ORACLE

GraalVM as a Static Analysis Framework

Christian Wimmer
GraalVM Native Image Project Lead
christian.wimmer@oracle.com

Safe Harbor Statement

The following is intended to outline our general product direction. 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, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

GraalVM Native Image technology (including Substrate VM) is Early Adopter technology. It is available only under an early adopter license and remains subject to potentially significant further changes, compatibility testing and certification.

Copyright © 2020, Oracle and/or its affiliates

GraalVM™

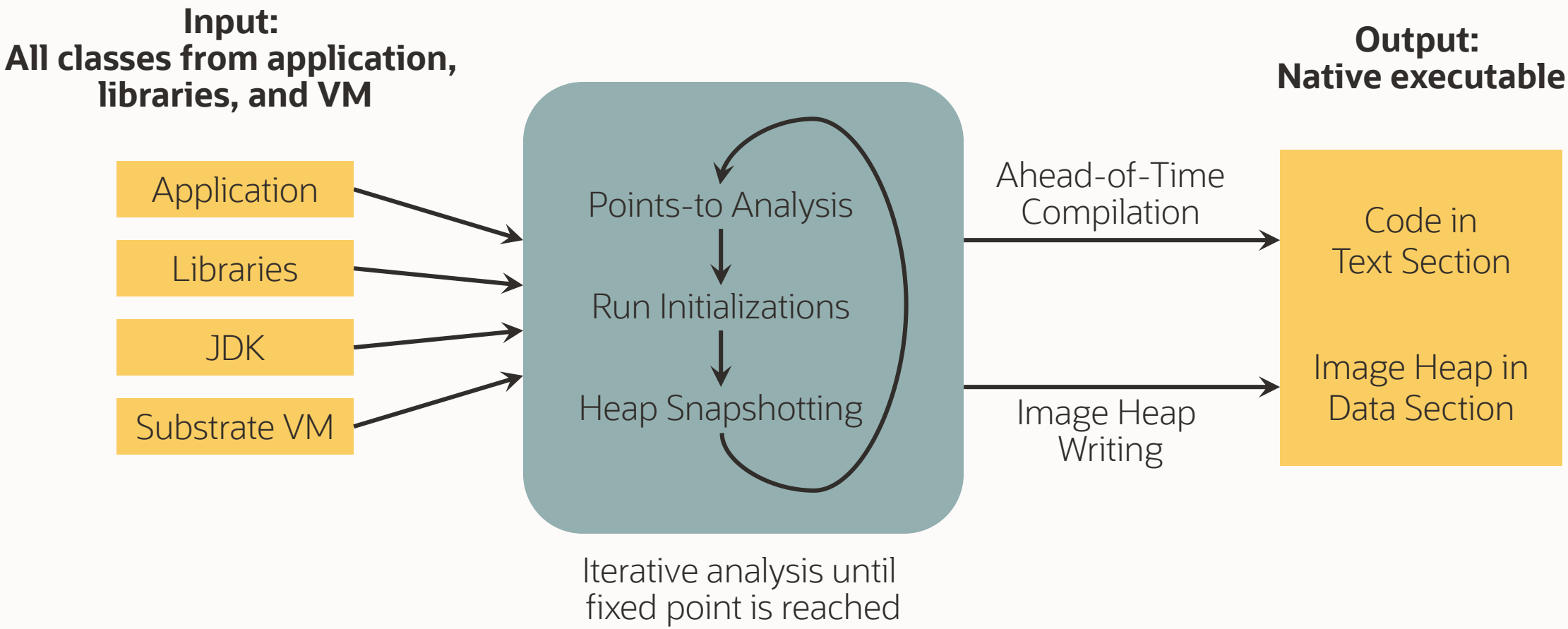
OpenJDK™

ORACLE®
Database

Native Image

Copyright © 2020, Oracle and/or its affiliates

GraalVM Native Image: Details



Copyright © 2020, Oracle and/or its affiliates

Static Analysis Frameworks

Most established frameworks are independent from a compiler. But the feature list reads like a compiler:

- Soot: https://sable.github.io/soot/
  - Jimple: a typed 3-address intermediate representation suitable for optimization
  - Shimple: an SSA variation of Jimple
  - Call-graph construction
  - Points-to analysis
- Doop: https://plast-lab.github.io/doop-pldi15-tutorial/
  - A declarative framework for static analysis, centered on pointer analysis algorithms
  - Points-to analysis implemented in Datalog
- Wala: http://wala.sourceforge.net/wiki/index.php/Main_Page
  - Interprocedural dataflow analysis (RHS solver)
  - Pointer analysis and call graph construction
  - SSA-based register-transfer language IR
  - General framework for iterative dataflow

So why not use a compiler for static analysis? Well, at least the front-end and some high-level optimizations.

