COMMUNITY
THE ASF CONFERENCE
CODE

Implement automatic observability of Tomcat applications under GraalVM static compilation

Zihao RAO, Alibaba Cloud

CONTENTS

1. Background
2. Solution
3. Demonstration
4. Future works

01

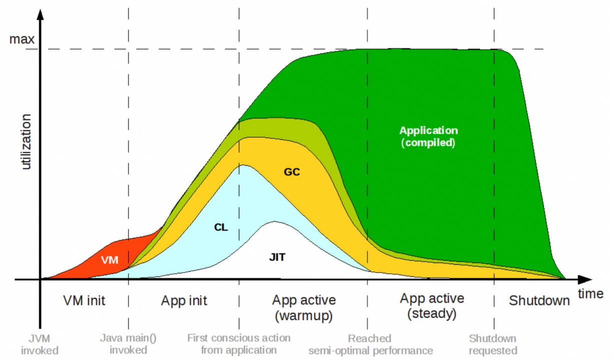
Background

Challenges for modern Java applications

Slow startup

High memory overhead

Lifecycle of Java applications: VM init, App init, warmup, App active and shutdown:



Lifecycle of Java apps

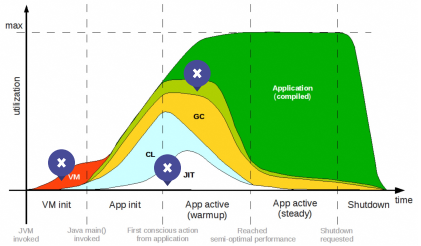
Picture by: https://shipilev.net/talks/j1-Oct2011-21682-benchmarking.pdf

Introduction of GraalVM native image

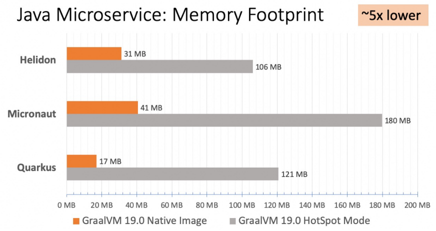
Compared to JVM-based environments, GraalVM offers the following advantages

Enhanced startup speed: By eliminating VM init, JIT, and interpretation overhead, the startup time is significantly reduced

Reduced memory overhead: By removing the memory footprint associated with the VM and applying numerous optimizations, memory usage is significantly reduced



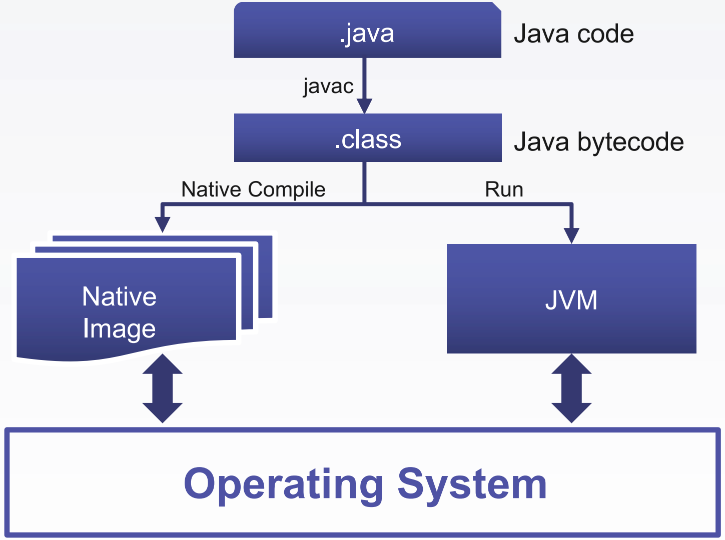
Lifecycle of Java apps under GraalVM



Improvements of different frameworks

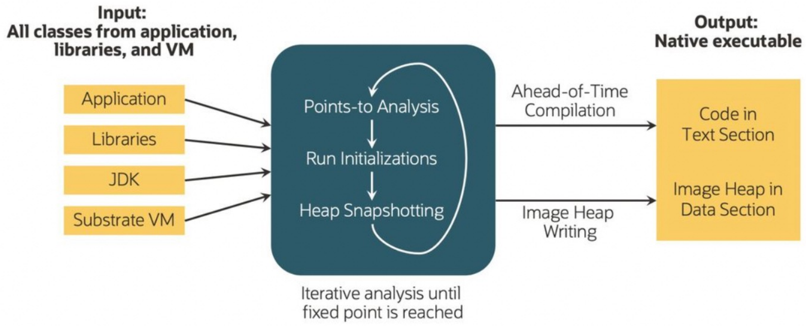
Picture by: https://medium.com/graalvm/lightweight-cloud-native-java-applications-35d56bc45673

