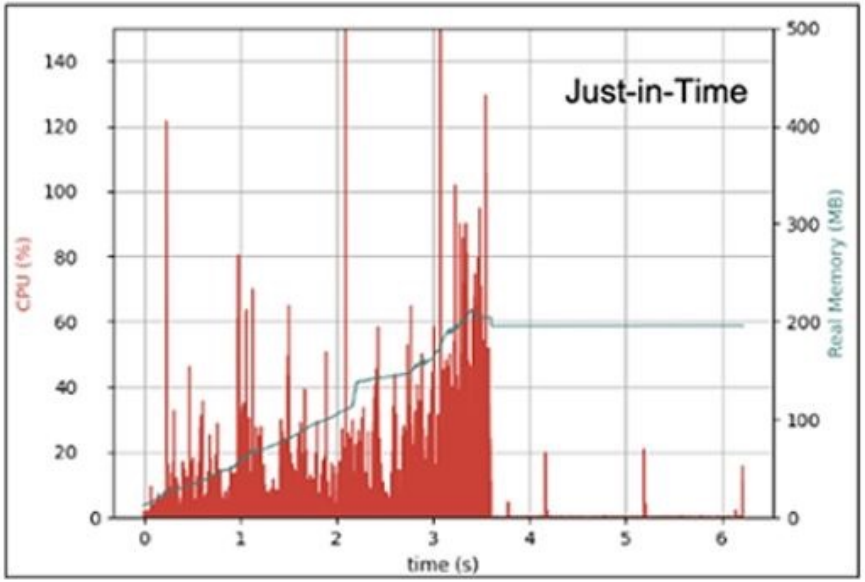
Cloud Services — Memory Footprint

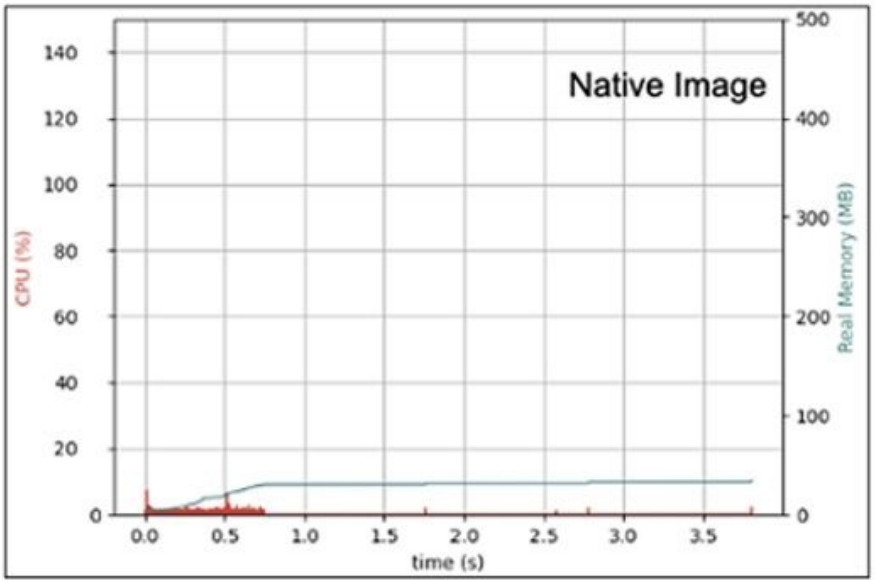


GraalVM — Memory and CPU Usage in JIT and Native Image Modes

Memory efficiency

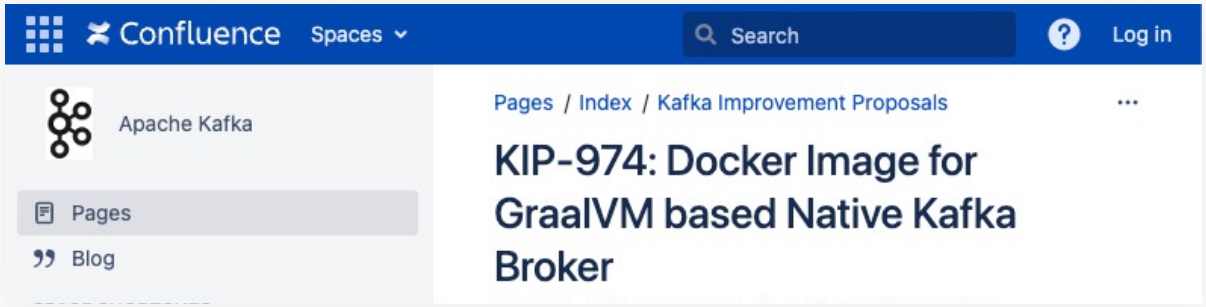
- A native executable requires neither the JVM and its JIT compilation infrastructure nor memory for compiled code, profile data, and bytecode caches.
- All it needs is memory for the executable and the application data.





Native Image

Kafka Broker with GraalVM for JDK 17



Avg Kafka Broker Startup Time GraalVM versus JVM

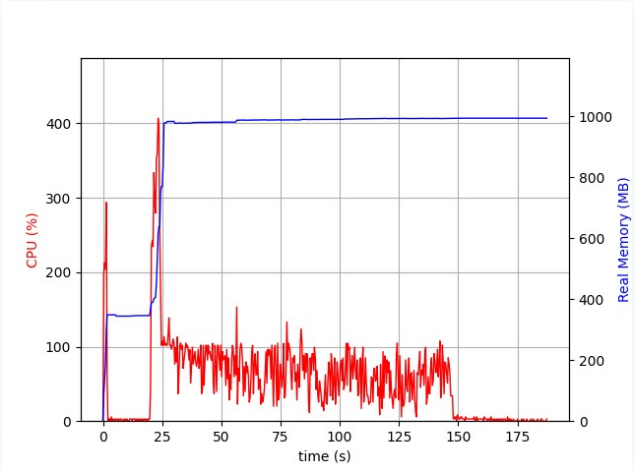
- Avg Kafka Server Startup Time Using JVM(default configs): ~1150 ms - 1200ms
- Max Memory: ~1GB
- Following is the table of startup times of GraalVM based Native Kafka Broker depending on GC and PGO:

GC	PGO	Kafka Native Binary size	Avg Kafka Broker Startup Time	Max Memory
serial	disabled	96MB	~130ms	~250MB
serial	enabled	67MB	~110ms	~230MB
G1	disabled	128MB	~140ms	~520MB
G1	enabled	82MB	~135ms	~540MB

