ORACLE®

One Compiler

Christian Wimmer

VM Research Group, Oracle Labs

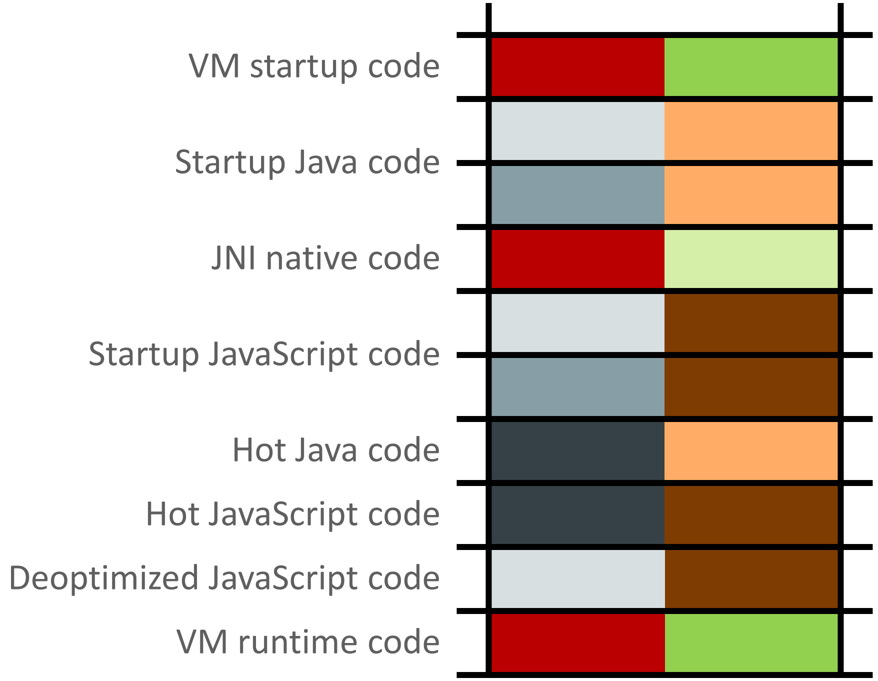
ORACLE

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.

ORACLE

Typical Stack of Java HotSpot VM Running Nashorn



Stack frame layout:

- Bytecode interpreter
- Client compiler
- Server compiler
- Native code

Source of code:

- Java code (bytecode from .java file)
- JavaScript (dynamically generated bytecode)
- Java HotSpot VM
- JDK native code

How do you find all the GC root pointers?

Duplication: Everything Implement Three Times

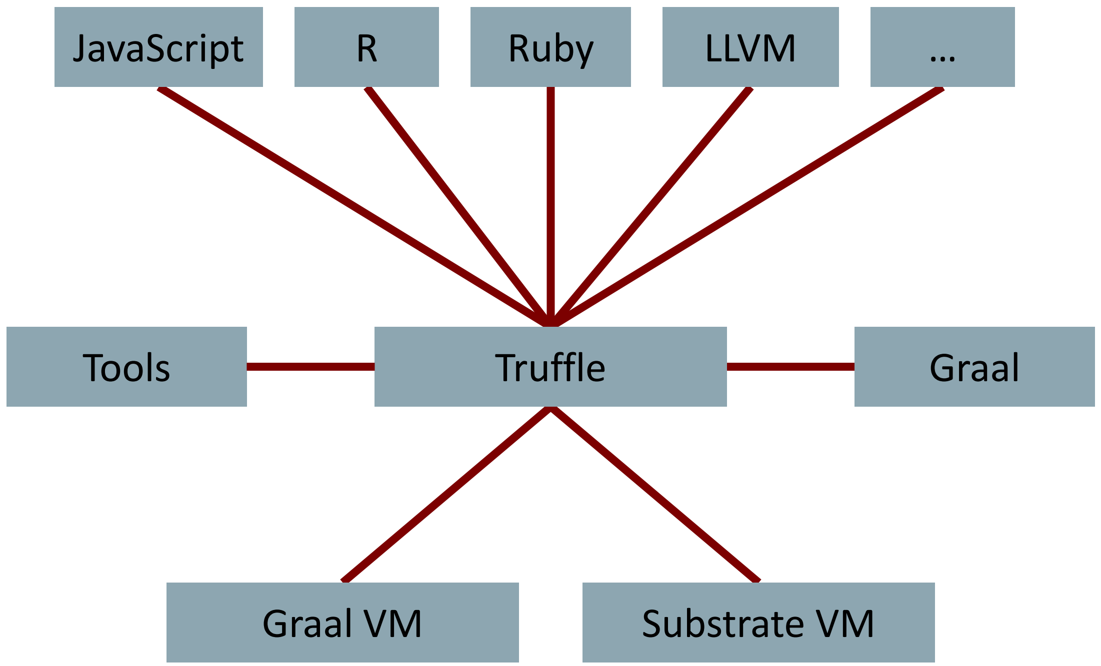
	Bytecode interpreter	Compiled bytecode	Native code (C/C++)
Stack frame layout	Close to JVM spec	Spill slots	Unspecified
Stack frame size	variable	fixed per method	unknown
Root pointers for GC	Bytecode liveness (expensive to compute)	Pointer map from compiler	Explicit handles (error prone)
Exception handling	Interpret metadata	Compiled in (mostly)	Explicit checks (error prone)
Porting to new architecture	Write assembly code	Write client compiler and server compiler backends	Write gcc backend
Debugging	Java debugger	Java debugger	gdb

ORACLE

Truffle System Structure

AST Interpreter for every language

Your language should be here!