GraalVM native image compilation process



Comparation of JVM and native compilation

The process of native compile:



Process of native compilation

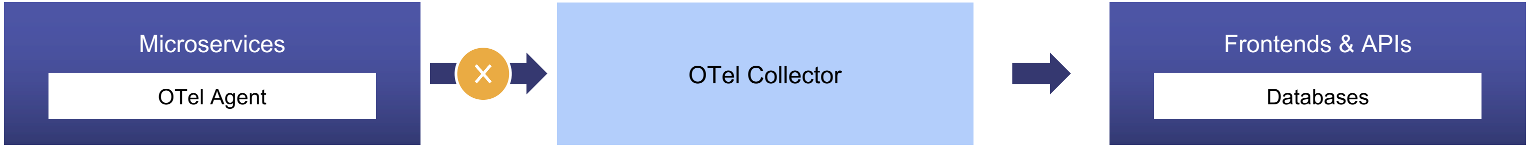
Picture by: https://www.infoq.com/articles/native-java-graalvm/

Impacts of GraalVM on the Java Ecosystem

Dynamic Features: Dynamic class loading, reflection, dynamic proxies, JNI, and serialization are no longer fully supported

Platform Independence: Without the JVM and bytecode, the platform independence that is a hallmark of the Java platform is no longer available

Ecosystem Tools: The original Java ecosystem tools for monitoring, debugging, and Java Agents are ineffective without the JVM and bytecode



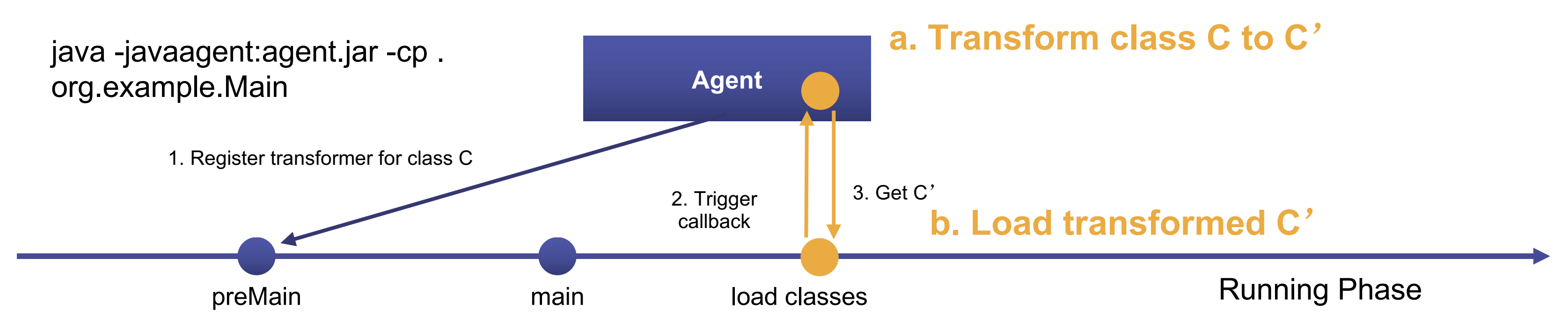
Impact of GraalVM in observability

02

Solution

Idea to instrument under GraalVM

Java Agent work process:



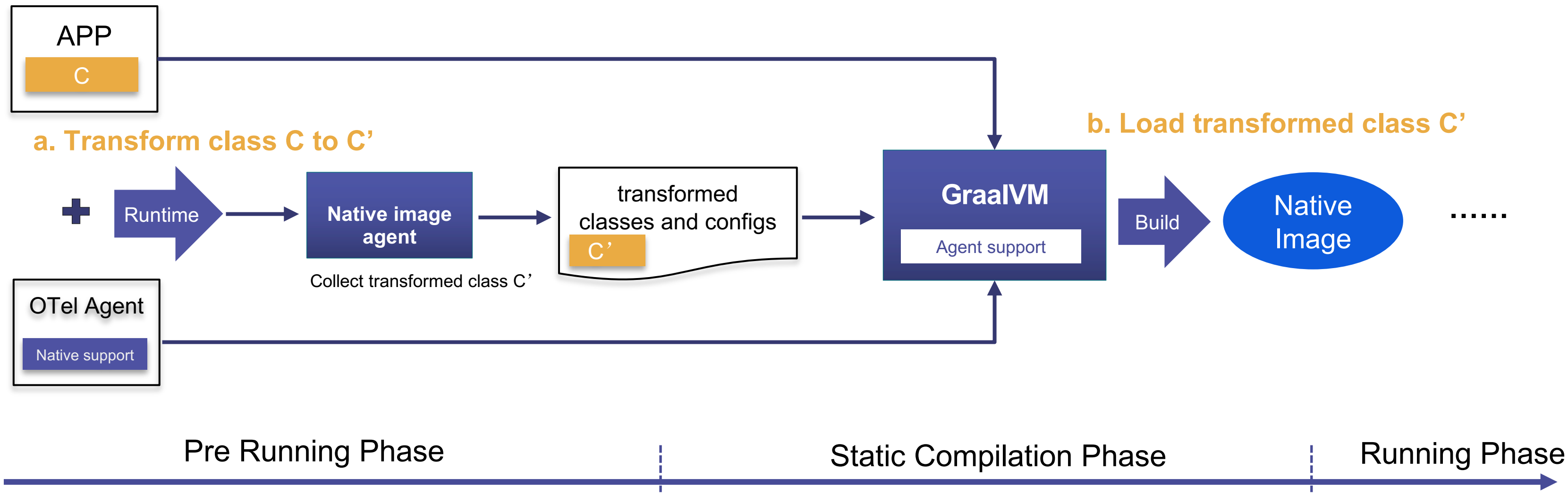
With GraalVM, bytecode is no longer used. Therefore, we aim to perform these enhancements during compilation:

a. How to transform target classes before runtime?

b. How to load transformed classes before runtime?

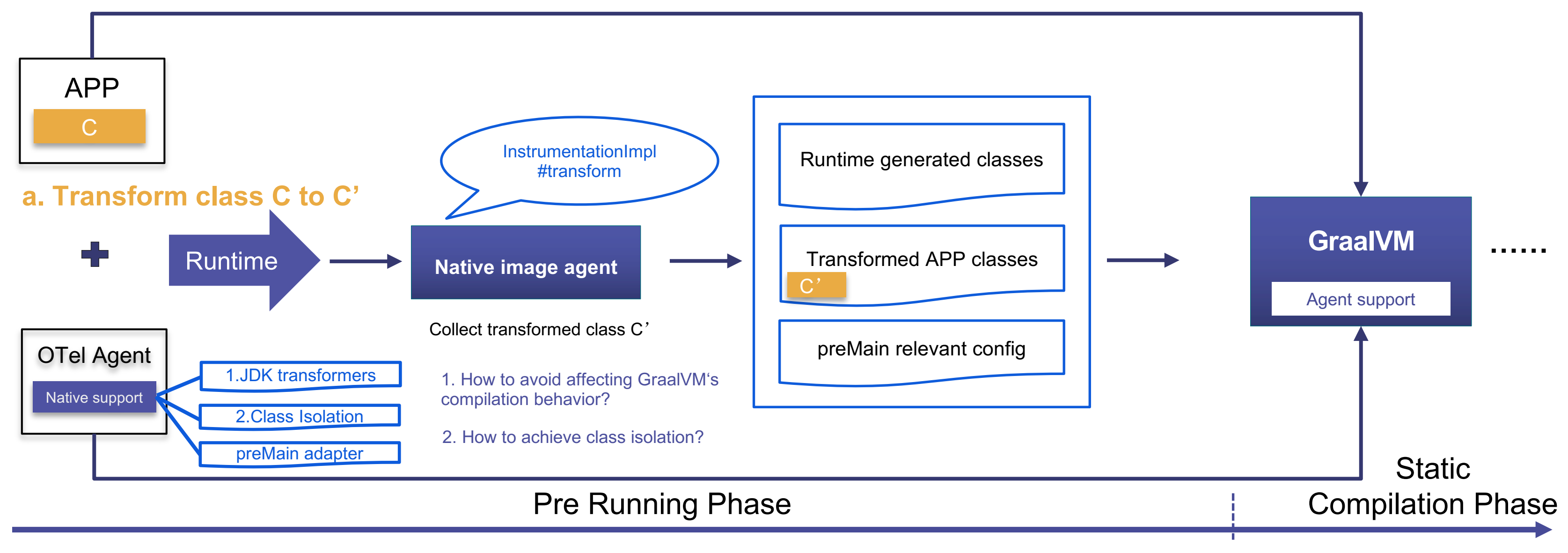
Overall design

Implemented static instrumentation before runtime:



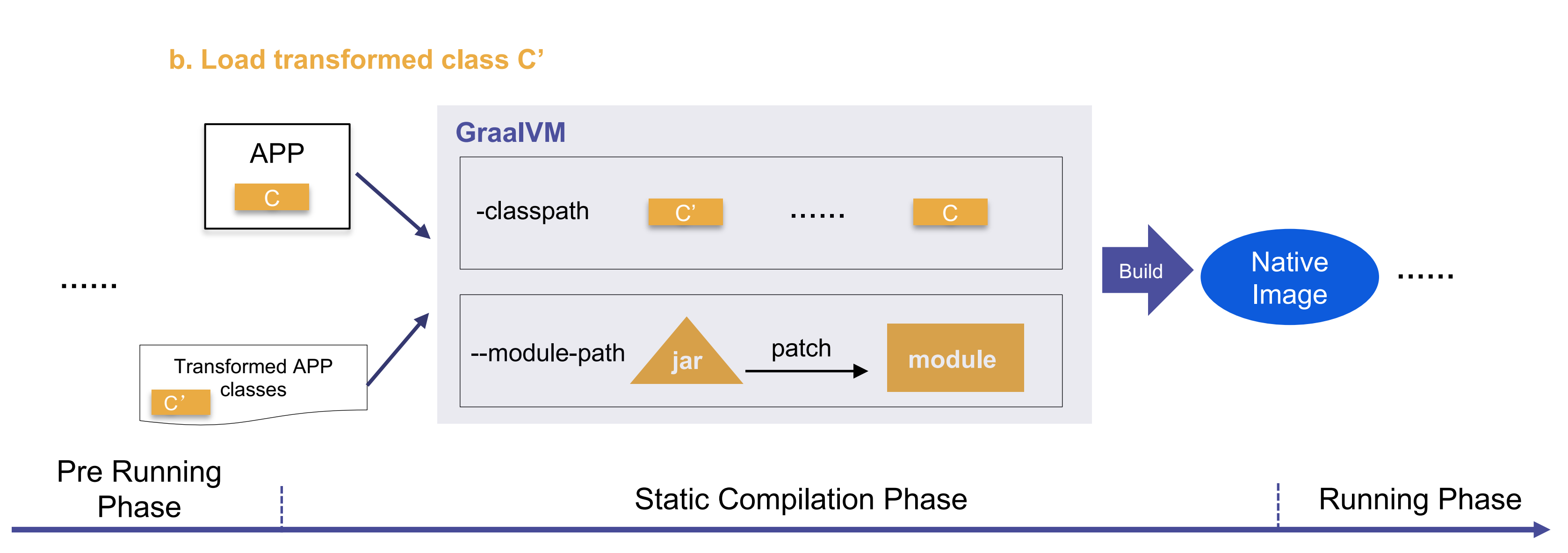
Transform and record classes

Implemented an interceptor in native image agent to collect transformed classes:



How to apply Transformed Classes

Load transformed classes by -classpath and --module-path:



03

Demonstration

Demonstration

Experimental Result

Comparison of startup speed and memory overhead: JVM vs. GraalVM native image with Java Agent

	Spring Boot	Kafka	Redis	MySQL
Startup Speed (JVM)	7.541s	11.323s	10.717s	8.116s
Memory Overhead (JVM)	402MB	408MB	420MB	394MB
Startup Speed (GraalVM)	0.117s （-98%）	0.168s （-98%）	0.152s （-98%）	0.119s （-98%）
Memory Overhead (GraalVM)	96MB （-75%）	141MB （-65%）	128MB （-69%）	107MB （-73%）

32 vCPU/64 GiB/5 Mbps

04

Future works

Future works

In the future, we plan to focus on the following aspects:

1. Conduct comprehensive test cases over multiple signals(metrics, trace, logs, and etc).
2. Consolidate the pre-running phase and the native compilation phase into a unified phase to ensure transformed classes are universally collected.

Thank you
Q&A