“Static Java”: The Programming Model of GraalVM Native Image

Christian Wimmer
Architect, GraalVM Native Image
christian.wimmer@oracle.com

Christian Wimmer

5+ years working on Java HotSpot VM

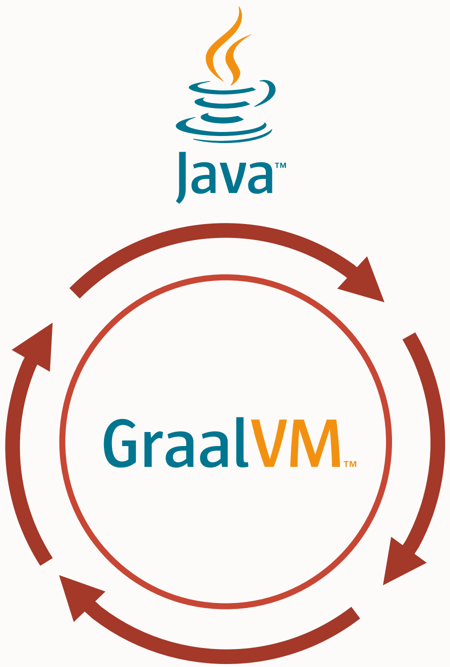
- SSA form and register allocation for the client compiler
- Research of object layout optimizations

3 years “detour” into language based security

10+ years working on GraalVM

- Native Image architect, from first commit to production

What is GraalVM?



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

Ahead-of-Time (AOT) “Native Image” generator

Multi-language support

Goals of this talk

1. Introduce and explain the programming model of GraalVM Native Image
2. Convince you that this is a worthwhile programming model for Java
3. Show where the closed-world assumption helps and where it hurts

Keeping Java vibrant

- Cloud and microservices change how software is written
- New languages like Go
- New ecosystems like NodeJS

- Java can be better than Go for application startup and memory footprint
- Java can have a better programming model than Go and NodeJS

Our world. Moved by Java.

25 years of ongoing innovation

It's been 25 years since Java set our world spinning with new possibilities, closing the divide between the impossible and the achievable.

So let's celebrate the people and innovations of the last quarter of a century—and those yet to come. Here's to the millions of Java developers everywhere.

Follow along and participate in the #MovedbyJava campaign on social media.

What is GraalVM Native Image

- An ahead-of-time (AOT) compiler for Java
  “gcc for Java”

  No, AOT compilation is an implementation detail

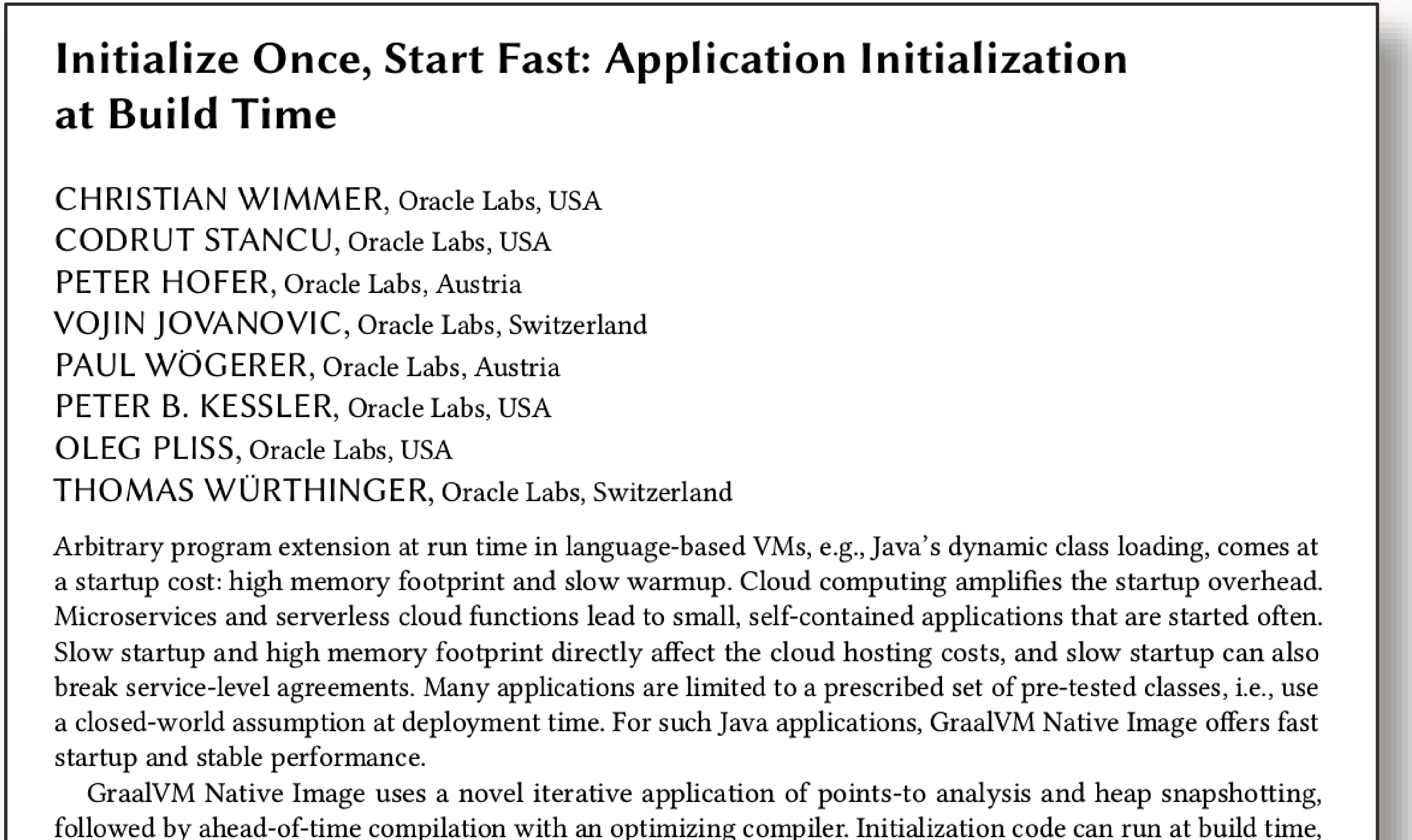
- A snapshotting tool for Java
  “CRIU for Java”

  Yes, but with an explicit control of snapshot building

- A new programming model for Java:
  “Application initialization at build time”

