GraalVM Native Images

Instant Startup and Low Footprint for Java

Christian Wimmer

ORACLE CODE

Live for the Code

ORACLE

Safe Harbor Statement

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

“Hello World”

Download GraalVM:

https://www.graalvm.org/downloads/

Build a native image:

```
$ echo 'public class HelloWorld { public static void main(String[] args) { System.out.println("Hello World!"); } }' > HelloWorld.java
$ ~/graalvm-ee-1.0.0-rc8/bin/javac HelloWorld.java
$ ~/graalvm-ee-1.0.0-rc8/bin/native-image HelloWorld
```

This gives you an executable "helloworld":

```
$ ./helloworld
Hello World!
```

Execution time and instruction count:

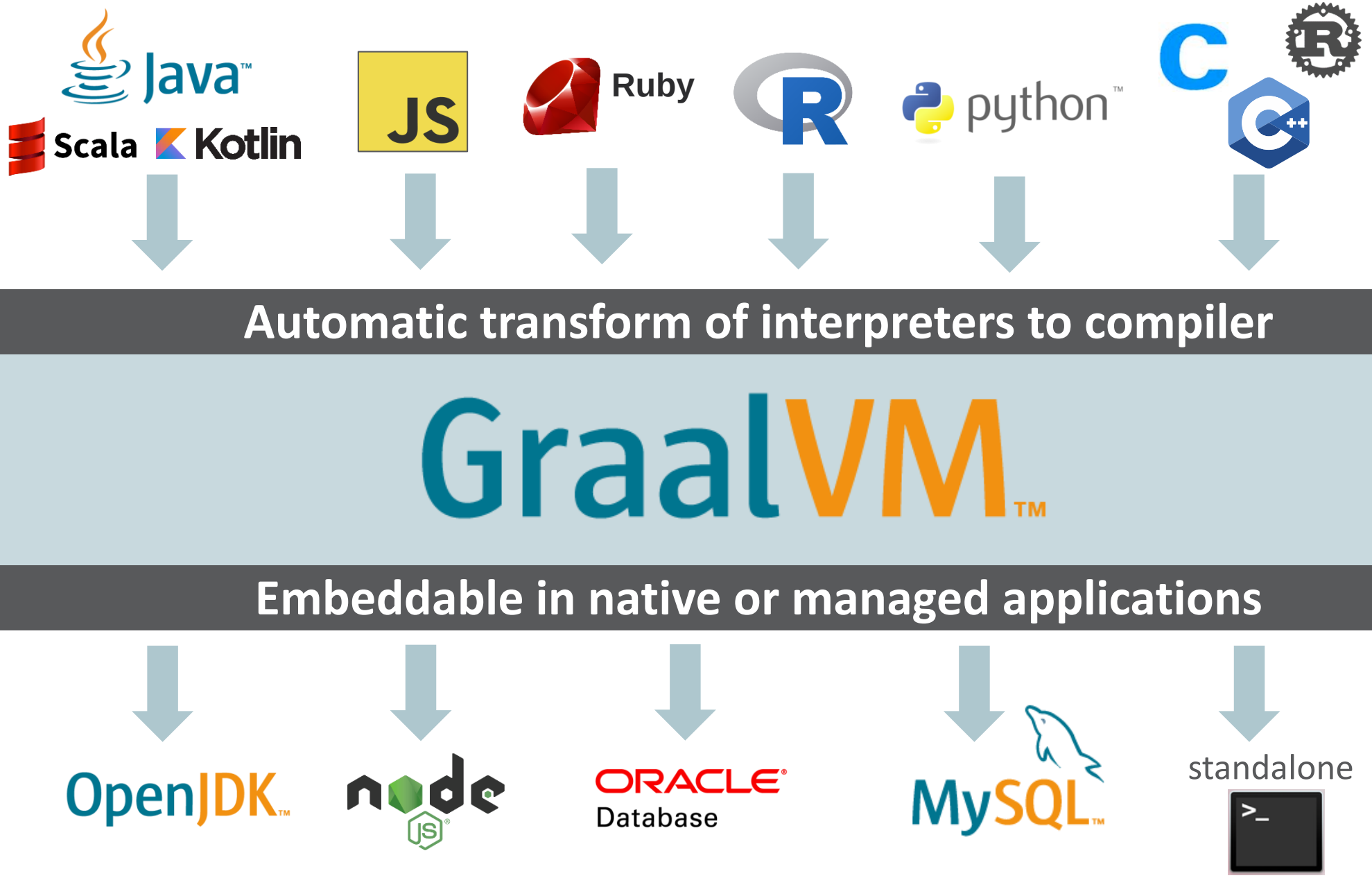
```
real      0m0.003s
user      0m0.000s
sys       0m0.000s

286,935  instructions:u
```

