ORACLE

GraalVM Native Image

Large-scale static analysis for Java

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

Christian Wimmer

5+ years working on Java HotSpot VM

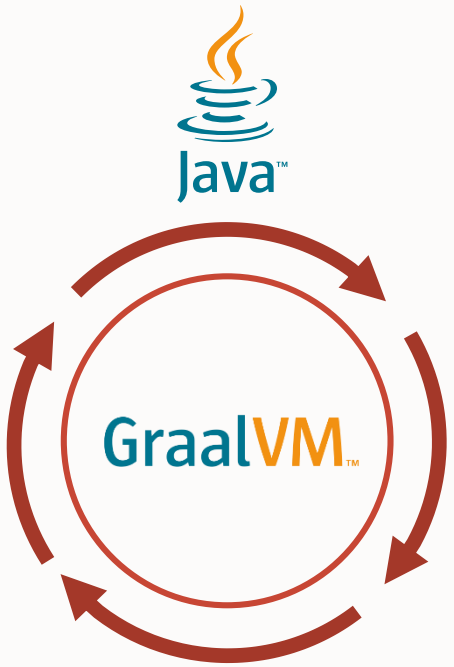
- SSA form and register allocation for the client compiler
- Research of object layout optimizations

3 years “detour” into language based security

10+ years working on GraalVM

- Native Image architect, from first commit to production

What is GraalVM?