Paper with details, examples, benchmarks

https://doi.org/10.1145/3360610



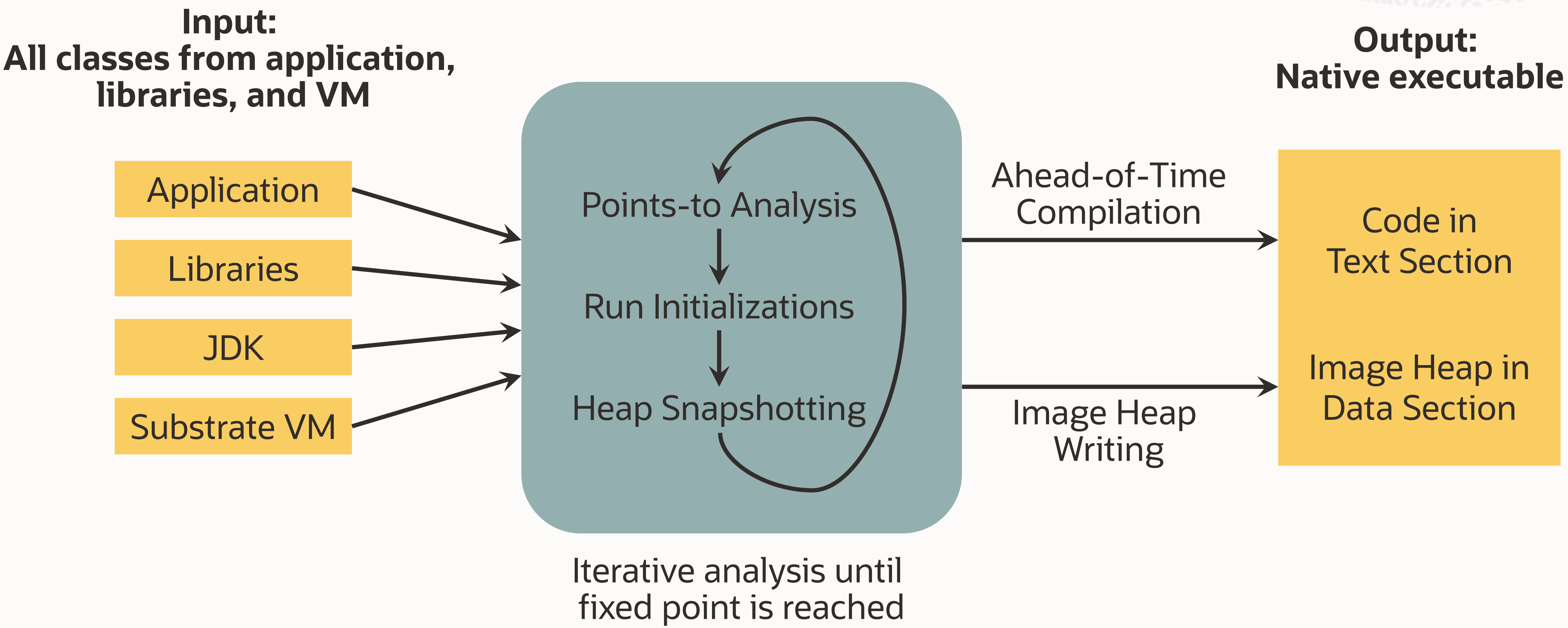
Initialize Once, Start Fast: Application Initialization at Build Time

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.

GraalVM Native Image uses a novel iterative application of points-to analysis and heap snapshotting, followed by ahead-of-time compilation with an optimizing compiler. Initialization code can run at build time,

Native image generation



Input:
All classes from application, libraries, and VM

Application

Libraries

JDK

Substrate VM

Points-to Analysis

Run Initializations

Heap Snapshotting

Iterative analysis until fixed point is reached

Ahead-of-Time Compilation

Image Heap Writing

Output:
Native executable

Code in Text Section

Image Heap in Data Section

Closed-world assumption

- The points-to analysis needs to see all bytecode
  - Otherwise aggressive AOT optimizations are not possible
  - Otherwise unused classes, methods, and fields cannot be removed
  - Otherwise a class loader / bytecode interpreter is necessary at run time

- Dynamic parts of Java require configuration at build time
  - Reflection, JNI, Proxy, resources, …