Comparison to Java HotSpot VM, JDK 11:

```
real      0m0.078s
user      0m0.096s
sys       0m0.012s

262,053,163  instructions:u
```



Multiplicative Value-Add of GraalVM Ecosystem

Languages	*	**GraalVM**	*	**Embeddings**
Java		Optimizations		HotSpot JVM
JavaScript		Tooling		MySQL
Ruby		Interoperability		Oracle Database
R		Security		Node.js
Python		…		Spark
C/C++, FORTRAN				…
…				

Add your own language or embedding or language-agnostic tools

Community Edition (CE)

GraalVM CE is available for free for development and production use. It is built from the GraalVM sources available on GitHub. We provide pre-built binaries for GraalVM CE for Linux on x86 64-bit systems.

DOWNLOAD FROM GITHUB

Enterprise Edition (EE)

GraalVM EE provides additional performance, security, and scalability relevant for running critical applications in production. It is free for evaluation uses and available for download from the Oracle Technology Network. We provide binaries for GraalVM EE for Linux or Mac OS X on x86 64-bit systems.

DOWNLOAD FROM OTN

www.graalvm.org/downloads

Maven Dependencies

- Either just depend on the API, or on everything (including internal classes)

Maven dependency for API only:

```
<dependency>
  <groupId>org.graalvm.sdk</groupId>
  <artifactId>graal-sdk</artifactId>
  <version>1.0.0-rc8</version>
  <scope>provided</scope>
</dependency>
```

Maven dependency that includes internal classes too:

```
<dependency>
  <groupId>com.oracle.substratevm</groupId>
  <artifactId>svm</artifactId>
  <version>1.0.0-rc8</version>
  <scope>provided</scope>
</dependency>
```

- Provide options to native-image tool
  - Multiple native-image.properties files can be in META-INF/native-image/*your-id*
  - Allows libraries to ship with native-image configuration

```
ImageName = netty-plot
Args = --features=com.oracle.svm.nettyplot.PlotterSingletonFeature \
       -H:ReflectionConfigurationResources=${.}/reflection-config.json \
       --delay-class-initialization-to-runtime=io.netty.handler.codec.http.HttpObjectEncoder \
       -H:+SpawnIsolates
```