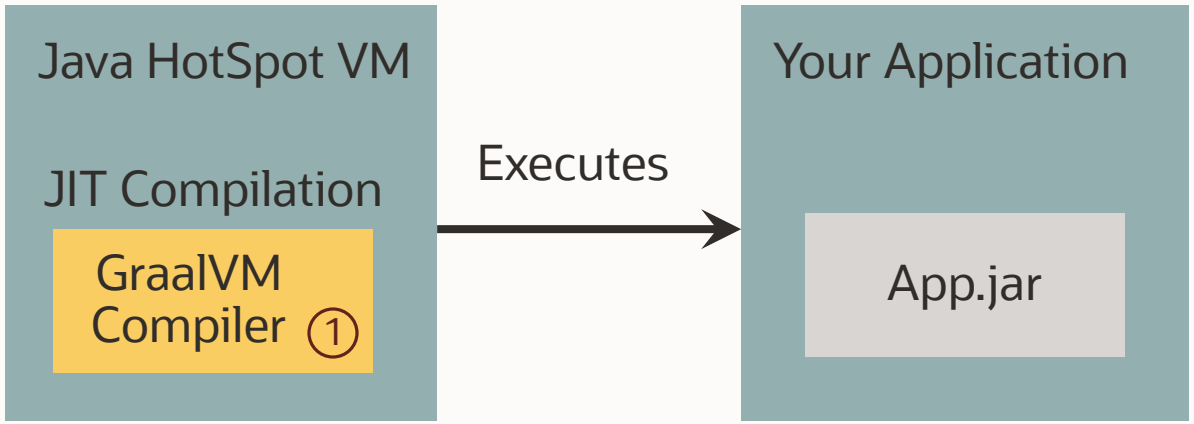


High-performance optimizing Just-in-Time (JIT) compiler

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

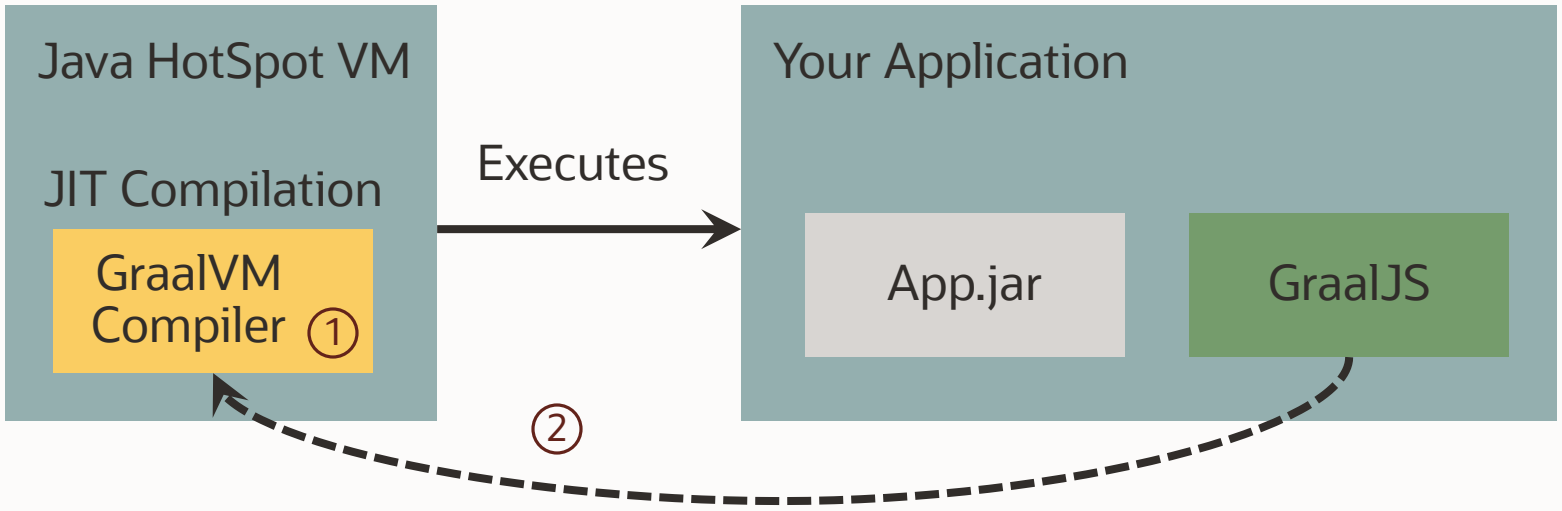
Multi-language support

One compiler, many configurations



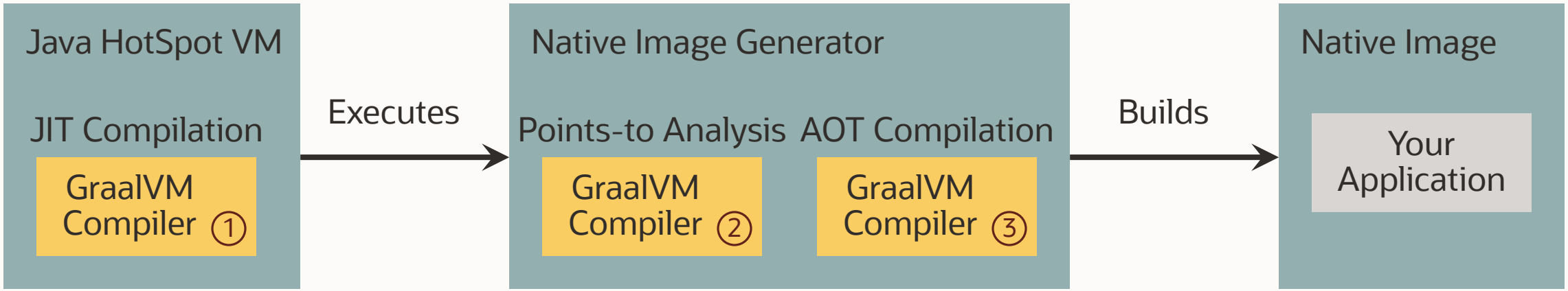
① Compiler configured for just-in-time compilation inside the Java HotSpot VM

One compiler, many configurations



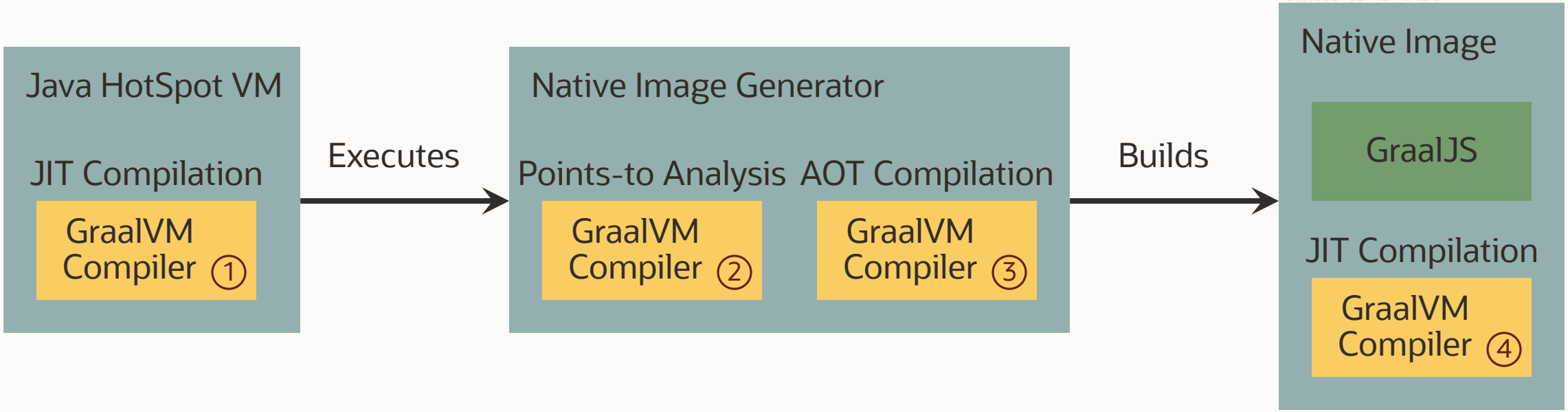
① Compiler configured for just-in-time compilation inside the Java HotSpot VM
② Compiler also used for just-in-time compilation of JavaScript code

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

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

One compiler, many configurations

https://doi.org/10.1145/2764907

Snippets: Taking the High Road to a Low Level

DOUG SIMON and CHRISTIAN WIMMER, Oracle Labs
BERNHARD URBAN, Institute for System Software, Johannes Kepler University Linz, Austria
GILLES DUBOSCQ, LUKAS STADLER, and THOMAS WÜRTHINGER, Oracle Labs

When building a compiler for a high-level language, certain intrinsic features of the language must be expressed in terms of the resulting low-level operations. Complex features are often expressed by explicitly weaving together bits of low-level IR, a process that is tedious, error prone, difficult to read, difficult to reason about, and machine dependent. In the Graal compiler for Java, we take a different approach: we use *snippets* of Java code to express semantics in a high-level, architecture-independent way. Two important restrictions make snippets feasible in practice: they are compiler specific, and they are explicitly prepared and specialized. Snippets make Graal simpler and more portable while still capable of generating machine code that can compete with other compilers of the Java HotSpot VM.

