ORACLE®

Polyglot on the JVM with Graal

Vojin Jovanovic
VM Research Group, Oracle Labs

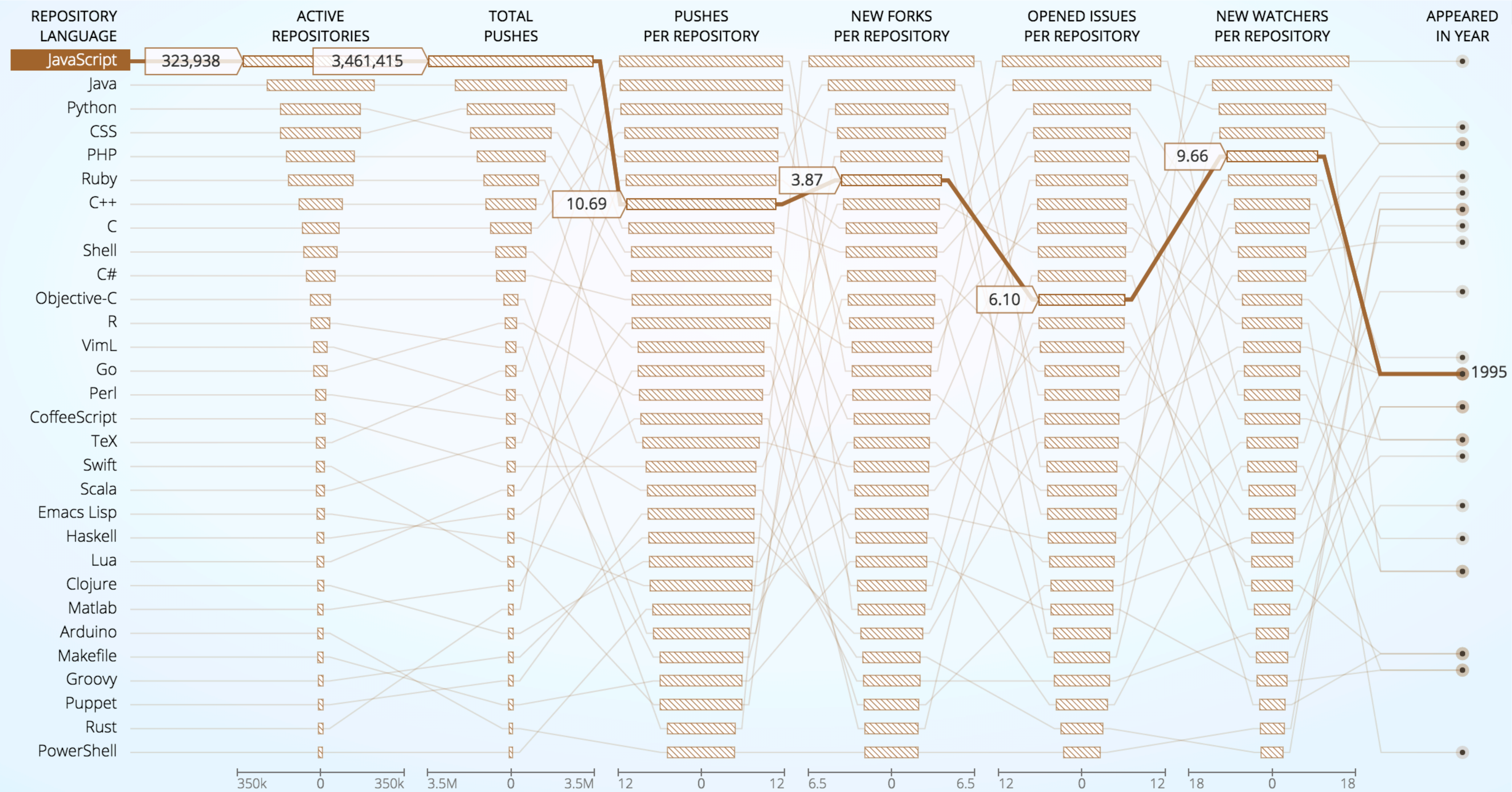
Github: @vjovanov
Twitter: @vojjov

ORACLE

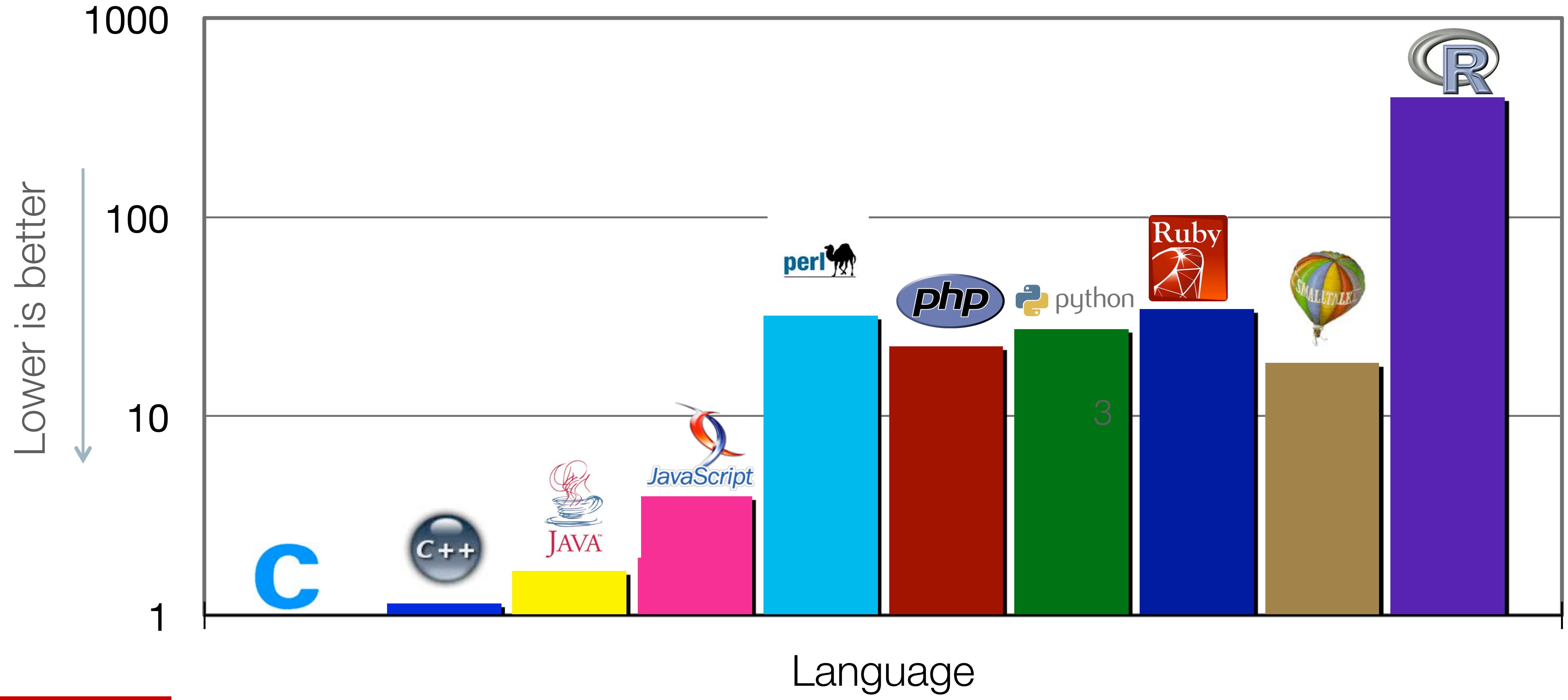
Safe Harbor Statement

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


REPOSITORY LANGUAGE
ACTIVE REPOSITORIES
TOTAL PUSHES
PUSHES PER REPOSITORY
NEW FORKS PER REPOSITORY
OPENED ISSUES PER REPOSITORY
NEW WATCHERS PER REPOSITORY
APPEARED IN YEAR
JavaScript
323,938
3,461,415
10.69
3.87
6.10
9.66
1995
Java
Python
CSS
PHP
Ruby
C++
C
Shell
C#
Objective-C
R
VimL
Go
Perl
CoffeeScript
TeX
Swift
Scala
Emacs Lisp
Haskell
Lua
Clojure
Matlab
Arduino
Makefile
Groovy
Puppet
Rust
PowerShell
350k 0 350k
3.5M 0 3.5M
12 0 12
6.5 0 6.5
12 0 12
18 0 18

The World is Polyglot: What About Performance?