Common API separates language implementation, optimization system, and tools (debugger)

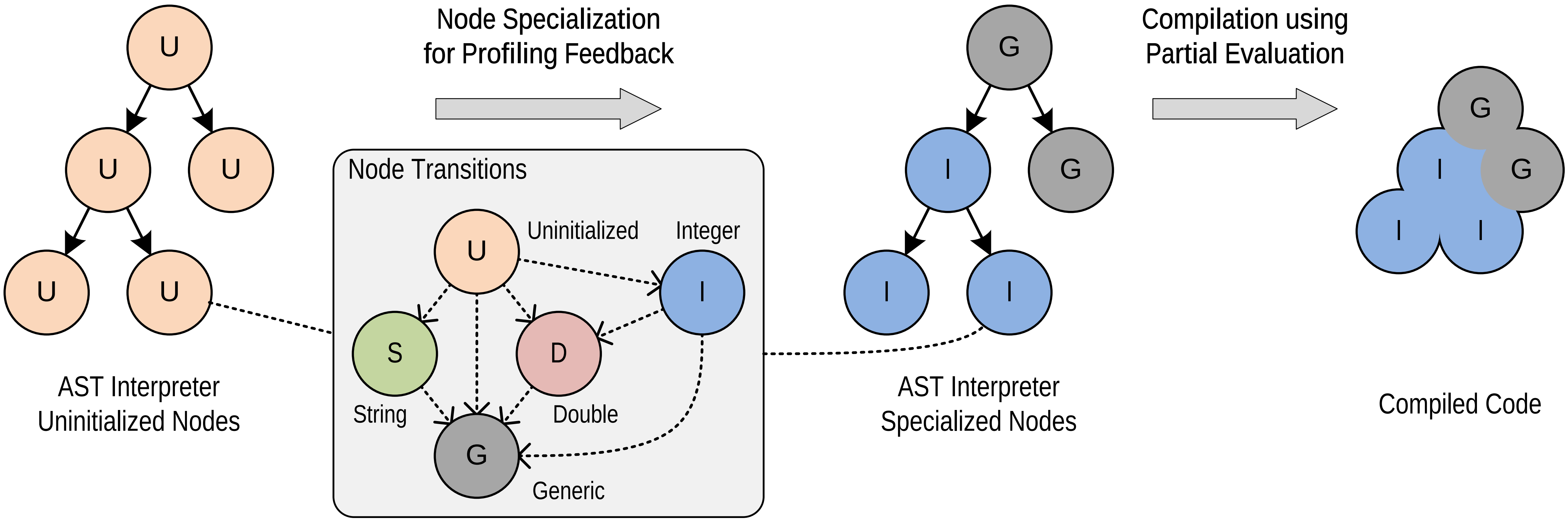
Language agnostic dynamic compiler

Integrate with Java applications

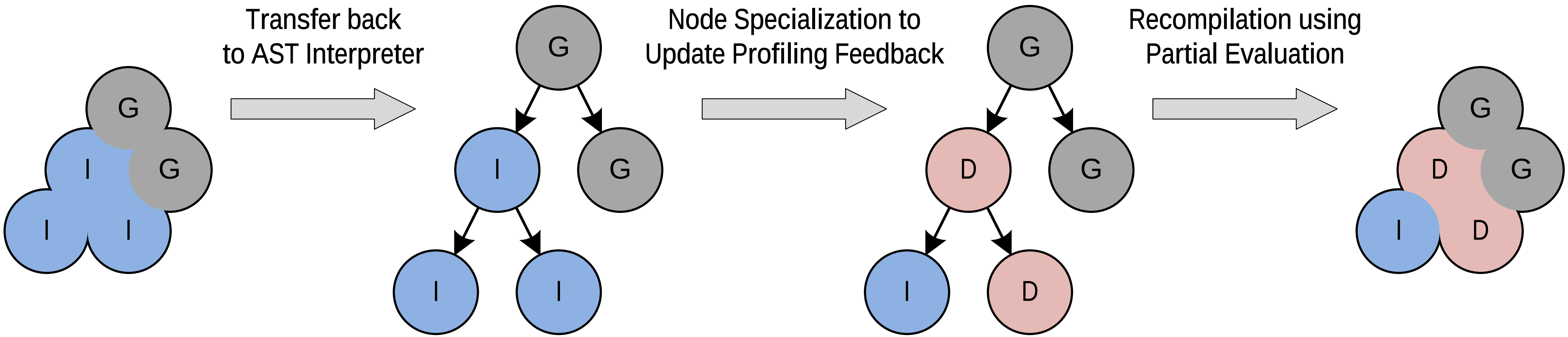
Low-footprint VM, also suitable for embedding

ORACLE

Speculate and Optimize …

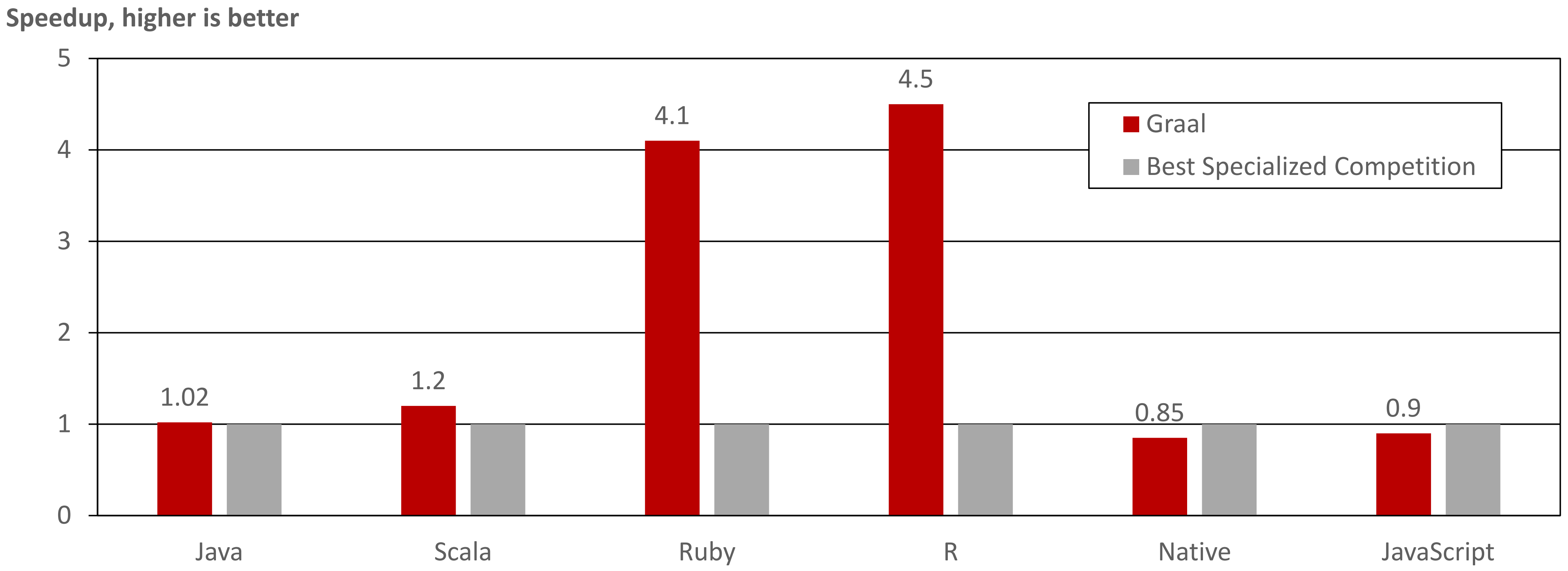


… and Transfer to Interpreter and Reoptimize!



