ORACLE®

GTLC
GLOBAL
TECH LEADERSHIP
CONFERENCE

全球技术领导力峰会

通往**年薪百万**的CTO的路上，

如何打造自己的技术**领导**力？

扫描二维码了解详情

GraalVM™ and Its Ecosystem

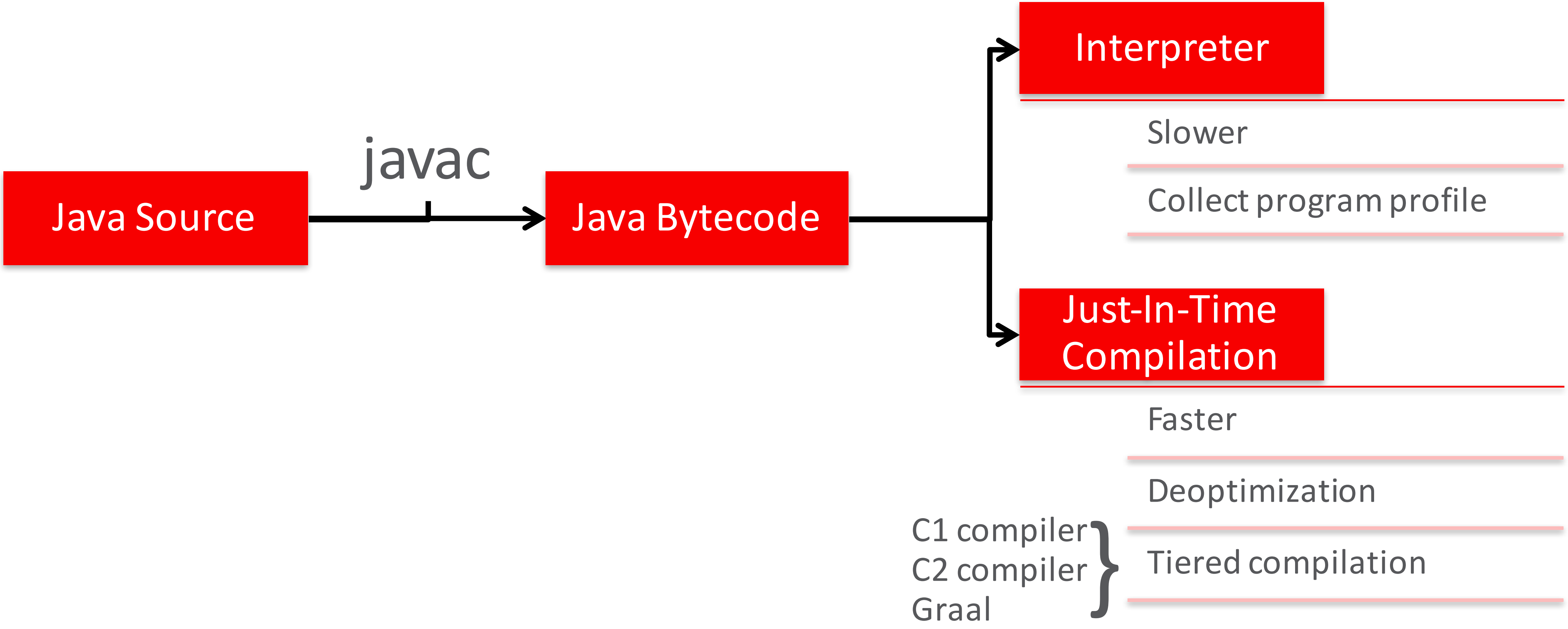
Yudi Zheng
Senior Researcher
Oracle Labs

April, 2018

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, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Java in a Nutshell



Java Source

javac

Java Bytecode

Interpreter

- Slower
- Collect program profile

Just-In-Time Compilation

- Faster
- Deoptimization
- Tiered compilation

C1 compiler
C2 compiler
Graal

“Things I won’t do again: write a VM in C/C++.”

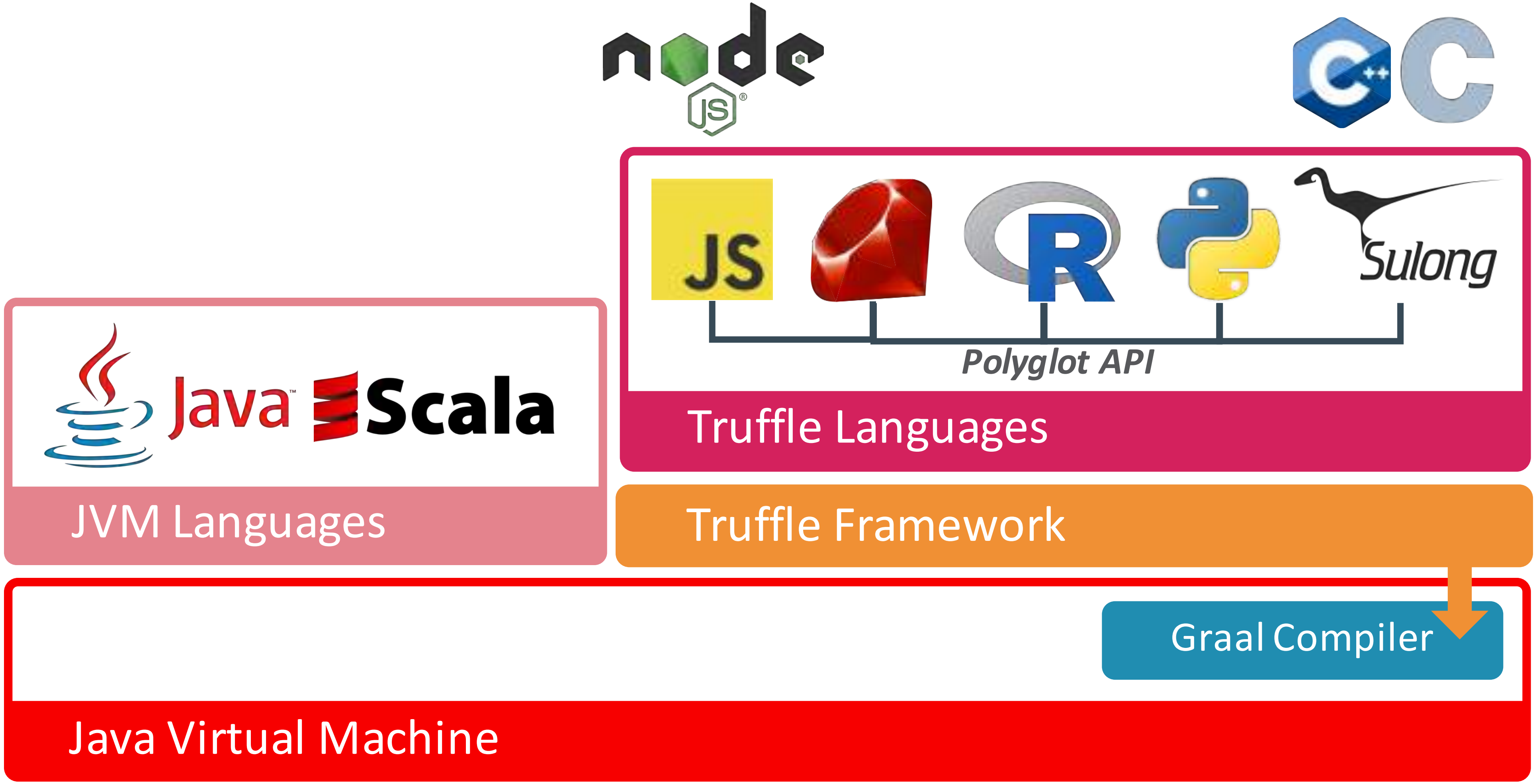
– Cliff Click, CTO, Neurensic; author of HotSpot C2 Compiler

> “C2 is old and very complex;
> Graal is easier to understand.”