- No loading of new classes at run time

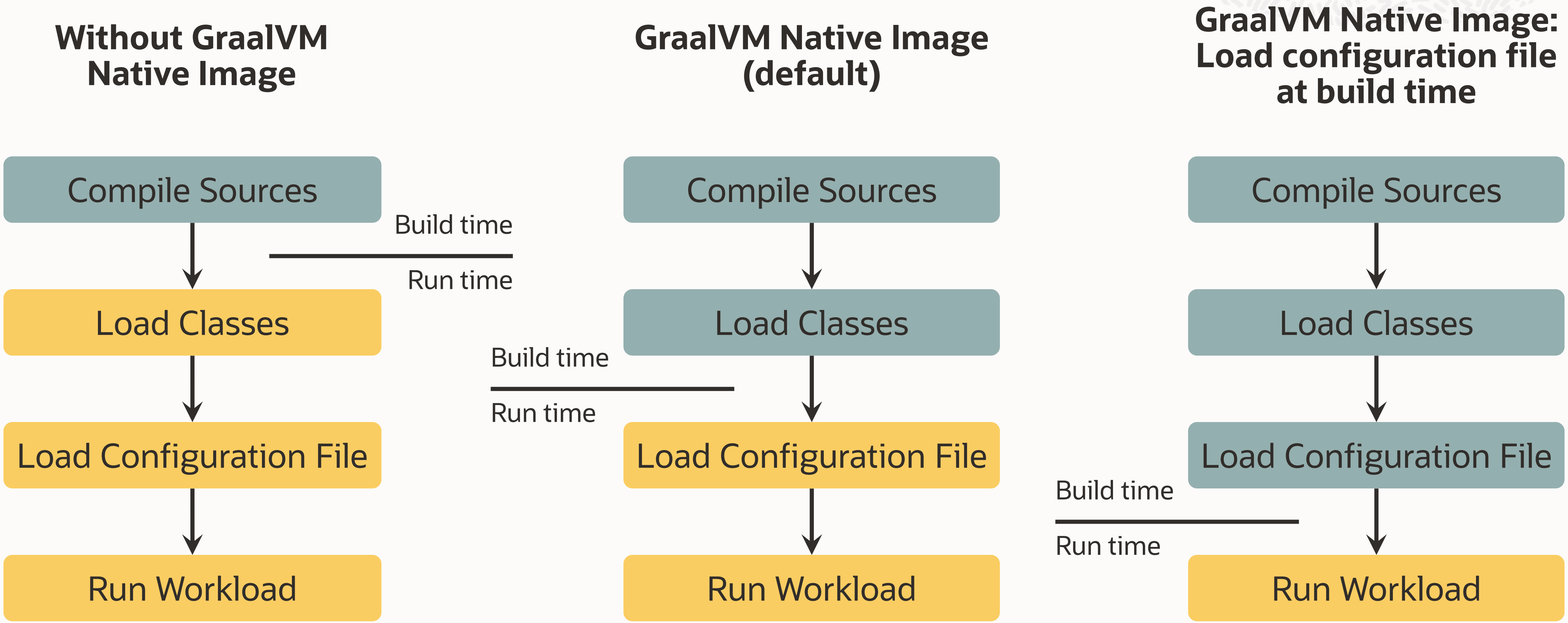
Image heap

- 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



Without GraalVM Native Image

Compile Sources

Build time

Run time

Load Classes

Load Configuration File

Run Workload

GraalVM Native Image (default)

Compile Sources

Load Classes

Build time

Run time

Load Configuration File

Run Workload

GraalVM Native Image: Load configuration file at build time

Compile Sources

Load Classes

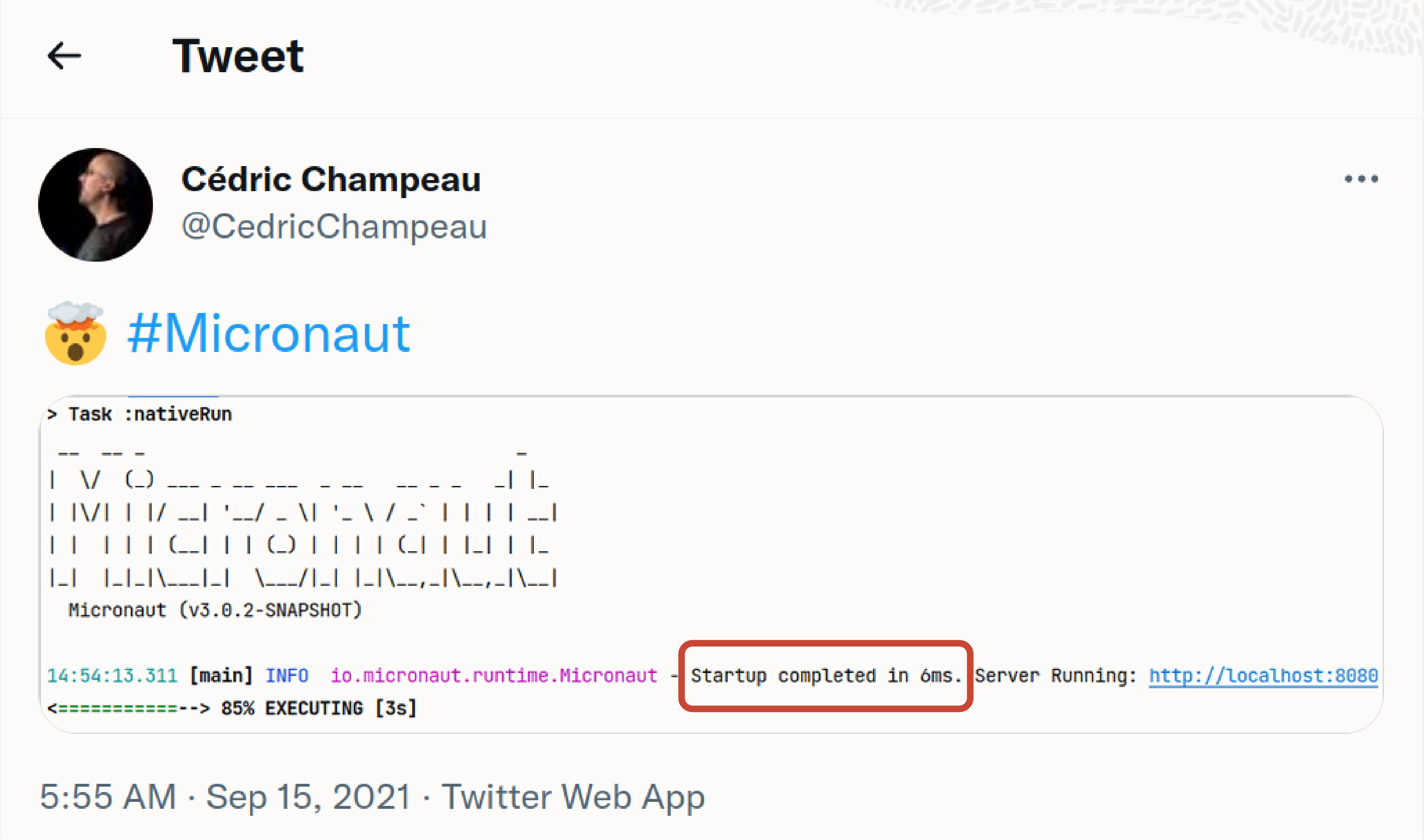
Load Configuration File

Build time

Run time

Run Workload

Nice theory, but does it work in practice?



Cédric Champeau
@CedricChampeau

#Micronaut

```
> Task :nativeRun
 __  __ _                                  _
|  \/  (_) ___ _ __ ___  _ __   __ _ _   _| |_
| |\/| | |/ __| '__/ _ \| '_ \ / _` | | | | __|
| |  | | | (__| | | (_) | | | | (_| | |_| | |_
|_|  |_|_|\___|_|  \___/|_| |_|\__,_|\__,_|\__|
  Micronaut (v3.0.2-SNAPSHOT)

14:54:13.311 [main] INFO  io.micronaut.runtime.Micronaut - Startup completed in 6ms. Server Running: http://localhost:8080
<===========--> 85% EXECUTING [3s]
```

5:55 AM · Sep 15, 2021 · Twitter Web App

Why not “just AOT compilation”?