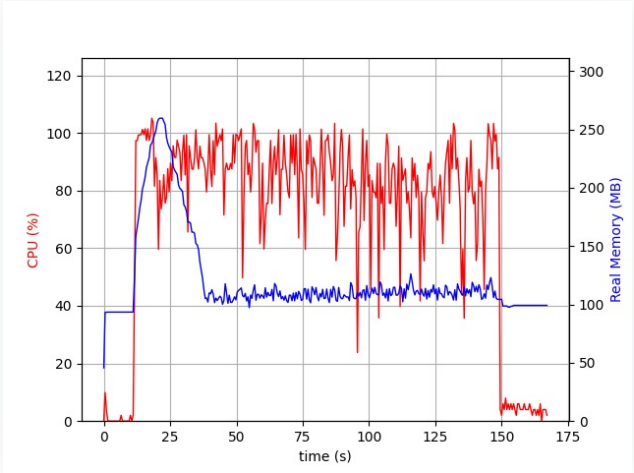
https://cwiki.apache.org/confluence/display/KAFKA/KIP-974%3A+Docker+Image+for+GraalVM+based+Native+Kafka+Broker

Kafka Broker CPU and Memory

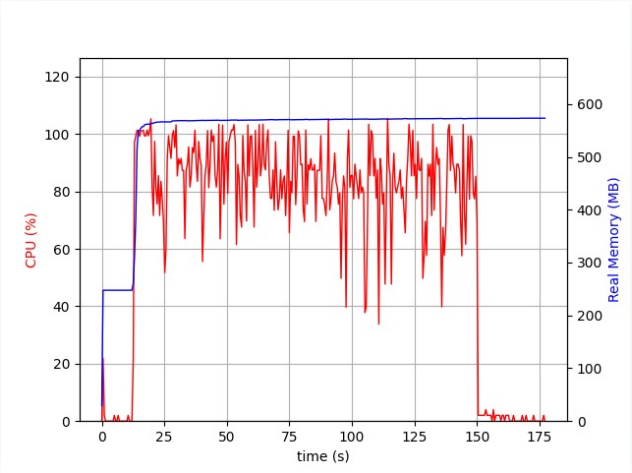
JIT vs. Native Image AOT



Kafka Broker using JIT (G1 GC)



Native Binary Serial GC



Native Binary G1 GC

Legend

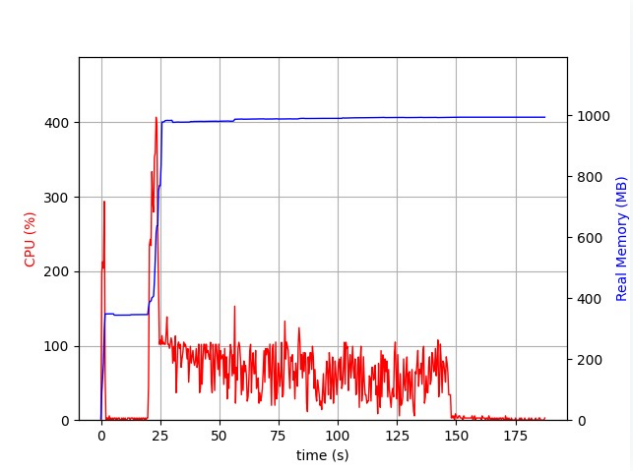
Real Memory

CPU

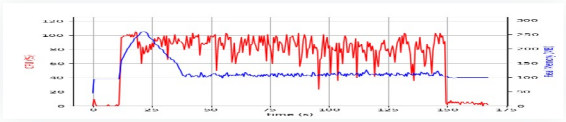
Kafka Broker CPU and Memory

JIT vs. Native Image AOT

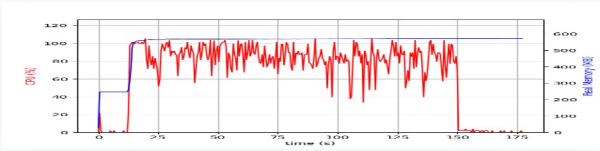
CPU aligned



Kafka Broker using JIT (G1 GC)



Native Binary Serial GC



Native Binary G1 GC

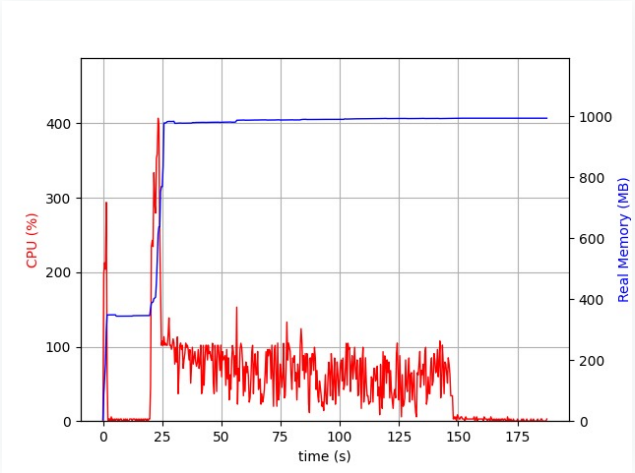
Legend

CPU

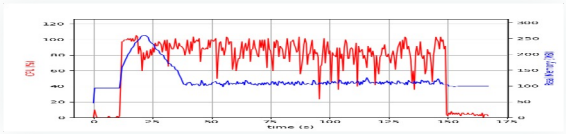
Kafka Broker CPU and Memory

JIT vs. Native Image AOT

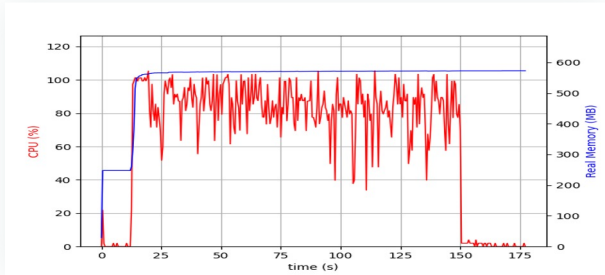
Memory aligned



Kafka Broker using JIT (G1 GC)