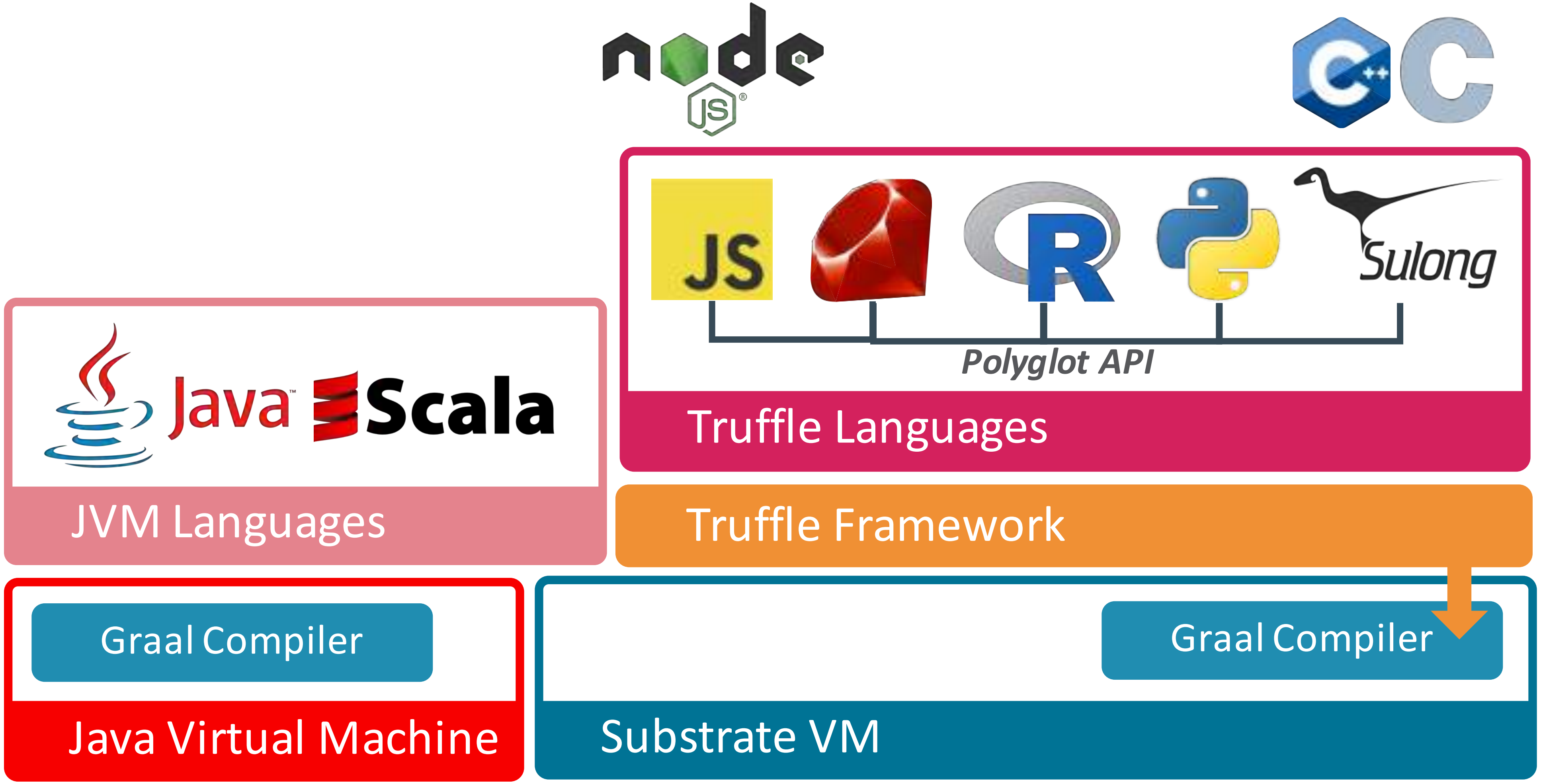
– Chris Thalinger, Staff Software Engineer, Twitter

More Than A JIT Compiler

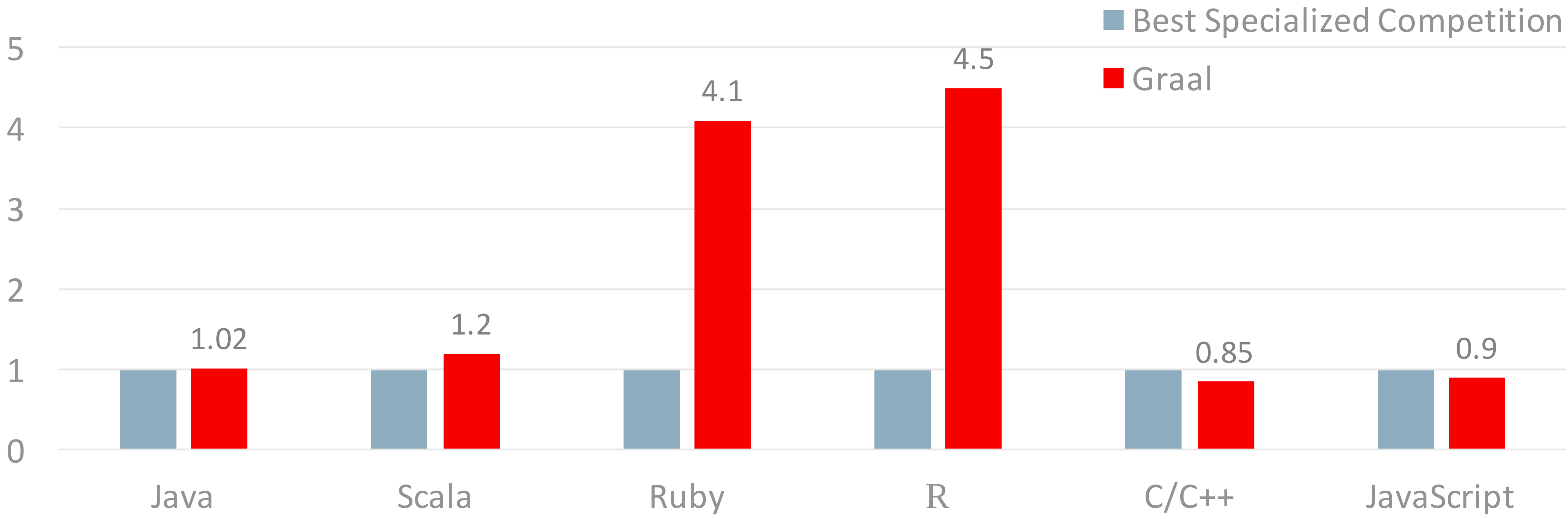
GraalVM™ and Its Ecosystem



GraalVM™ and Its Ecosystem



Performance: GraalVM



Performance relative to:
HotSpot/C2, HotSpot/C2 running JRuby, GNU R, LLVM AOT compiled, V8

Part 1: The Graal Compiler

Key Features of Graal Compiler

- Designed for aggressive speculative optimizations
  - Specialization based on program profile
  - Metadata for deoptimization is propagated through all optimization phases
    - Target method & bytecode index
    - State of local variables and expression stack

Deoptimization

- Switch to interpreter in the middle of compiled machine code
  - Example:

```
int negate(int n) {
  if (n == Integer.MIN_VALUE)
    throw new ArithmeticException();
  return -n;
}
```

→

```
int negate(int n) {
  if (n == Integer.MIN_VALUE)
    deoptimize();
  return -n;
}
```

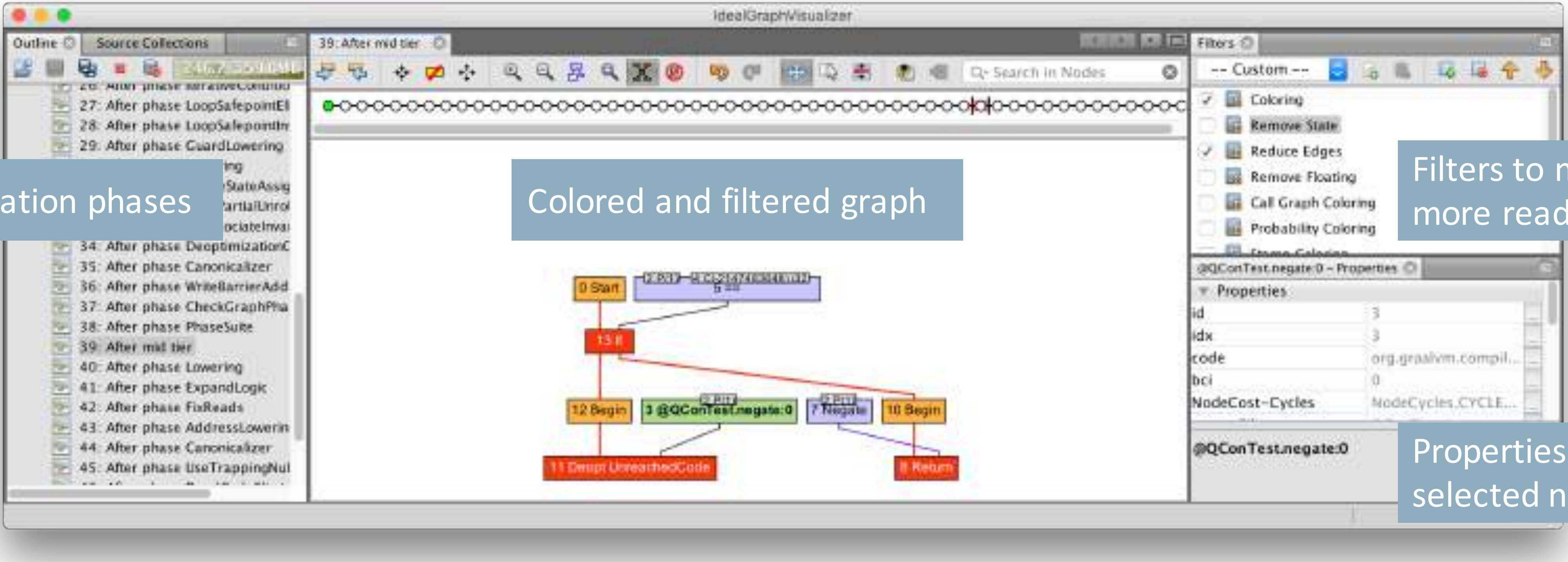
  - Less compilation time; more compact emitted code
  - Expensive, better put in slow path
  - Java-level assumption