ORACLE

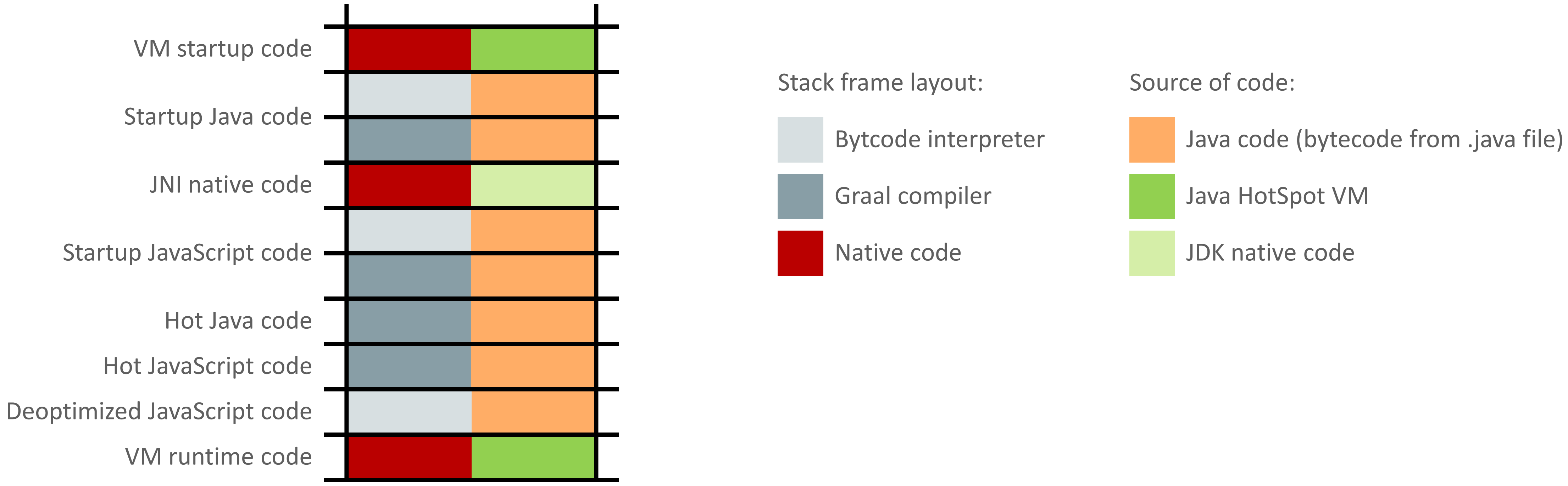
Performance: Graal VM



Performance relative to:

HotSpot/Server, HotSpot/Server running JRuby, GNU R, LLVM AOT compiled, V8

Possible Stack of Java HotSpot VM Running Truffle



Our default configuration of Truffle still uses Client and Server compiler for Java code

The Substrate VM is …

… an **embeddable** VM

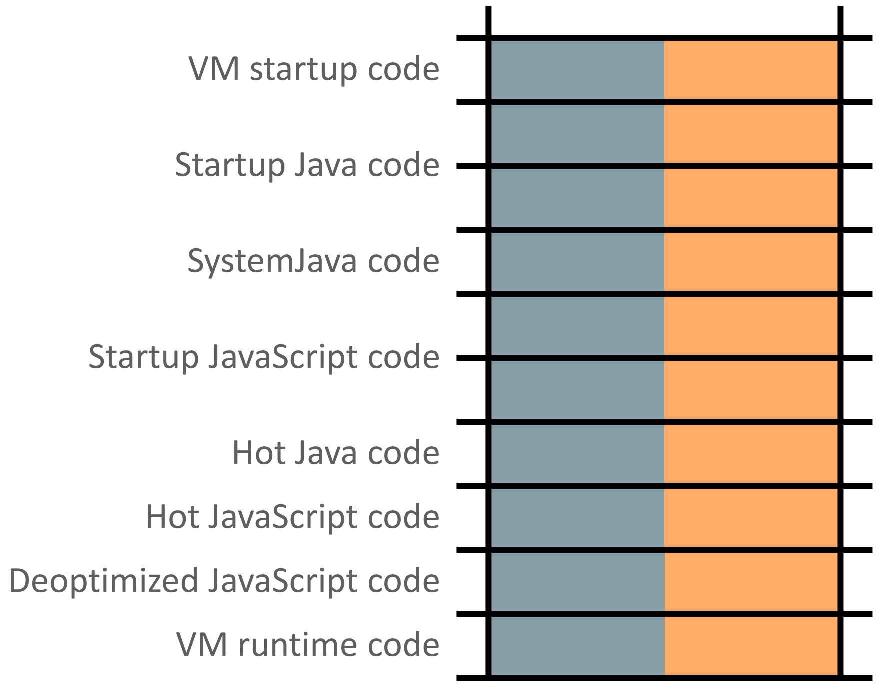
for, and written in, a **subset of Java**

optimized to **execute Truffle** languages

ahead-of-time compiled using Graal

integrating with **native development tools.**

Typical Stack of Substrate VM Running Truffle



Stack frame layout:

Graal compiler

Source of code:

Java code (bytecode from .java file)

Substrate VM runtime is written in Java

Same compiler for ahead-of-time compiled Java code and dynamically compiled Truffle AST

Transfer to AST interpreter (deoptimization) to Graal compiled code with extra deoptimization entry points

ORACLE

Substrate VM: Execution Model

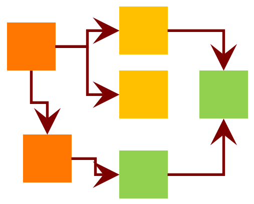
Points-To Analysis

Ahead-of-Time Compilation

Truffle Language

JDK

Substrate VM



Machine Code

Initial Heap

DWARF Info

ELF / MachO Binary

All Java classes from Truffle language (or any application), JDK, and Substrate VM