Copyright © 2020, Oracle and/or its affiliates

Using the GraalVM Compiler as a Static Analysis Framework

GraalVM compiler and the hosting Java VM provide

- Class loading (parse the class file)
- Access the bytecodes of a method (via JVMCI)
- Access to the Java type hierarchy, type checks (via JVMCI)

Bytecode parsing for points-to analysis and compilation use same intermediate representation

- Simplifies using the analysis results for optimizations

Goals of points-to analysis

- Identify all methods reachable from a root method
- Identify the types assigned to each field
- Identify all instantiated types

Fixed point iteration of type flows: Types propagated from sources (allocation) to usage

Copyright © 2020, Oracle and/or its affiliates

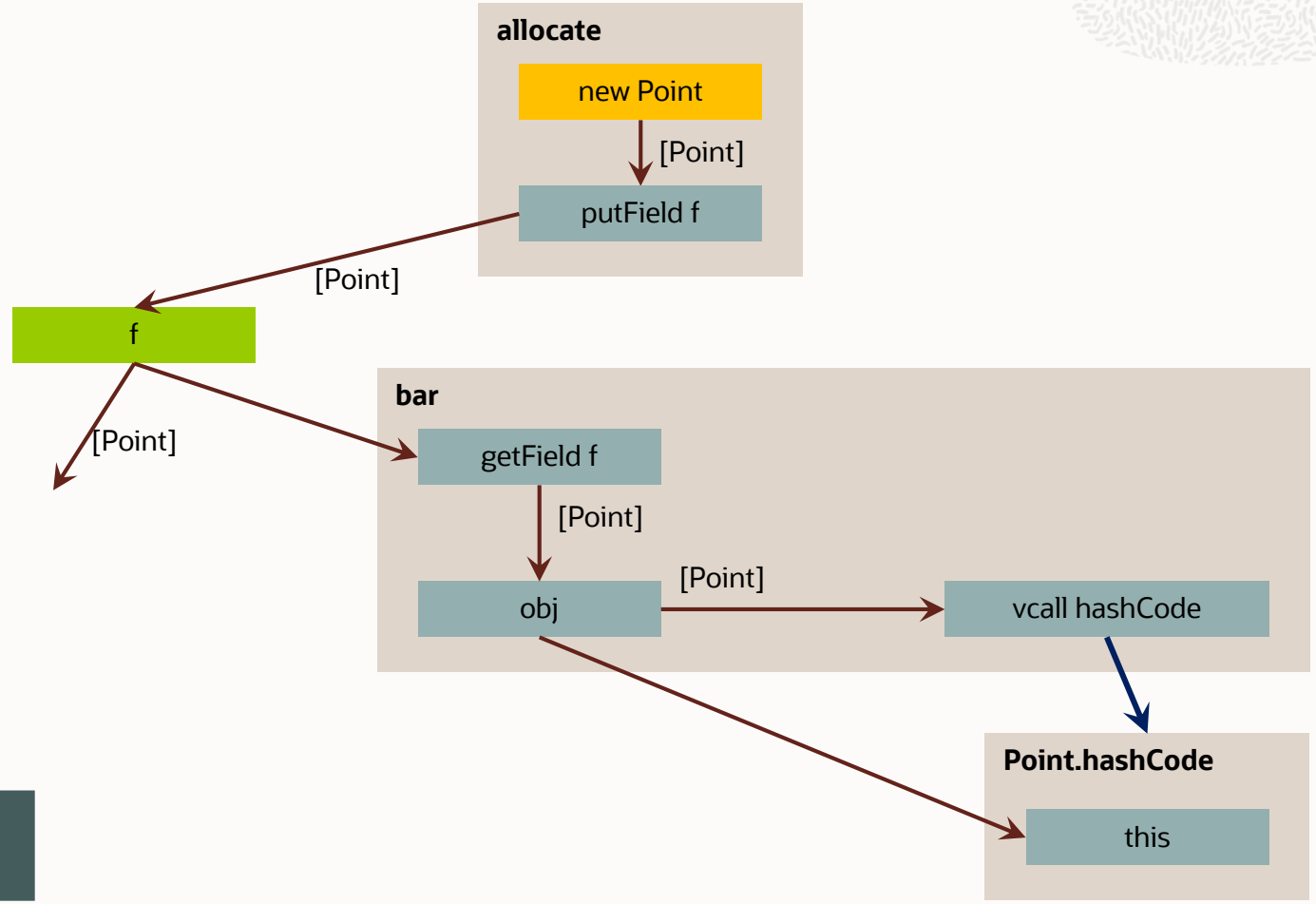
Example Type Flow Graph

```
Object f;

void foo() {
  allocate();
  bar();
}

Object allocate() {
  f = new Point()
}

int bar() {
  return f.hashCode();
}
```