Key Features of Graal Compiler

- Designed for aggressive speculative optimizations
  - Specialization based on program profile
  - Metadata for deoptimization is propagated through all optimization phases
    - Target method & bytecode index
    - State of local variables and expression stack
- Graph-based intermediate representation

Ideal Graph Visualizer

Visualizing compilation on-the-fly



27: After phase LoopSafepointEl
28: After phase LoopSafepointIn
29: After phase GuardLowering
34: After phase DeoptimizationC
35: After phase Canonicalizer
36: After phase WriteBarrierAdd
37: After phase CheckGraphPha
38: After phase PhaseSuite
39: After mid tier
40: After phase Lowering
41: After phase ExpandLogic
42: After phase FixReads
43: After phase AddressLowerin
44: After phase Canonicalizer
45: After phase UseTrappingNul

-- Custom --

☑ Coloring
☐ Remove State
☑ Reduce Edges
☐ Remove Floating
☐ Call Graph Coloring
☐ Probability Coloring

@QConTest.negate:0 – Properties

Properties

id	3
idx	3
code	org.graalvm.compil...
bci	0
NodeCost-Cycles	NodeCycles.CYCLE...

@QConTest.negate:0

Optimization phases

Colored and filtered graph

Filters to make graph more readable

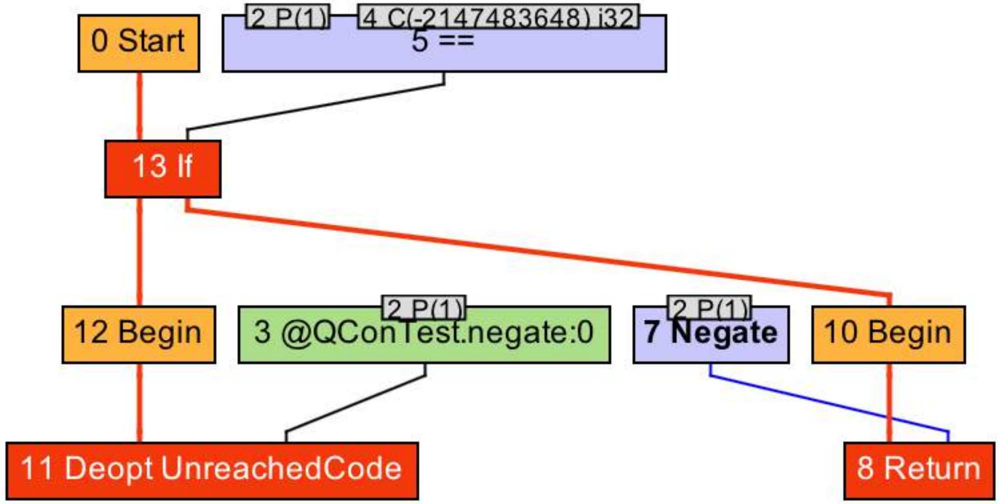
Properties for the selected node

Ideal Graph Visualizer

Visualizing compilation on-the-fly

- Control flow (in **red**) v.s. data flow
- Fixed node v.s. floating node
- Schedule
- Global value numbering
- FrameState

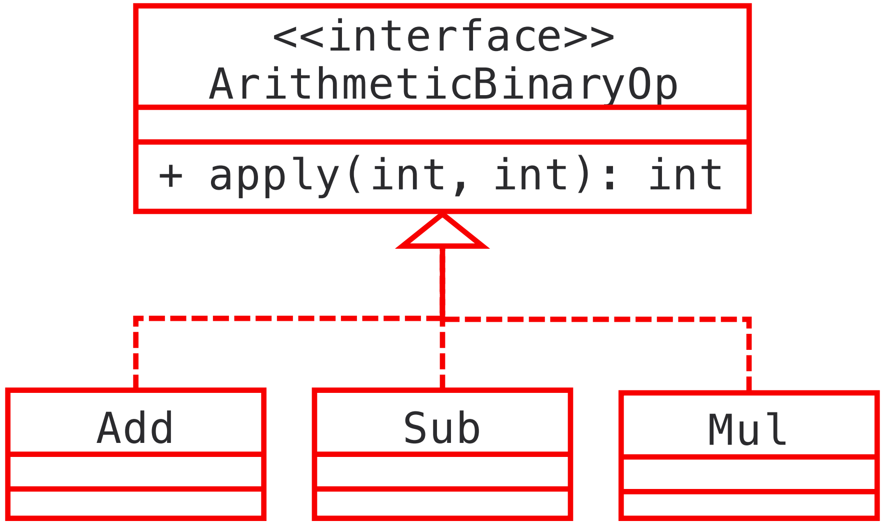
```
int negate(int n) {
  if (n == Integer.MIN_VALUE)
    deoptimize();
  return -n;
}
```



Speculative Optimization: Inlining of Virtual Methods

- What is inlining?

```java
int apply(ArithmeticBinaryOp op,
          int x, int y) {
  return op.apply(x, y); // Add: 50%
                         // Sub: 50%
}
```



<<interface>>
ArithmeticBinaryOp

+ apply(int, int): int

Add

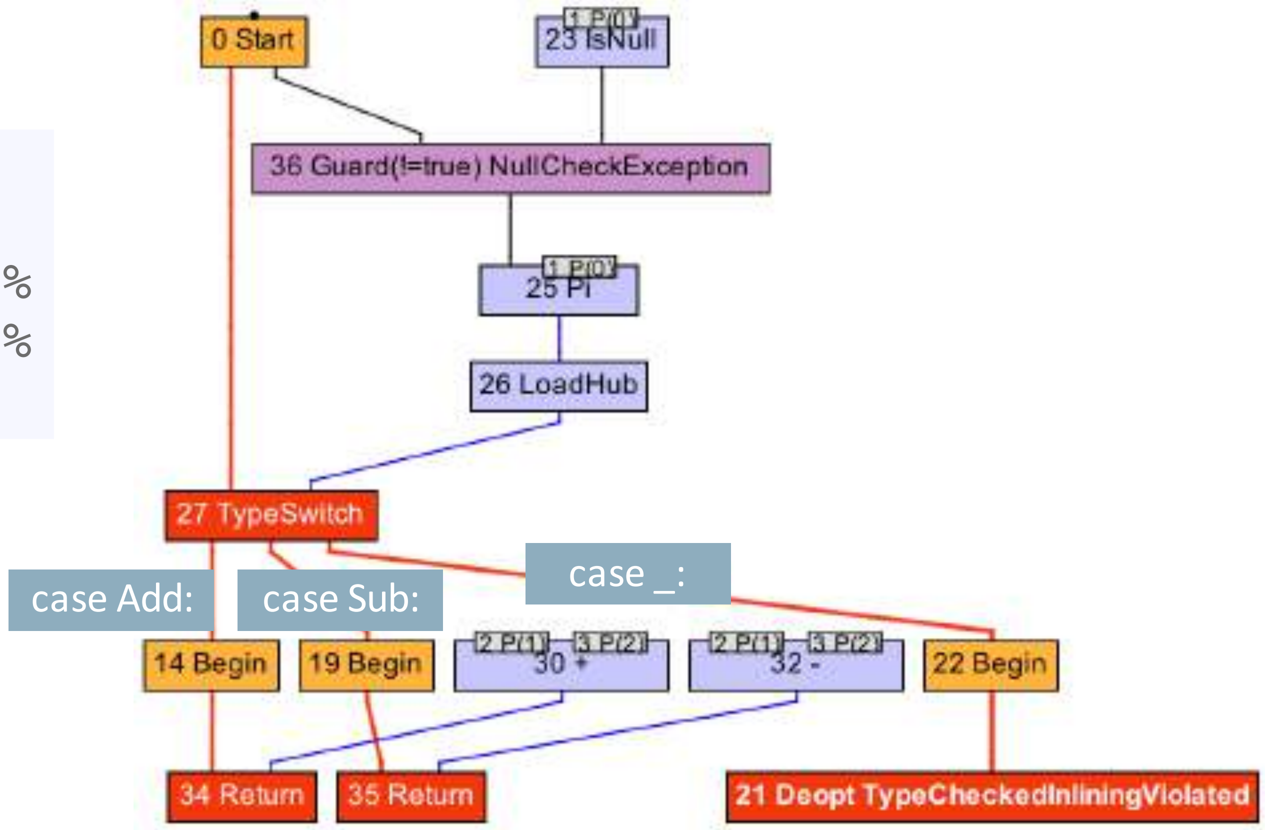
Sub

Mul

- Assumption: the receiver type of this callsite will always be Add/Sub

Speculative Optimization: Inlining of Virtual Methods

```
int apply(ArithmeticBinaryOp op,
            int x, int y) {
  return op.apply(x, y); // Add: 50%
                         // Sub: 50%
}
```