Reachable methods, fields, and classes

Application running without dependency on JDK and without Java class loading

ORACLE

Substrate VM Building Blocks

- Reduced runtime system, all written in Java
  - Stack walking, exception handling, garbage collector, deoptimization
  - Graal for ahead-of-time compilation and dynamic compilation
- Points-to analysis
  - Closed-world assumption: no dynamic class loading, no reflection
  - Using Graal for bytecode parsing
  - Fixed-point iteration: propagate type states through methods
- SystemJava for integration with C code
  - Machine-word sized value, represented as Java interface, but unboxed by compiler
  - Import of C functions and C structs to Java
- Substitutions for JDK methods that use unsupported features
  - JNI code replaced with SystemJava code that directly calls to C library

ORACLE

Key Features of Graal

- Designed for speculative optimizations and deoptimization
  – Metadata for deoptimization is propagated through all optimization phases
- Designed for exact garbage collection
  – Read/write barriers, pointer maps for garbage collector
- Aggressive high-level optimizations
  – Example: partial escape analysis
- Modular architecture
  – Configurable compiler phases
  – Compiler-VM separation: snippets, provider interfaces
- Written in Java to lower the entry barrier
  – Graal compiling and optimizing itself is also a good optimization opportunity

ORACLE

Deoptimization

Deoptimization

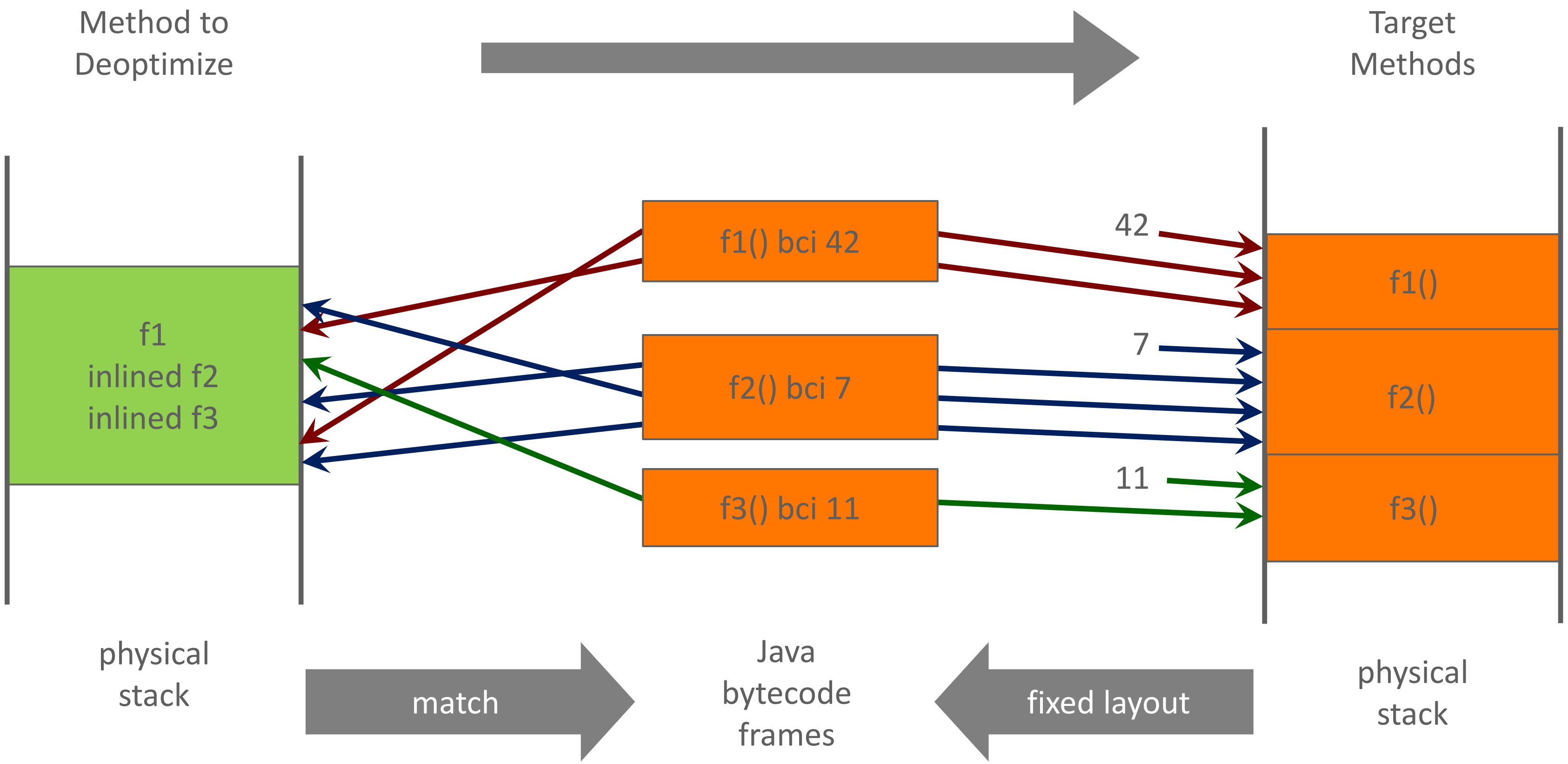
- Transfer from optimized machine code back to unoptimized code

- Enables speculative optimizations
  - Optimized code does not need to deal with corner cases
    - No control flow merges from slow-path code back into the fast path
    - More potential for optimizations
  - Optimized code does not need to check assumptions
    - Instead, it gets invalidated externally when assumption is no longer valid

- Speculative optimizations are essential for optimizing dynamic languages
  - Speculate on JavaScript type stability
  - Speculate that Ruby operators for primitive types are not changed by program
  - Polymorphic inline caches for function calls, property accesses, ...