“Write Your Own Language”

Current situation	How it should be

Current situation

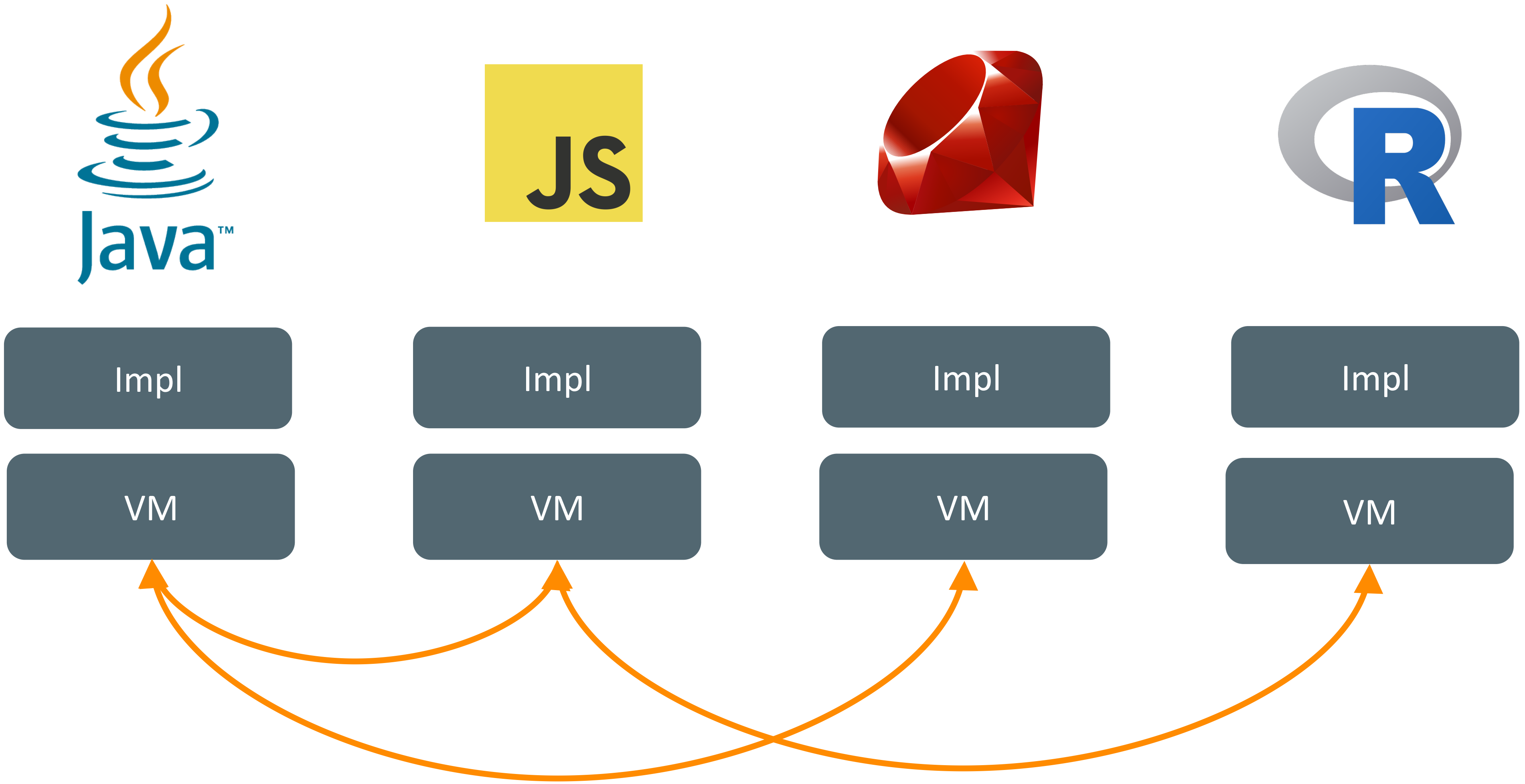
- Prototype a new language
  - Parser and language work to build syntax tree (AST), AST Interpreter
- Write a “real” VM
  - In C/C++, still using AST interpreter, spend a lot of time implementing runtime system, GC, …
- People start using it
- People complain about performance
  - Define a bytecode format and write bytecode interpreter
- Performance is still bad
  - Write a JIT compiler, improve the garbage collector

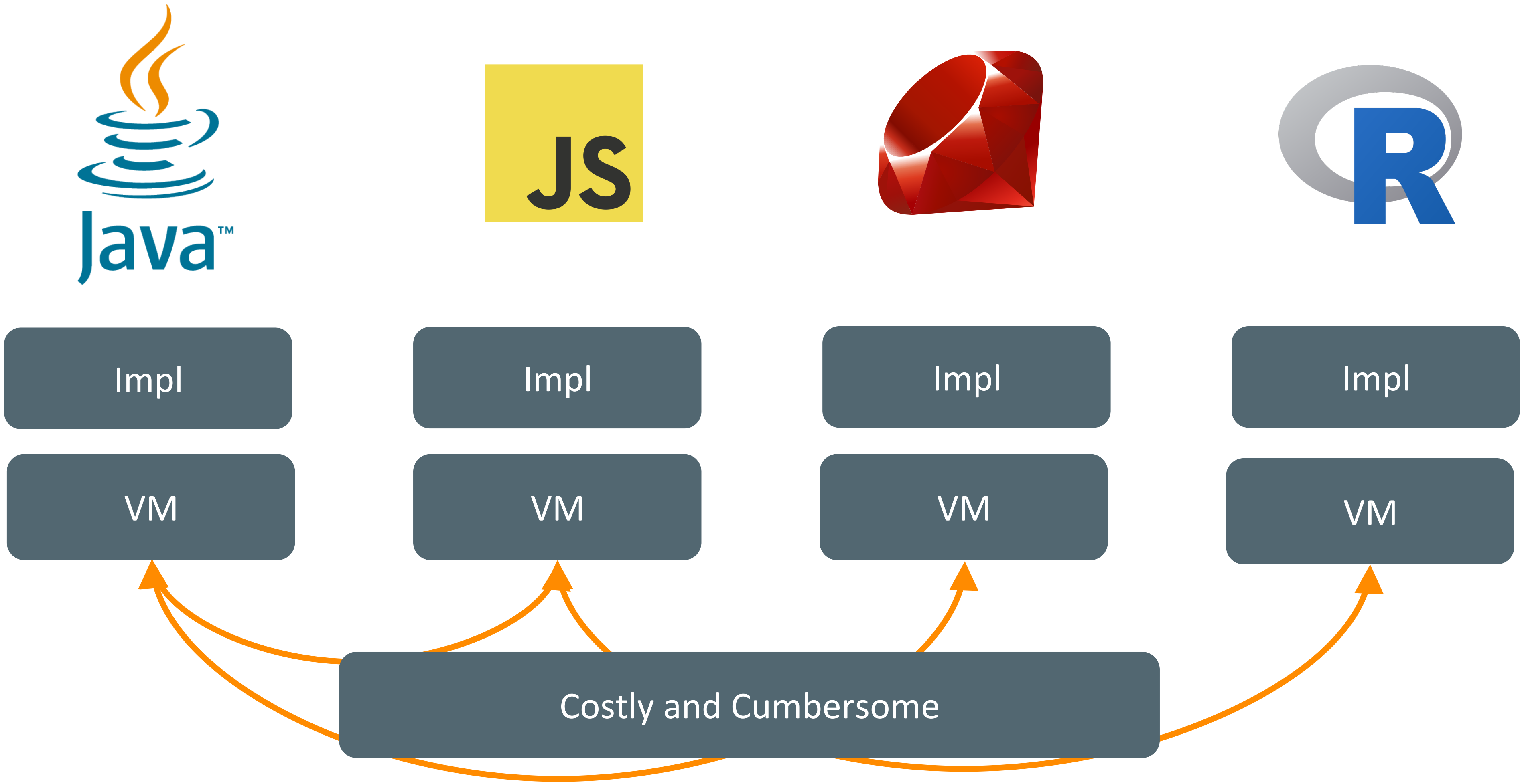
How it should be

- Prototype a new language in Java
  - 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., a debugger**