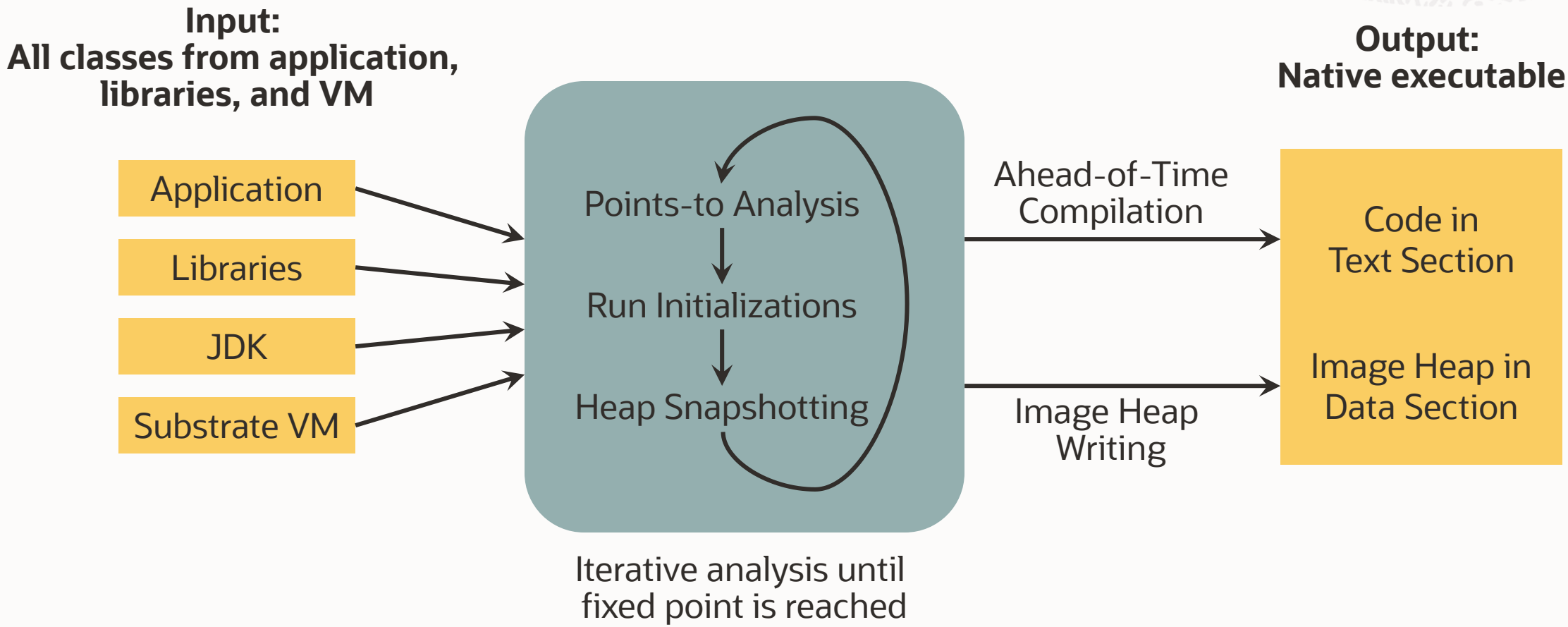
Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers, Run-time environments*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Java, Graal, snippet, compiler, dynamic compilation, just-in-time compilation