Aggressive Optimization: Partial Escape Analysis

- Escape analysis determines the dynamic scope of an Object
  - – Synchronization elision
  - – Heap allocation -> stack allocation
    - Breaking up objects & scalar replacement

```
int add(int x, int y) {
  return new Add().apply(x, y);
}
```

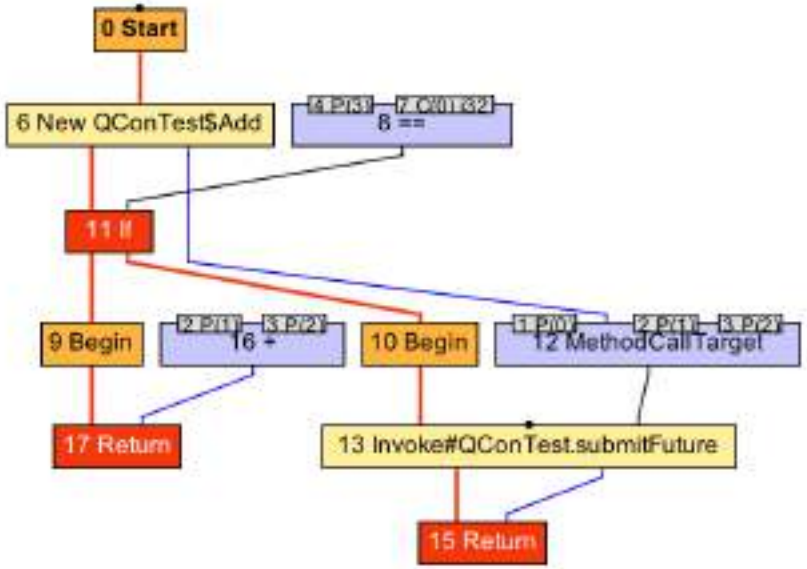
→

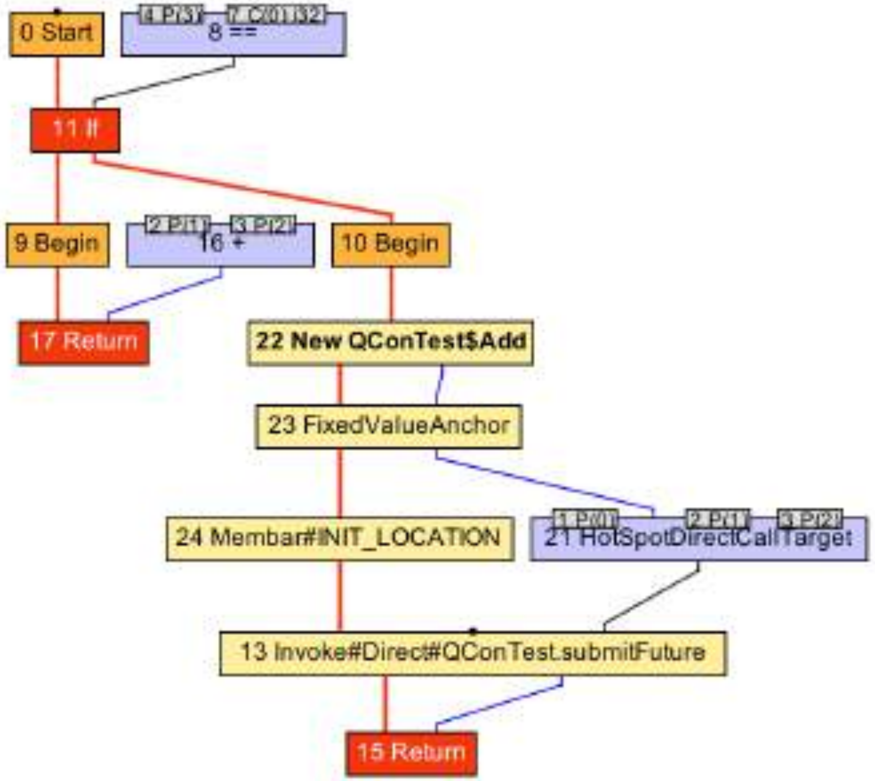
```
int add(int x, int y) {
  return x + y;
}
```

- Partial escape analysis
  - – Control flow sensitive
  - – Defer allocation into sub-branches where needed

Aggressive Optimization: Partial Escape Analysis

```
int add(int x, int y, boolean cond) {
  Add op = new Add();
  if (cond)
    return submitFuture(op, x, y);
  return op.apply(x, y);
}
```

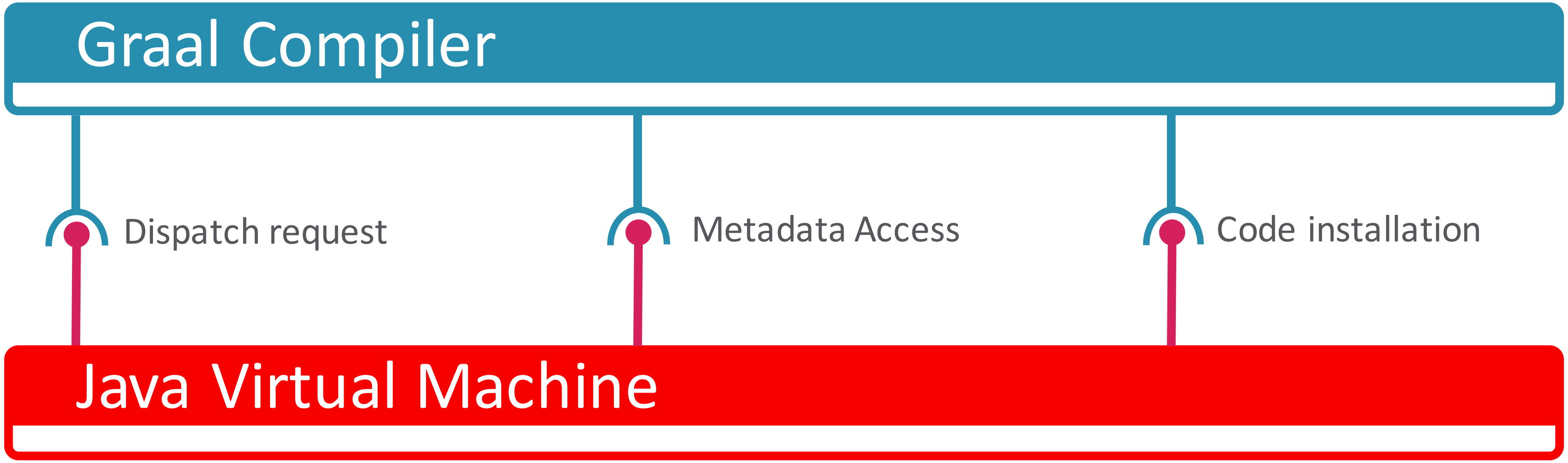




Key Features of Graal

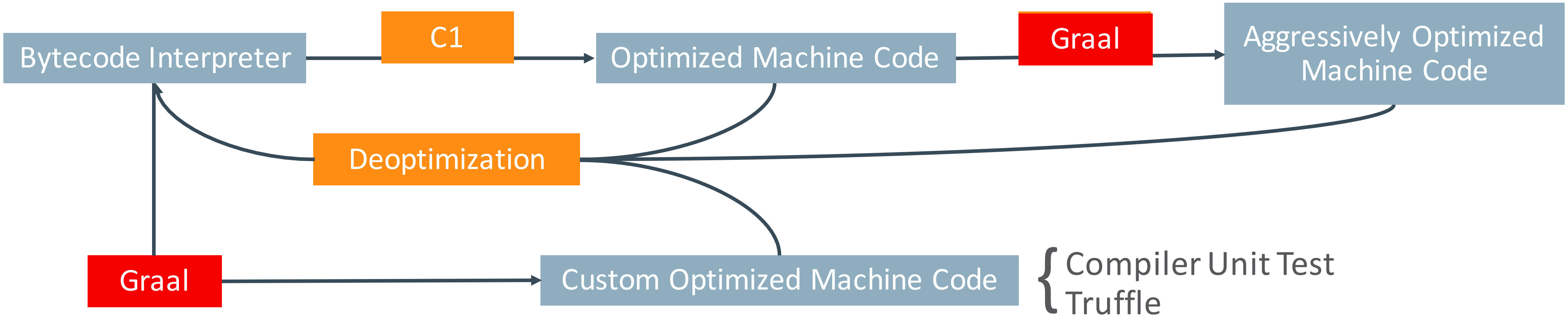
- Designed for aggressive speculative optimizations
  - Specialization based on program profile
  - Metadata for deoptimization is propagated through all optimization phases
    - Target method & bytecode index
    - State of local variables and expression stack
- Graph-based intermediate representation
- Modular architecture
  - Compiler-VM separation