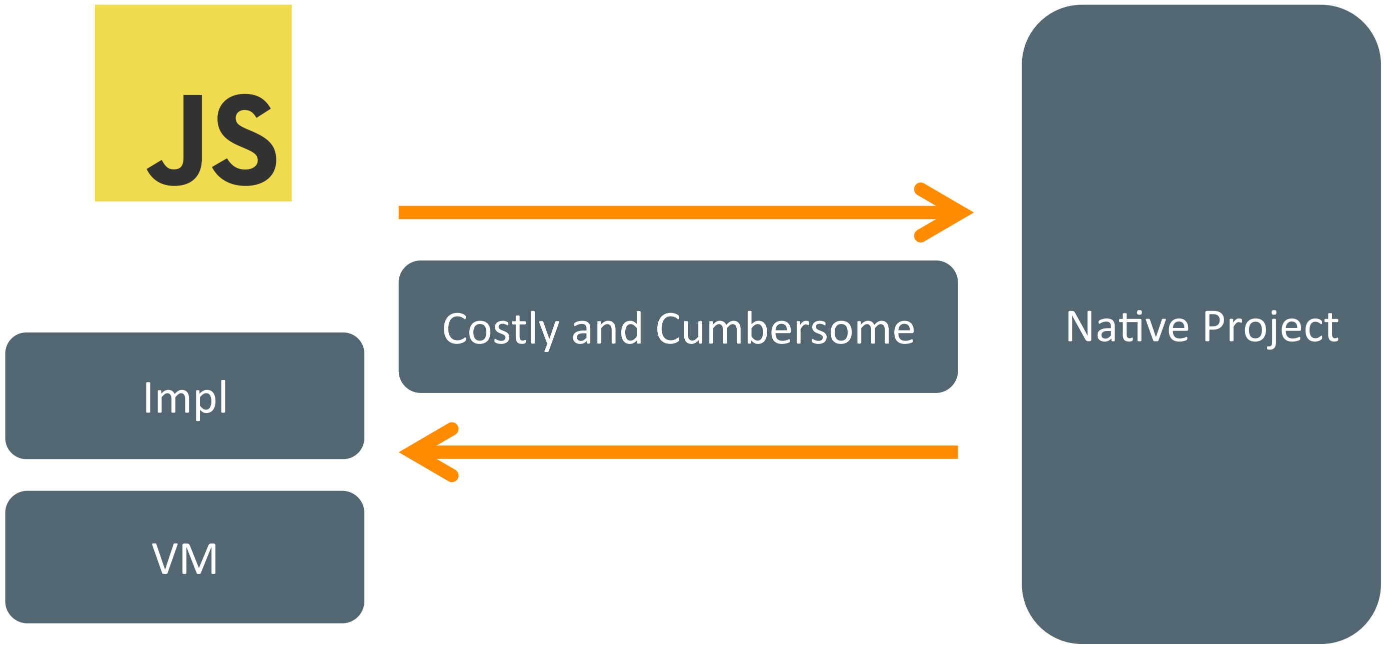
ORACLE

Java	JS	Ruby	R
Impl	Impl	Impl	Impl
VM	VM	VM	VM


Java
JS
Impl
Impl
Impl
Impl
VM
VM
VM
VM


Java™
JS
Impl
Impl
Impl
Impl
VM
VM
VM
VM
Costly and Cumbersome

Communication with Native Code is Expensive



Summary

- Fast languages are hard to implement
- Interoperability between the languages is cumbersome and costly
- Barrier between languages and native projects

Graal: One Compiler for Managed Languages

Graal VM Architecture

Java™ Scala Groovy

Graal Compiler

JVM Compiler Interface (JVMCI) JEP 243

Java HotSpot Runtime

ORACLE®

Key Features of Graal

- Written in Java
  - Eases development and maintenance
- Modular architecture
  - Configurable compiler phases
  - Compiler-VM separation: snippets, provider interfaces
- 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

ORACLE

Example Optimization: Partial Escape Analysis (1)

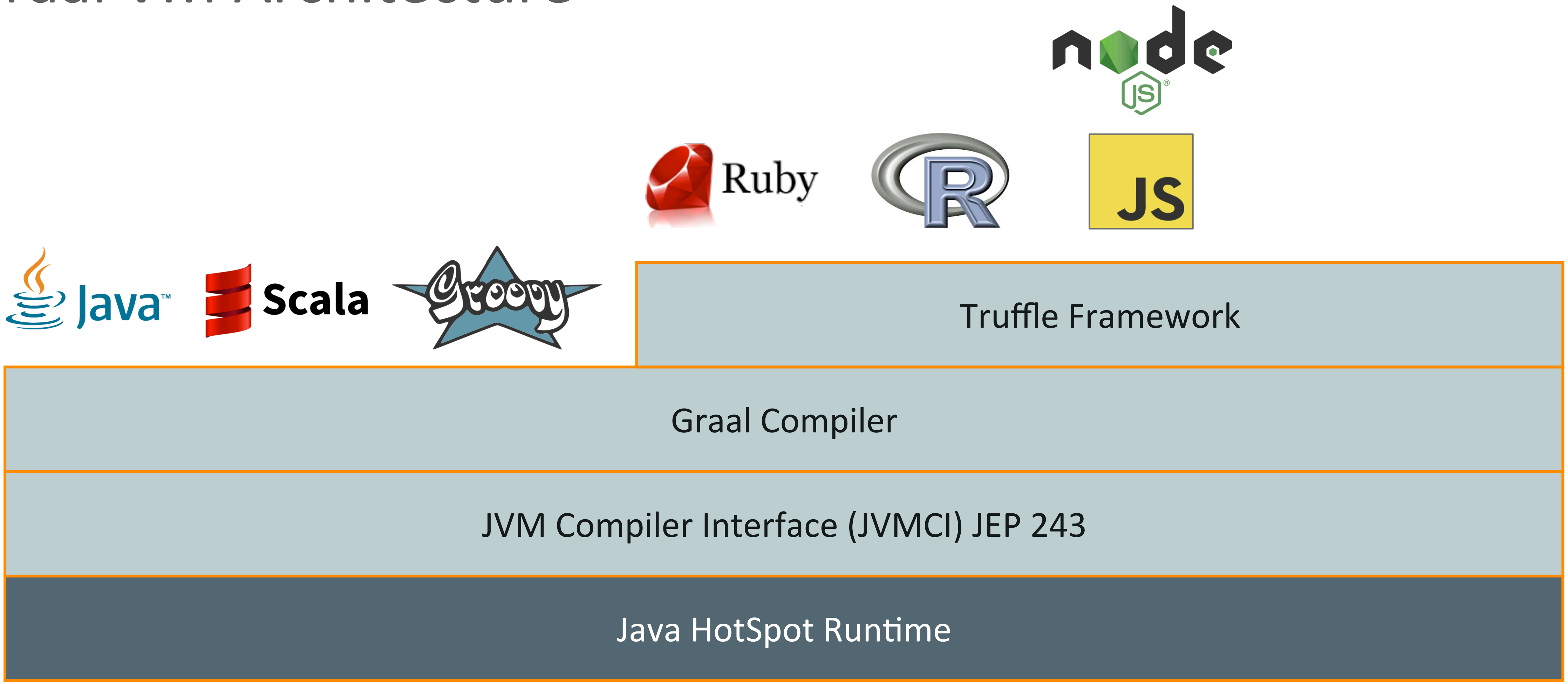
```java
public static Car getCached(int hp, String name) {
    Car car = new Car(hp, name, null);
    Car cacheEntry = null;
    for (int i = 0; i < cache.length; i++) {
        if (car.hp == cache[i].hp &&
              car.name == cache[i].name) {
            cacheEntry = cache[i];
            break;
        }
    }
    if (cacheEntry != null) {
        return cacheEntry;
    } else {

        addToCache(car);
        return car;
    }
}
```

Example Optimization: Partial Escape Analysis (2)