Native Binary Serial GC



Native Binary G1 GC

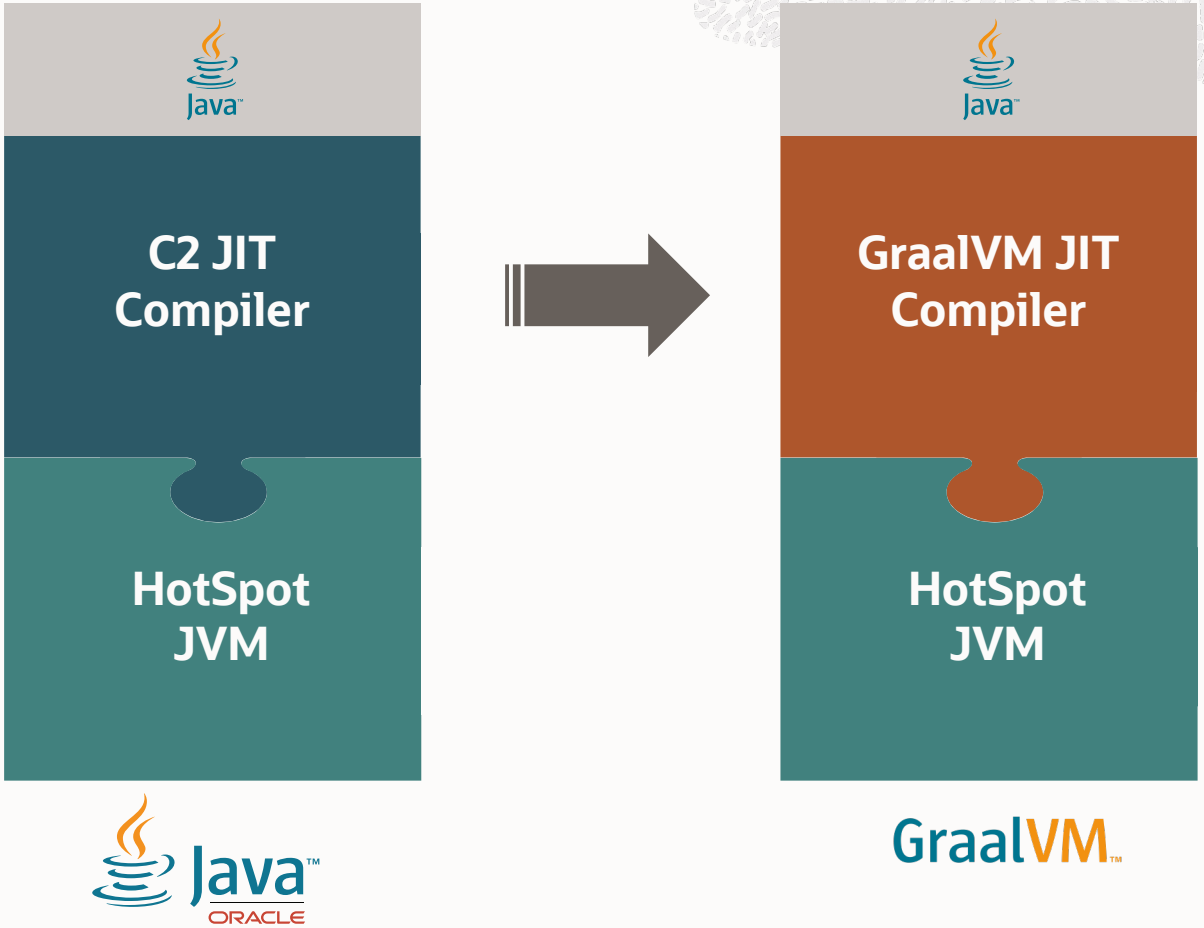
Legend

Real Memory ——

ORACLE

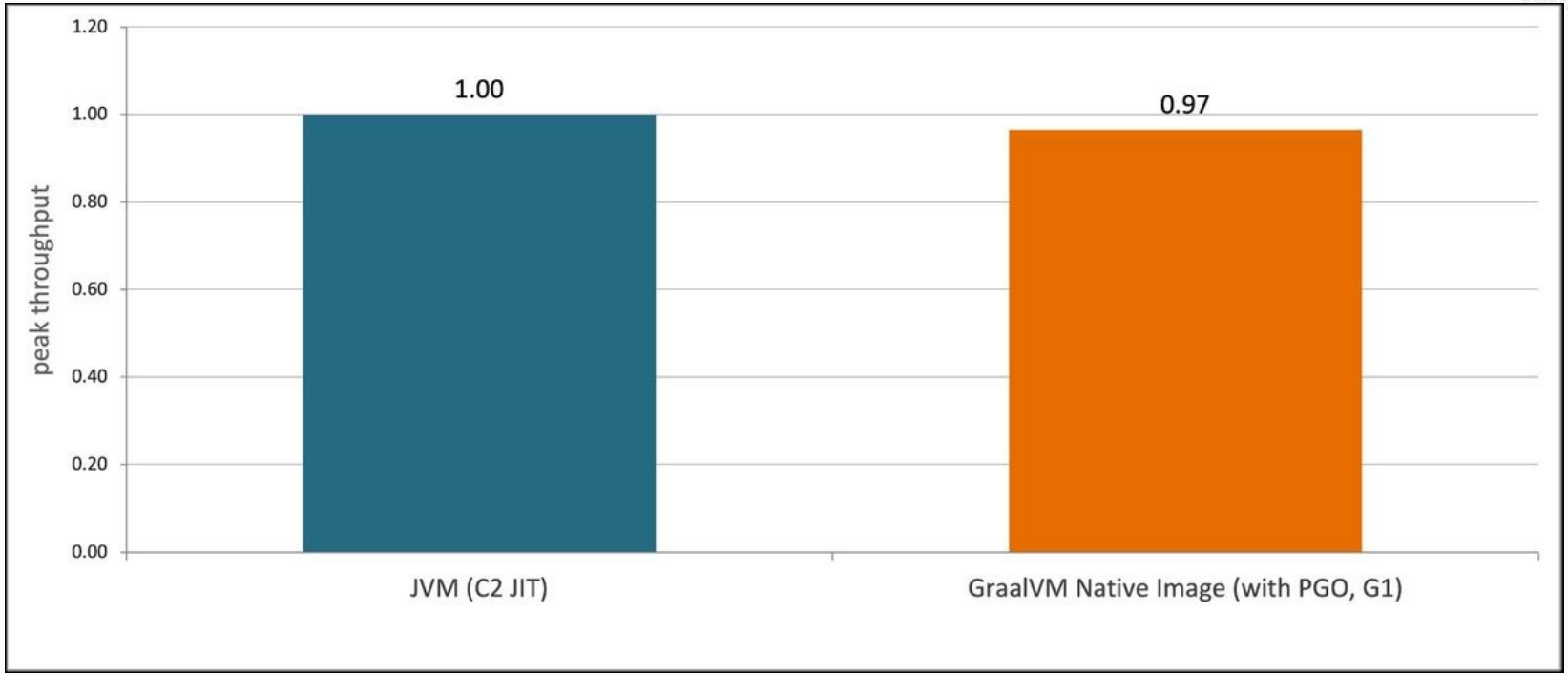
GraalVM Performance

GraalVM: Oracle JDK + GraalVM JIT Compiler



GraalVM PGO and the G1 GC — Native executables achieve peak performance on par with the JVM

Geomean of Renaissance and DaCapo Benchmarks