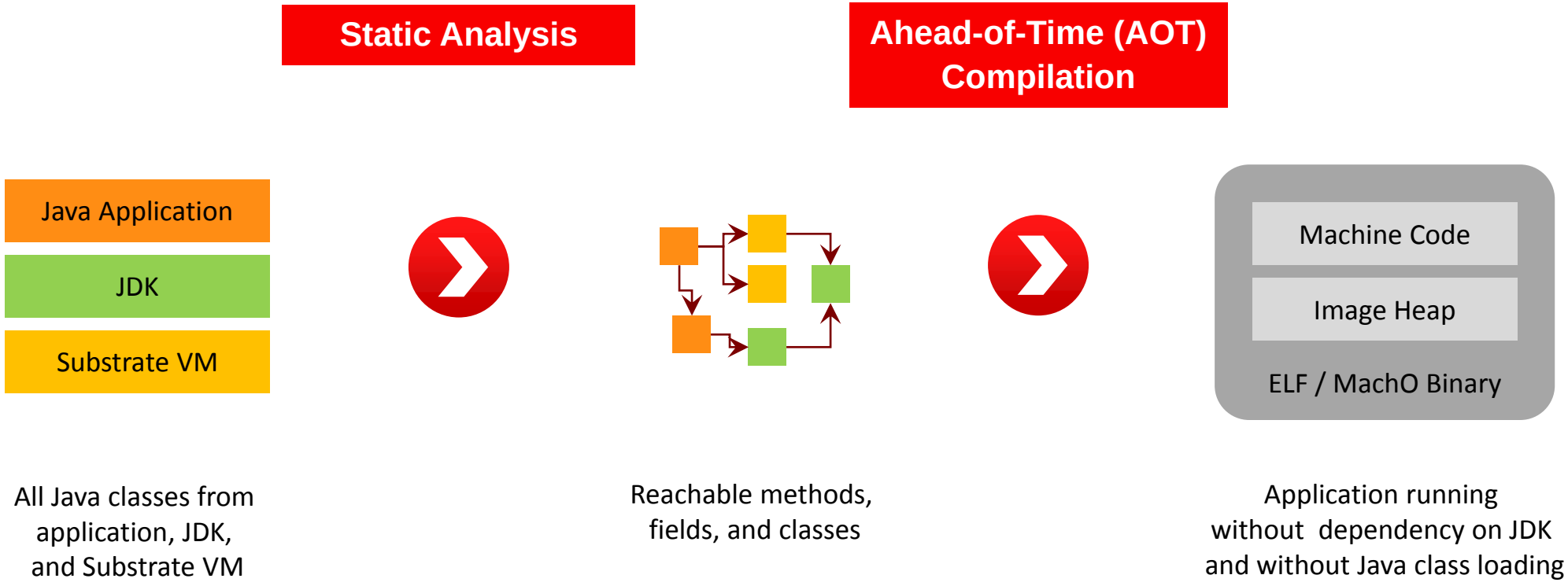
  - Get your library native-image ready!

Native Image vs. Regular Java VM

- Use native image when
  - Startup time matters
    - Short-running command line applications
    - Serverless cloud functions, e.g., Fn Project
  - Memory footprint matters
    - Small to medium-sized heaps (100 MByte – a few GByte)
  - All Java code is known ahead of time
    - Single-application cloud server, e.g., Helidon

- Use a regular Java VM when
  - Heaps size is large
    - Big Data analytics
    - Multiple GByte – TByte heap size
  - Classes are only known at run time
    - “Traditional” Java application server

Native Image Generation



Closed World Assumption

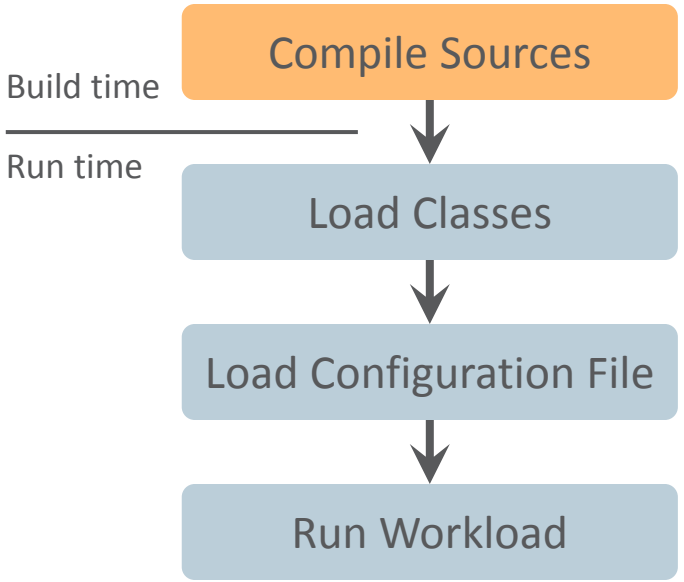
- The static analysis needs to see all bytecode
  - Otherwise aggressive AOT optimizations are not possible
  - Otherwise unused classes, methods, and fields cannot be removed
  - Otherwise a class loader / bytecode interpreter is necessary at run time
- General pattern: configuration files / options passed to the image generator
- Examples for supported features
  - Class.forName() for classes registered using -H:ReflectionConfigurationResources=...
  - Proxy.getProxyClass() for interfaces registered using -H:DynamicProxyConfigurationResources=...
  - ClassLoader.getResource() for resources registered using -H:IncludeResources=...
- Examples for unsupported features
  - Load new classes from the user / the internet using a URLClassLoader