Analysis is context insensitive:
One type state per field

Copyright © 2020, Oracle and/or its affiliates

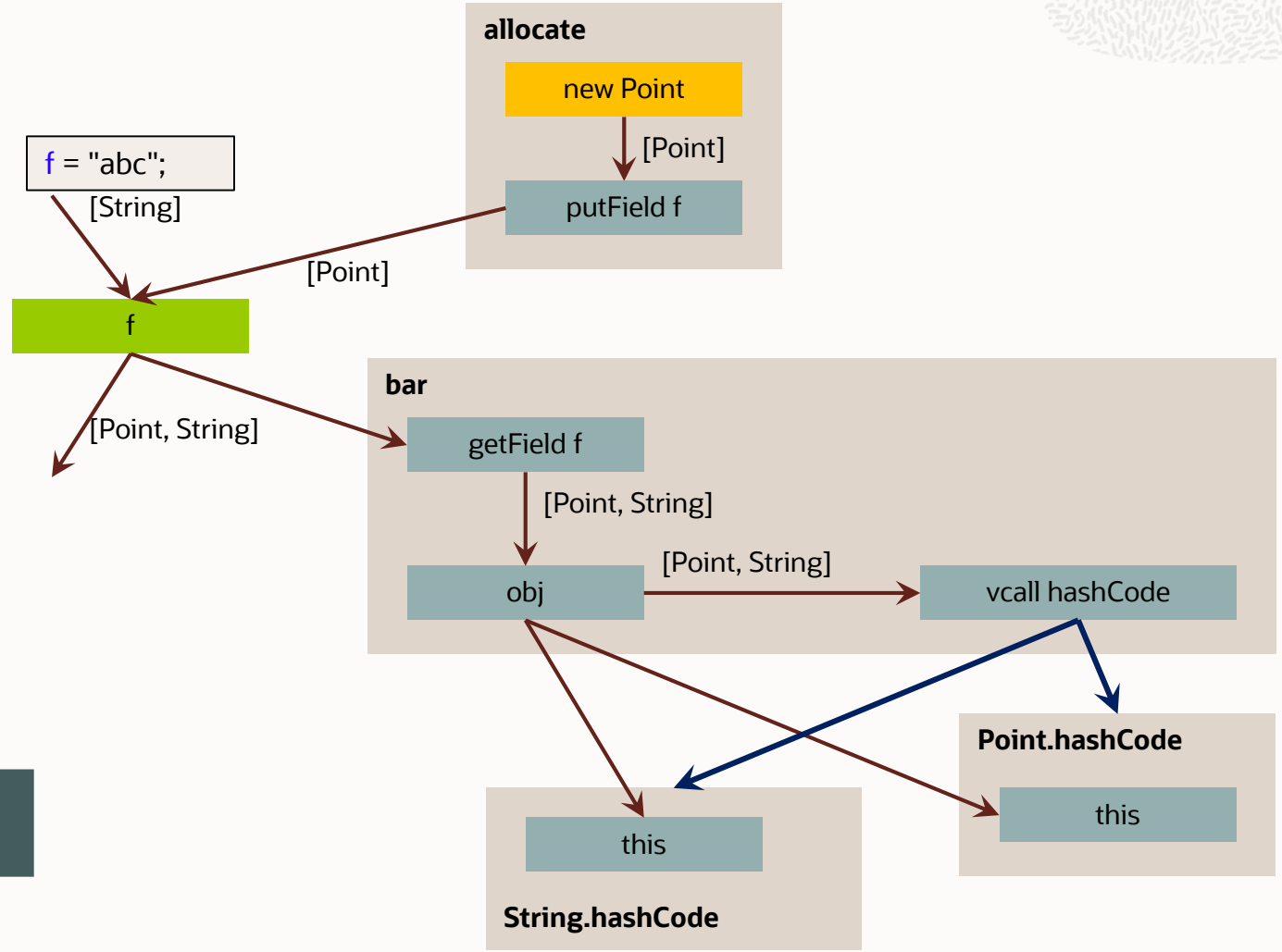
Example Type Flow Graph

```
Object f;

void foo() {
  allocate();
  bar();
}

Object allocate() {
  f = new Point()
}

int bar() {
  return f.hashCode();
}
```



Analysis is context insensitive:
One type state per field

Copyright © 2020, Oracle and/or its affiliates

Context Sensitive Analysis

So to improve analysis precision, we “just” have to make the analysis context sensitive. And there is no shortage of papers (and entire conferences) about that.

But what nobody really tells you: A realistic improvement in precision requires a very deep context

- Java has deep call chains
- Java has deep object structures. For example, just look at java.util.HashMap

Java is messy: about every call can reach JDK methods that lazily initialize global state

- About everything can call String.format which has huge reachability
- String.format initializes global state
- Cannot have context that covers all of String formatting, so you loose all precision

We believe that a context sensitive analysis is infeasible in production. At least we tried and failed.

Copyright © 2020, Oracle and/or its affiliates

Region-Based Memory Management

https://doi.org/10.1145/2754169.2754185

Safe and Efficient Hybrid Memory Management for Java

Codruţ Stancu*† Christian Wimmer* Stefan Brunthaler† Per Larsen† Michael Franz†

*Oracle Labs, USA †University of California, Irvine, USA

c.stancu@uci.edu christian.wimmer@oracle.com s.brunthaler@uci.edu perl@uci.edu franz@uci.edu

Abstract

Java uses automatic memory management, usually implemented as a garbage-collected heap. That lifts the burden of manually allocating and deallocating memory, but it can incur significant runtime overhead and increase the memory footprint of applications. We propose a hybrid memory management scheme that utilizes region-based memory management to deallocate objects automatically on region exits. Static program analysis detects allocation sites that are safe for region allocation, i.e., the static analysis proves that the objects allocated at such a site are not reachable after the region exit. A regular garbage-collected heap is used for objects that are not region allocatable.