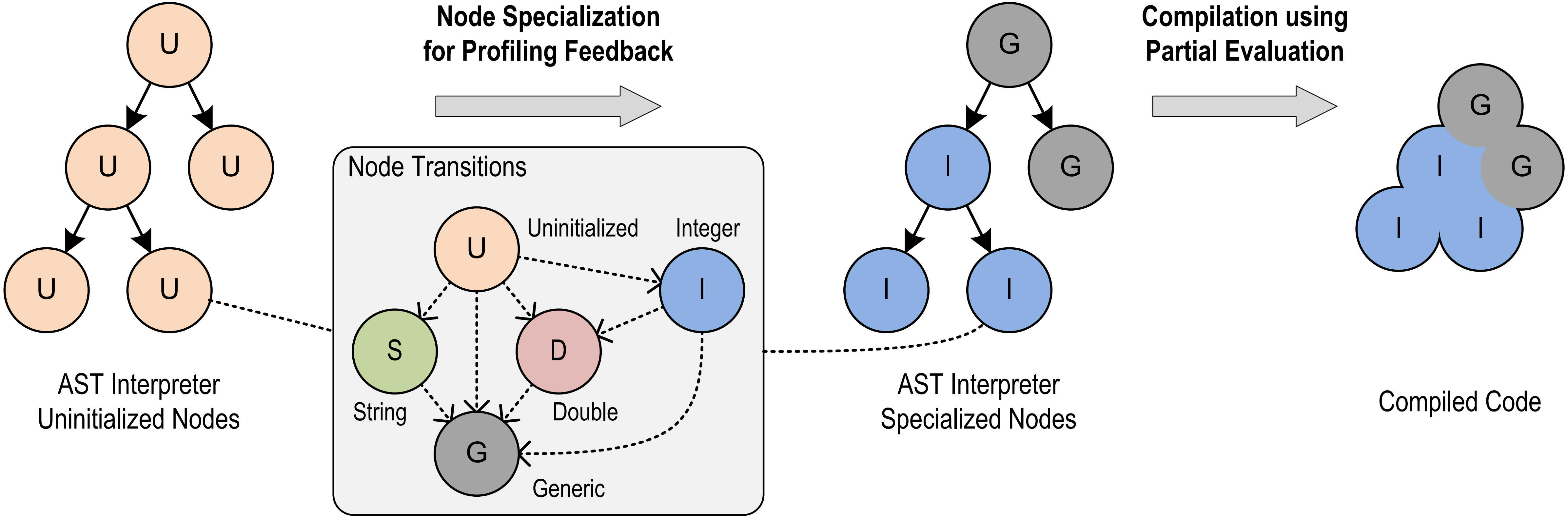
```
public static Car getCached(int hp, String name) {

    Car cacheEntry = null;
    for (int i = 0; i < cache.length; i++) {
        if (hp == cache[i].hp &&
              name == cache[i].name) {
            cacheEntry = cache[i];
            break;
        }
    }
    if (cacheEntry != null) {
        return cacheEntry;
    } else {
        Car car = new Car(hp, name, null);
        addToCache(car);
        return car;
    }
}
```

- `new Car(...)` escapes at:
  - `addToCache(car);`
  - `return car;`
- Might be a very unlikely path
- No allocation in frequent path

Graal VM Architecture

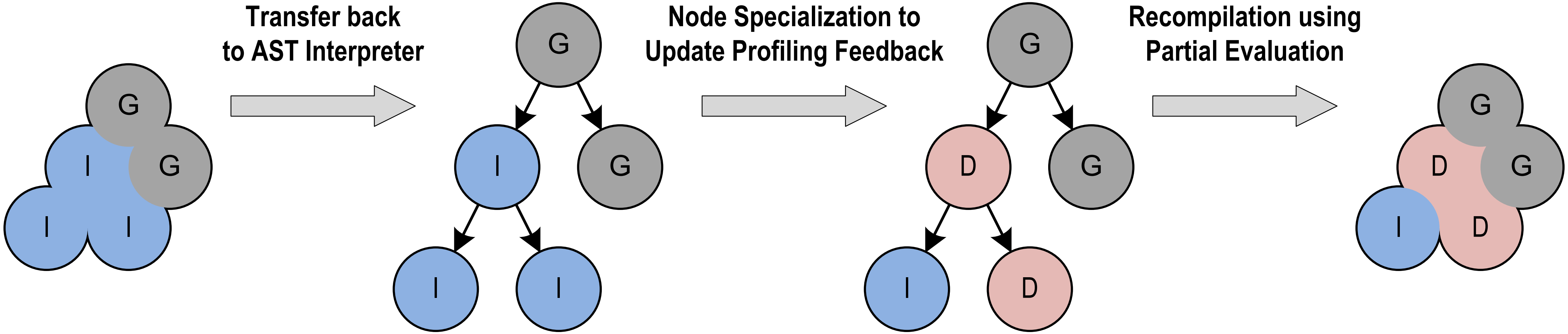


Speculate and Optimize …



ORACLE

… and Transfer to Interpreter and Reoptimize!



How effective is this approach?

```ruby
def add(a, b)
  a + b
end

def sum(n)
  i = 0
  a = 0
  while i < n
    i += 1
    a = add(a, n)
  end
  a
end
```

Looking at this loop here

ORACLE

```ruby
def add(a, b)
  a + b
end

def sum(n)
  i = 0
  a = 0
  while i < n
    i += 1
    a = add(a, n)
  end
  a
end
```

```
0x0000000103a7dc70: mov     esi,edi
0x0000000103a7dc72: add     esi,r9d
0x0000000103a7dc75: jo      0x0000000103a7dda2
0x0000000103a7dc7b: inc     ecx
0x0000000103a7dc7d: mov     edi,esi
0x0000000103a7dc7f: cmp     r9d,ecx
0x0000000103a7dc82: jg      0x0000000103a7dc70
```

```
def add(a, b)
  a + b
end
```

```
0x0000000103a7dc70: mov     esi,edi
0x0000000103a7dc72: add     esi,r9d
0x0000000103a7dc75: jo      0x0000000103a7dda2
0x0000000103a7dc7b: inc     ecx
0x0000000103a7dc7d: mov     edi,esi
0x0000000103a7dc7f: cmp     r9d,ecx
0x0000000103a7dc82: jg      0x0000000103a7dc70
```

```
  end
  a
end
```

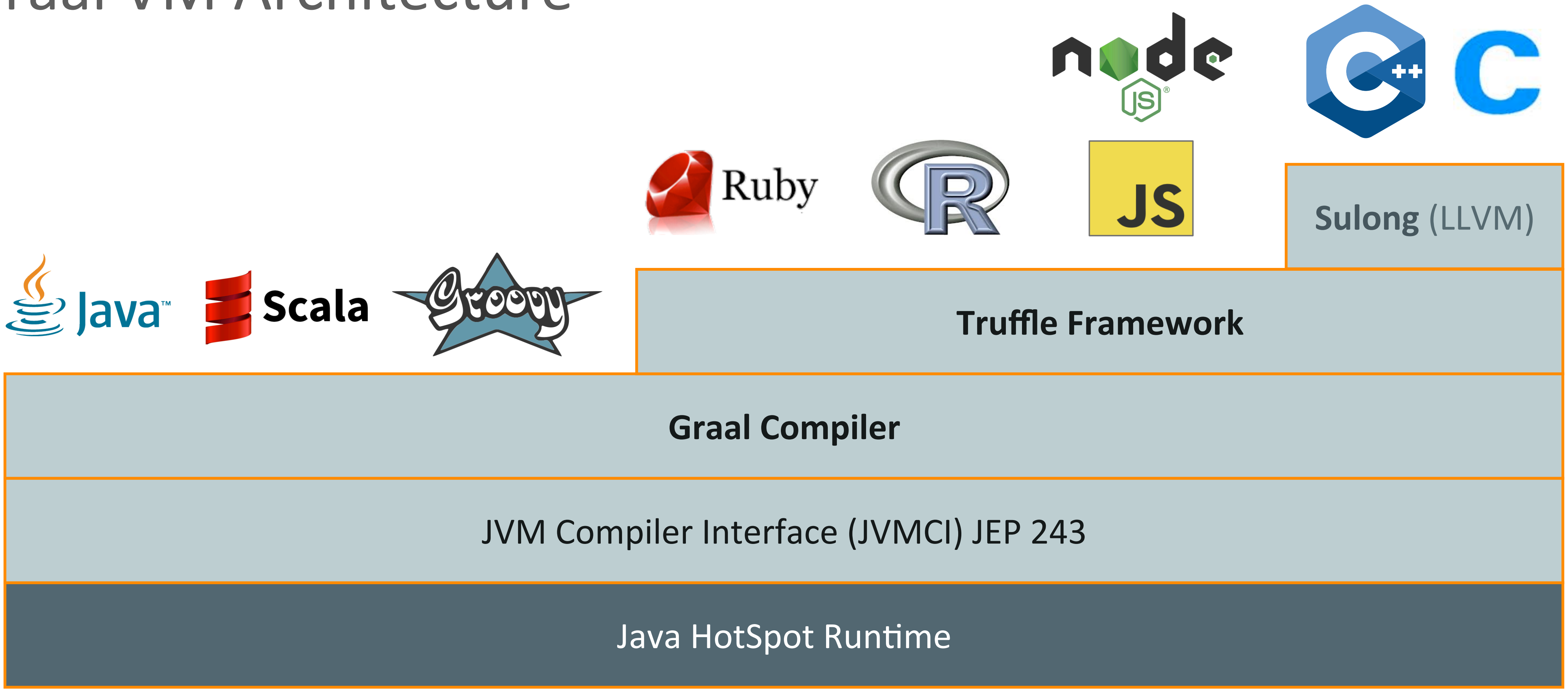
0x0000000103a7dc82: jg   0x0000000103a7dc70

```ruby
def add(a, b)
  a + b
end
```

```
0x0000000103a7dc70: mov     esi,edi
0x0000000103a7dc72: add     esi,r9d
0x0000000103a7dc75: jo      0x0000000103a7dda2
0x0000000103a7dc7b: inc     ecx
0x0000000103a7dc7d: mov     edi,esi
0x0000000103a7dc7f: cmp     r9d,ecx
0x0000000103a7dc82: jg      0x0000000103a7dc70
```

```ruby
  end
  a
end
```