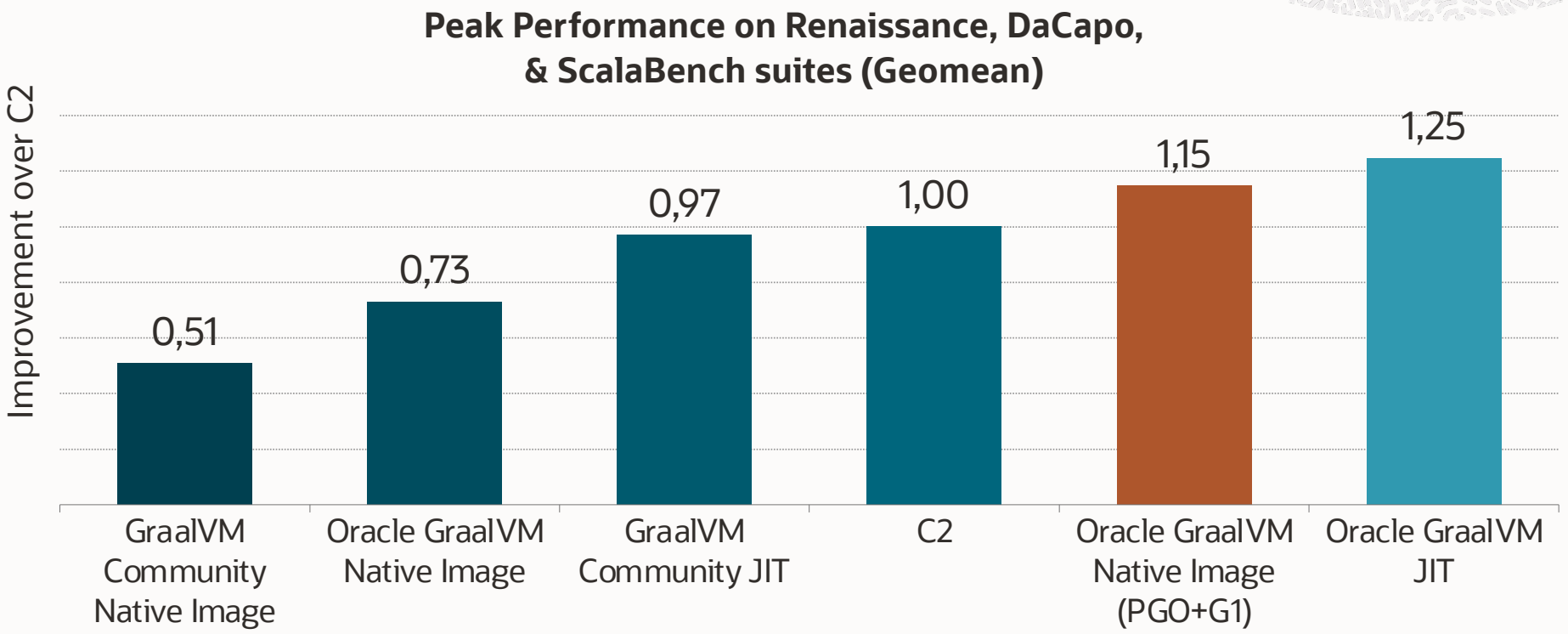


With these options, you can maximize every performance dimension of your application with Native Image:

Startup Time, Memory efficiency, and Peak Throughput

Oracle GraalVM 23 — JDK 17 Performance

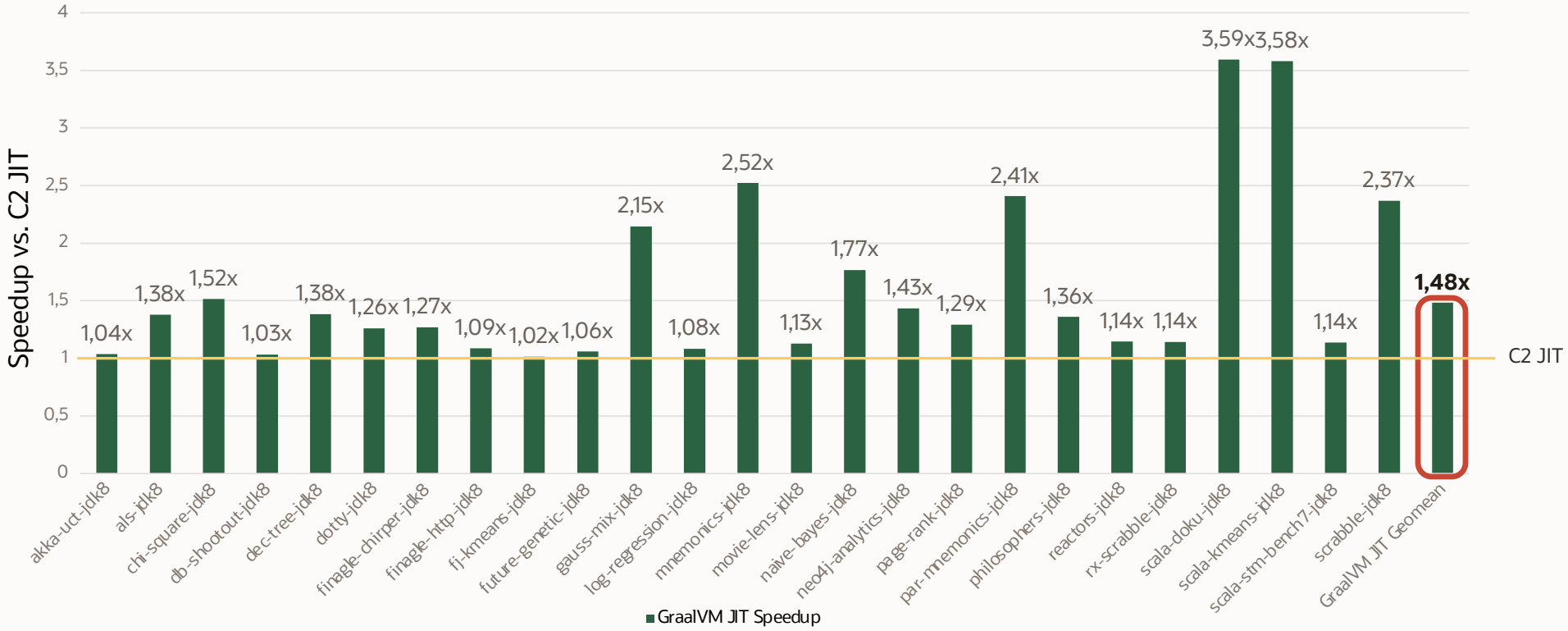
Native Image competitive with JIT and getting faster each release!