Several AOT compilers for Java exist or existed

- jaotc (part of OpenJDK, using the GraalVM compiler)
- gcj
- Excelsior JET

But Java code is hard to optimize without data

- Java code is very object oriented
- AOT compilation only covers the “code” aspect of objects and ignores the “data” aspect
- Simple example: You cannot optimize Java `enum` usages without having the actual `enum` instances
- To get to data (Java objects), you need to run parts of your application

Terminology

Many terms can and have been used for this programming model:

- Snapshotting (but focuses too much on the “data” aspect)
- Staging (too generic term)
- Phase awareness (too generic term)
- Partial evaluation (already used by Truffle language framework in GraalVM)
- Hosted execution (used by meta-circular VMs such as the Maxine VM)
- **Image generation**

For GraalVM Native Image, we mostly use “image generation” (and sometimes “hosted execution”)

- Consistent with the `jlink` terminology of a “custom runtime image”

Programming model vs. implementation detail

Parts of GraalVM Native Image that are necessary for the programming model:

- Closed-world assumption: if you do not know your entire application, you cannot aggressively optimize
- Static analysis: computes reachable classes, methods, and fields based on closed-world assumption
- Heap snapshotting: the “data” part of OOP
- Running application code at image build time
  - Implicitly when initializing classes at image build time
  - Explicitly based on triggers from the static analysis (better and more controllable than class initializer)

Implementation details

- AOT compilation: interpretation and JIT compilation work as well
- Points-to analysis: the actual kind of static analysis does not matter (can be a simpler analysis)
- A runtime system written in Java (“Substrate VM”)
- Building a single self-contained executable

Detour: history of GraalVM Native Image

A lot of implementation details are due to the original use case: Oracle Database Multilingual Engine (MLE)
- Integration of the GraalVM language framework (Truffle) into the Oracle database

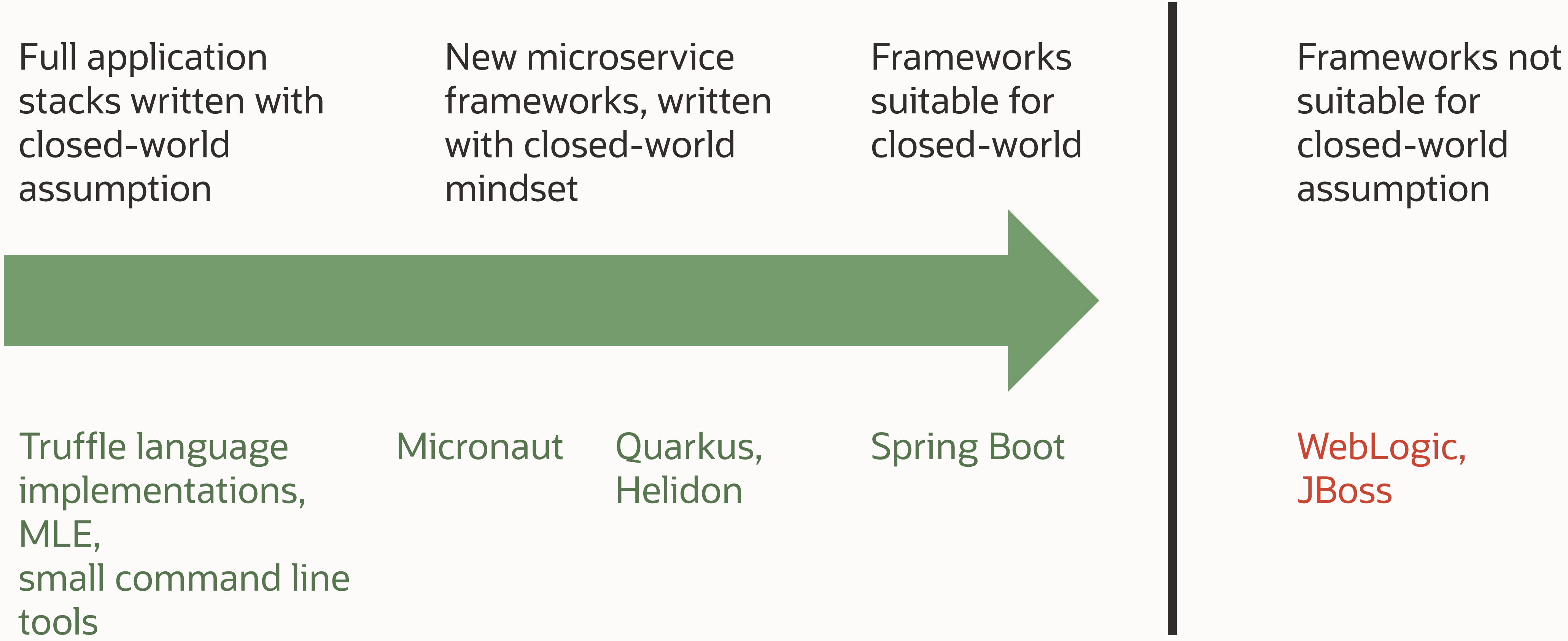
MLE allowed many simplifying assumptions:
- All code apart from the JDK is developed by same team and can adapt to restrictions
- One production OS: the OS abstraction layer of the database
  - Linux and MacOS support for development
  - Only basic file system and network support is necessary.
- No Java reflection or JNI support is needed
- Fast initialization of a database session and low memory footprint are essential

Result: a quite restrictive but very well performing system

Many other uses cases profit from fast startup and low memory footprint
- Microservices, command line applications

But “real-world code” does not follow the initial restrictions
- How far can we go and lift restrictions without sacrificing the main benefits?
- What other applications can profit from the new programming model (application initialization at build time)?

Pushing the boundaries



Full application stacks written with closed-world assumption

New microservice frameworks, written with closed-world mindset

Frameworks suitable for closed-world

Frameworks not suitable for closed-world assumption

Truffle language implementations,
MLE,
small command line tools

Micronaut

Quarkus,
Helidon

Spring Boot

WebLogic,
JBoss

Class initialization vs. application initialization

“Class initialization at build time” and “application initialization at build time” are equivalent

- If you want to initialize your application at build time, you can write a class initializer for this and mark the class as “initialize at build time”
- Initializing the application at build time requires initialization of some application classes

Java class initialization is tricky

- Some reasons for class initialization are not obvious
  - Example: adding a default method in an interface changes class initialization behavior
- Class initialization order is non-deterministic when class dependencies are cyclic
- A static analysis cannot determine the order in which class initialization happens at run time

An explicit API for application initialization is better than relying on side effects of class initialization

- And in addition the explicit API provides access to static analysis results
- Especially useful for frameworks: “if XYZ is used by the application, also include/initialize ABC”