Graal VM Architecture



Sulong

- Enable LLVM bitcode as just another ”Truffle language”
- Why?
  - Particular interest in running C, C++, and Fortran programs.
  - High-performance native extensions for managed languages.
  - Low overhead of security-related instrumentations such as bounds checks.
  - Apply dynamic optimization techniques to static context.

```
FUNCTION add(x, y)
  INTEGER :: add
  INTEGER :: a
  INTEGER :: b
  add = a + b
  RETURN
END FUNCTION
```

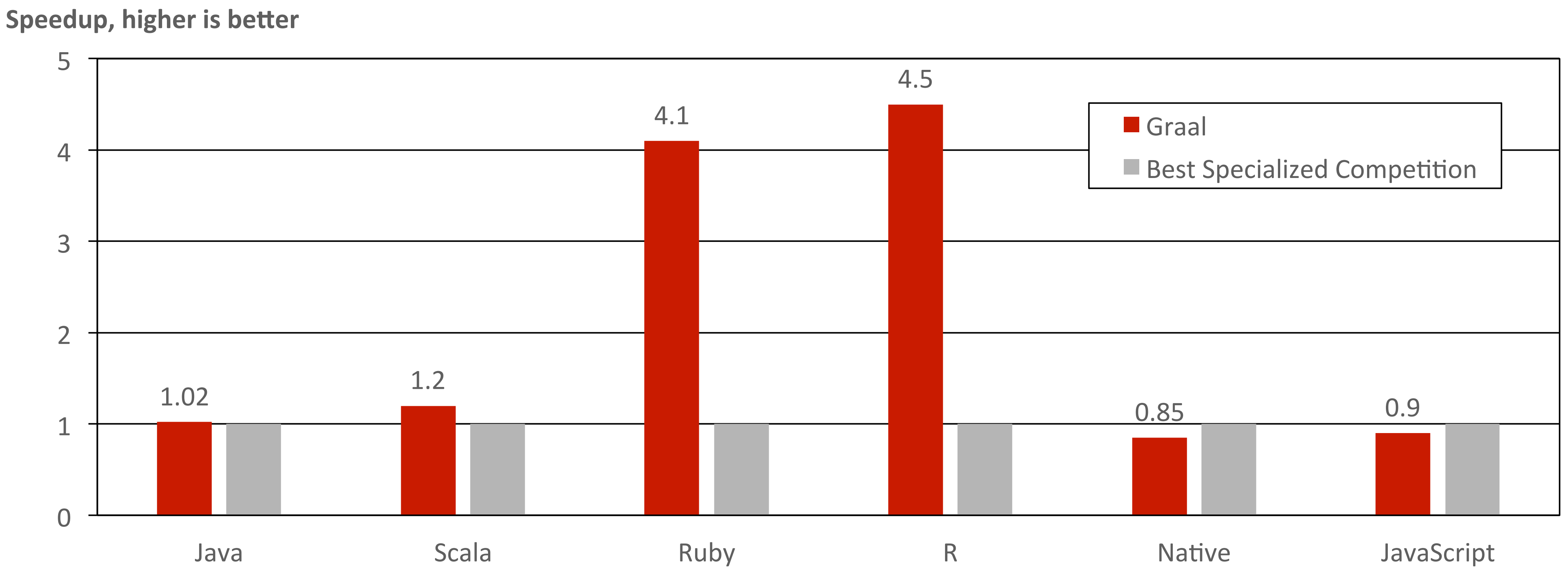
LLVM frontend →

```
define i32 @add(i32 %x, i32 %y) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  store i32 %y, i32* %2, align 4
  %3 = load i32* %1, align 4
  %4 = load i32* %2, align 4
  %5 = add nsw i32 %3, %4
  ret i32 %5
}
```