Results as of 2023-03-14

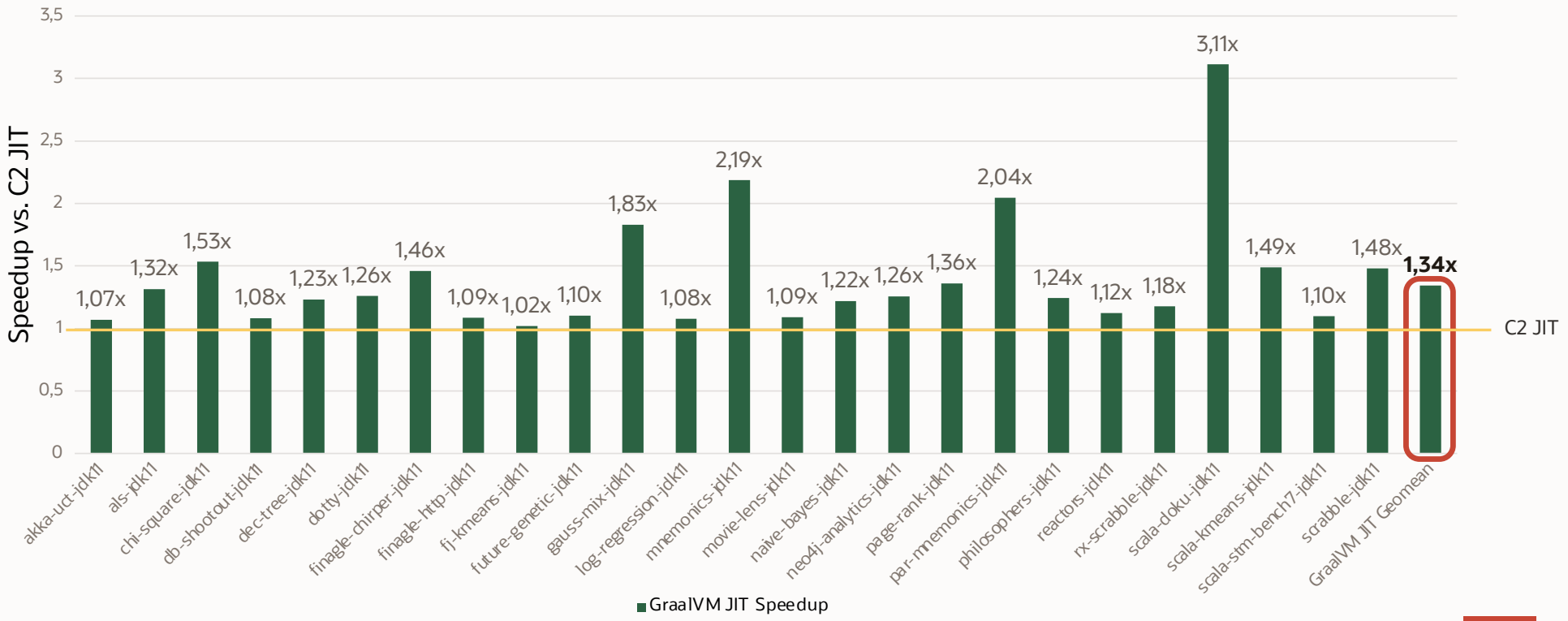
GraalVM Enterprise — Faster

Increased performance in real-world application benchmarks



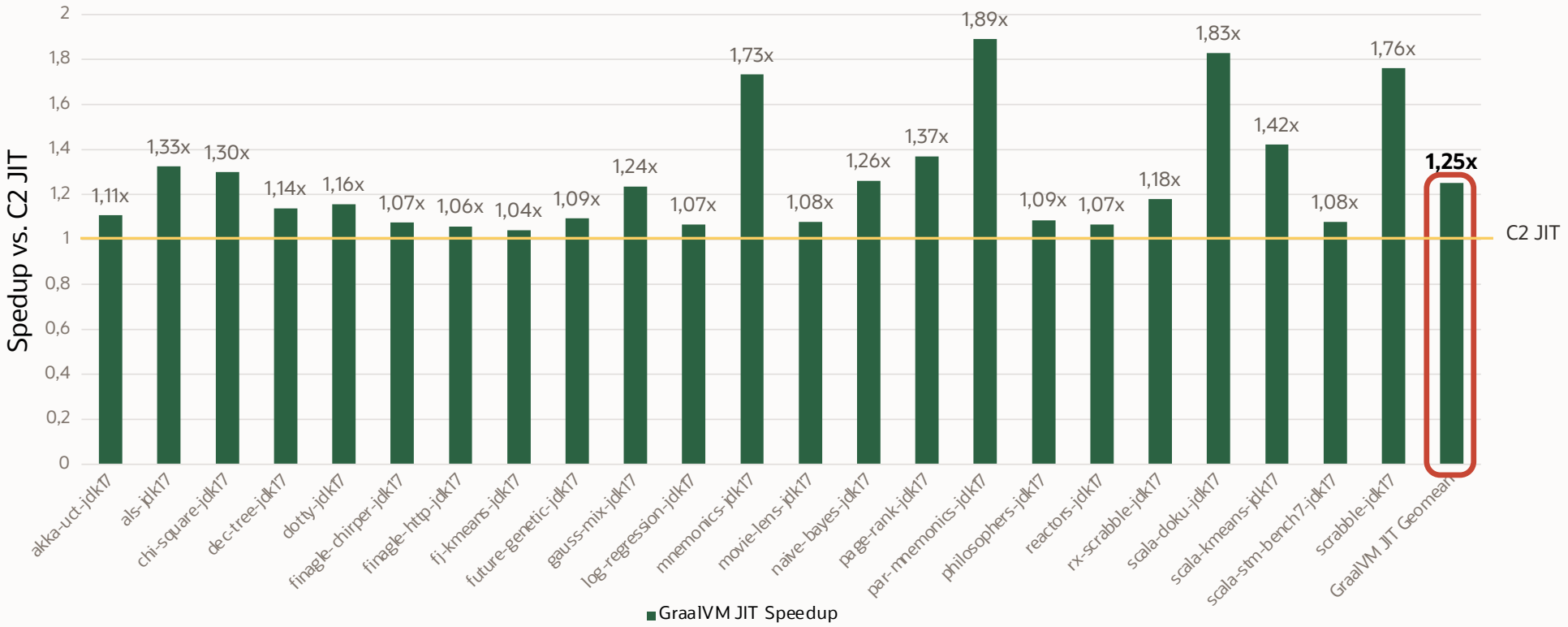
GraalVM Enterprise — Faster

Increased performance in real-world application benchmarks



GraalVM Enterprise — Faster

Increased performance in real-world application benchmarks



Spring Petclinic — Faster applications, less memory — GraalVM for JDK 23

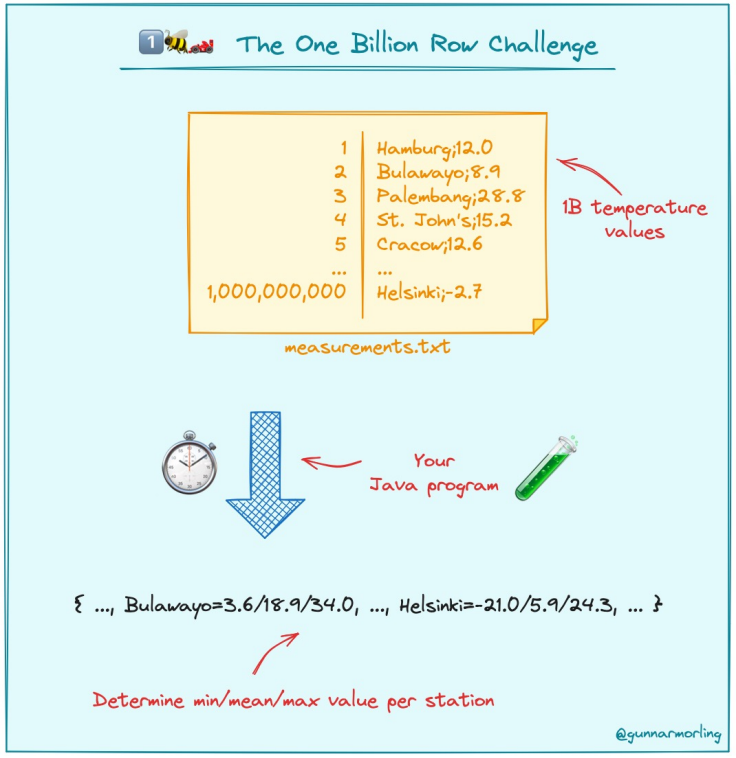
- Native executables delivering superior performance across-the-board on Spring PetClinic:
  - Higher throughput, lower memory requirements, lower latency, faster startup
  - Faster build times

Metric / runtime	GraalVM CE JIT	Oracle GraalVM Native Image	Difference
Memory Usage (max RSS)	910 MB	591 MB	35% better
Peak Throughput	9669 req/s	10358 req/s	7% faster
Throughput per memory	10637 req/(GB*s)	17526 req/(GB*s)	65% faster
Tail latency (p99)	26.3 ms	14.6 ms	45% better
Startup	14653 ms	353 ms	41x faster

Performance of Spring Petclinic with Oracle GraalVM Native Image and GraalVM CE JIT.
The benchmarking experiment ran the latest Spring Petclinic on Ampere A1 servers, restricting the workload to 16 CPUs and setting a maximum heap size of 512MB.

GraalVM — Parsing 1 Billion Rows in Java in 0.3 Seconds (1)

- The One Billion Row Challenge (1BRC) is a fun exploration of how far modern **Java can be pushed for aggregating one billion rows from a text file**
- Grab all your (virtual) threads, reach out to SIMD, optimize your GC, or pull any other trick, and create the fastest implementation for solving this task!