JEP 243: Java-Level JVM Compiler Interface (JVMCI)



Mixed-Mode Execution

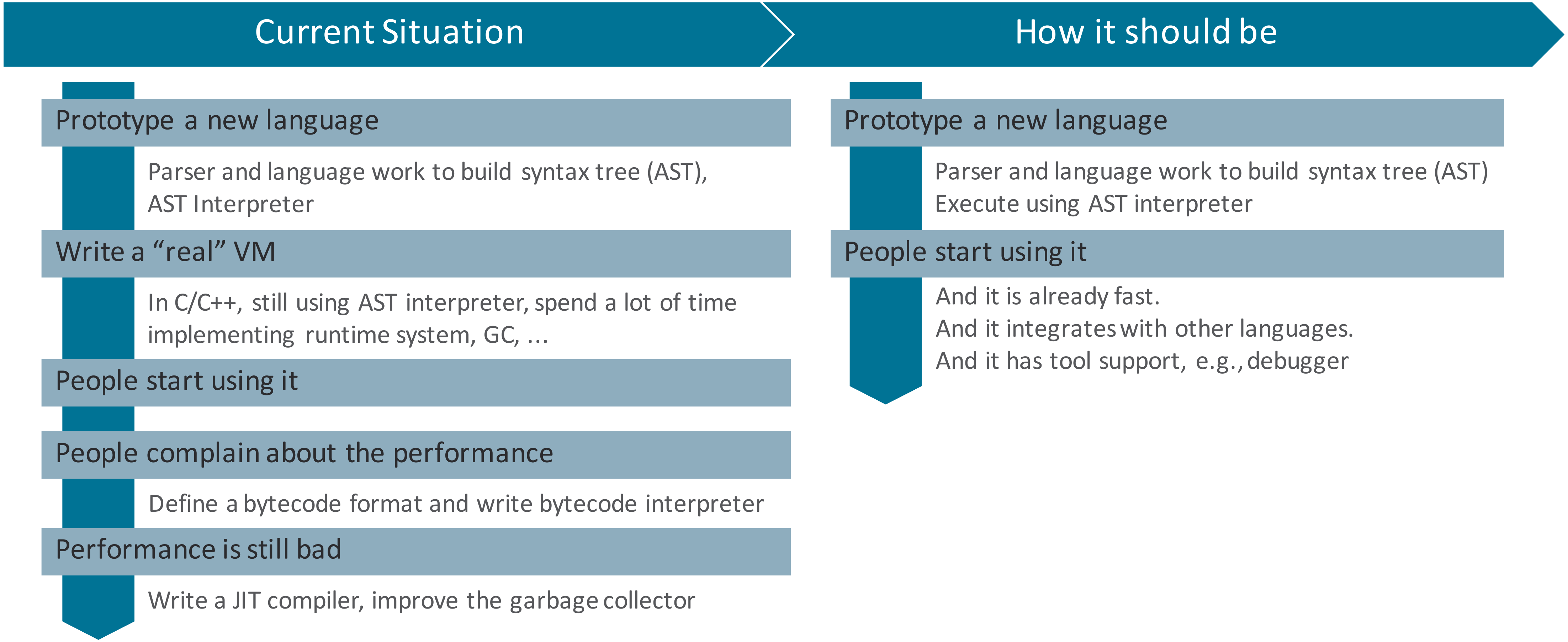
- Default configuration of Java HotSpot VM in production:
- Graal replaces the C2 compiler:
- Graal used only for custom compilations:



Part 2: GraalVM’s Ecosystem

ORACLE

“Write Your Own Language”



Current Situation

Prototype a new language

Parser and language work to build syntax tree (AST),
AST Interpreter

Write a “real” VM

In C/C++, still using AST interpreter, spend a lot of time implementing runtime system, GC, ...

People start using it

People complain about the performance

Define a bytecode format and write bytecode interpreter

Performance is still bad

Write a JIT compiler, improve the garbage collector

How it should be

Prototype a new language

Parser and language work to build syntax tree (AST)
Execute using AST interpreter

People start using it

And it is already fast.
And it integrates with other languages.
And it has tool support, e.g., debugger

Partial Evaluation

```
// Sample program (arg[0] + arg[1]) + arg[2]
sample = new Add(new Add(new Arg(0), new Arg(1)), new Arg(2));
sample.execute(new int[]{0, 1, 2});
```

```
abstract class Node {
  abstract int execute(int[] args);
}
```

```
class Arg extends Node {
  final int index;

  Arg(int i) { this.index = i; }

  int execute(int[] args) {
    return args[index];
  }
}
```

```
class Add extends Node {
  final Node left, right;

  Add(Node left, Node right) {
    this.left = left;
    this.right = right;
  }

  int execute(int[] args) {
    return left.execute(args) +
           right.execute(args);
  }
}
```

Partial Evaluation

```
// Sample program (arg[0] + arg[1]) + arg[2]
sample = new Add(new Add(new Arg(0), new Arg(1)), new Arg(2));
```

```
int interpret(Node node, int[] args) {
  return node.execute(args);
}
```

(`sample`)

Partial Evaluation

```
// Sample program (arg[0] + arg[1]) + arg[2]
sample = new Add(new Add(new Arg(0), new Arg(1)), new Arg(2));
```

```
int interpret(int[] args) {
  return sample .execute(args);
}
```

(

```
class Add
int execute(int[] args) {
  return left.execute(args) +
         right.execute(args);
}
```

)

Partial Evaluation

```
// Sample program (arg[0] + arg[1]) + arg[2]
sample = new Add(new Add(new Arg(0), new Arg(1)), new Arg(2));
```

```
int interpret(int[] args) {
  return sample.left .execute(args) +
         sample.right .execute(args);
}
```

(

```
class Add
int execute(int[] args) {
  return left.execute(args) +
         right.execute(args);
}
```

,

```
class Arg
int execute(int[] args) {
  return args[ index ];
}
```

)

Partial Evaluation

```
// Sample program (arg[0] + arg[1]) + arg[2]
sample = new Add(new Add(new Arg(0), new Arg(1)), new Arg(2));
```

```
int interpret(int[] args) {
  return sample.left.left .execute(args) +
         sample.left.right .execute(args) +
         args[ sample.right.index ];
}
```

(

```
class Arg
int execute(int[] args) {
  return args[ index ];
}
```

)

Partial Evaluation

```
// Sample program (arg[0] + arg[1]) + arg[2]
sample = new Add(new Add(new Arg(0), new Arg(1)), new Arg(2));
```

```
int interpret(int[] args) {
  return args[ sample.left.left.index ] +
         args[ sample.left.right.index ] +
         args[2];
}
```

