ORACLE®

Graal

Christian Wimmer

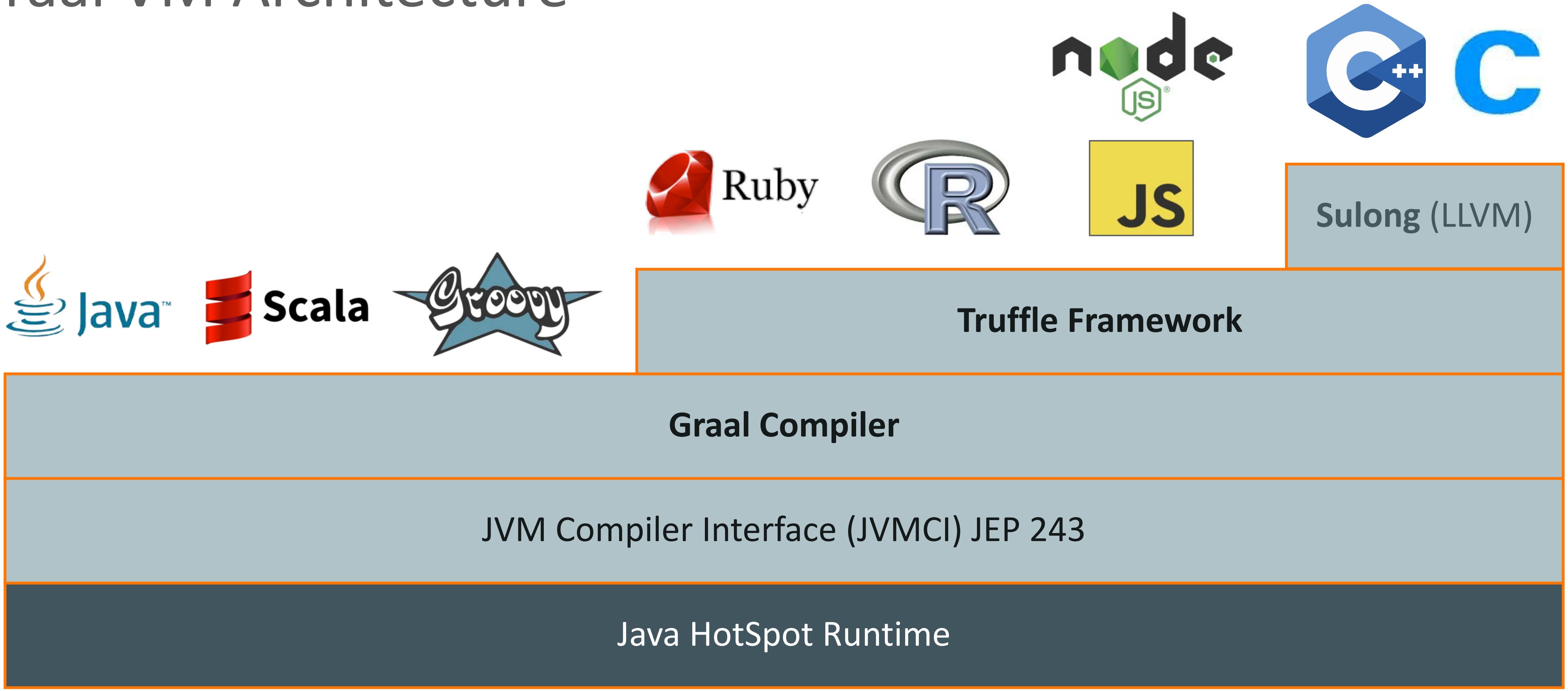
VM Research Group, Oracle Labs

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

Graal VM Architecture

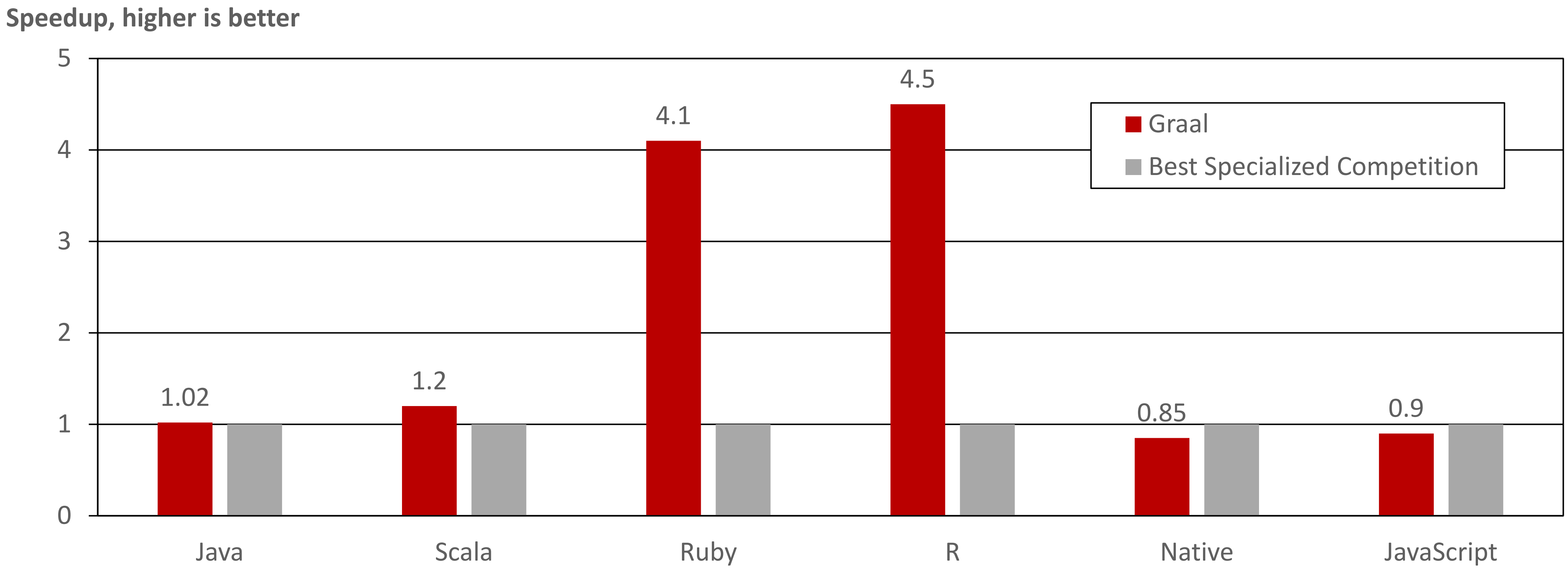


Tutorial Outline

- The Graal compiler
  - Key distinguishing features of Graal, a high-performance dynamic compiler for Java written in Java
  - Introduction to the Graal intermediate representation: structure, instructions, and optimization phases
  - Speculative optimizations: first-class support for optimistic optimizations and deoptimization
  - JVMCI: 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
  - Custom compilations with Graal: integration of the compiler with an application or library
- The GraalVM ecosystem
  - The Truffle framework for dynamic programming language implementation
  - Graal as a compiler for dynamic programming languages in the Truffle framework
  - Polyglot Native: ahead-of-time compilation of Java (and Scala, Kotlin, ...) and integration with C code