Image Heap

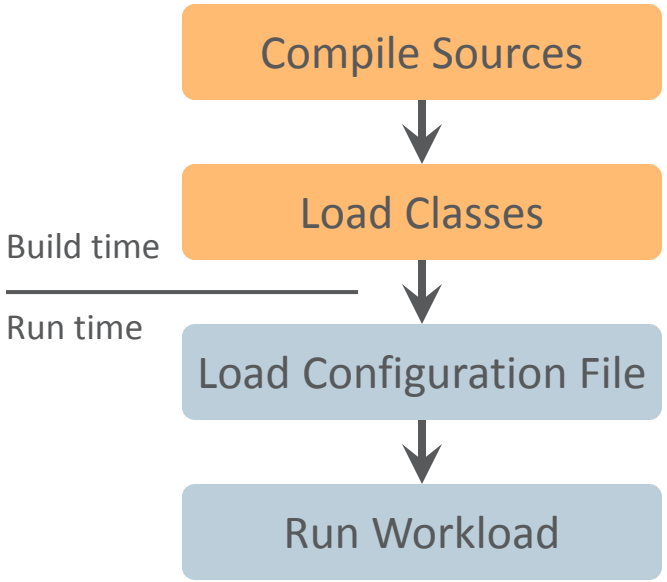
- Execution at run time starts with an initial heap: the “image heap”
  - Objects are allocated in the Java VM that runs the image generator
- Why?
  - Do things once at build time instead at every application startup
- Who allocates the objects?
  - Class initializers, initializers for static and static final fields
  - Explicit code that is part of a so-called “Feature”
- Examples for objects on the image heap
  - java.lang.Class objects
  - enum constants
  - Localization and charset data

Benefits of Image Heap

Execution on regular Java VM

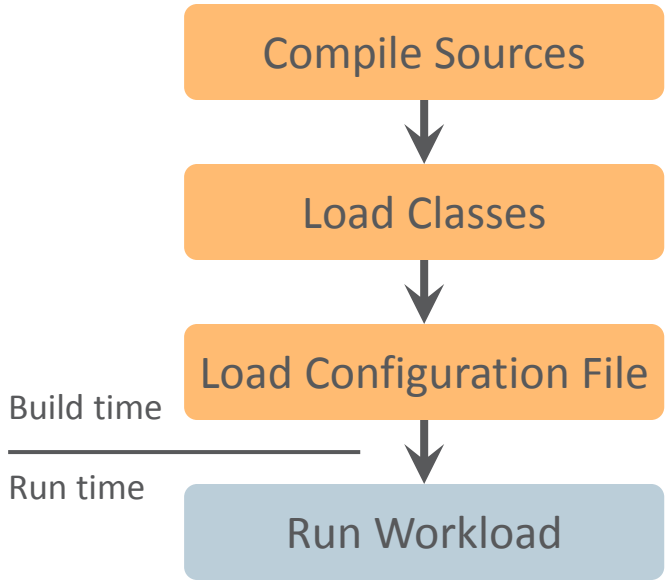


Native Image



Classes, static data
on Image Heap

Native Image
with custom feature to pre-load
configuration file at build time



Configuration objects on Image Heap
for faster startup

Isolates

- Start multiple independent VM instances in one process
- Strict isolation of Java objects
  - Strong security guarantees
  - Free memory without GC when isolate is shut down
  - GC is independent of other isolates
- Java API and C API available to manage isolates
  - Need to use -H:+SpawnIsolates (will be default in one of the next releases)