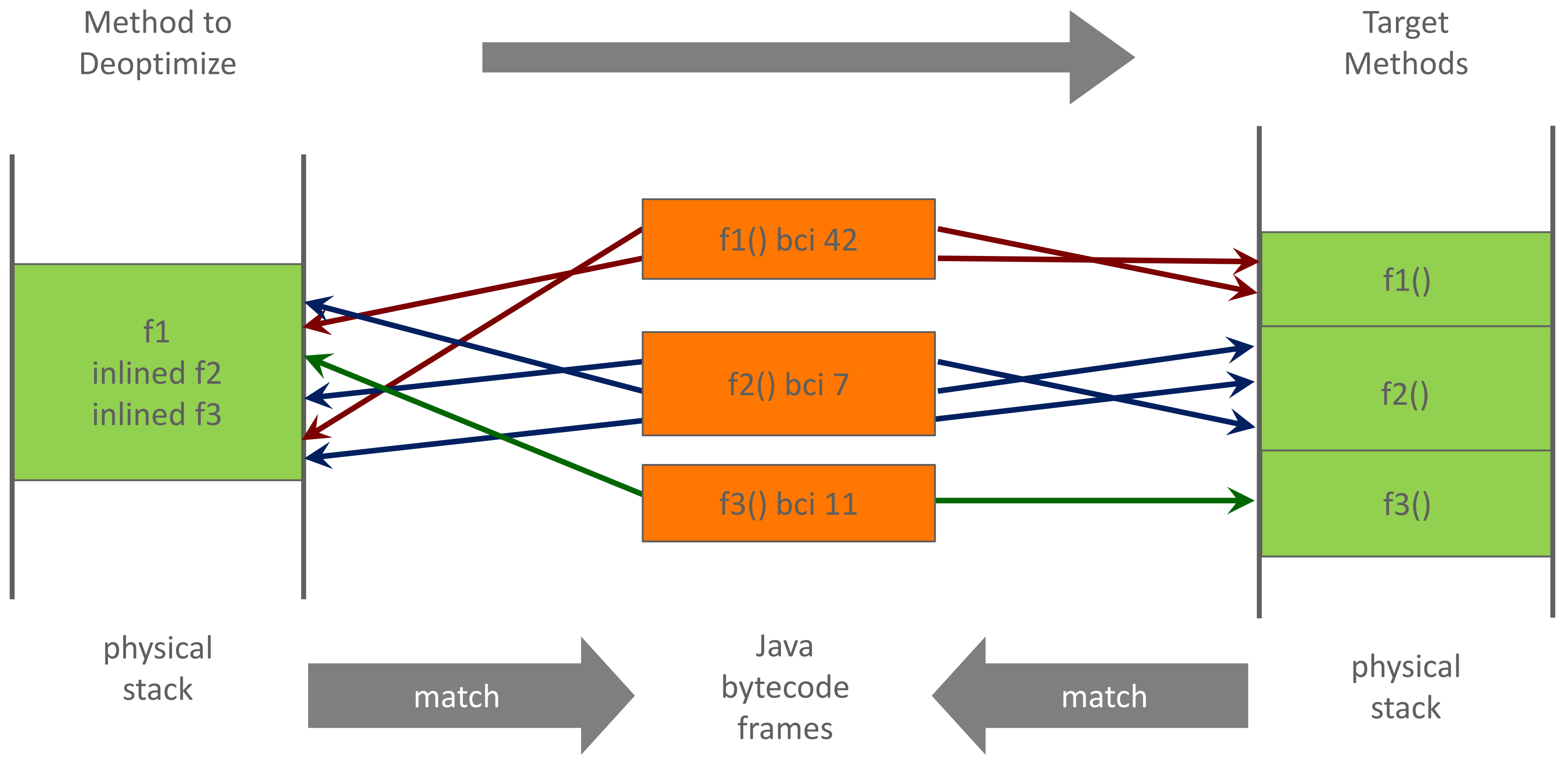
Deoptimization on HotSpot VM

Mapping from optimized to bytecode interpreter frames



Deoptimization on Substrate VM

Mapping from optimized to unoptimized stack frames



Deoptimization on Substrate VM

- Source and target are Graal compiled frames
  - Both have metadata that describes the layout with respect to JVM specification
  - Stack frame location of all used local variables and expression stack elements
  - Source and target describe the same bytecode index (bci), i.e., a matching state

- Source is a fully optimized Graal frame
  - Method inlining: multiple target frames for one source frame
  - Escape analysis: virtual objects that are re-allocated during deoptimization
  - Global value numbering: elimination of duplicate computations
- Targets are Graal frames with limited optimizations
  - No method inlining: multiple target frames restored when source frame has inlined methods
  - No escape analysis: all objects are re-allocated during deoptimization
  - Limited value numbering: only values in Java frame state can be live across a deoptimization entry point

ORACLE

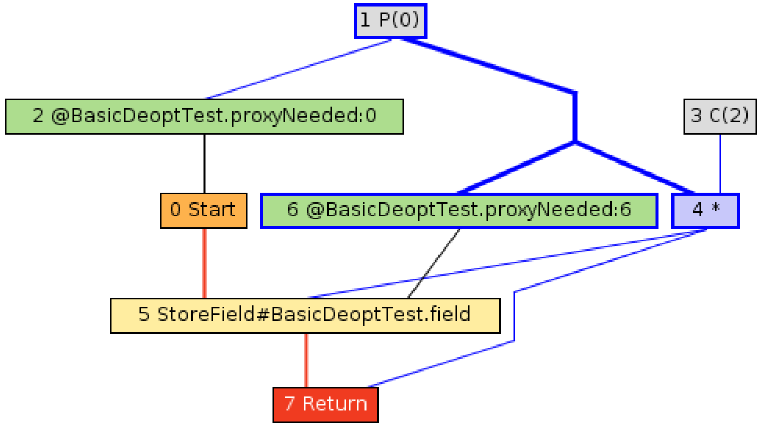
Example: Graal IR for Deoptimization

Java source code:

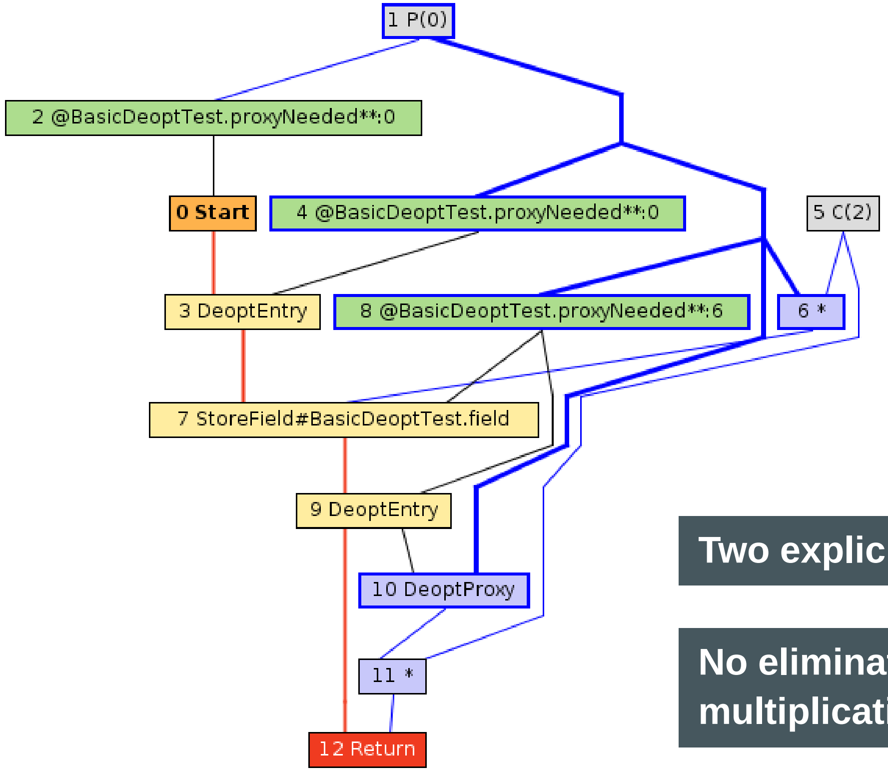
```java
public class BasicDeoptTest {
  static int field;

  static int proxyNeeded(int x) {
    field = x * 2;
    return x * 2;
  }
}
```