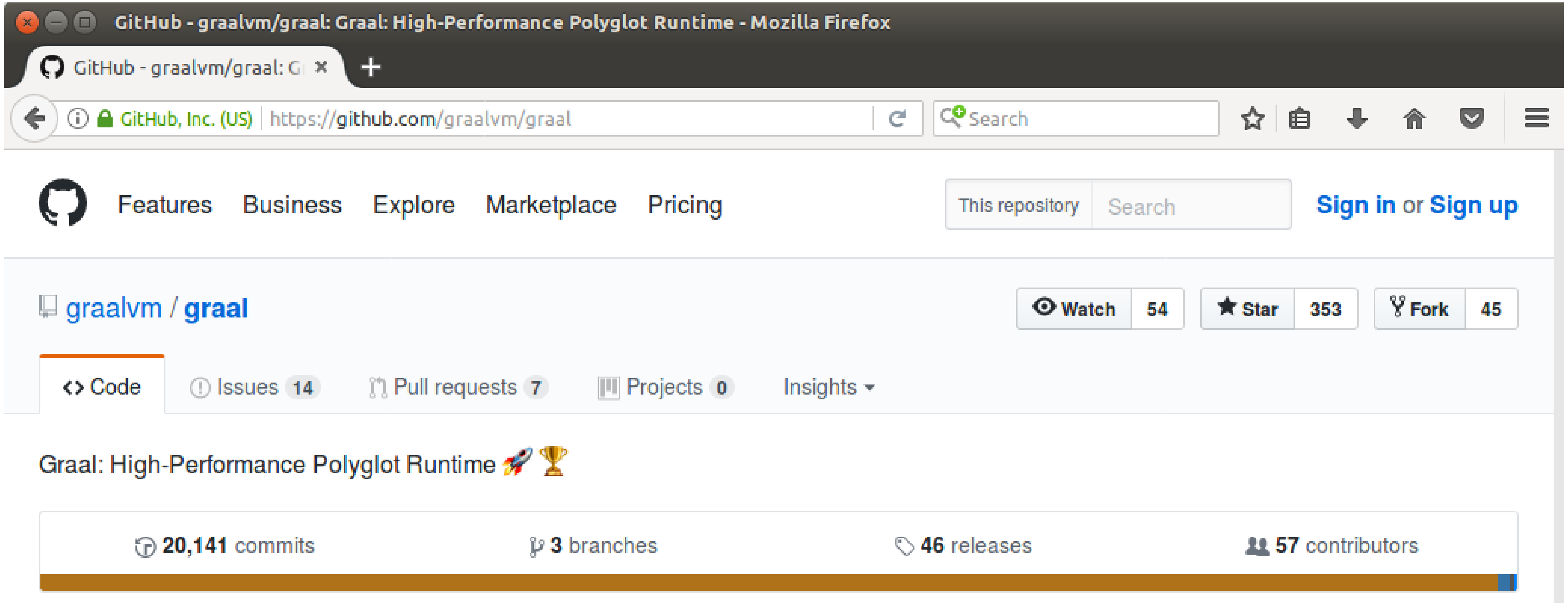
ORACLE

Performance: Graal VM



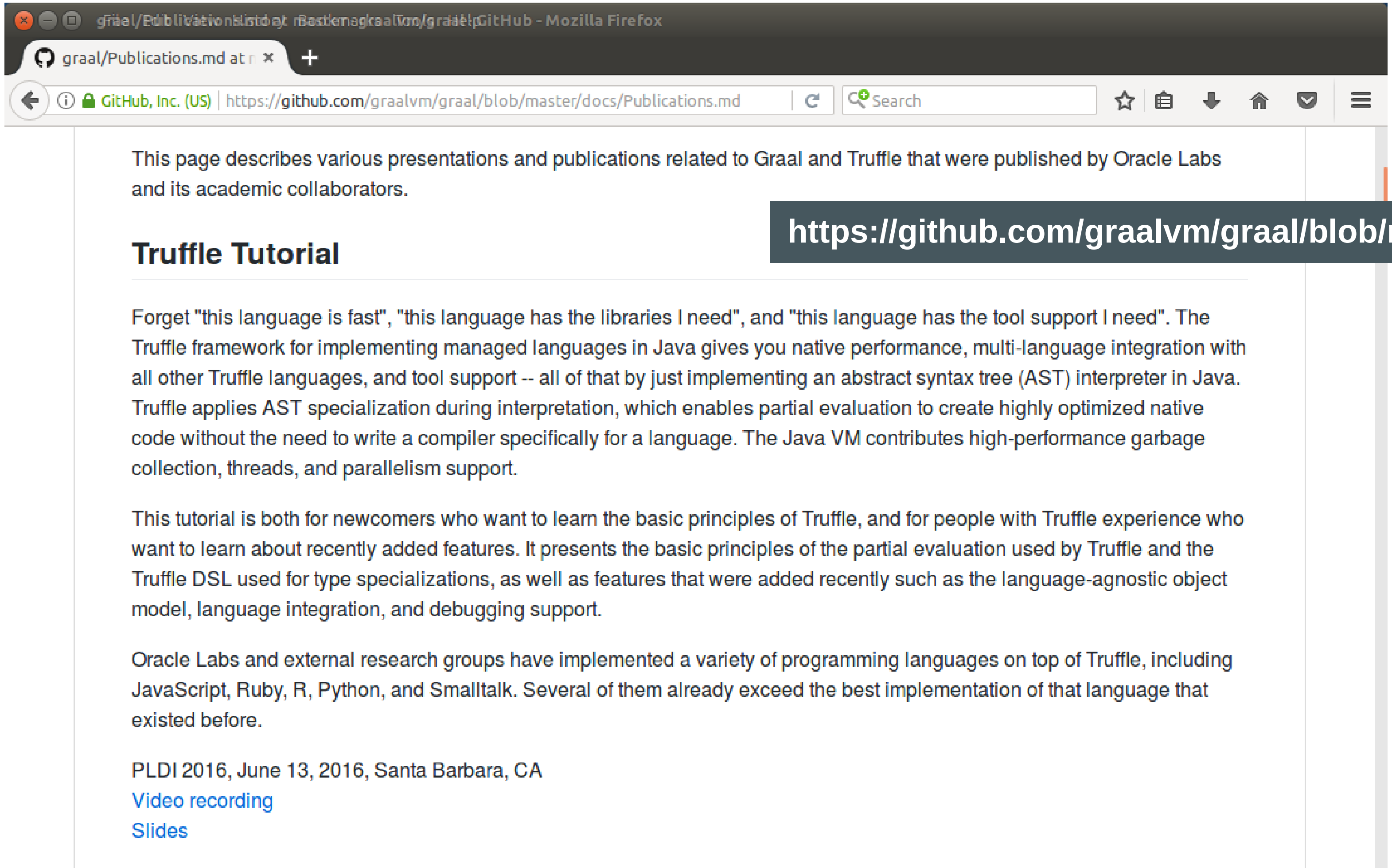
Performance relative to:
HotSpot/Server, HotSpot/Server running JRuby, GNU R, LLVM AOT compiled, V8

Open Source Code on GitHub



https://github.com/graalvm

Publications and Tutorials



This page describes various presentations and publications related to Graal and Truffle that were published by Oracle Labs and its academic collaborators.