Static analysis API exposed to application

Active API: register callbacks for analysis status changes

```
/* Invoke callback when one of the provided elements (can be Class, Field, or Executable) gets reachable. */
void registerReachabilityHandler(Consumer<DuringAnalysisAccess> callback, Object... elements);

/* Invoke callback when a new subtype of the provided type gets reachable. */
void registerSubtypeReachabilityHandler(BiConsumer<DuringAnalysisAccess, Class<?>> callback, Class<?> baseClass);

/* Invoke callback when a new override of the provided method gets reachable. */
void registerMethodOverrideReachabilityHandler(BiConsumer<DuringAnalysisAccess, Executable> callback, Executable baseMethod);
```

Passive API: query current analysis status

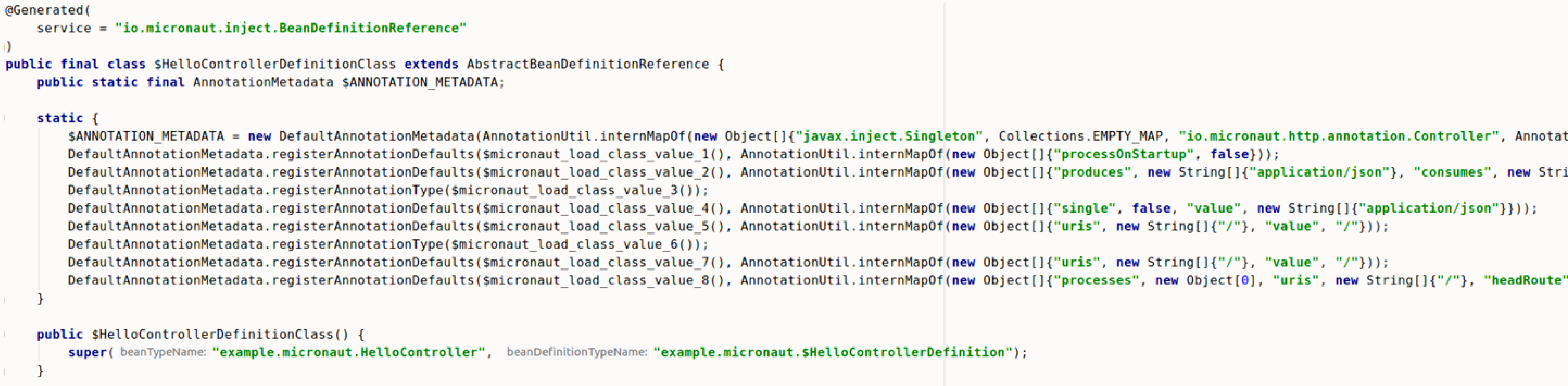
```
boolean isReachable(Class<?> clazz);
boolean isReachable(Field field);
boolean isReachable(Executable method);

Set<Class<?>> getReachableSubtypes(Class<?> baseClass);
Set<Executable> getReachableMethodOverrides(Executable baseMethod);
```

Participate in heap snapshotting: transform entire object or transform individual field value before it is added to image heap

```
void registerObjectTransformer(Function<Object, Object> transformer);  // actually called registerObjectReplacer right now
void registerFieldValueTransformer(Field field, Function transformer); // actually done via @Alias and @RecomputeFieldValue
```

Example: Micronaut metadata

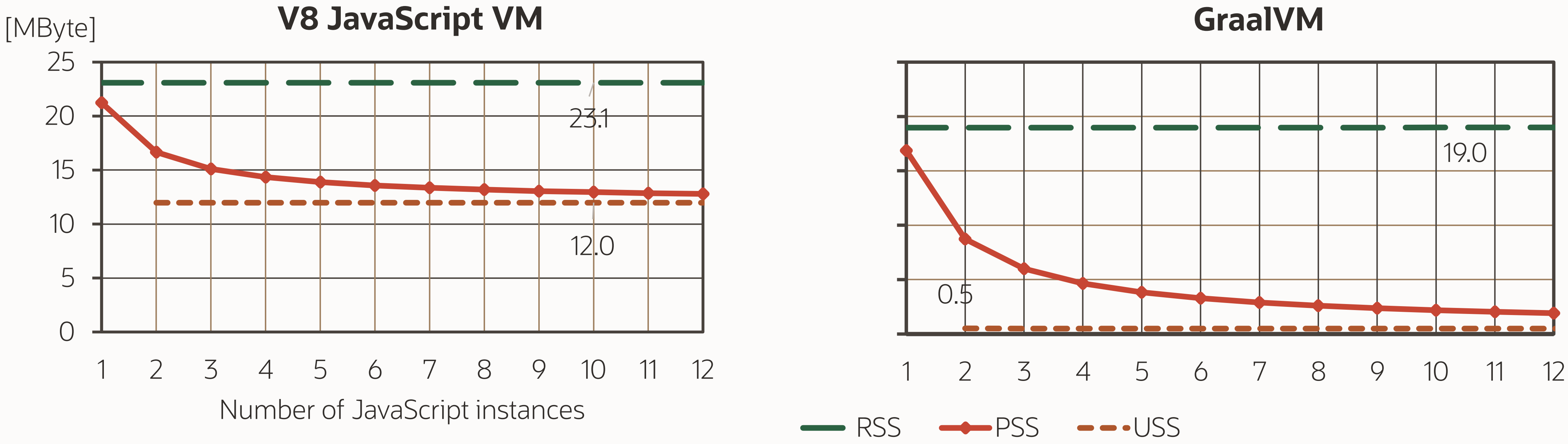
- Micronaut annotation processor generates metadata holder classes to avoid reflection at run time
- Class initializer initializes large data structure used by runtime lookup
- Such initialization code can run at image build time
    - Framework can guarantee that code is safe for build-time execution



```
@Generated(
    service = "io.micronaut.inject.BeanDefinitionReference"
)
public final class $HelloControllerDefinitionClass extends AbstractBeanDefinitionReference {
    public static final AnnotationMetadata $ANNOTATION_METADATA;

    static {
        $ANNOTATION_METADATA = new DefaultAnnotationMetadata(AnnotationUtil.internMapOf(new Object[]{"javax.inject.Singleton", Collections.EMPTY_MAP, "io.micronaut.http.annotation.Controller", Annotat
        DefaultAnnotationMetadata.registerAnnotationDefaults($micronaut_load_class_value_1(), AnnotationUtil.internMapOf(new Object[]{"processOnStartup", false}));
        DefaultAnnotationMetadata.registerAnnotationDefaults($micronaut_load_class_value_2(), AnnotationUtil.internMapOf(new Object[]{"produces", new String[]{"application/json"}, "consumes", new Stri
        DefaultAnnotationMetadata.registerAnnotationType($micronaut_load_class_value_3());
        DefaultAnnotationMetadata.registerAnnotationDefaults($micronaut_load_class_value_4(), AnnotationUtil.internMapOf(new Object[]{"single", false, "value", new String[]{"application/json"}}));
        DefaultAnnotationMetadata.registerAnnotationDefaults($micronaut_load_class_value_5(), AnnotationUtil.internMapOf(new Object[]{"uris", new String[]{"/"}, "value", "/"}));
        DefaultAnnotationMetadata.registerAnnotationType($micronaut_load_class_value_6());
        DefaultAnnotationMetadata.registerAnnotationDefaults($micronaut_load_class_value_7(), AnnotationUtil.internMapOf(new Object[]{"uris", new String[]{"/"}, "value", "/"}));
        DefaultAnnotationMetadata.registerAnnotationDefaults($micronaut_load_class_value_8(), AnnotationUtil.internMapOf(new Object[]{"processes", new Object[0], "uris", new String[]{"/"}, "headRoute"
    }

    public $HelloControllerDefinitionClass() {
        super( beanTypeName: "example.micronaut.HelloController",  beanDefinitionTypeName: "example.micronaut.$HelloControllerDefinition");
    }
```