Partial Evaluation

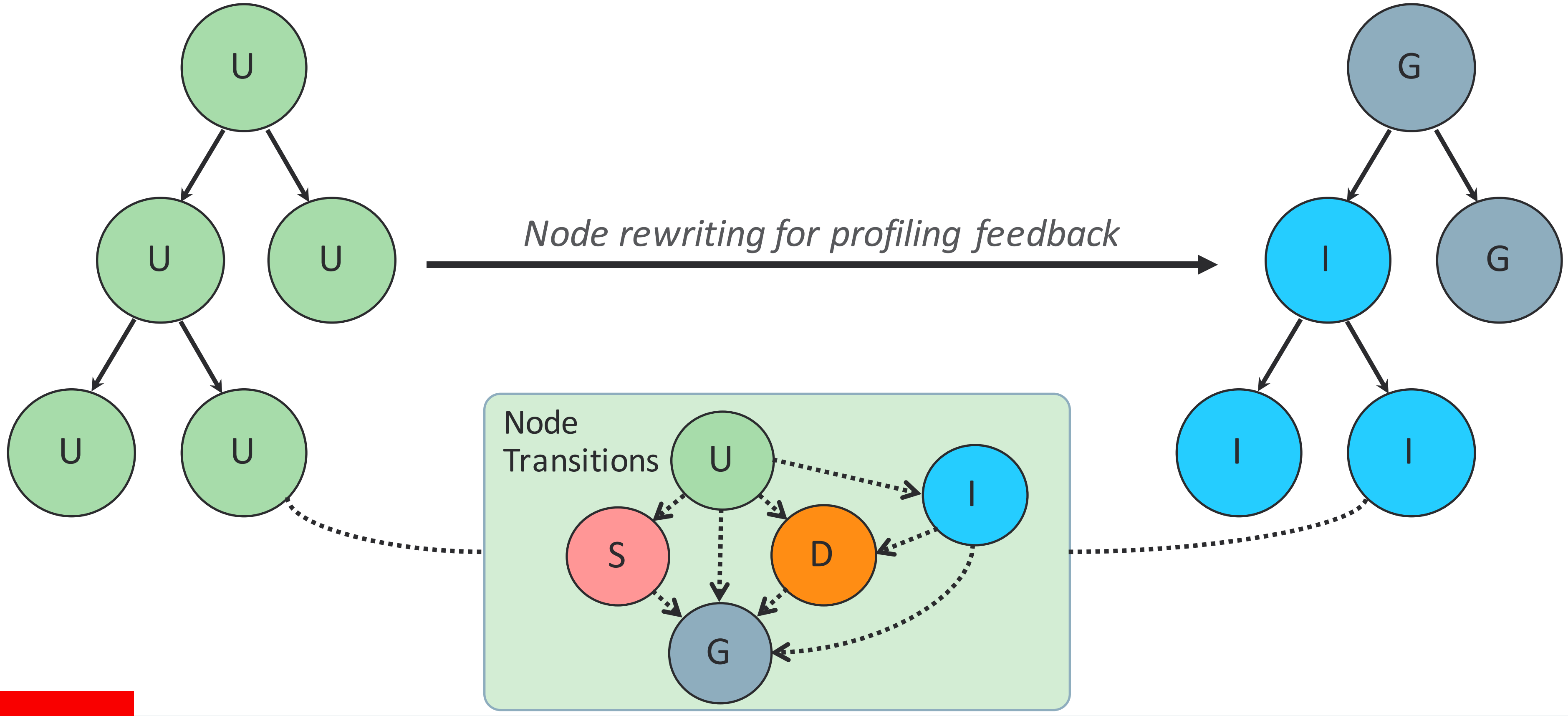
```
// Sample program (arg[0] + arg[1]) + arg[2]
sample = new Add(new Add(new Arg(0), new Arg(1)), new Arg(2));
```

```
int interpret(int[] args) {
  return args[0] + args[1] + args[2];
}
```

Truffle

- A Language Implementation Framework that uses Graal for Custom Compilation

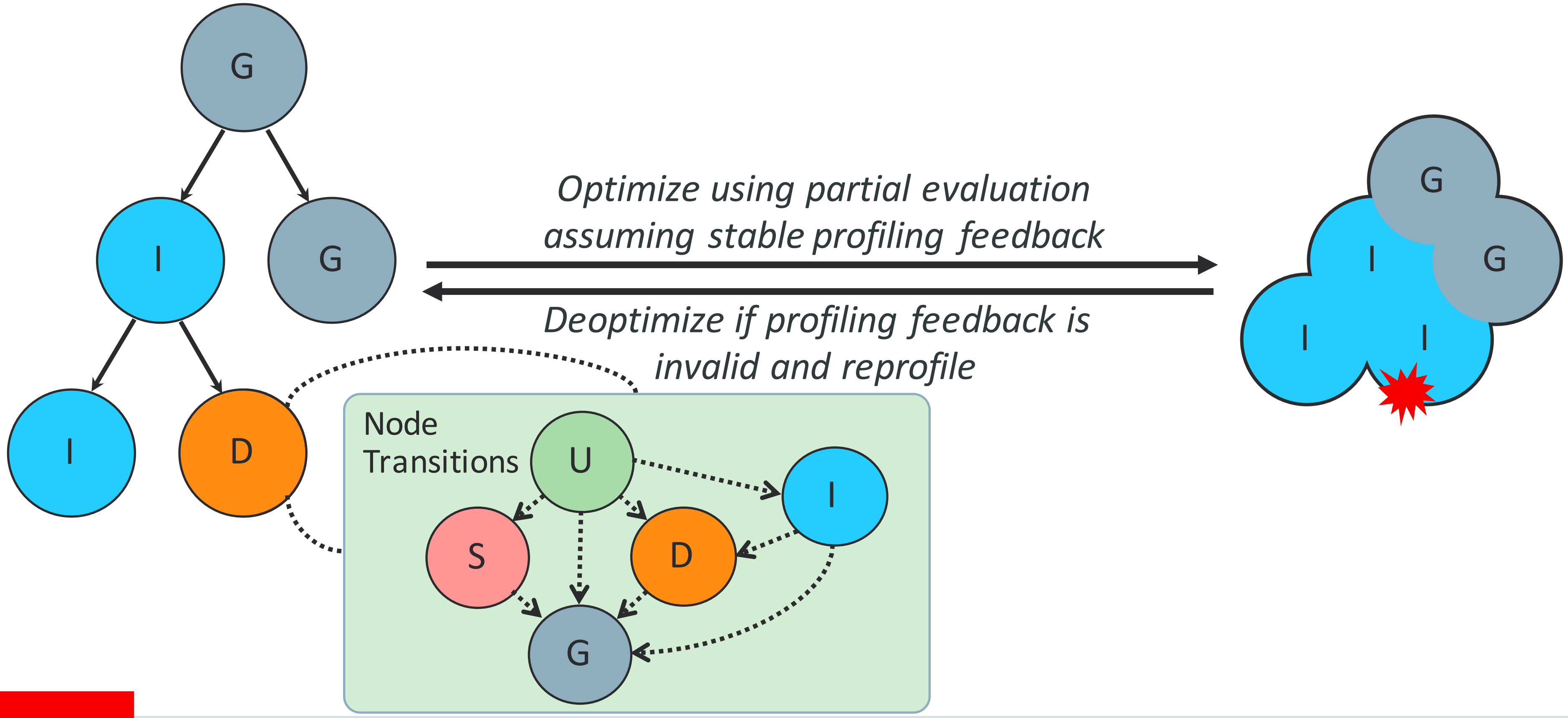
```
// Sample program (arg[0] + arg[1]) + arg[2]
```



Truffle

- A Language Implementation Framework that uses Graal for Custom Compilation

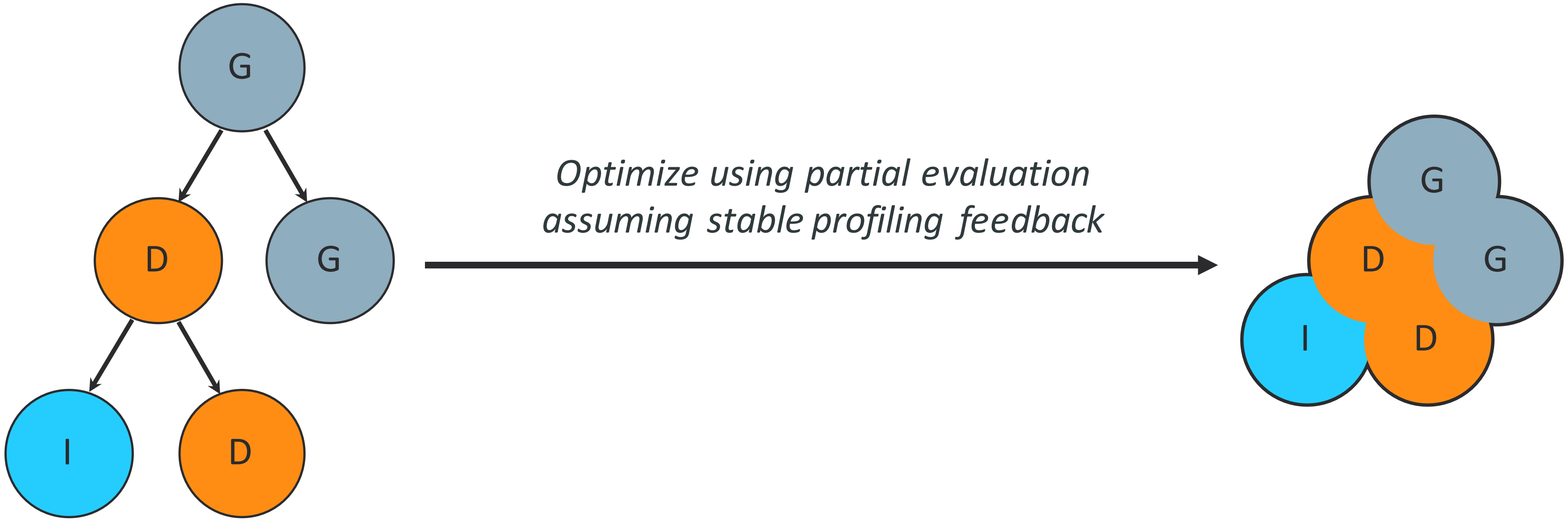
```
// Sample program (arg[0] + arg[1]) + arg[2]
```


G
I
G
I
D
Optimize using partial evaluation assuming stable profiling feedback
Deoptimize if profiling feedback is invalid and reprofile
Node Transitions
U
S
D
I
G
G
I
G
I
I