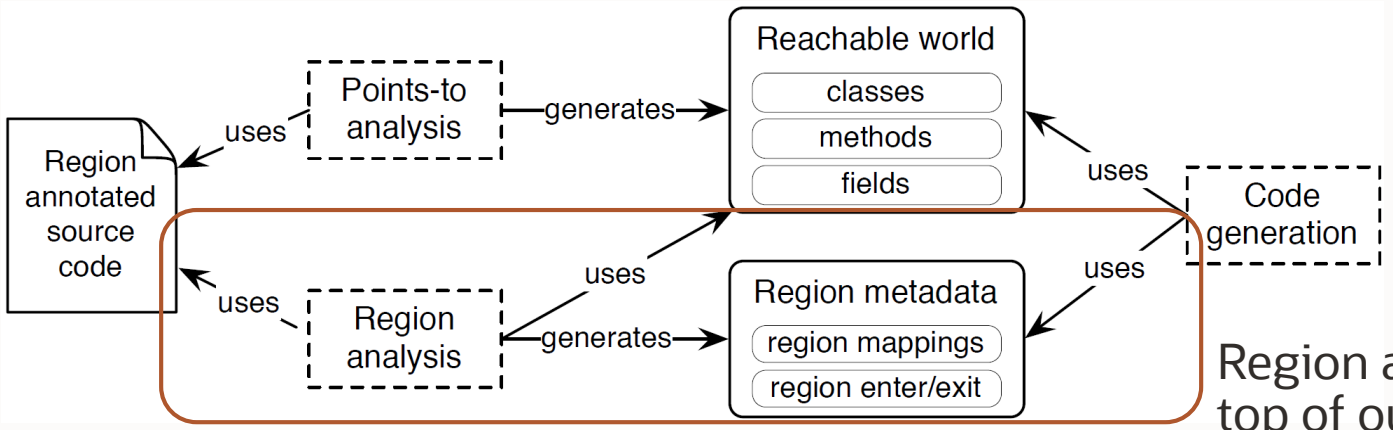
Keywords Static program analysis, region-based memory management, garbage collection

1. Introduction

Many memory intensive applications follow a regular execution pattern that can be divided into execution phases. For example, an application server responds to user requests; a database performs transactions; or a compiler applies optimization phases during compilation of a method. These applications allocate phase-local temporary memory that is used only for the duration of the phase. Such coarse-grain phases can be identified by the application developer with a

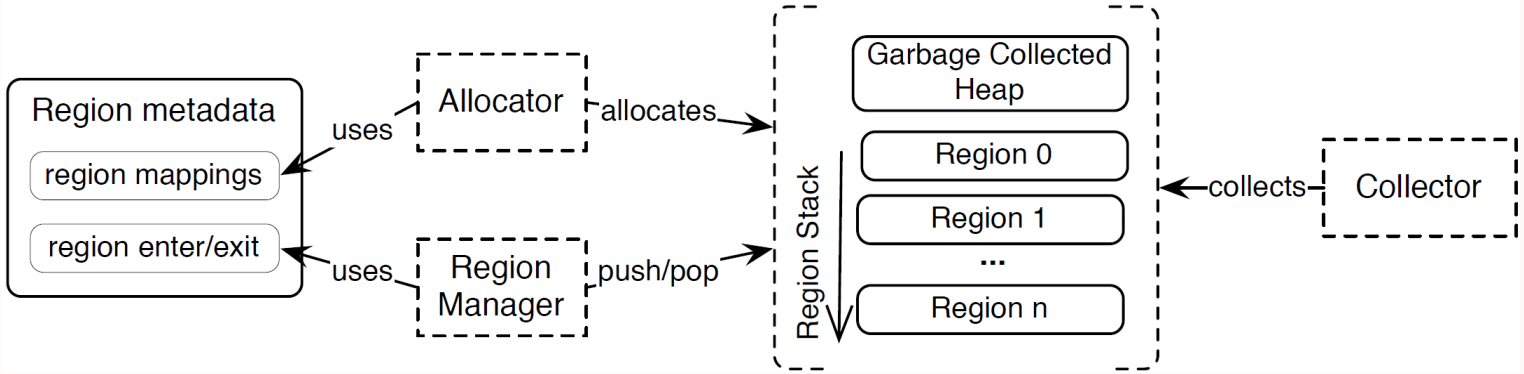
Region-Based Memory Management

Static Analysis



Region analysis is built on top of our static analysis

Runtime Data Structures



Copyright © 2020, Oracle and/or its affiliates

Semantic Models

https://doi.org/10.1145/3377555.3377885

Scalable Pointer Analysis of Data Structures using Semantic Models

Pratik Fegade
Oracle Labs and Carnegie Mellon University, USA
ppf@cs.cmu.edu

Christian Wimmer
Oracle Labs, USA
christian.wimmer@oracle.com

Abstract

Pointer analysis is widely used as a base for different kinds of static analyses and compiler optimizations. Designing a scalable pointer analysis with acceptable precision for use in production compilers is still an open question. Modern object oriented languages like Java and Scala promote abstractions and code reuse, both of which make it difficult to achieve precision. Collection data structures are an example of a pervasively used component in such languages. But analyzing collection implementations with full context sensitivity leads to prohibitively long analysis times.

We use *semantic models* to reduce the complex internal implementation of, e.g., a collection to a small and concise model. Analyzing the model with context sensitivity leads to precise results with only a modest increase in analysis

1 Introduction

Whole program pointer analysis [48] has applications in a variety of different compiler analyses and optimizations. It has been used for autoparallelization [15, 44], security analysis of applications [29], bugfinding [20], high level synthesis [46] among other applications. Significant amount of work has been done in improving the precision and/or scalability of pointer analysis [18, 28, 51, 58]. Despite this, precise pointer analysis remains expensive and often not scalable.