Graal IR for optimized compilation:



Graal IR for compilation with deoptimization entry points:



Two explicit `DeoptEntry` points

No elimination of second multiplication

SystemJava

SystemJava

- Legacy C code integration
  - Need a convenient way to access preexisting C functions and structures
  - Example: libc, database
- Legacy Java code integration
  - Leverage preexisting Java libraries
  - "Patch" violations of our reduced Java rules
  - Example: JDK class library
- Call Java from C code
  - Entry points into our Java code

SystemJava vs. JNI

- Java Native Interface (JNI)
  - Write custom C code to integrate existing C code with Java
  - C code knows about Java types
  - Java objects passed to C code using handles

- SystemJava
  - Write custom Java code to integrate existing C code with Java
  - Java code knows about C types
  - No need to pass Java objects to C code

ORACLE

Word type for low-level memory access

- Requirements
  - Support raw memory access and pointer arithmetic
  - No extension of the Java programming language
  - Pointer type modeled as a class to prevent mixing with, e.g., `long`
  - Transparent bit width (32 bit or 64 bit) in code using it
- Base interface `Word`
  - Looks like an object to the Java IDE, but is a primitive value at run time
  - Graal does the transformation
- Subclasses for type safety
  - `Pointer`: C equivalent `void*`
  - `Unsigned`: C equivalent `size_t`
  - `Signed`: C equivalent `ssize_t`

```
public static Unsigned strlen(CharPointer str) {
  Unsigned n = Word.zero();
  while (str.read(n) != 0) {
    n = n.add(1);
  }
  return n;
}
```

Java Annotations to Import C Elements

```
@CFunction static native int clock_gettime(int clock_id, timespec tp);
```

```
int clock_gettime(clockid_t __clock_id, struct timespec *__tp)
```

```
@CConstant static native int CLOCK_MONOTONIC();
```

```
#define CLOCK_MONOTONIC 1
```

```
@CStruct interface timespec extends PointerBase {
  @CField long tv_sec();
  @CField long tv_nsec();
}
```

```
struct timespec {
  __time_t tv_sec;
  __syscall_slong_t tv_nsec;
};
```

```
@CPointerTo(nameOfCType="int") interface CIntPointer extends PointerBase {
  int read();
  void write(int value);
}
```

```
int* pint;
```

```
@CPointerTo(CIntPointer.class) interface CIntPointerPointer ...
```

```
int** ppint;
```

```
@CContext(PosixDirectives.class)
```

```
#include <time.h>
```

```
@CLibrary("rt")
```

```
-lrt
```

Implementation of `System.nanoTime()` using SystemJava:

```
static long nanoTime() {
  timespec tp = StackValue.get(SizeOf.get(timespec.class));
  clock_gettime(CLOCK_MONOTONIC(), tp);
  return tp.tv_sec() * 1_000_000_000L + tp.tv_nsec();
}
```

ORACLE

Points-To Analysis

Graal as a Static Analysis Framework

- Graal 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
  - Build a high-level IR graph in SSA form
  - Linking / method resolution of method calls
- Static points-to analysis and compilation use same intermediate representation
  - Simplifies applying 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 are propagated from sources (allocations) to usages