Truffle

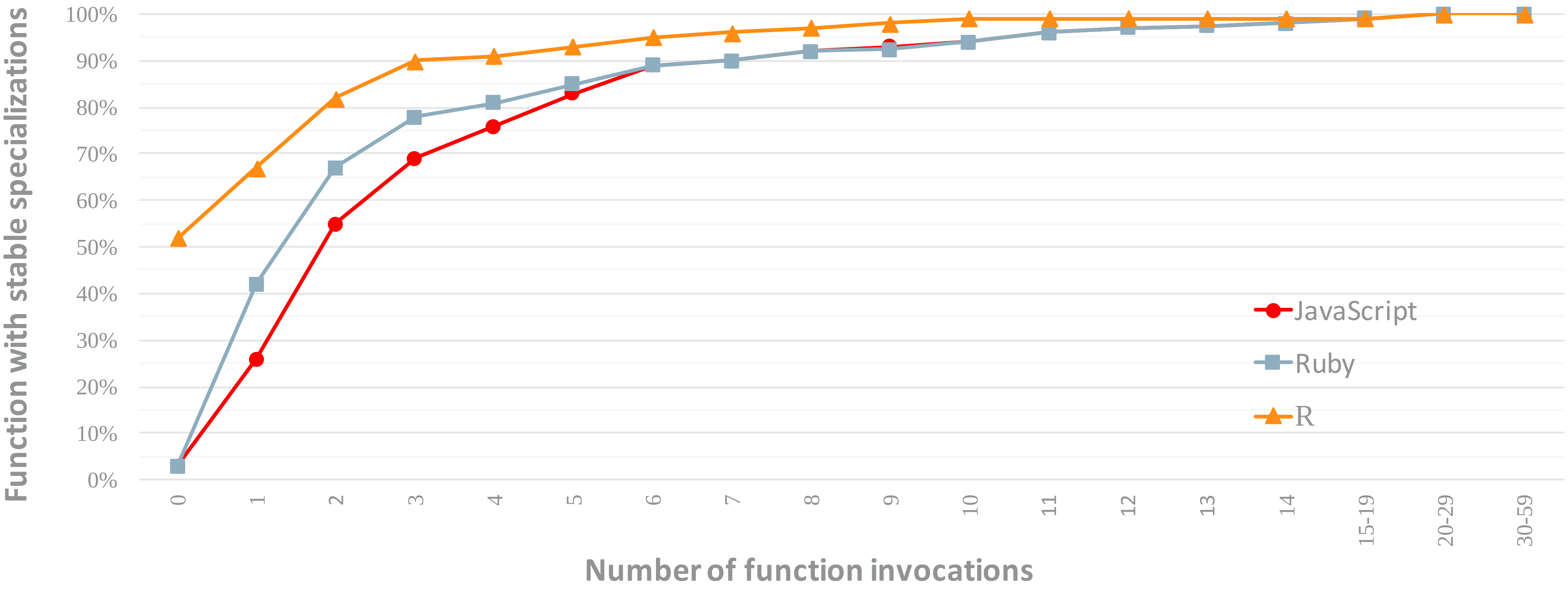
- A Language Implementation Framework that uses Graal for Custom Compilation

```
// Sample program (arg[0] + arg[1]) + arg[2]
```



Optimize using partial evaluation assuming stable profiling feedback

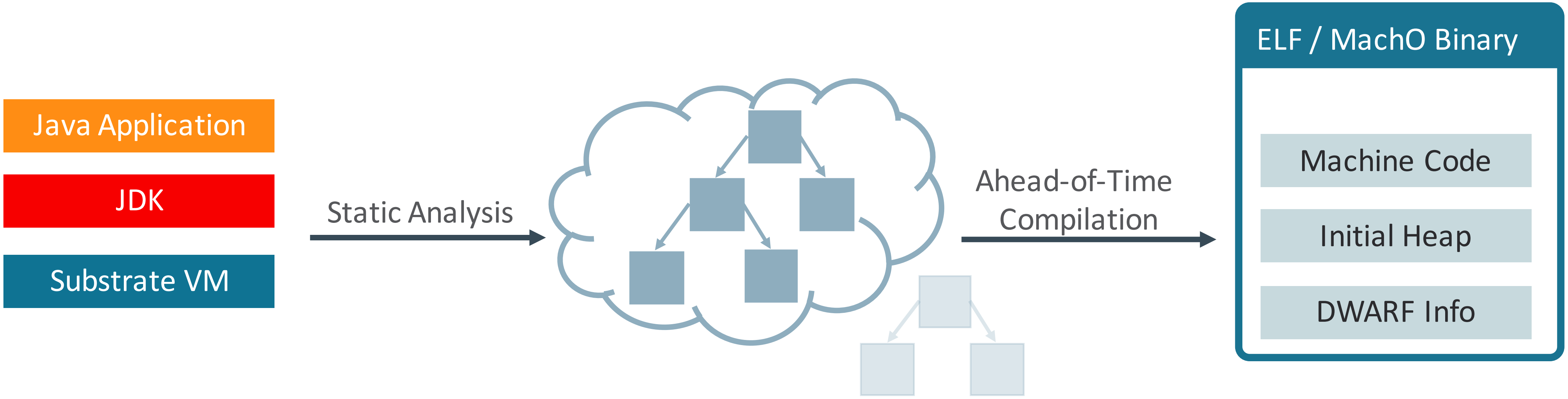
Stability



Function with stable specializations
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15-19
20-29
30-59
Number of function invocations
JavaScript
Ruby
R

Substrate VM

Static Analysis and Ahead-of-Time Compilation using Graal



All Java classes from application, JDK, and Substrate VM

Reachable methods, fields, and classes

Application running without dependency on JDK and without Java class loading

"Hello World" in C, Java

	C	GNU	Java/JVM	Java/SVM
🕒	<10 ms	<10 ms	40 ms	<10 ms
RAM	450 KB	800 KB	24 MB	850 KB
101 011	100 K	300 K	140 M	220 K

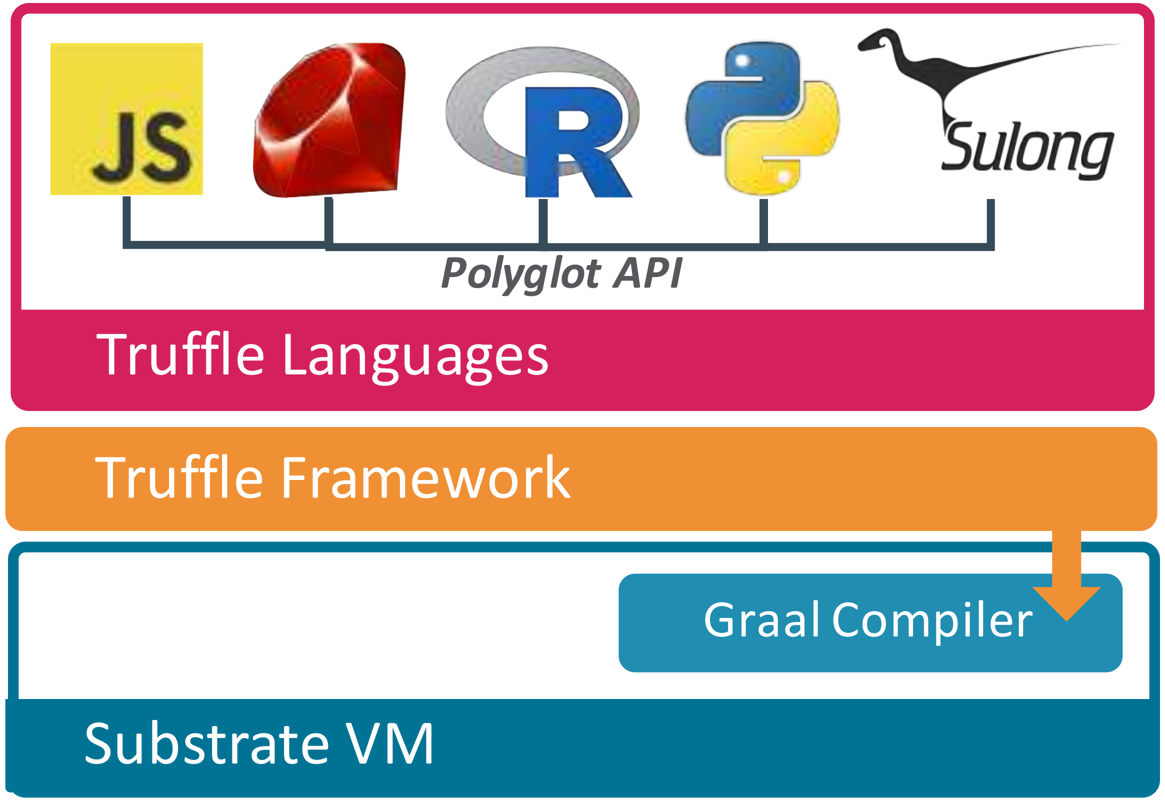
* *Operating system*: **Linux**; *Time,* Memory: **/usr/bin/time** ... ; *Instructions*: **valgrind** --tool=callgrind ...;