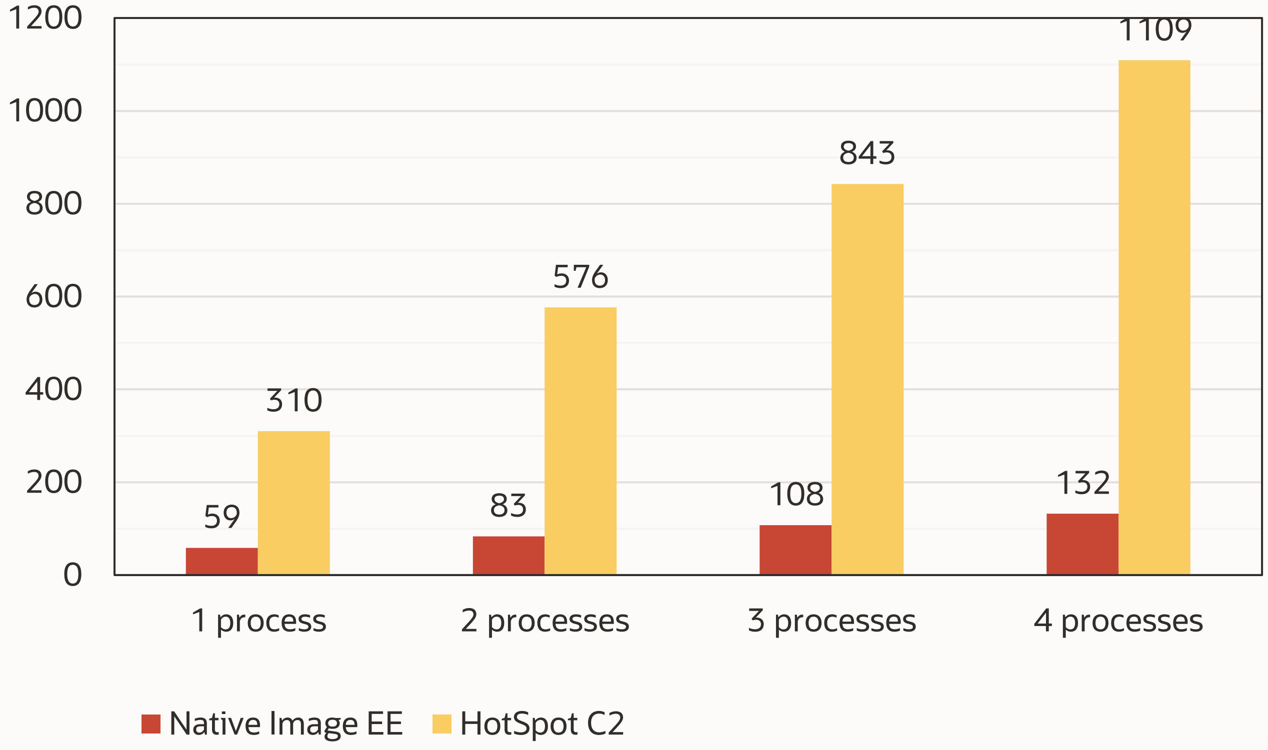
Example: GraalVM JavaScript engine

- Written entirely in Java, but better startup time and memory footprint than V8
- Context pre-initialization
  - At image build time, all data structures for a JavaScript execution context are built
  - At run time, only a bit of patching and configuration is necessary



Example: horizontal scaling of microservices

Memory Usage in MByte
Quarkus Apache Tika ODT in a "tiny" configuration and with the serial GC
(1 CPU core per process, -Xms32m -Xmx128m) – JDK 11



Java HotSpot VM

- **4 VM instances = 4 times the memory**

Native Image

- **4 VM instances = 2 times the memory**
- Image heap shared between processes
- Machine code shared between processes

Static analysis: the good, the bad, and the ugly

The good
- It works to find a reasonably small closed world for real-world applications
- Tightly integrating static analysis with the compiler allows re-use optimization phases before static analysis

The bad
- Precision is not great regarding types that can reach a variable or field
- Attempts to use context-sensitivity have so far not been useful
  - Any useful improvement of precision requires a very deep context
  - Java has deep call chains and deep object structures. For example, just look at `java.util.HashMap`
  - Java arrays have no type information: every array can be cast to `Object[]`

The ugly
- Reflection, JNI, `Unsafe`, … need configuration at image build time
  - We see over-registration of methods to make configuration easier
- Reflection passes arguments in `Object[]` array
- About every JDK method can call `String.format` which has huge reachability
- Solution: avoid reflection, see for example the new reflection-free JSON framework from Micronaut

Benefits of a closed world: security

Closed-world has security benefits

- Implicit fix for Object deserialization vulnerabilities
- “Software bill of material” (SBOM): audit all code that can possibly execute
  - Check for vulnerable library versions, disallowed crypto algorithms, …
- No need to rely on `SecurityManager` for “untrusted” code

Snapshotting can lead to security problems

- Leaking of build-time information into the executable
  - Username, working directory, passwords, IP address, server name, …
  - The native image generator has some checks in place, but it cannot provide guarantees
- Application code runs at image build time
  - Malicious code can take over the build infrastructure
- JDK / library security updates (like quarterly CPU releases) require rebuild of images

Benefits of a closed-world AOT compilation: predictable performance

No “deoptimization”