Repeated analysis of methods under different calling contexts dominates execution time for top-down pointer analysis [59]. Commonly used components, as well as a high degree of abstractions in the form of pointer indirections thus lead to either high analysis costs or low analysis precision.

Semantic Models

Semantic models

- Simpler to analyze
- Model API behavior
- But don't model all behavior

Internal methods skipped if only semantic models analyzed

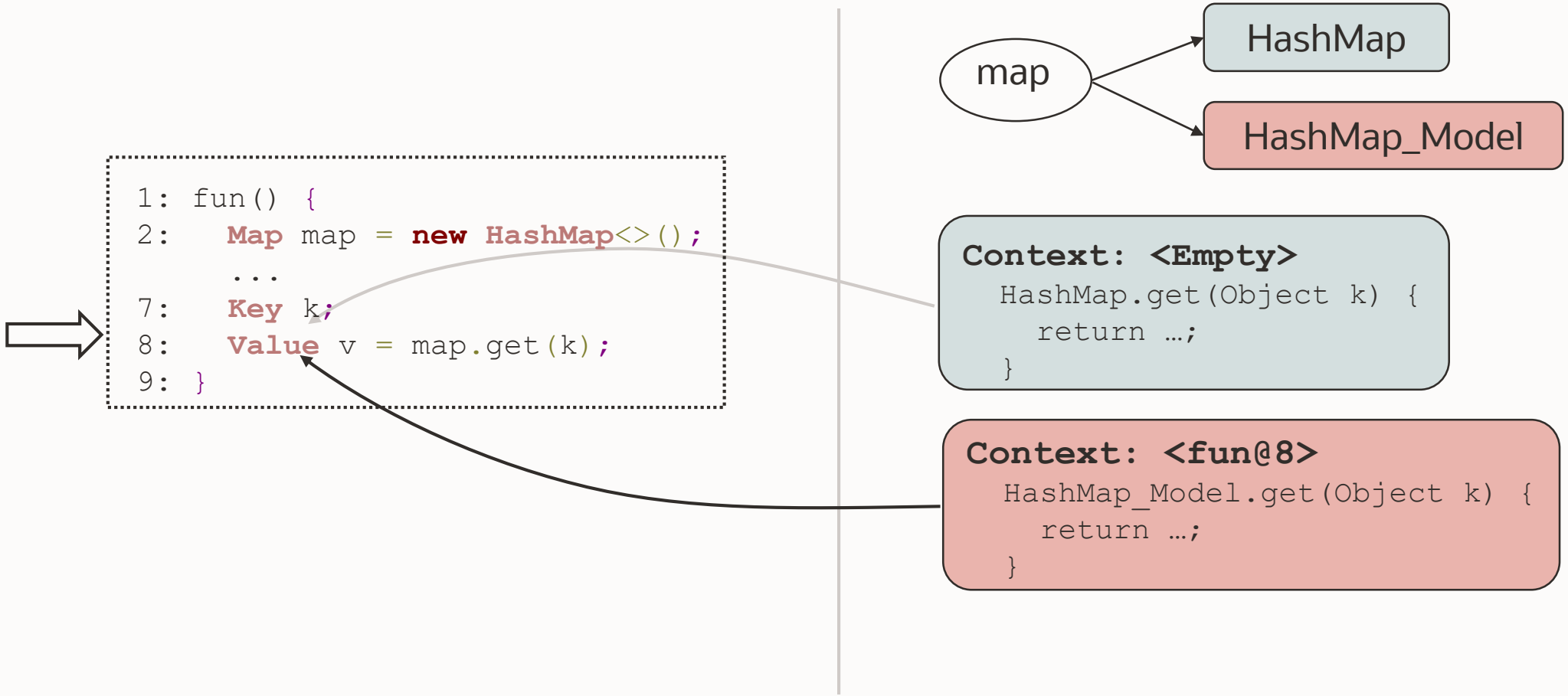
Analyze both, but with different contexts

```
V HashMap_Model.put(K k, V v) {
  this.allKeys = k;
  this.allValues = v;
  return this.allValues;
}
```

```
V HashMap.put(K k, V v) {
  Node n = new Node(k, v);
  .. // Insert node in array
}
```