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
- Reflection example: Gson library to serialize / deserialize Java objects

Data class:

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

Serialize Java object to JSON:

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

Deserialize JSON to Java object:

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

Reflection configuration:

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

Class Initialization

- By default, all classes are initialized during image generation
  - Avoid class initialization checks at run time
  - Allow instances to be on the image heap
  - Allow aggressive constant folding of final fields
- But what about...
  - Class initializers that load native libraries
  - Class initializers that depend on system properties, e.g., user name
  - Class initializers that depend on configuration files
- Manually provide list of classes that are initialized at run time
  - List of classes provided to the image generator
  - We are working to improve usability: difficult to trace transitive class initializations
    - All classes that depend on a runtime initialized class must also be initialized at run time

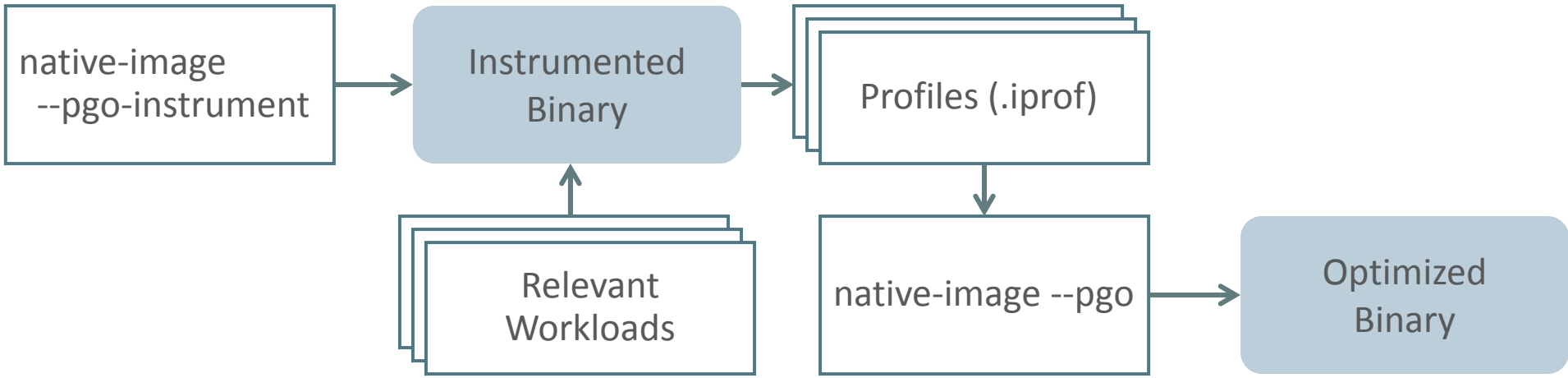
Service Loader

- Allows dynamic configuration of Java applications
  - Look up classes implementing a service interface
  - Requires a combination of resources (in META-INF directory) and reflection

- The image generator 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

Profile-Guided Optimizations (PGO)

- AOT compiled code cannot optimize itself at run time
  - No dynamic “hot spot” compilation as regular Java VMs
- PGO requires running relevant workloads before building an image
  - Advantage: optimized code runs immediately at startup, no “warmup” curve



Jump Start Your Project

- The image generator aggressively checks for unsupported features
  - Report as many errors at build time instead of at run time
- But that can be overwhelming
  - How do I know my application will run after fixing all issues, without first fixing them?
  - Use -H:+ReportUnsupportedElementsAtRuntime