ORACLE

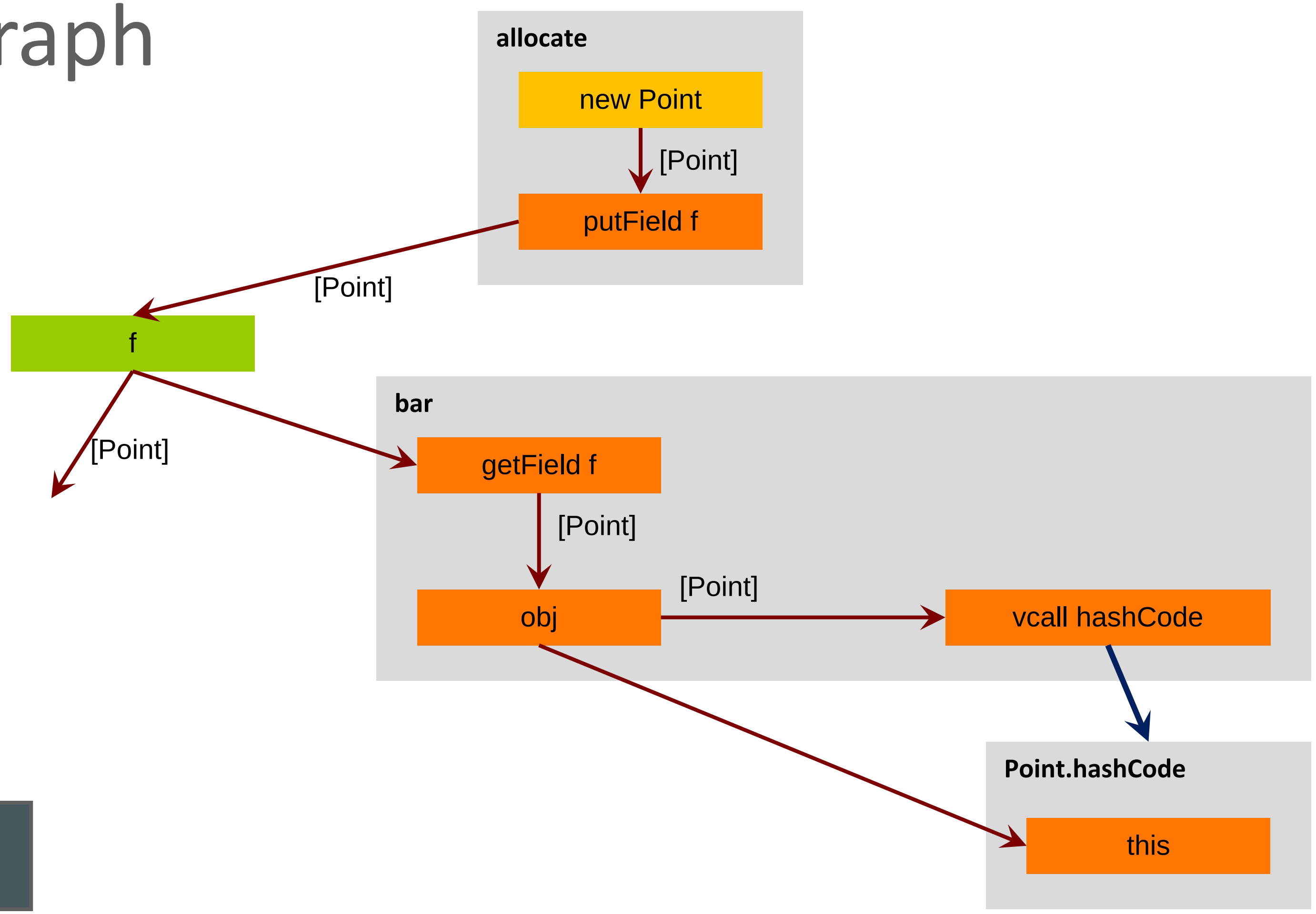
Example Type Flow Graph

```
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

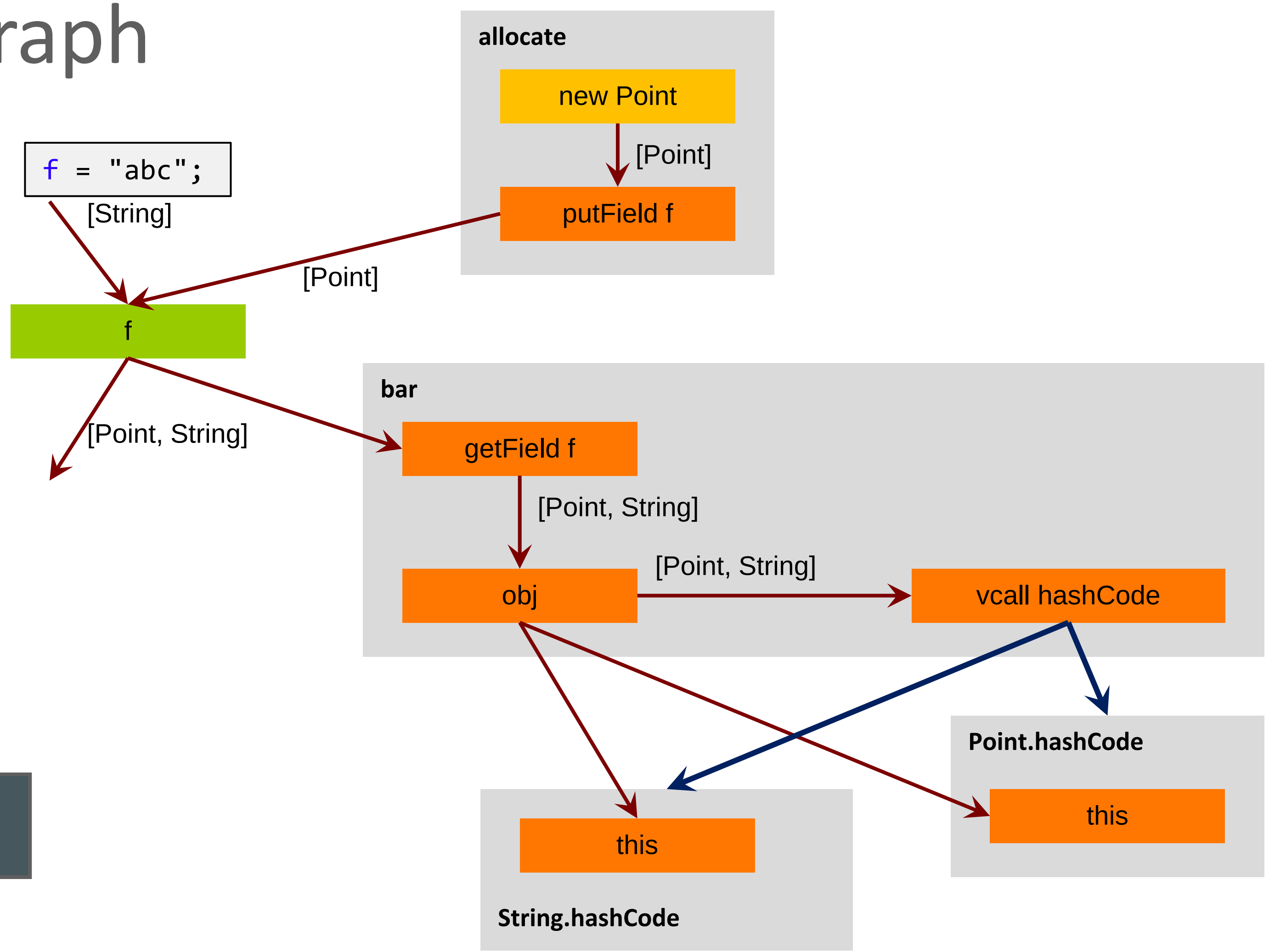
Example Type Flow Graph

```java
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

Results

Microbenchmark for Startup and Peak Performance (1)

```
function benchmark(n) {
   var obj = {i: 0, result: 0};
   while (obj.i <= n) {
      obj.result = obj.result + obj.i;
      obj.i = obj.i + 1;
   }
   return obj.result;
}
```