Copyright © 2020, Oracle and/or its affiliates

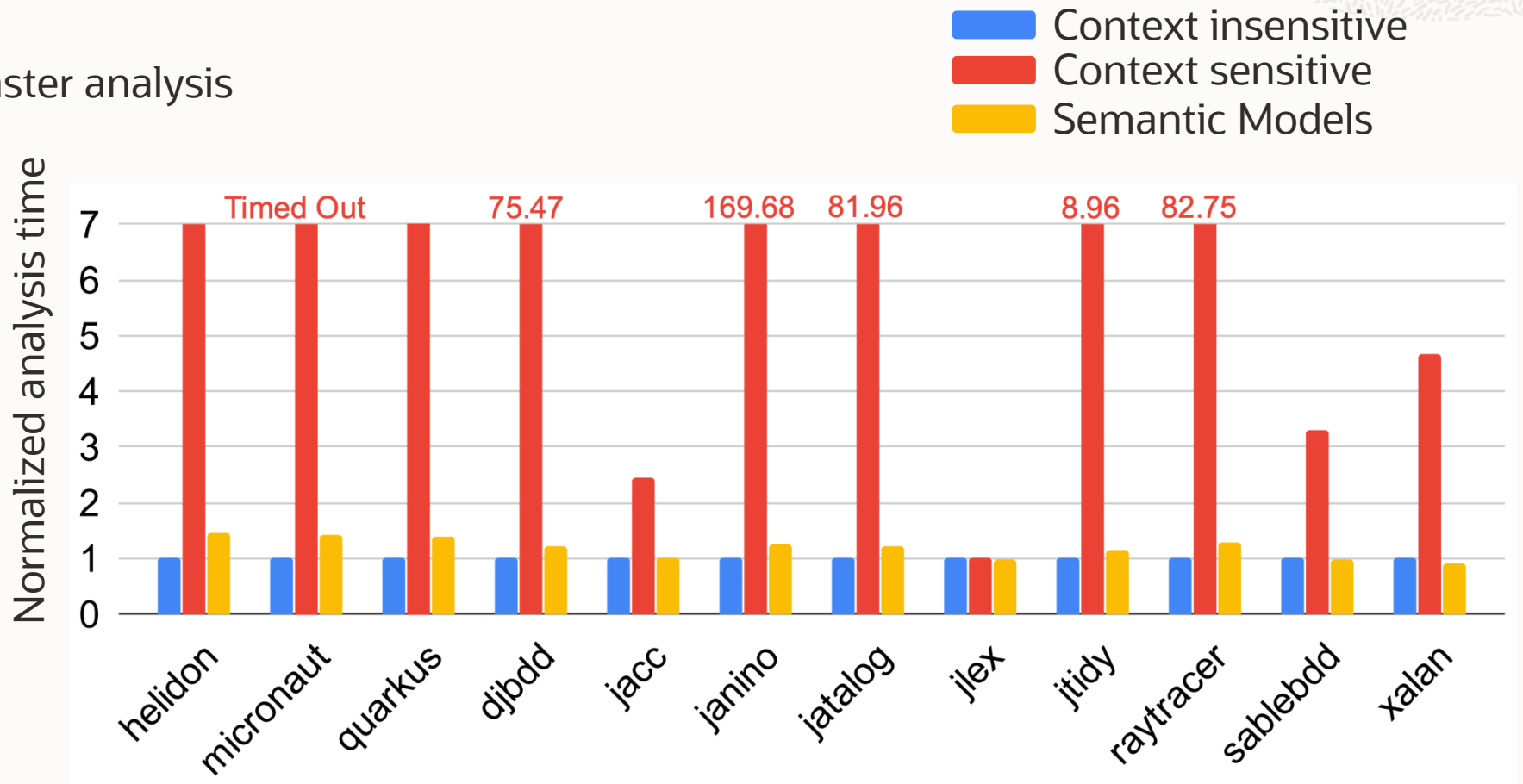
Semantic Models: Call Resolution



Copyright © 2020, Oracle and/or its affiliates

Cost: Normalized Total Analysis Time

Lower is better → faster analysis



Copyright © 2020, Oracle and/or its affiliates

Current GraalVM Static Analysis Projects

Improve static analysis memory footprint and time

- Idea: “saturate” type states with many types to the declared type
- How much will it reduce analysis precision?

Inline methods before static analysis

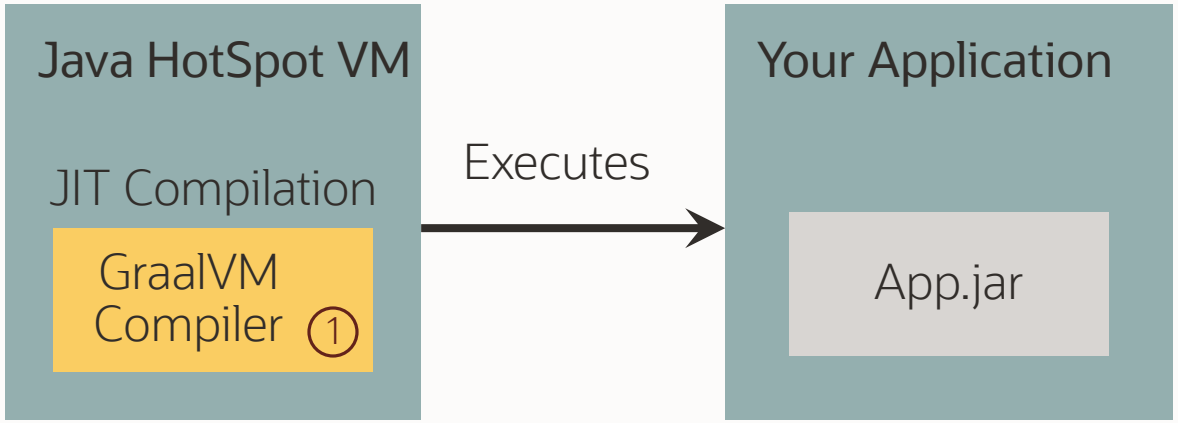
- Similar benefits (and costs) as context sensitive analysis
- But when done only for small methods, cost should not increase
  - Compiler: Inlining of small methods reduces compilation time and compiled code size

Use static analysis results to initialize classes early (at native image build time)

- When static initializer does not depend on external state
- Also interesting to find cycles in class initializers