- No performance cliff when reverting from highly optimized to unoptimized code
- Fast and predictable performance already for the first request

Indirect method calls are simple and always constant time

- `invokevirtual` and `invokeinterface` are always vtable calls with fixed vtable index

Dynamic type checks are simple and always constant time

Larger code transformations at build time without worrying about deoptimization (= interpreter state)

- Outlining of allocations
- String concatenation: replace StringBuilder with pre-sized concatenation methods
- Optimization of String.format(): pre-parsing of format strings
- String inlining: combine the String object and the byte[] array to a single hybrid object

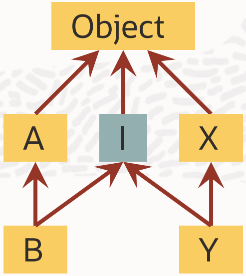
Example: type checks

- Type checks can be seen as a binary matrix of types
  - “1” === “assignable”
- A binary matrix has the Consecutive Ones Property (C1P) when there is a permutation of its rows that leaves the 1's consecutive in every column.
  - “Consecutive Ones” === “Range Check”
- Precompute the minimum number of sub-matrices that all fulfill C1P
- Number of matrices === number of type-check slots in each type
  - Every type has a typeID for each slot

The actual snippet for the lowering of all type checks:

```
short typeCheckStart = type.typeCheckStart;
short typeCheckRange = type.typeCheckRange;
short typeCheckSlot = type.typeCheckSlot;
short checkedTypeID = checkedType.typeIDs[typeCheckSlot];
return checkedTypeID - typeCheckStart < typeCheckRange;
```

- 1 memory read when type is a compile-time constant (`instanceof` bytecode)
- 4 memory reads when type is not a compile-time constant (`Class.isInstance`, `Class.isAssignableFrom`)



	O	A	B	X	Y	I
O	1	1	1	1	1	1
A	0	1	1	0	0	0
B	0	0	1	0	0	0
X	0	0	0	1	1	0
Y	0	0	0	0	1	0
I	0	0	1	0	1	1

Slot 0:

	O	A	B	X	Y	I
O	1	1	1	1	1	1
A	0	1	1	0	0	0
B	0	0	1	0	0	0
X	0	0	0	1	1	0
Y	0	0	0	0	1	0

Slot 1:

	O	A	B	Y	I	X
I	0	0	1	1	1	0

Example: allocation outlining, StringBuilder optimization

Exception allocations

```java
public static <T> T requireNonNull(T obj) {
  if (obj == null)
    throw new NullPointerException();
  return obj;
}
```

- Smaller code
- Performance neutral

```java
public static <T> T requireNonNull(T obj) {
  if (obj == null)
    throw createAndThrowNullPointerException();
  return obj;
}

// Shared by all places that throw a NullPointerException
NeverReturningMethod createAndThrowNullPointerException() {
  throw new NullPointerException();
}
```

String concatenation

```java
Object object = …
int number = …
String s = "literal" + object + number;
```

- Smaller code
- Better performance
  - No copying of data array
  - No incremental array size increases
  - Like invokedynamic-based string concatenation

```java
String objectStr = String.valueOf(object);
String s = concat_S_S_I("literal", objectStr, number)

// Shared by all places that concatenate the same types
String concat_S_S_I(String s1, String s2, int i3) {
  int len = s1.length + s2.length + intLen(i3);
  byte[] data = new byte[len];
  int pos = 0;
  pos = copy(data, pos, s1);
  pos = copy(data, pos, s2);
  pos = copy(data, pos, i3);
  return new String(data, false); // non-copying constructor
}
```

Example: String inlining

Combine the `java.lang.String` object and the `byte[]` array

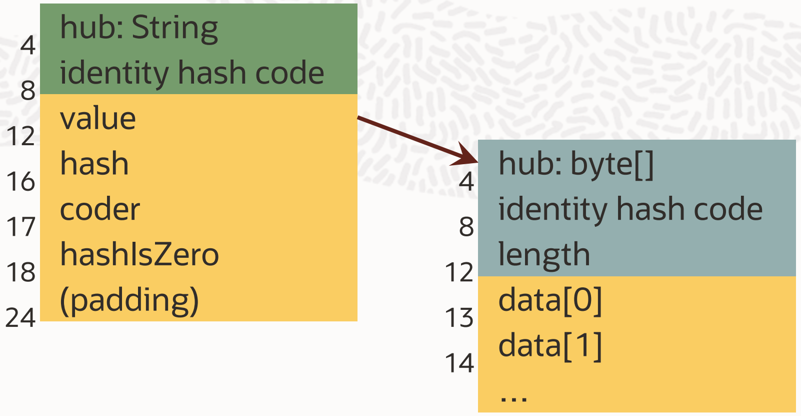
- Reduce memory footprint
- Improve cache behavior

Single “hybrid” object that has fields from `String`, and array parts

- Loading the field `String.value` is a no-op
- Access of array elements use larger array base offset

Real-world complications

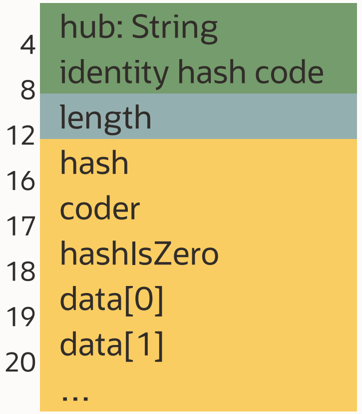
- Access of `String.value` via reflection: disable optimization at image build time
- Access of `String.value` via JNI (yes, of course the JDK C code is doing that): change the JNI functions for array access



hub: String
4
identity hash code
8
value
12
hash
16
coder
17
hashIsZero
18
(padding)
24

hub: byte[]
4
identity hash code
8
length
12
data[0]
13
data[1]
14
…

String with 6 ASCII characters: **48 byte**
String object: 24 byte
byte[] array: 24 byte



hub: String
4
identity hash code
8
length
12
hash
16
coder
17
hashIsZero
18
data[0]
19
data[1]
20
…

Inlined string with 6 ASCII characters: **24 byte**

Challenges of a closed-world AOT compilation

Precise exception semantics

- Java specification requires exact exception class at the exact place
- Cannot hoist null checks / bounds checks out of loops
  - Speculative guard movement phase used for JIT compilation does no work
- Solution: loop duplication phase, explicit loop invariant code motion phase

Must compile for lowest common denominator of CPU features

- Intel: SSE2 (maybe AVX2 soon)
- Solution: loop duplication phase, method duplication