GraalVM Native Image

Native image generation



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

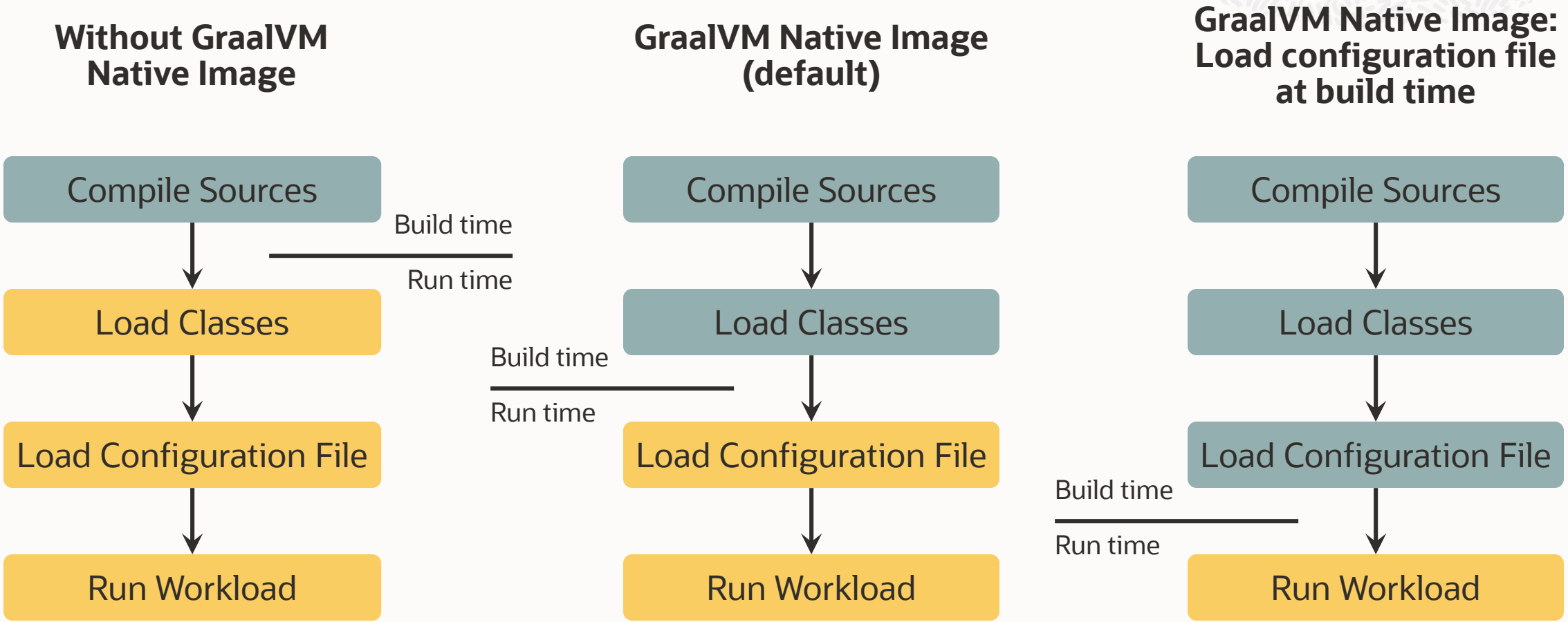
Image heap

- 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



Paper with details, examples, benchmarks

https://doi.org/10.1145/3360610

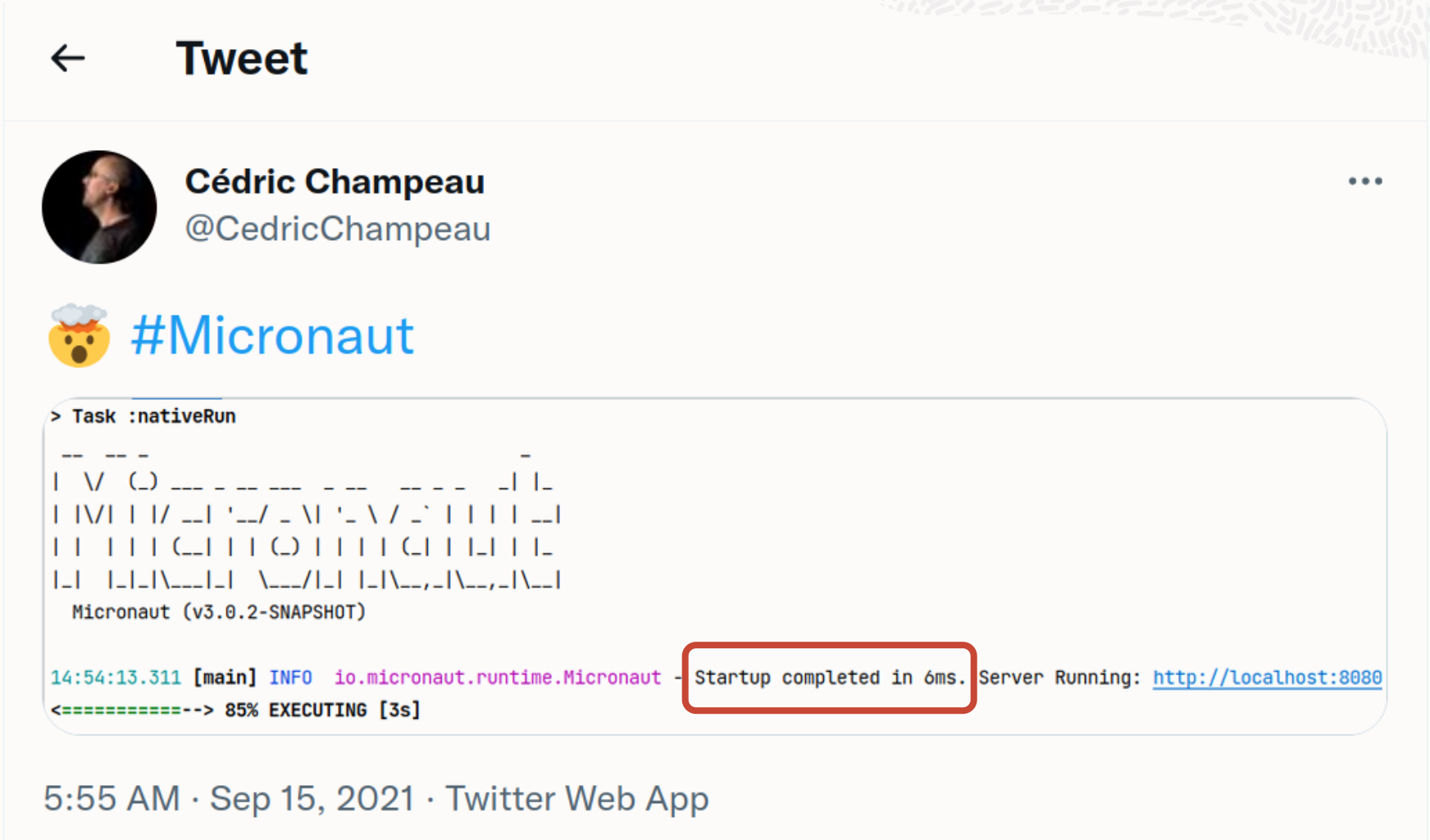
Initialize Once, Start Fast: Application Initialization at Build Time

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,

Nice theory, but does it work in practice?



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

Static analysis using the JVM compiler interface (JVMCI)