The text file contains temperature values for a range of weather stations. Each row is one measurement in the format `<string: station name>;<double: measurement>`, with the measurement value having exactly one fractional digit. The following shows ten rows as an example:

```
Hamburg;12.0
Bulawayo;8.9
Palembang;38.8
St. John's;15.2
Cracow;12.6
Bridgetown;26.9
Istanbul;6.2
Roseau;34.4
Conakry;31.2
Istanbul;23.0
```

The task is to write a Java program which reads the file, calculates the min, mean, and max temperature value per weather station, and emits the results on stdout like this (i.e. sorted alphabetically by station name, and the result values per station in the format `<min>/<mean>/<max>`, rounded to one fractional digit):

```
{Abha=-23.0/18.0/59.2, Abidjan=-16.2/26.0/67.3, Abéché=-10.0/29.4/69.0, Accra=-10.1/26.4/66.4, Addis Ababa=-23.7/16.(
```

GraalVM — Parsing 1 Billion Rows in Java in 0.3 Seconds (2)

- About the 1BRC — **a challenge** by Gunnar Morling **to see how modern Java can be used to process a file containing one billion rows as fast as possible**
- It seems like for two months until 31st of January 2024 the entire Java performance community had their eyes on this challenge - and so did the Oracle Labs
- There's much to learn from this challenge, which we will get to in a second, but first, let's look at the results
- The baseline solution proposed by Gunnar took 04:49 minutes to process the file; **then contestants, applying various performance optimizations, reduced the time** to as low as 01.53s on 8 cores, **and absolutely impressive 0.3s on 32 cores**!