Size of the AOT compiled code

- Especially with aggressive method inlining, code size can explode
- Solution: profile-guided optimizations to compile hot code for performance, cold code for size

No profile information from current execution

- Profile-guided optimization with either instrumentation-based or sampling-based profiling
- Applying profiles requires re-build of native image

Persistent heaps

Basic idea: “extend” the image heap at run time

Example: persist configuration for fast application startup
- Keeping an app continuously provisioned just for occasional short queries wastes resources
- Native Image loads code fast – but what about data?
- The image heap cannot be updated without a rebuild — minutes of compute

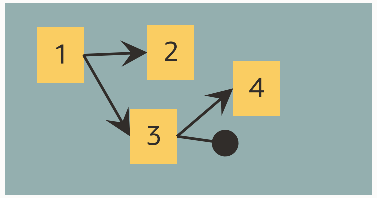
Example: persist long-lived and slowly evolving caches
- Persist cache on shutdown of application

Need to be able to load pre-populated parts of the heap quickly and efficiently
- Deserialization is much too slow and inefficient
- Small updates also necessitate fast, incremental saves
- Copy-on-write sharing like image heap

Example: creating a persistent heap

New process with just image heap

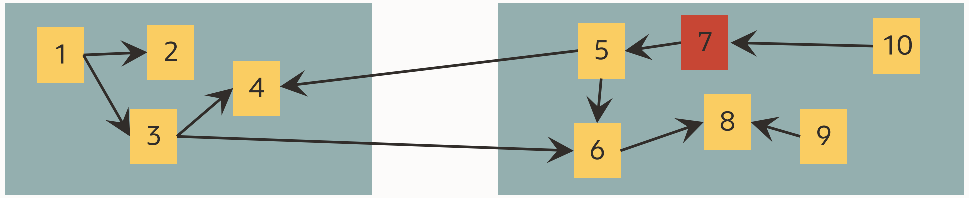
Image heap



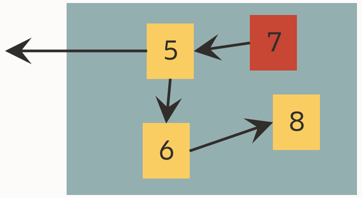
Execution modifies image heap and run time heap

Image heap

Run-time heap



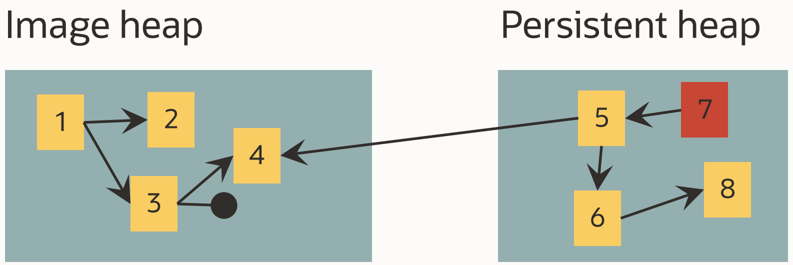
Persisted heap



🟥 Root for persistent heap

Example: loading a persistent heap

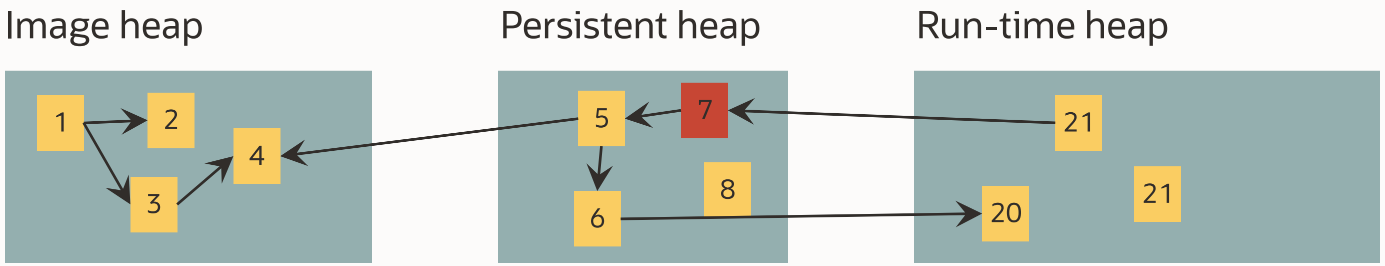
New process with loaded persistent heap



Reference from snapshot to image heap (5 to 4) is preserved

Reference from image to snapshot heap (3 to 6) is NOT preserved

Execution modifies image heap, snapshot heap, and run-time heap



When do you load your microservice configuration?

```
public static void main(String[] args) {
  Configuration config = Configuration.loadFromFile();
  System.out.println(config.handler.handle());
}
```

Load configuration at run time

11.4 million instructions

Instruction counts are for parsing a 1-line JSON file using Jackson, to build a configuration that prints “Hello, World”

```
class ConfigureAtBuildTimeFeature implements Feature {
  public void beforeAnalysis(BeforeAnalysisAccess access) {
    // This code runs at image build time
    ImageSingletons.add(Configuration.class, Configuration.loadFromFile());
  }
}

public static void main(String[] args) {
  Configuration config = ImageSingletons.lookup(Configuration.class);
  System.out.println(config.handler.handle());
}
```

Load configuration at image build time

1.0 million instructions

This is the same instruction count as printing “Hello, World” directly without any configuration.

```
public static void main(String[] args) {
  File heapFile = ...
  if (!heapFile.exists()) {
    config = Configuration.loadFromFile();
    persistHeap(config, heapFile);
  } else {
    Heap heap = Heap.map(heapFile);
    config = heap.lookup(Configuration.class);
  }
  System.out.println(config.handler.handle());
}
```

Load at run time and cache in persistent heap

First run (load file and persistent heap): 23 million instructions

All other runs (load persistent heap): 1.1 million instructions

Summary

Introduce and explain the programming model of GraalVM Native Image

- Application initialization at build time is a new programming model for Java where applications have explicit control over snapshot building and static analysis
- Ahead-of-time compilation is not part of the programming model, only an implementation detail

Convince you that this is a worthwhile programming model for Java

- The GraalVM language implementations are a large-scale case study showcasing this model
- Microservice frameworks like Micronaut and Quarkus are using it already

Show where the closed-world assumption helps and where it hurts

- Predictable performance
- Good peak performance with profile-guided optimizations
- Image size is a big concern

Thank you

https://www.graalvm.org