Copyright © 2020, Oracle and/or its affiliates

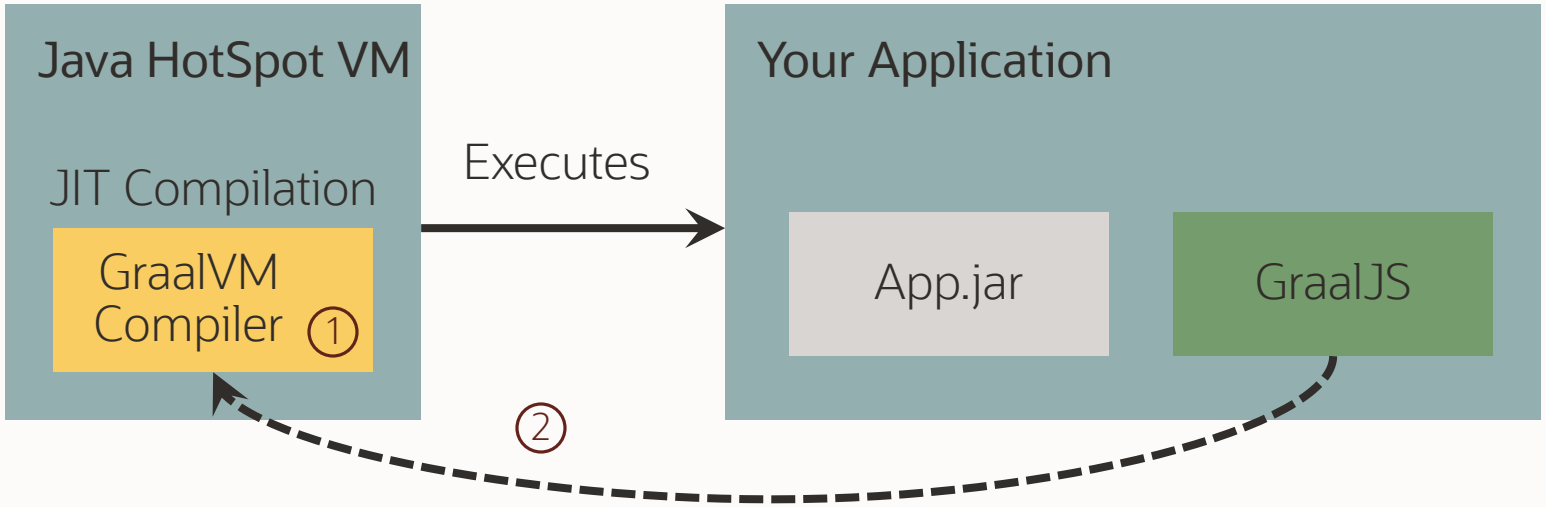
One Compiler, Many Configurations



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

Copyright © 2020, Oracle and/or its affiliates

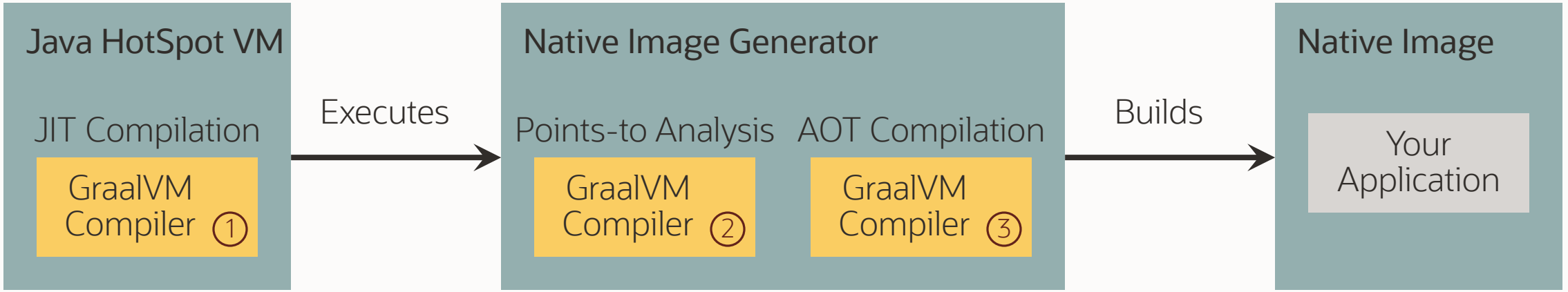
One Compiler, Many Configurations



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

Copyright © 2020, Oracle and/or its affiliates