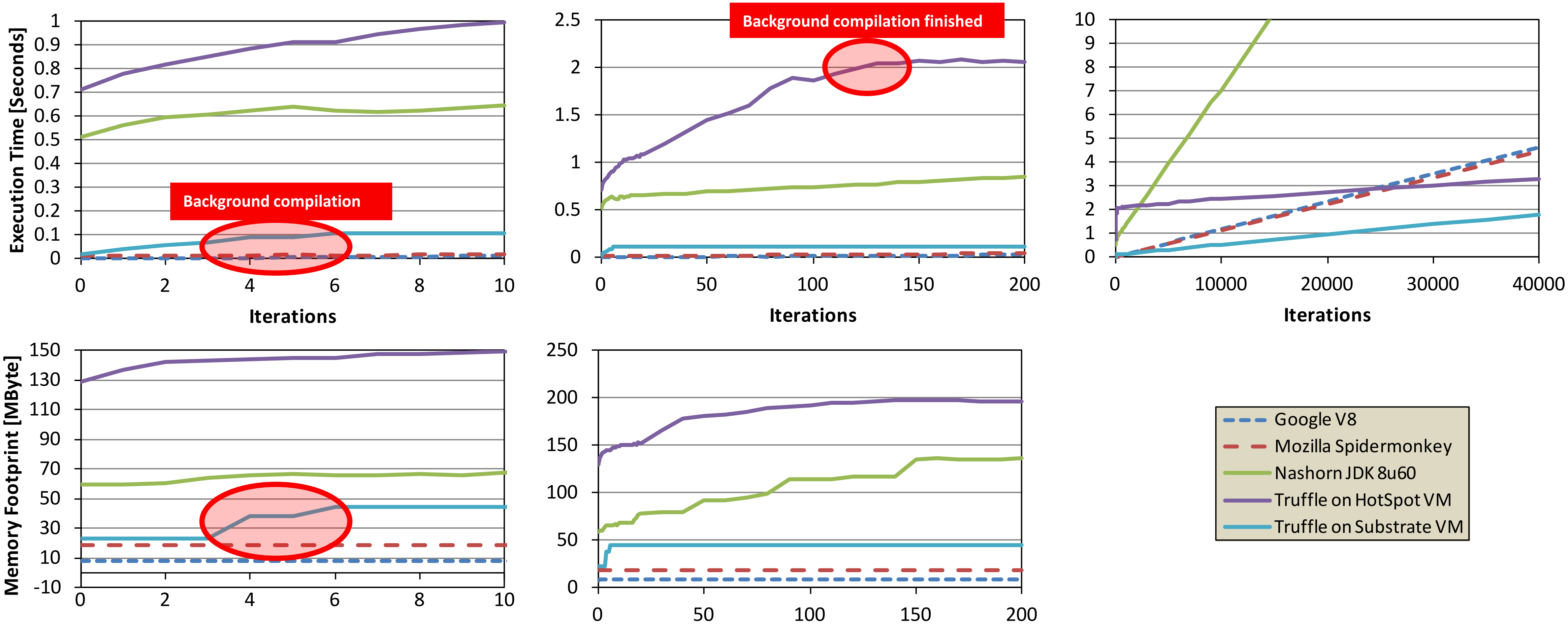
Function benchmark is invoked in a loop by harness (0 to 40000 iterations)

n fixed to 50000 for all iterations

JavaScript VM	Version	Command Line Flags
Google V8	Version 4.2.27	[none]
Mozilla Spidermonkey	Version JavaScript-C45.0a1	[none]
Nashorn JDK 8 update 60	build 1.8.0_60-b27	-J-Xmx256M
Truffle on HotSpot VM	graal-js changeset a8947301fd1e from Nov 30, 2015 graal-enterprise changeset f47fff503e49 from Nov 30, 2015	-J-Xmx256M
Truffle on Substrate VM	substratevm changeset 45c61d192d43 from Dec 1, 2015 graal-enterprise changeset d8ee392c83e3 from Nov 21, 2015	[none]

ORACLE

Microbenchmark for Startup and Peak Performance (2)



Summary

- Substrate VM uses a "One Compiler" approach
  - For ahead-of-time compilation and dynamic compilation
  - For all levels: Java, SystemJava, JavaScript, all other Truffle languages
  - For deoptimization entry points
  - For static points-to analysis

- Graal is flexible enough to support all these use cases
  - Snippets for compiler-VM separation
  - Configuration of phases

ORACLE

Acknowledgements

Oracle
Danilo Ansaloni
Stefan Anzinger
Cosmin Basca
Daniele Bonetta
Matthias Brantner
Petr Chalupa
Jürgen Christ
Laurent Daynès
Gilles Duboscq
Martin Entlicher
Bastian Hossbach
Christian Humer
Mick Jordan
Vojin Jovanovic
Peter Kessler
David Leopoldseder
Kevin Menard
Jakub Podlešák
Aleksandar Prokopec
Tom Rodriguez

Oracle (continued)
Roland Schatz
Chris Seaton
Doug Simon
Štěpán Šindelář
Zbyněk Šlajchrt
Lukas Stadler
Codrut Stancu
Jan Štola
Jaroslav Tulach
Michael Van De Vanter
Adam Welc
Christian Wimmer
Christian Wirth
Paul Wögerer
Mario Wolczko
Andreas Wöß
Thomas Würthinger

Oracle Interns
Brian Belleville
Miguel Garcia
Shams Imam
Alexey Karyakin
Stephen Kell
Andreas Kunft
Volker Lanting
Gero Leinemann
Julian Lettner
Joe Nash
David Piorkowski
Gregor Richards
Robert Seilbeck
Rifat Shariyar

Alumni
Erik Eckstein
Michael Haupt
Christos Kotselidis
Hyunjin Lee
David Leibs
Chris Thalinger
Till Westmann

JKU Linz
Prof. Hanspeter Mössenböck
Benoit Daloze
Josef Eisl
Thomas Feichtinger
Matthias Grimmer
Christian Häubl
Josef Haider
Christian Huber
Stefan Marr
Manuel Rigger
Stefan Rumzucker
Bernhard Urban

University of Edinburgh
Christophe Dubach
Juan José Fumero Alfonso
Ranjeet Singh
Toomas Remmelg

LaBRI
Floréal Morandat

University of California, Irvine
Prof. Michael Franz
Gulfem Savrun Yeniceri
Wei Zhang

Purdue University
Prof. Jan Vitek
Tomas Kalibera
Petr Maj
Lei Zhao

T. U. Dortmund
Prof. Peter Marwedel
Helena Kotthaus
Ingo Korb

University of California, Davis
Prof. Duncan Temple Lang
Nicholas Ulle

University of Lugano, Switzerland
Prof. Walter Binder
Sun Haiyang
Yudi Zheng

ORACLE

Integrated Cloud

Applications & Platform Services

ORACLE

ORACLE®