JVMCI and the hosting Java VM provide

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

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

- Simplifies using the analysis results for optimizations

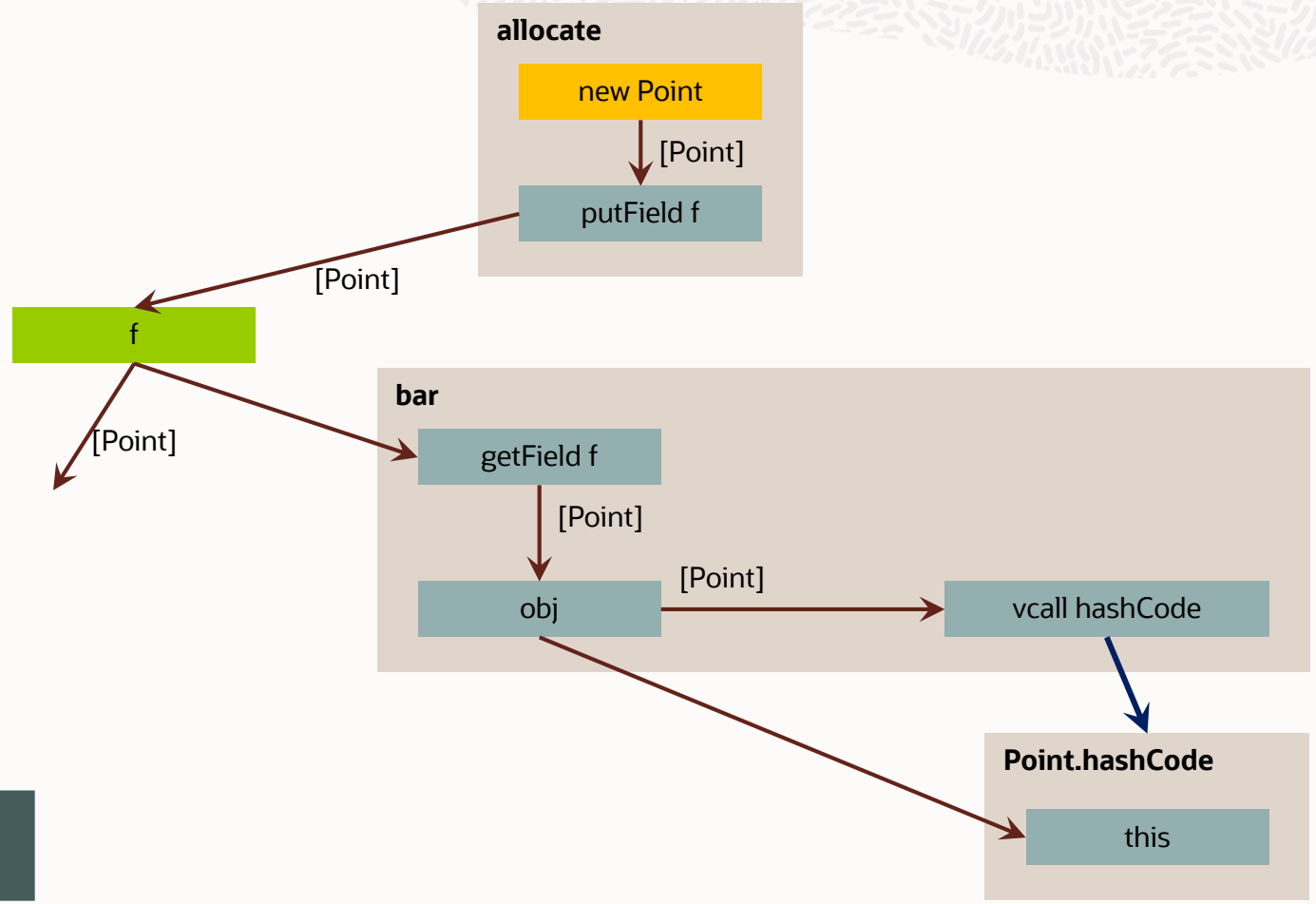
Goals of points-to analysis

- 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

Points-to analysis

```
Object f;

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

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

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

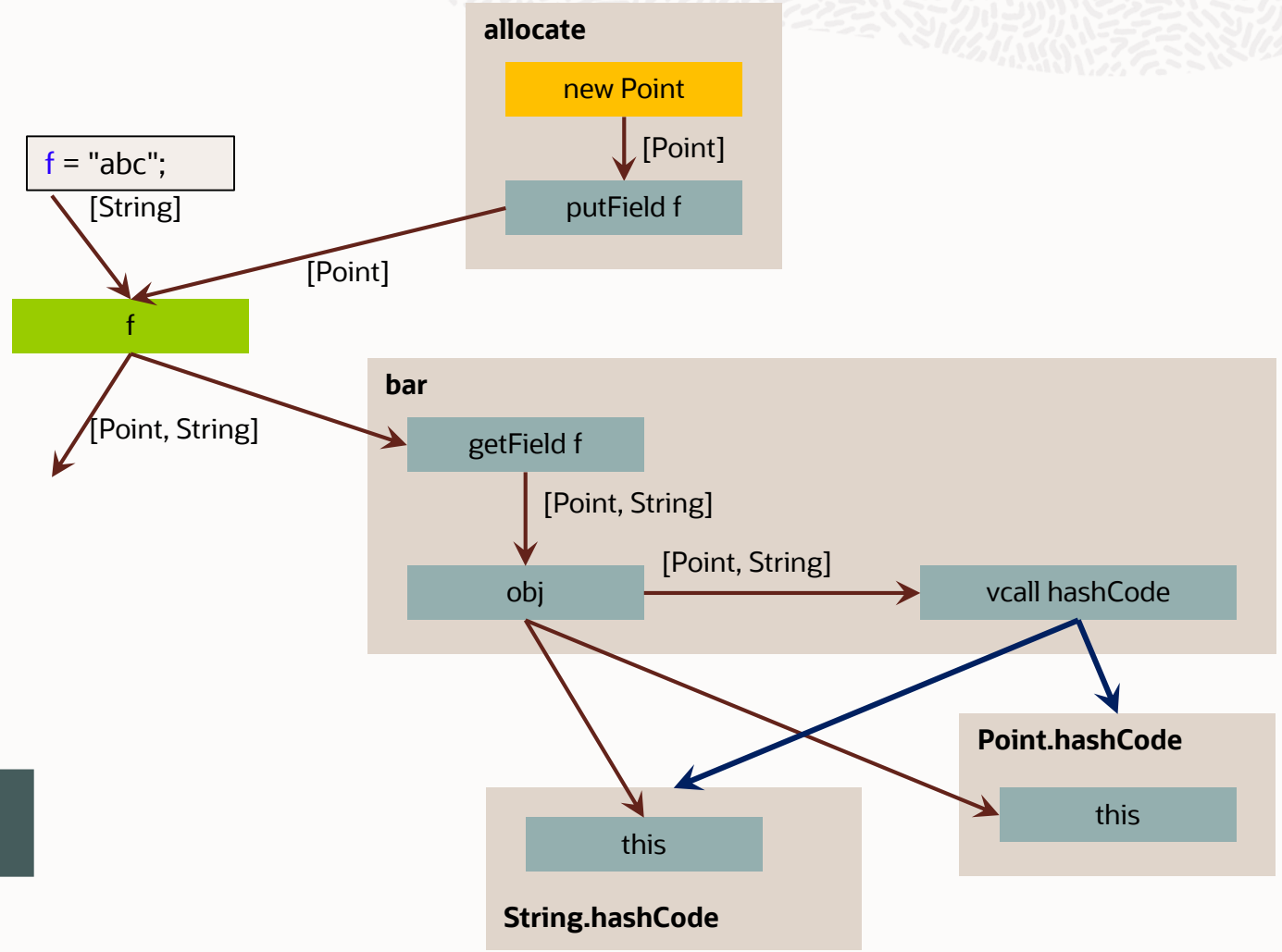


Analysis is context insensitive:
One type state per field

Points-to analysis