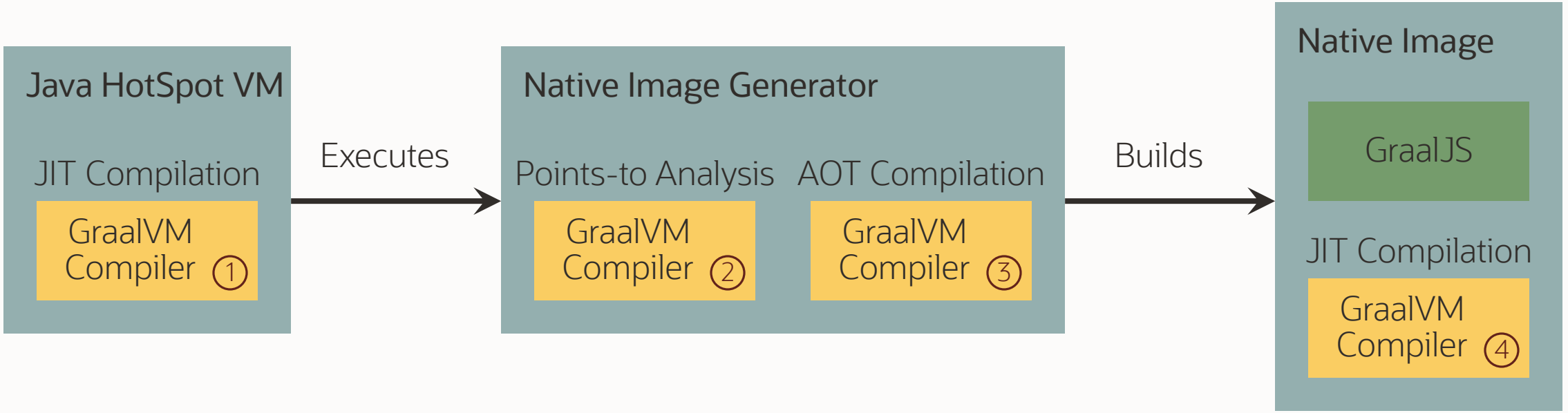
One Compiler, Many Configurations



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

Copyright © 2020, Oracle and/or its affiliates

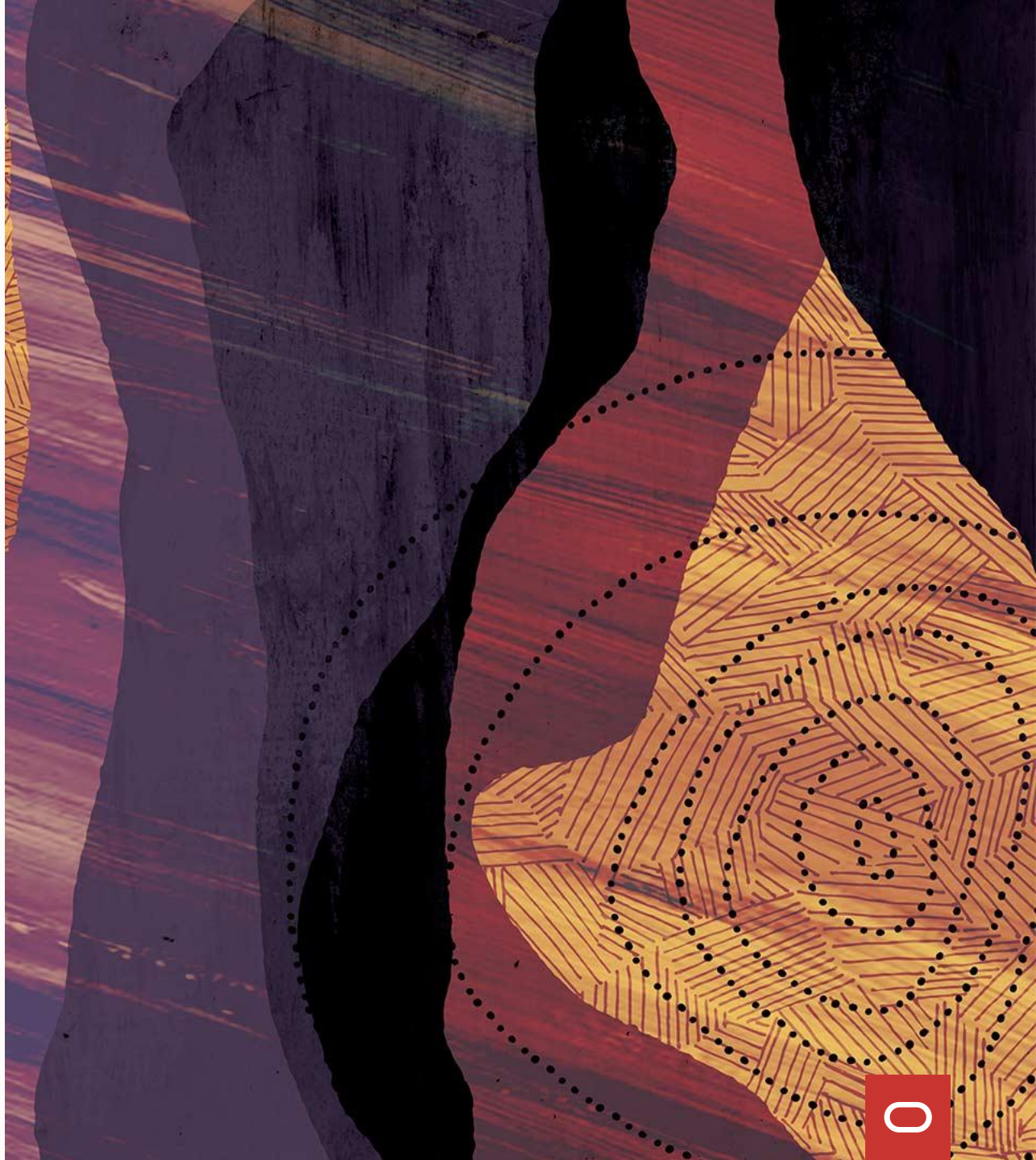
One Compiler, Many Configurations



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

Copyright © 2020, Oracle and/or its affiliates

Thank you



Copyright © 2020, Oracle and/or its affiliates

ORACLE