GraalVM — Parsing 1 Billion Rows in Java in 0.3 Seconds (3)

Results

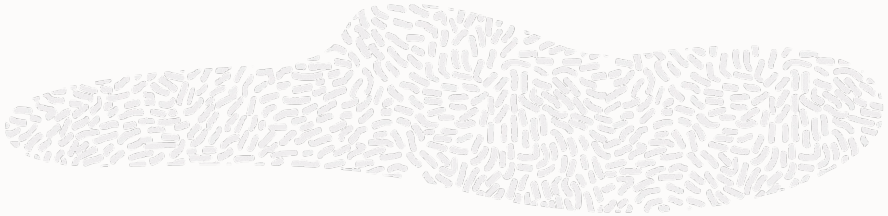
These are the results from running all entries into the challenge on eight cores of a Hetzner AX161 dedicated server (32 core AMD EPYC™ 7502P (Zen2), 128 GB RAM).

#	Result (m:s.ms)	Implementation	JDK	Submitter	Notes	Certificates
1	00:01.535	link	21.0.2-graal	Thomas Wuerthinger, Quan Anh Mai, Alfonso² Peterssen	GraalVM native binary, uses Unsafe	Certificate
2	00:01.587	link	21.0.2-graal	Artsiom Korzun	GraalVM native binary, uses Unsafe	Certificate
3	00:01.608	link	21.0.2-graal	Jaromir Hamala	GraalVM native binary, uses Unsafe	Certificate
	00:01.880	link	21.0.1-open	Serkan ÖZAL	uses Unsafe	Certificate
	00:01.921	link	21.0.2-graal	Van Phu DO	GraalVM native binary, uses Unsafe	Certificate

ORACLE

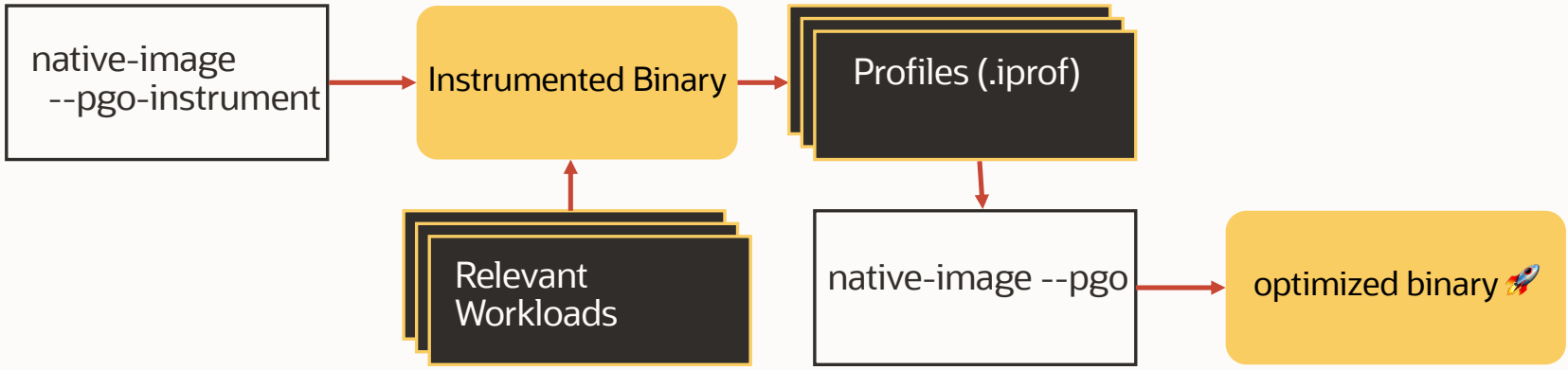
GraalVM Performance Optimization and Monitoring

Oracle GraalVM Native Image
What's included



- **Profile-guided optimizations** and more compiler optimizations for best peak performance
- **G1 GC** for running applications with large heaps and minimal pause times
- **Compressed object headers and pointers** for an even lower memory footprint
- **Machine learning** to infer profiling information automatically
- **Additional security features** with SBOM support

Optimizing performance of native image



Monitor performance with JFR

- Monitor and optimize performance of native images in production deployments
- include JFR at image build time:

```
native-image –enable-monitoring=jfr JavaApplication
```

- To enable JFR and start a recording:

```
./javaapplication -XX:+FlightRecorder -XX:StartFlightRecording="filename=recording.jfr"
```