```
Object f;

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

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

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

void someMethod() {
  f = "abc";
}
```

Analysis is context insensitive:
One type state per field



Context sensitive analysis

Theory: 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.

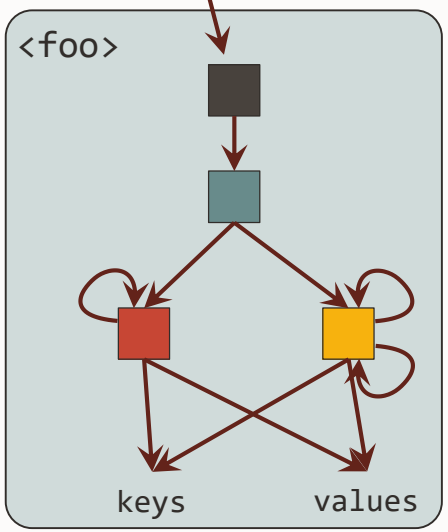
But what nobody really tells you: Any useful improvement of precision requires a very deep context

- Java has deep call chains
- Java has deep object structures. For example, just look at `java.util.HashMap`
- Java arrays have no type information: every array can be cast to `Object[]`
- Reflection is pervasively used, and reflection passes arguments in Object[] array
- About every JDK method can call `String.format` which has huge reachability

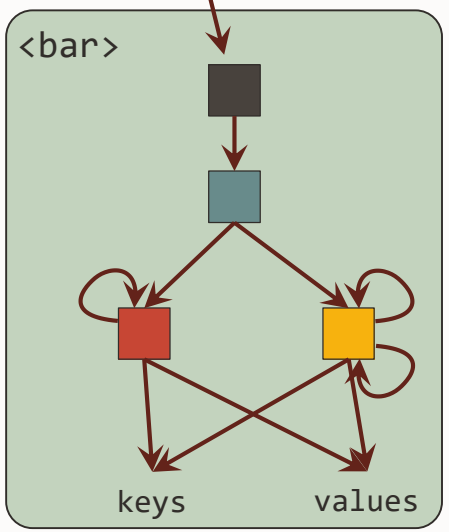
We believe that a context sensitive analysis is infeasible in production. At least we tried and failed.

Problems of context sensitive analysis: HashMap