→ Graal VM via Truffle

ORACLE

Performance: Graal VM



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

Completeness

99% Ruby language

JRuby passes 94%

99% ECMA Script 2015

Missing Unicode Regexes

96% Ruby core library

JRuby passes 95%

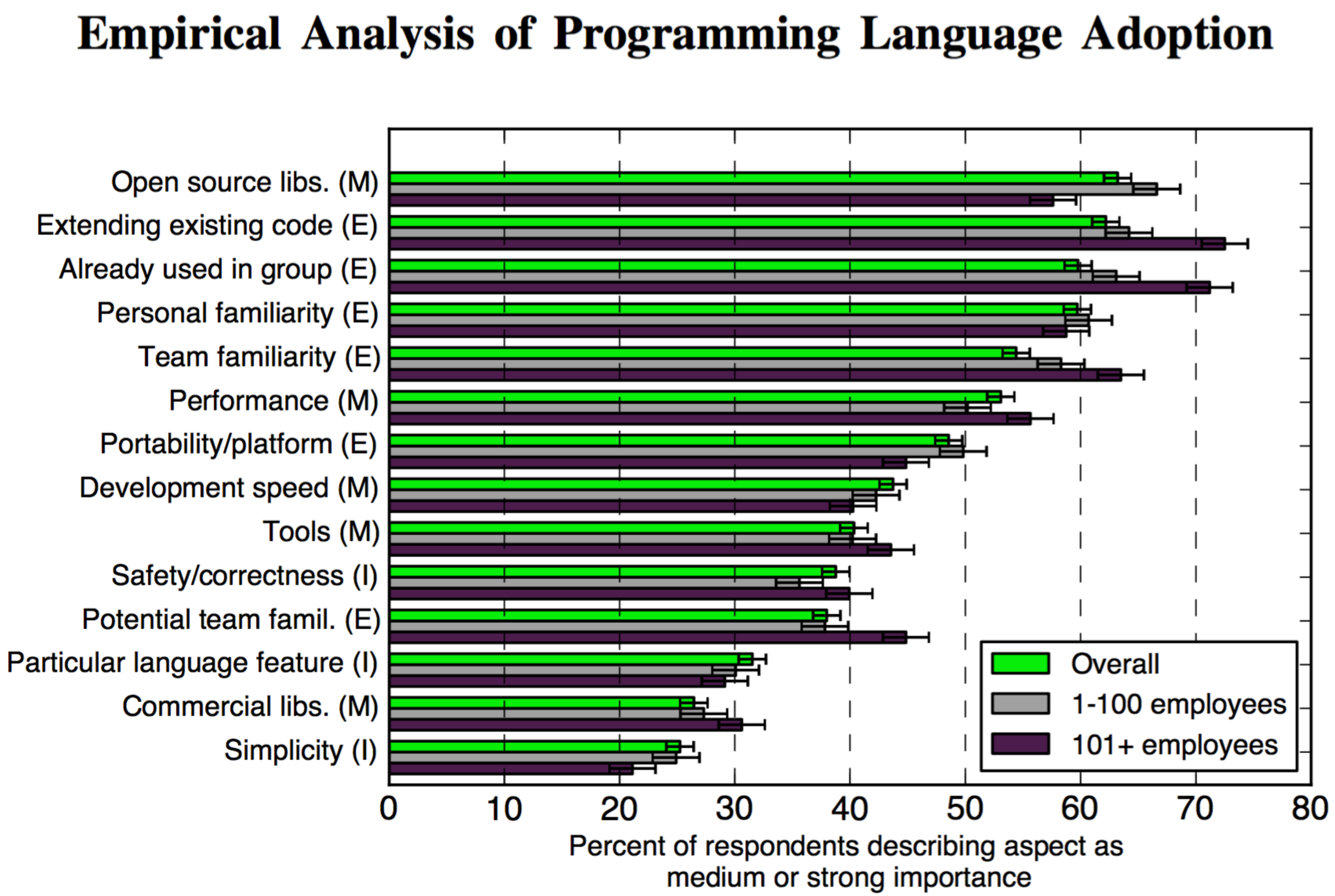
91% ECMA Script 2016

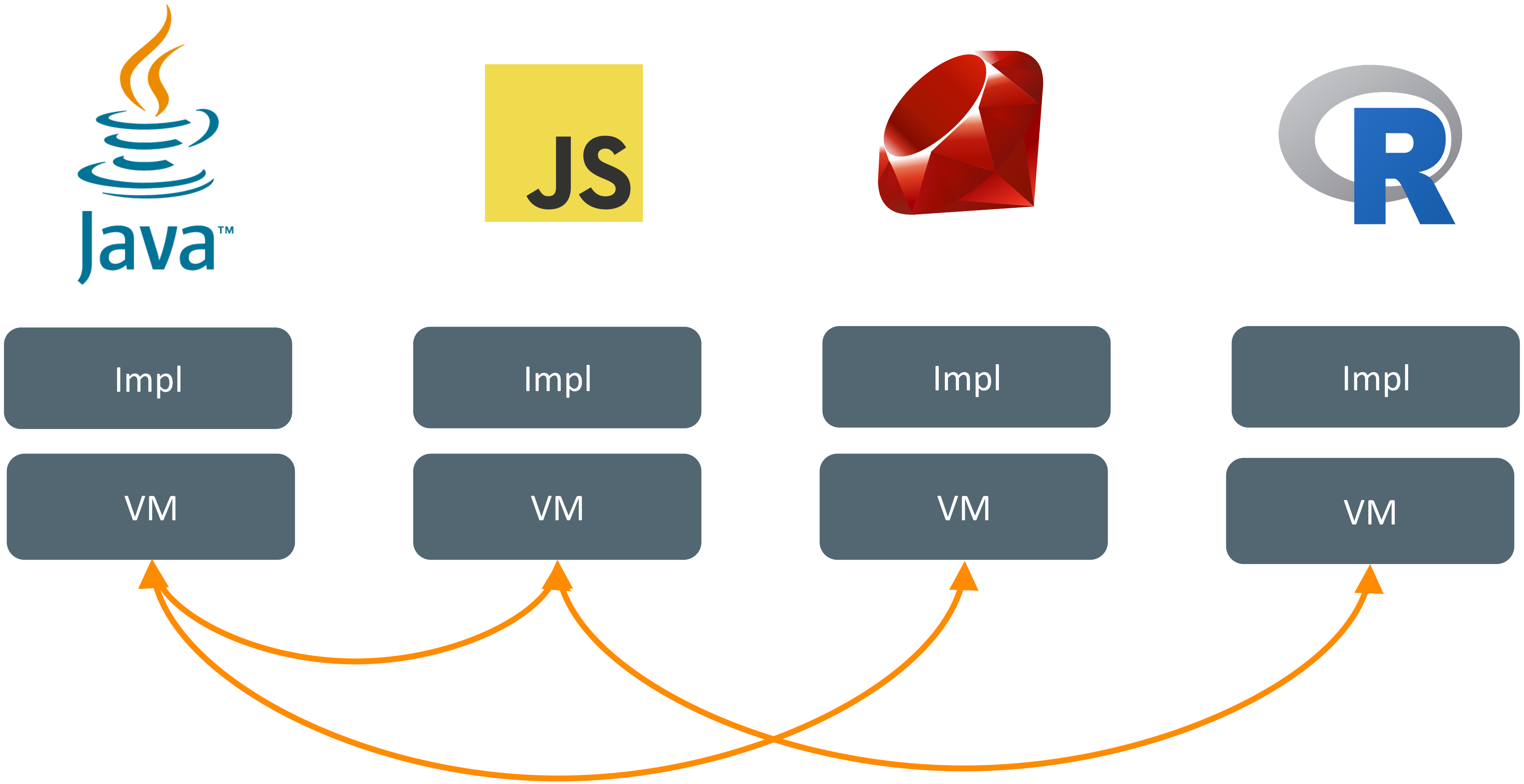
V8 (5.4.500.6) passes 91.1%

ORACLE

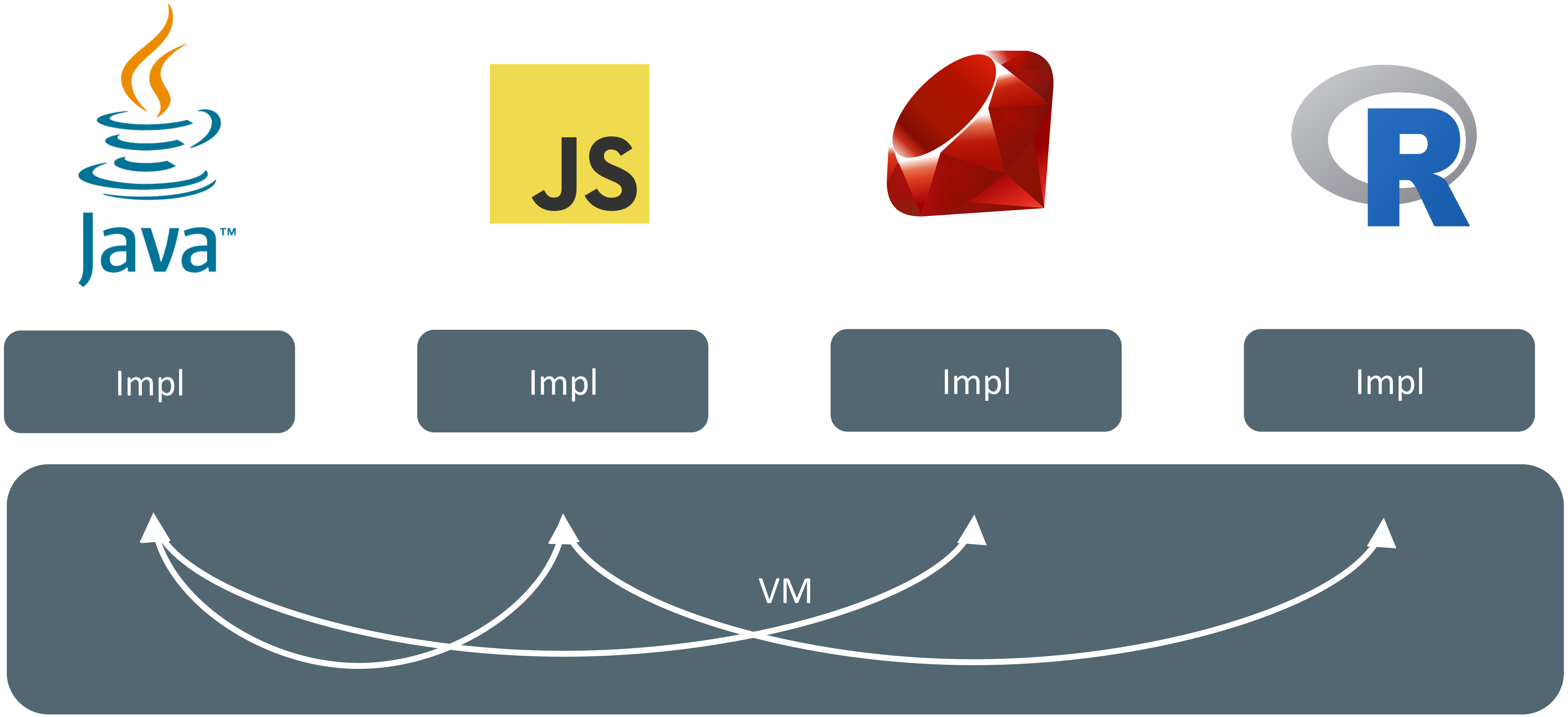
Graal VM: Going Polyglot

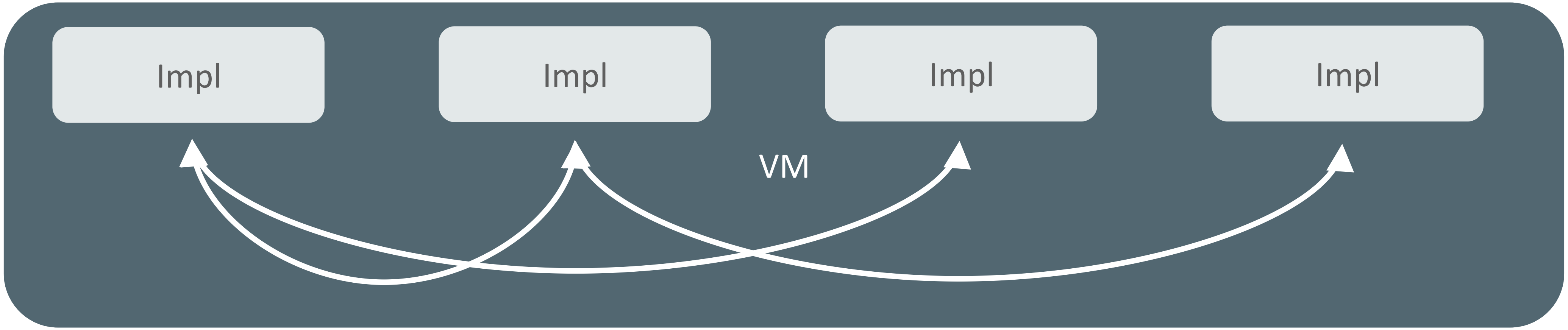
How important are the libraries you use?





ORACLE®


Java
JS
R
Impl
Impl
Impl
Impl
VM



Zero Overhead Interoperability

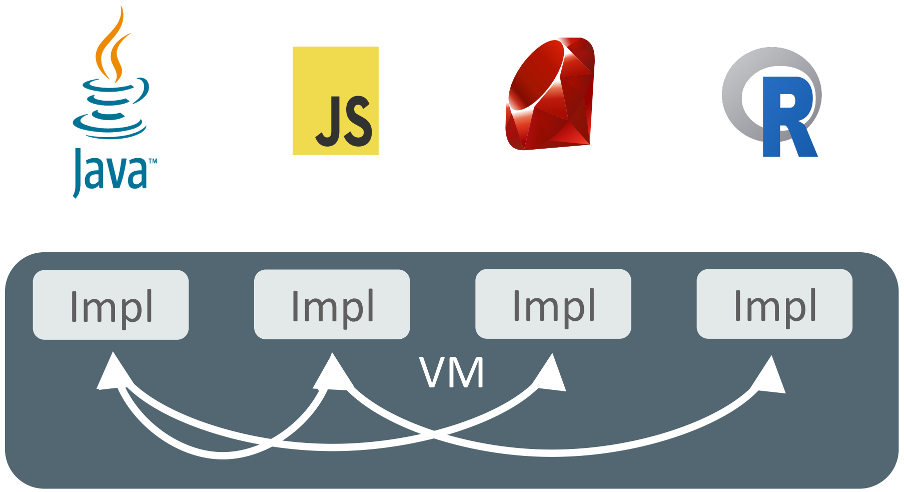
How we do polyglot in GraalVM?

```ruby
Truffle::Interop.eval('application/language', source)

value = Truffle::Interop.import(name)

Truffle::Interop.export(name)
```


Memory Managed
Code on the JVM
Native Code

Embedding a VM

Any Native Project



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.**

Substrate VM: Execution Model

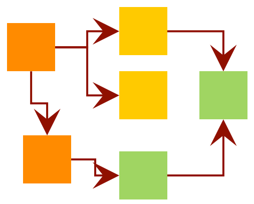
Points-To Analysis

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

SystemJava

SystemJava

- Legacy C code integration
  - Need a convenient way to access preexisting C functions and structures
  - Example: libc, legacy code
- 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