ORACLE

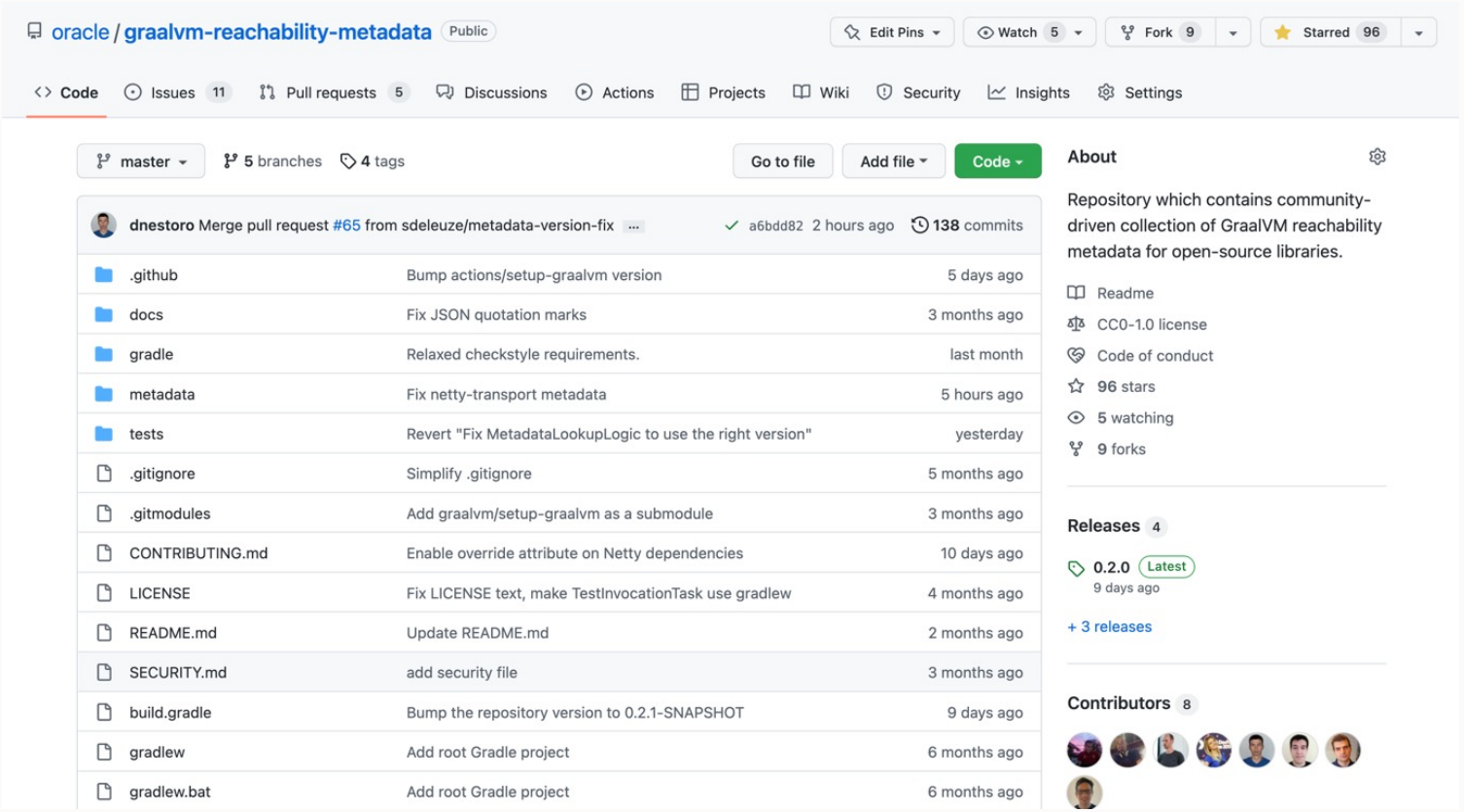
GraalVM & Reflection

GraalVM & Reflection — GraalVM meets Reflection ..

- **Collect Metadata with the Tracing Agent**
  - Most of the times you don't need it because libraries integrate with GraalVM out of the box
  - But in case one of them doesn't, you can get metadata via a tracing agent
    - https://www.graalvm.org/latest/reference-manual/native-image/metadata/AutomaticMetadataCollection/
- **Native Image tries to resolve the target elements through a static analysis that detects calls to the Reflection API**
  - If the analysis can not automatically detect your use of reflection, you might need additional configuration
- **Trace reflection, JNI, resource usage on the JVM with the tracing agent:**
  - Agent to record usage and produce configuration files for native images
    - `java -agentlib:native-image-agent=config-output-dir=META-INF/native-image ...`
  - Manual adjustment / addition might still be necessary
- **It's possible to exclude certain classes, such those that have caller methods or are targets of dynamic access**
  - Agent Advanced Usage
    - Caller-based Filters

```
{ "rules": [
    {"excludeClasses": "com.oracle.svm.**"},
    {"includeClasses": "com.oracle.svm.tutorial.*"},
    {"excludeClasses": "com.oracle.svm.tutorial.HostedHelper"}
  ],
  "regexRules": [
    {"includeClasses": ".*"},
    {"excludeClasses": ".*\\$\\$Generated[0-9]+"}
  ]
}
```


What about reflection in 3rd-party libraries?



https://github.com/oracle/graalvm-reachability-metadata/blob/master/CONTRIBUTING.md
https://github.com/oracle/graalvm-reachability-metadata/blob/master/library-and-framework-list.json

Is there an easier way to handle reflection? — Yes, using metadata

```
<plugin>
    <groupId>org.graalvm.buildtools</groupId>
    <artifactId>native-maven-plugin</artifactId>
    <version>${native.maven.plugin.version}</version>
    <extensions>true</extensions>
    <executions>
        <execution>
            <id>build-native</id>
            <goals>
                <goal>compile-no-fork</goal>
            </goals>
            <phase>package</phase>
        </execution>
    </executions>
    <configuration>
        <!-- tag::metadata-default[] -->
        <metadataRepository>
            <enabled>true</enabled>
        </metadataRepository>
        <!-- end::metadata-default[] -->
    </configuration>
</plugin>
```

ORACLE

GraalVM Contribution

Contribute to GraalVM

https://www.graalvm.org/community/

GraalVM is an open source Oracle project where people from all over the world contribute their work, help each other, and make GraalVM innovative

- We are thankful for past contributions both big and small and always welcome new collaborators
- Only a large community can make this project a strong language virtualization technology

There are two common ways to collaborate:

- **By submitting GitHub issues for bug reports, questions, or requests for enhancements**
  - ❖ Note, that a security vulnerability should be reported to secalert_us@oracle.com
  - ➢ https://github.com/oracle/graal/issues

- **By creating GitHub pull requests**
  - ➢ https://github.com/oracle/graal/pulls

If you consider contributing solely to the documentation, please check this guide:

- ❖ **How to Contribute to GraalVM Documentation**
- ➢ https://github.com/oracle/graal/blob/master/docs/README.md

Summary: GraalVM Readiness Checklist — Use for Production

1	Performance	Graal JIT, Graal Native Image PGO	high throughput
2	Startup Time	Graal Native Image	fast startup
3	Polyglot	Graal Language Runtimes	JS, Py, Wasm, others
4	Compatibility	GraalVM, JVM	Java
5	Security	Graal Sandbox	limit resources on CPU & MEM
6	Monitoring & Tooling	GraalVM Tooling eco system	VisualVM, JFR, DataDog, others
7	Development & Debug	GraalVM Native Image	IntelliJ IDEA, ..
8	Reflection	GraalVM Native Image	Tracing Agent to gather metadata and prepare configuration files
9	Frameworks	GraalVM Native Image	Spring Boot, Microservices, ..
10	Fast LLM inference engine in Java	Java Vector API, GraalVM, AOT CPU base performance	Integrating with LangChain4j, ..
11	License & Support	GFTC, and commercial	Three years for free, and Java SE Subscription

Danke!

Wolfgang.Weigend@oracle.com

Bluesky: @wolfgangweigend.bsky.social
Twitter/X: @wolflook