```
void foo() {
  Map map = new HashMap<>();
  ...
}
```




```
void bar() {
  Map map = new HashMap<>();
  ...
}
```

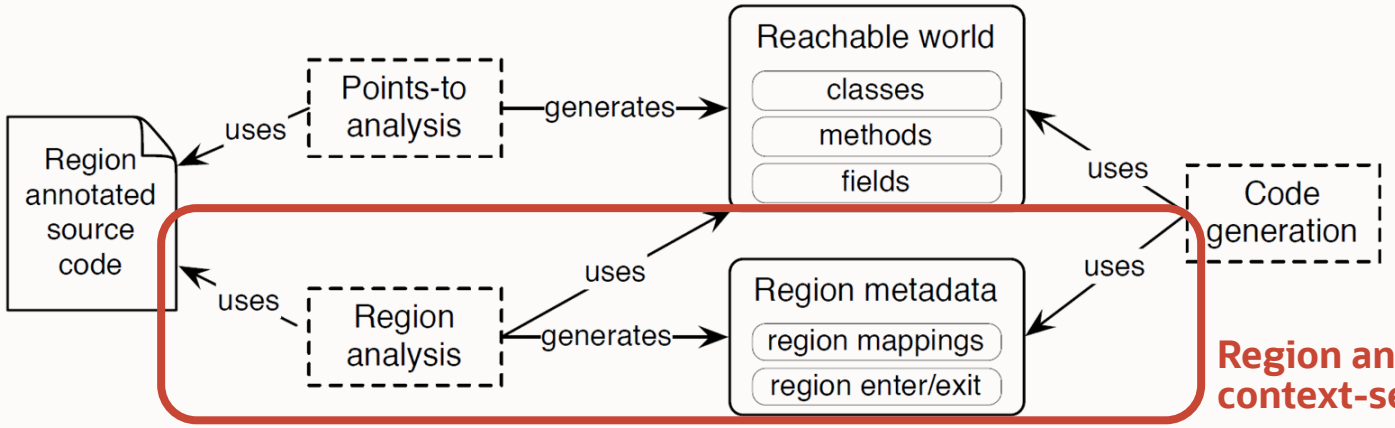


At least a 3-level heap context is required for any useful optimization

Region-based memory management

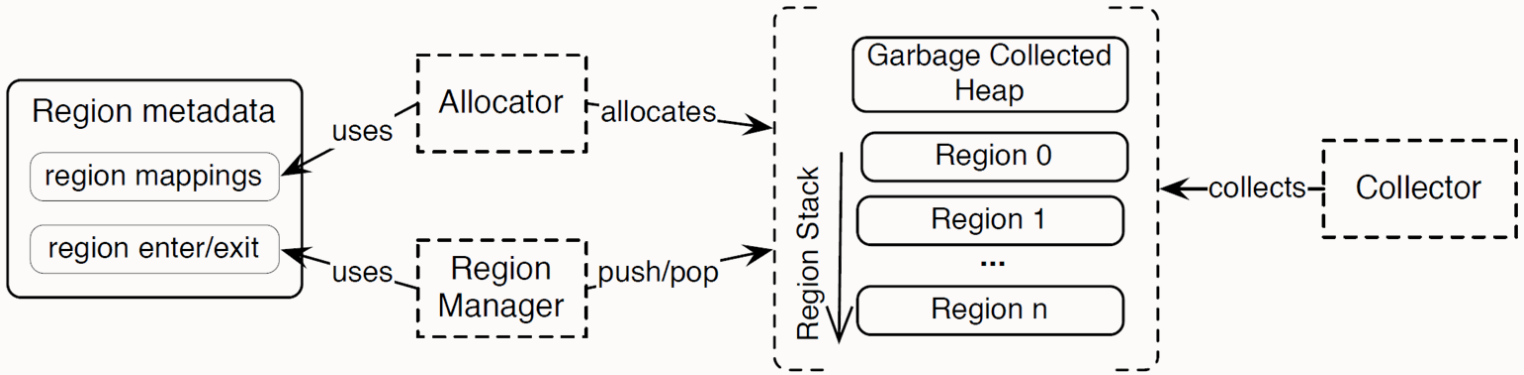
Stancu et al, **Safe and Efficient Hybrid Memory Management for Java**
https://doi.org/10.1145/2754169.2754185

Static Analysis



Region analysis is built as a context-sensitive static analysis

Runtime Data Structures



Semantic models

Fegade et al, **Scalable Pointer Analysis of Data Structures using Semantic Models**
https://doi.org/10.1145/3377555.3377885

- Model API behavior
- Are simple to analyze
- But do not model all behavior

The static analysis links method calls to both the original and the model implementation

- Original implementation without context
- Semantic model with context
  - 1-level deep heap context is sufficient

Only the more precise return values from model are propagated

Model for HashMap:

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

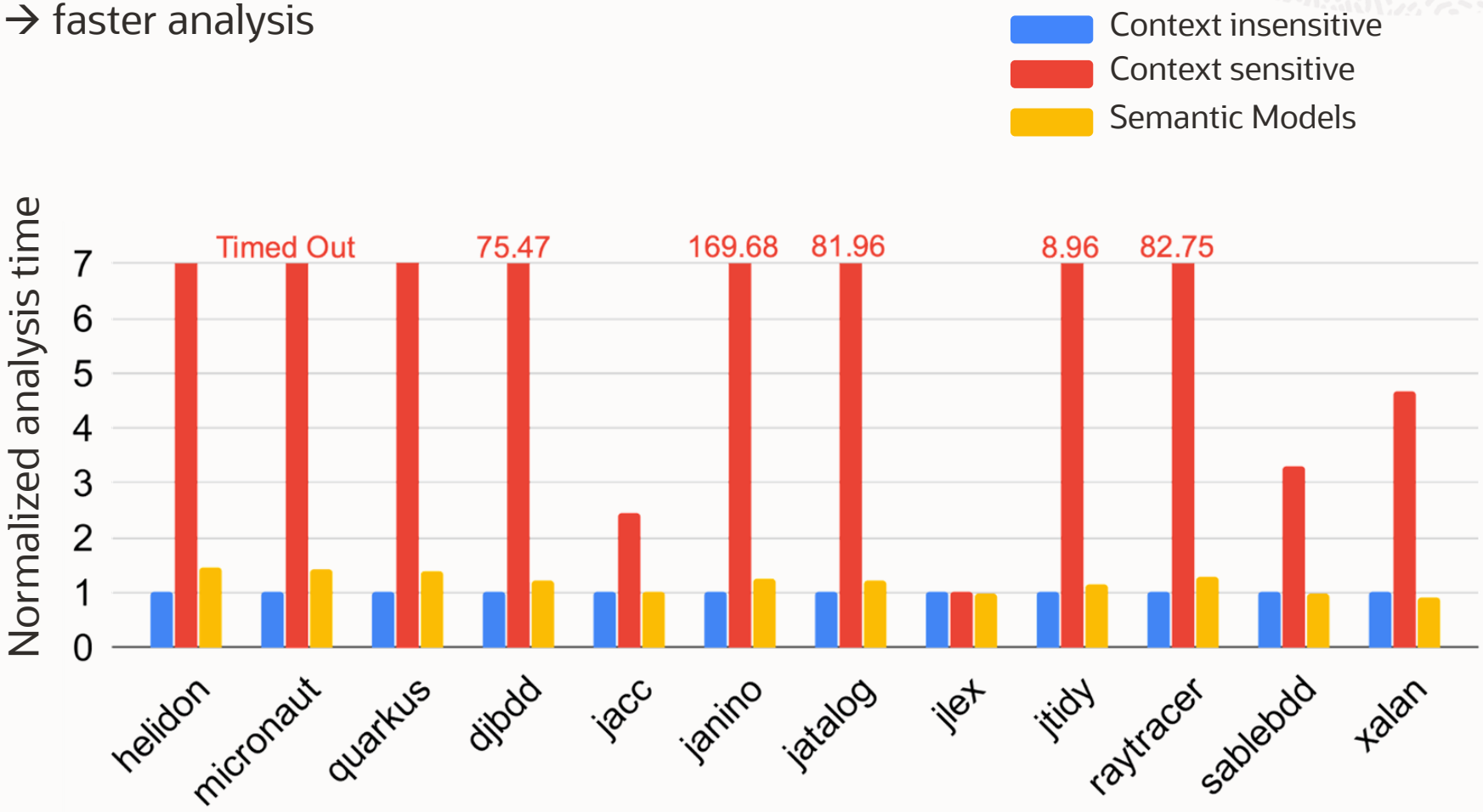
```
V HashMap_Model.get(Object key) {
   return this.allValues;
}
```

Quite intuitive

- “Anything that gets put in can come out”

Semantic models: analysis time

Lower is better → faster analysis



Improving analysis time and memory footprint

- Improving analysis precision is really great
- But in reality, no user has complained about the precision of our current context-insensitive analysis
  - And we only just started using all the computed information
- In contrast, many users have complained about long analysis time and high memory footprint
  - Currently minutes of analysis time, 10 GByte memory footprint
  - GitHub issue #3043: “Image building fails on my Raspberry Pi”

Points-to results are moderately useful for optimizing peak performance

- Useful to know if a certain value has an exact type, or a few types, or is never `null`
- But 10 types vs. 1000 types makes no difference

Solution: “saturated type states”

- Only track small type states individually

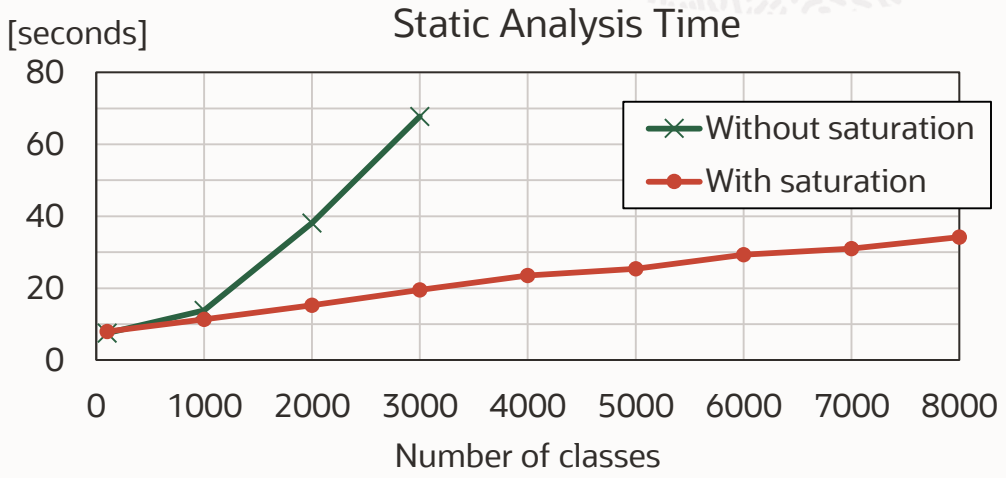
Saturated type states

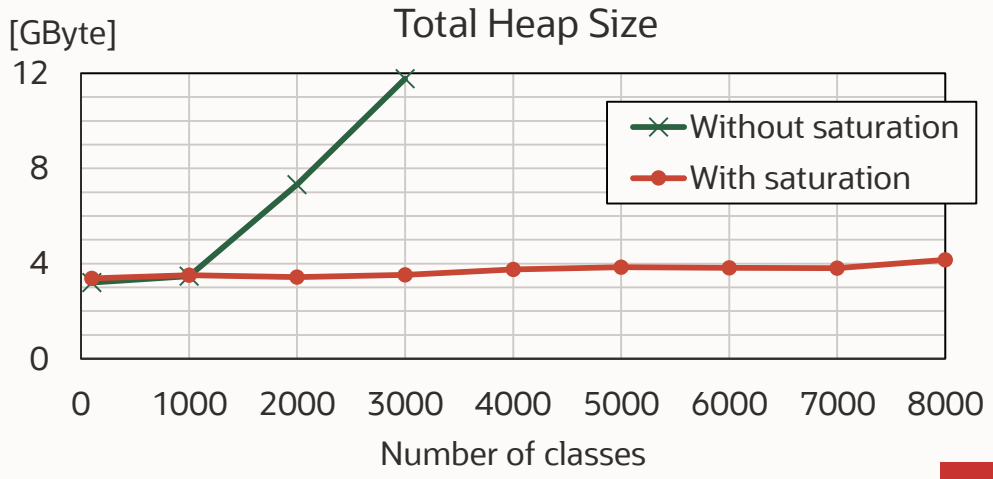
Goal: reduce the memory footprint (and analysis time) for large image builds

- Make the analysis “linear enough”
- Saturate type states: Do not track detailed type information when *number of types > n*
- Link virtual method calls with saturated receivers only once
- Enabled since GraalVM 20.2

Benchmark numbers: synthetic benchmark with 10 virtual methods per class, and increasing number of classes

- Out of memory without saturation for 4000 and more classes





Inline methods before static analysis

- Context-sensitivity with a method context is conceptually similar to method inlining
  - Both expand the scope from one method to a group of methods
- Inlining only for small methods does not increase static analysis time
  - Compiler: Inlining of small methods reduces compilation time and compiled code size

- Method inlining is even more powerful than a method context
  - Context sensitive analysis only improves quality of values that have an object type
  - Method inlining also improves the quality of primitive value
  - Enables more constant folding and dead code elimination before static analysis:

```
// Either foo or bar is reachable
if (WINDOWS) foo(); else bar();

// Without method inlining, both
// foo and bar are reachable
if (isWindows()) foo(); else bar();
```

```
static final boolean WINDOWS = ...

static boolean isWindows() {
  return WINDOWS;
}
```

- Enabled since GraalVM 21.3

Compiler optimizations to run before static analysis

Constant folding
- Propagate final field values from the image heap

Dead code elimination
- Removing unreachable code early makes less other code reachable

Method inlining
- Makes the static analysis context sensitive

Conditional elimination
- Remove redundant if-statements to make the analysis faster

Escape analysis
- Enable more constant folding and inlining by combining object/array allocation and initialization

```
HelloWorld.class.getDeclaredMethod("foo", String.class, int.class);
```

is really

```
HelloWorld.class.getDeclaredMethod("foo", new Class[] {String.class, int.class});
```

and needs escape analysis of the array allocation and initialization before `getDeclaredMethod` can be constant folded

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
```

Summary: GraalVM Native Image static analysis

- Context-insensitive points-to analysis
- Image heap scanning during analysis
- Application code runs during static analysis and can react to reachability information

- Context-sensitive analysis worked for research projects, but too slow for production
- Recent focus on reducing analysis time and footprint, not improving precision
  - Propagate only “useful” information through the type flow graph
- Some compiler optimizations run before the static analysis
  - Constant folding, method inlining

Thank you

https://www.graalvm.org