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;
}
```

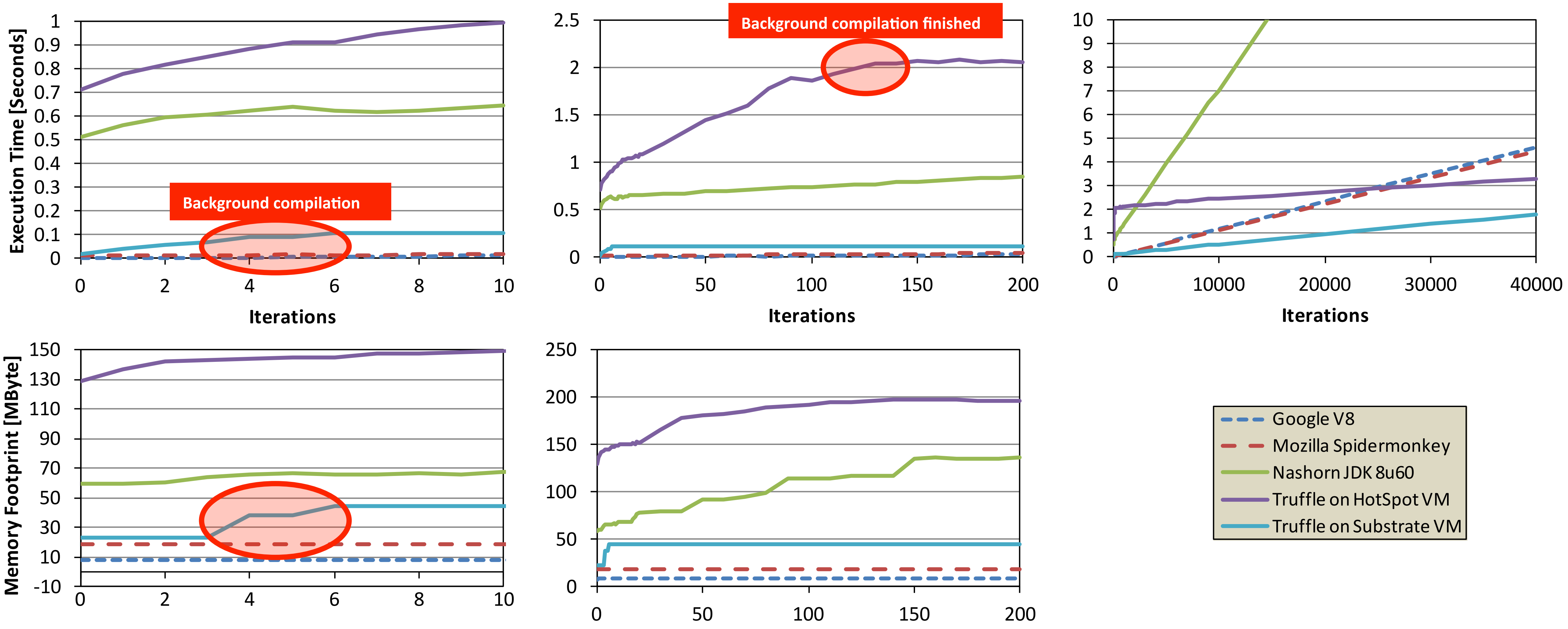
Function benchmark is invoked in a loop by harness (0 to 40000 iterations)

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)



Embedding the VM

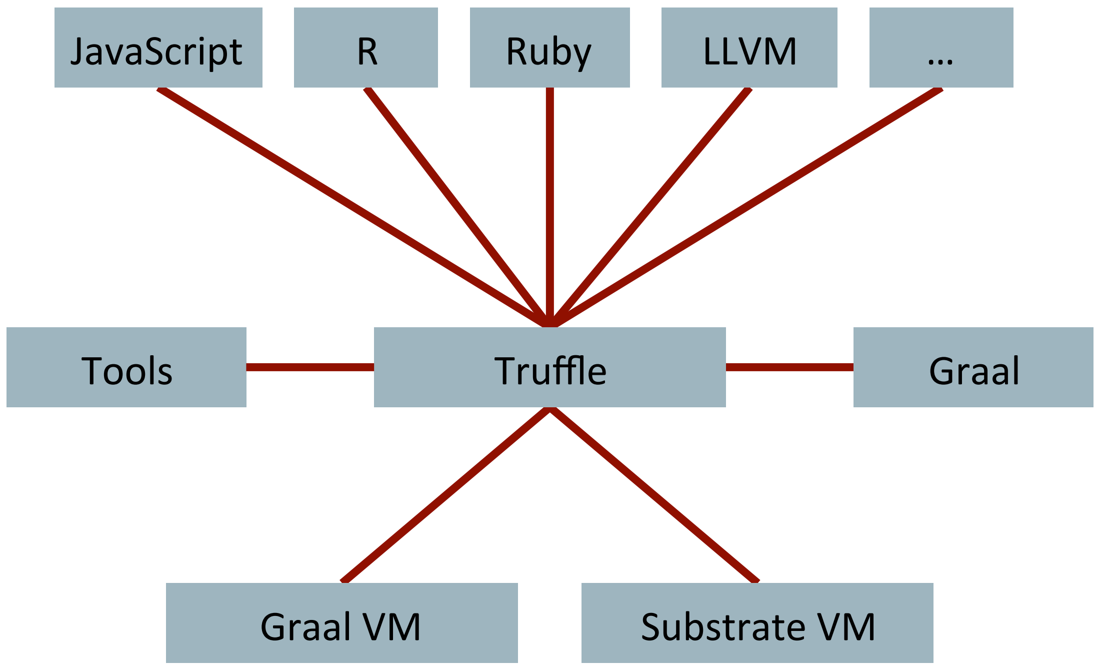
Truffle System Structure

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

Summary

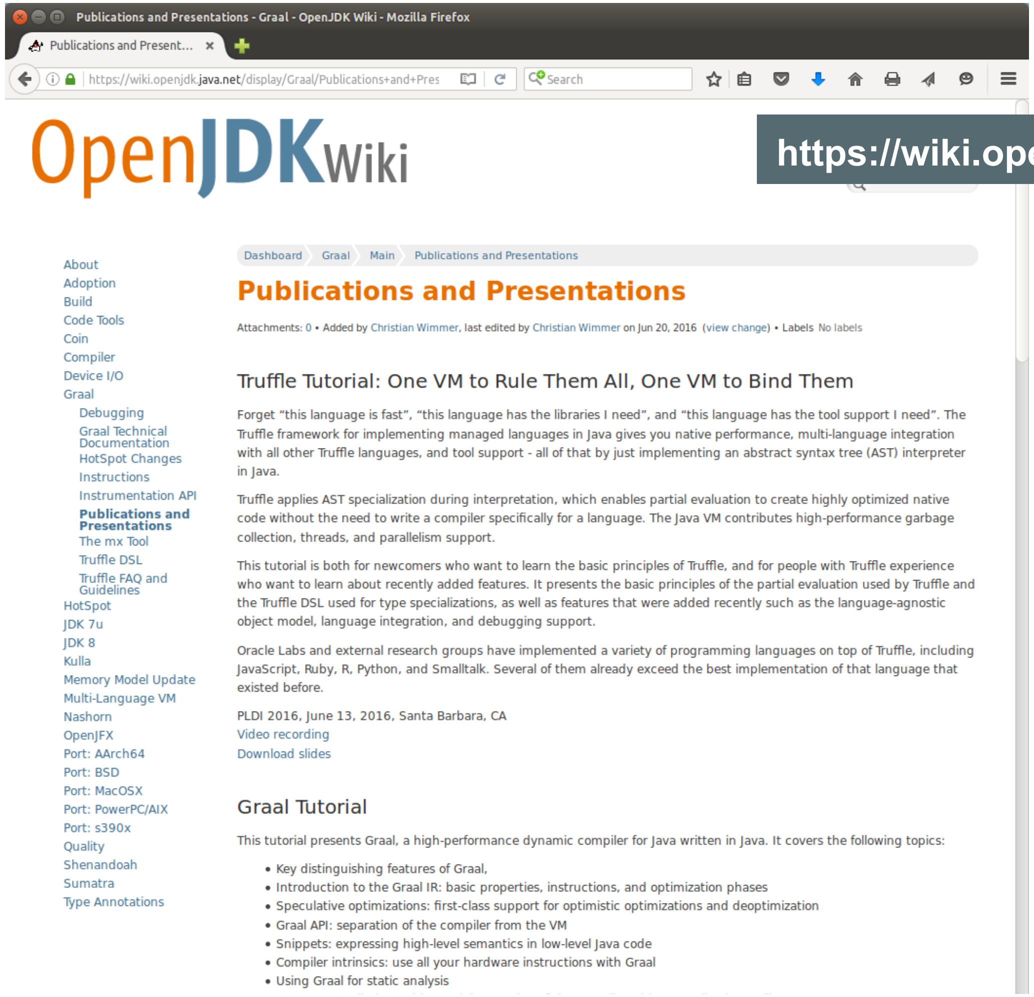
- Fast and easy-to-implement languages
- Interoperability between the languages with zero overhead
- Embeddable in native code via Substrate VM

Open Source

- github.com/graalvm/
- graal-core: dynamic compiler technology
- truffle: language implementation framework
- fastr: implementation of the R runtime
- sulong: execution of LLVM-based languages
- rubytruffle: implementation of the Ruby runtime
- simplelanguage: example language for getting started

ORACLE

Graal and Truffle Tutorials


Publications and Presentations - Graal - OpenJDK Wiki - Mozilla Firefox

Publications and Present...

https://wiki.openjdk.java.net/display/Graal/Publications+and+Pres

Search

OpenJDK Wiki

About
Adoption
Build
Code Tools
Coin
Compiler
Device I/O
Graal
- Debugging
- Graal Technical Documentation
- HotSpot Changes
- Instructions
- Instrumentation API
- **Publications and Presentations**
- The mx Tool
- Truffle DSL
- Truffle FAQ and Guidelines

HotSpot
JDK 7u
JDK 8
Kulla
Memory Model Update
Multi-Language VM
Nashorn
OpenJFX
Port: AArch64
Port: BSD
Port: MacOSX
Port: PowerPC/AIX
Port: s390x
Quality
Shenandoah
Sumatra
Type Annotations

Dashboard › Graal › Main › Publications and Presentations

Publications and Presentations

Attachments: 0 • Added by Christian Wimmer, last edited by Christian Wimmer on Jun 20, 2016 (view change) • Labels No labels

Truffle Tutorial: One VM to Rule Them All, One VM to Bind Them

Forget “this language is fast”, “this language has the libraries I need”, and “this language has the tool support I need”. The Truffle framework for implementing managed languages in Java gives you native performance, multi-language integration with all other Truffle languages, and tool support - all of that by just implementing an abstract syntax tree (AST) interpreter in Java.

Truffle applies AST specialization during interpretation, which enables partial evaluation to create highly optimized native code without the need to write a compiler specifically for a language. The Java VM contributes high-performance garbage collection, threads, and parallelism support.

This tutorial is both for newcomers who want to learn the basic principles of Truffle, and for people with Truffle experience who want to learn about recently added features. It presents the basic principles of the partial evaluation used by Truffle and the Truffle DSL used for type specializations, as well as features that were added recently such as the language-agnostic object model, language integration, and debugging support.

Oracle Labs and external research groups have implemented a variety of programming languages on top of Truffle, including JavaScript, Ruby, R, Python, and Smalltalk. Several of them already exceed the best implementation of that language that existed before.

PLDI 2016, June 13, 2016, Santa Barbara, CA
Video recording
Download slides

Graal Tutorial

This tutorial presents Graal, a high-performance dynamic compiler for Java written in Java. It covers the following topics:

- Key distinguishing features of Graal,
- Introduction to the Graal IR: basic properties, instructions, and optimization phases
- Speculative optimizations: first-class support for optimistic optimizations and deoptimization
- Graal API: separation of the compiler from the VM
- Snippets: expressing high-level semantics in low-level Java code
- Compiler intrinsics: use all your hardware instructions with Graal
- Using Graal for static analysis

https://wiki.openjdk.java.net/display/Graal/Publications+and+Presentations

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