"Hello World" in JavaScript

	V8	Spider Monkey	Nashorn	SVM
🕒	<10 ms	30 ms	450 ms	<10 ms
RAM	18 MB	10 MB	56 MB	4 MB
101 011	10 M	77 M	N/A	520 K

* *Operating system*: **Linux**; *Time,* Memory: **/usr/bin/time** ... ; *Instructions*: **valgrind** --tool=callgrind ...;

Embeddability

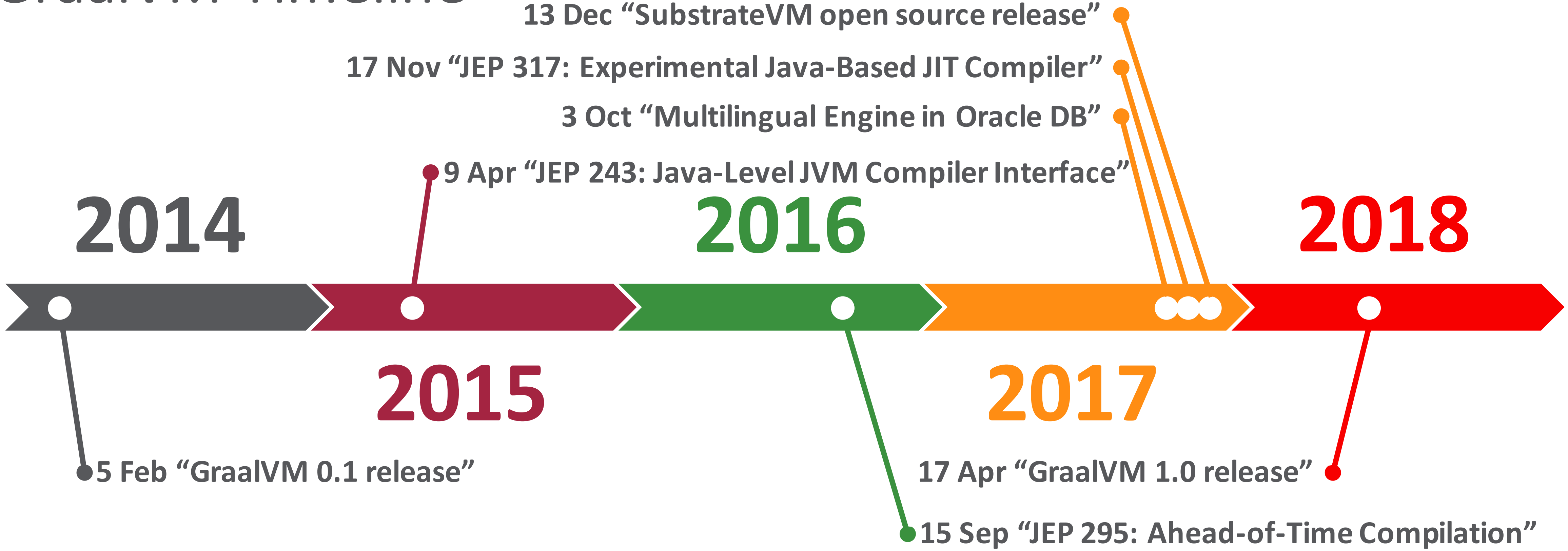


ORACLE
DATABASE

* http://www.oracle.com/technetwork/database/multilingual-engine/downloads/index.html

Summary

GraalVM Timeline



5 Feb “GraalVM 0.1 release”

9 Apr “JEP 243: Java-Level JVM Compiler Interface”

15 Sep “JEP 295: Ahead-of-Time Compilation”

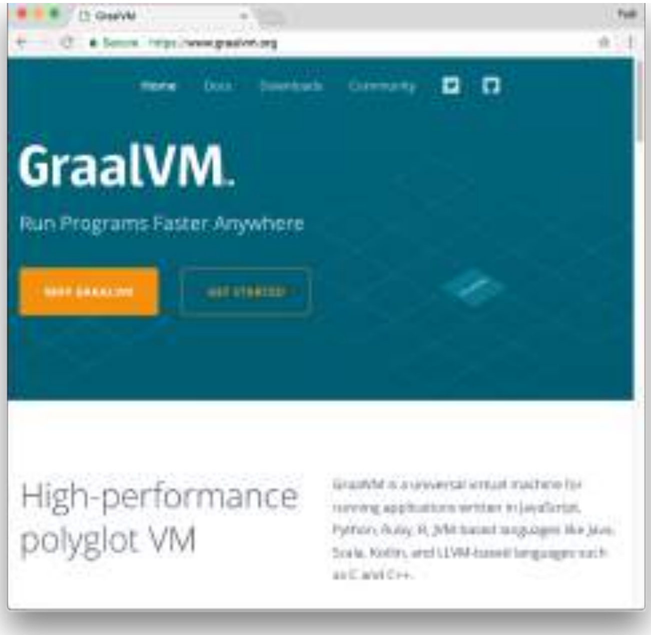
3 Oct “Multilingual Engine in Oracle DB”

17 Nov “JEP 317: Experimental Java-Based JIT Compiler”

13 Dec “SubstrateVM open source release”

17 Apr “GraalVM 1.0 release”

Get Started

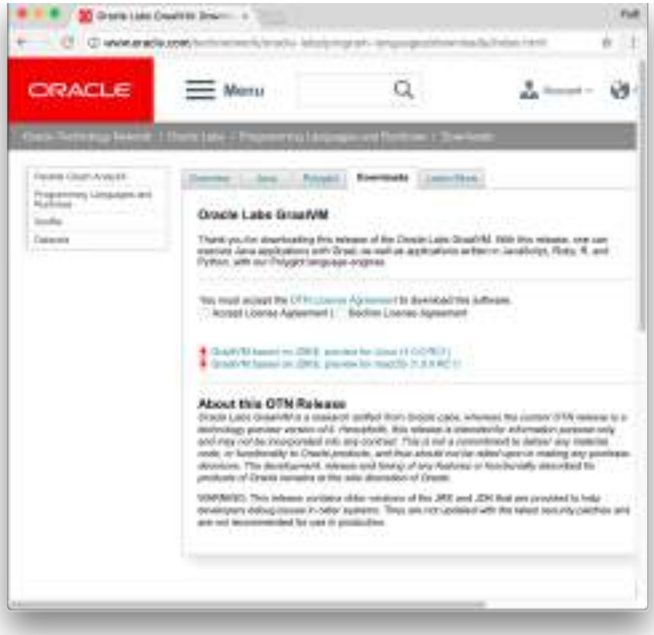


High-performance polyglot VM

GraalVM is a universal virtual machine for running applications written in JavaScript, Python, Ruby, R, JVM-based languages like Java, Scala, Kotlin, and LLVM-based languages such as C and C++.

Documentation

https://www.graalvm.org/

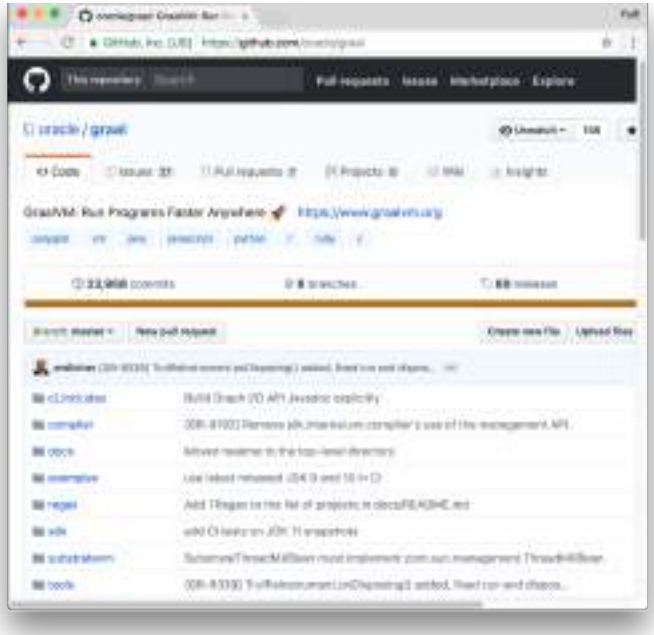


Oracle Labs GraalVM

About this OTN Release

Enterprise Release

Search for "OTN Graal"



GraalVM: Run Programs Faster Anywhere 🚀 https://www.graalvm.org

Open source on GitHub

https://github.com/oracle/graal

ORACLE®

扫码关注

获取更多培训信息