Substitutions

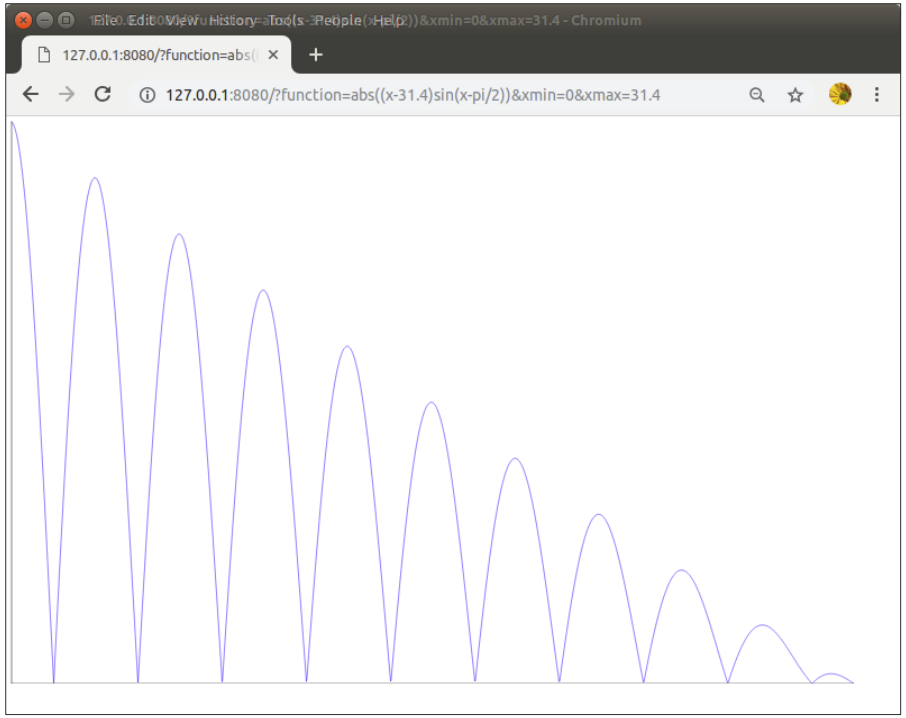
- Ideally, all code in an application can be adapted for native images
  - But in reality, there are dependencies that are beyond your control
  - It is difficult to maintain forks or patches for libraries
- Substitutions are a band-aid to patch libraries on-the-fly during image generation
  - Provide substitutions for methods that you cannot change otherwise
  - Use with care and as little as possible
    - Easy to make mistakes that are hard to debug
  - Not part of our official API: we do not want to promote it

```
@TargetClass(className = "io.netty.util.internal.logging.InternalLoggerFactory")
final class Target_io_netty_util_internal_logging_InternalLoggerFactory {
    @Substitute
    private static InternalLoggerFactory newDefaultFactory(String name) {
        return JdkLoggerFactory.INSTANCE;
    }
}
```

Demo: Netty Web Server

- Complete but minimal example
  - Using Netty web server
  - Maven for building
  - Isolates for plotting
  - Reflection configuration file for Netty
  - Substitutions for Netty

- Blog post with details
  - https://medium.com/graalvm/
- Source code on GitHub
  - https://github.com/graalvm/graalvm-demos/



Questions?

www.graalvm.org

@graalvm

- GraalVM: Vision and Roadmap
  - Thursday, 2:00 pm, Moscone West - Room 2006 [DEV5580]
- Ten Things You Can Do With GraalVM
  - Wednesday, 9:00 am, Moscone West - Room 2001A [HOL5576]
- Instant Startup and Low Footprint for Java
  - Wednesday, 4:00 pm, Moscone West - Room 2006 [DEV5705]
- Oracle Database MLE: JavaScript, Python, and More in the Database
  - Monday, 12:30 pm, Moscone West – Room 2022
- Running JavaScript Stored Programs Inside MySQL Server
  - Tuesday, 11:15 am, Park Central (Floor 2) - Olympic [PRO2091]