Truffle Tutorial

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.

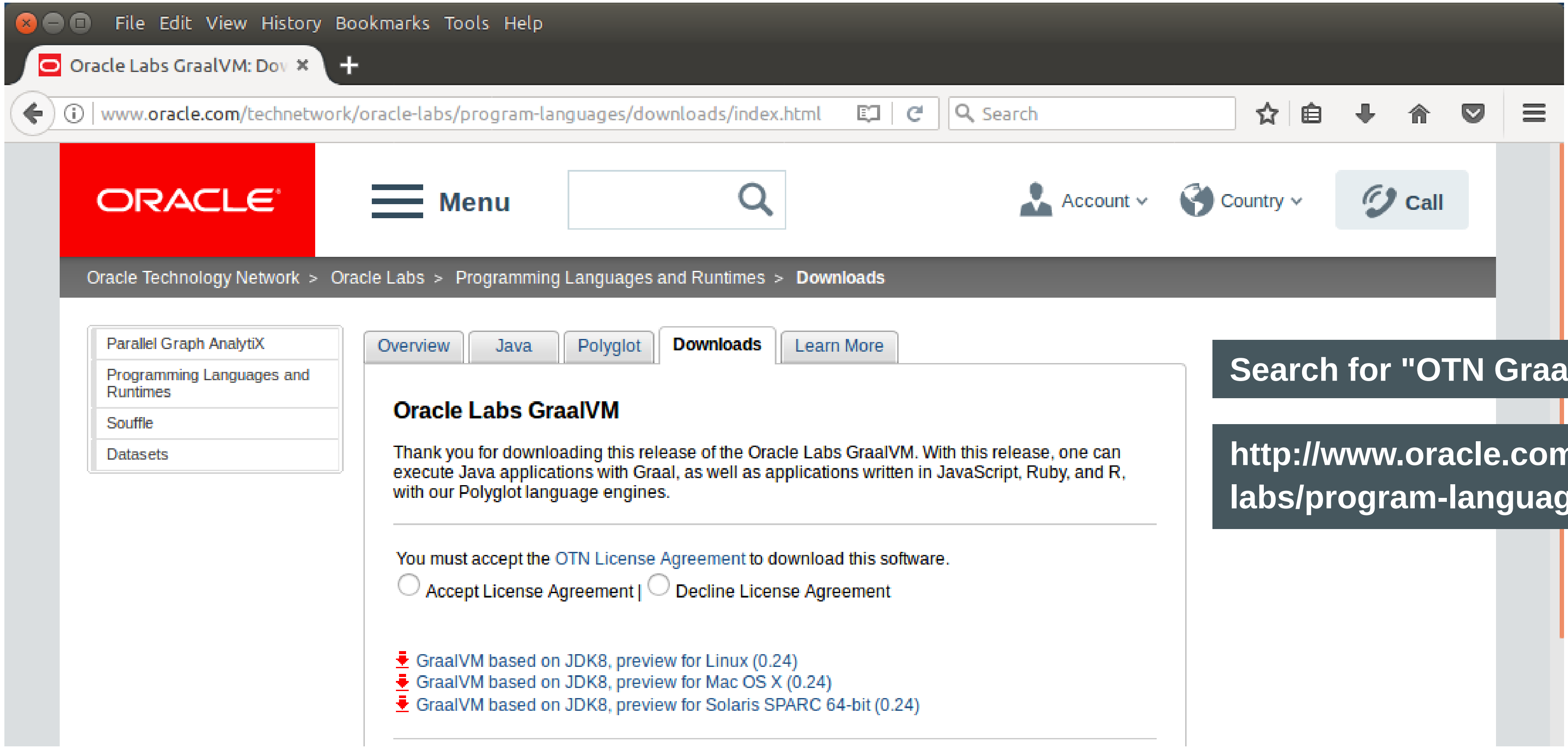
This tutorial is both for newcomers who want to learn the basic principles of Truffle, and for people with Truffle experience who want to learn about recently added features. It presents the basic principles of the partial evaluation used by Truffle and the Truffle DSL used for type specializations, 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
Slides

https://github.com/graalvm/graal/blob/master/docs/Publications.md

Binary Snapshots on OTN



Search for "OTN Graal"

http://www.oracle.com/technetwork/oracle-labs/program-languages/downloads/

Team

Oracle
Florian Angerer
Danilo Ansaloni
Stefan Anzinger
Martin Balin
Cosmin Basca
Daniele Bonetta
Dušan Bálek
Matthias Brantner
Lucas Braun
Petr Chalupa
Jürgen Christ
Laurent Daynès
Gilles Duboscq
Svatopluk Dědic
Martin Entlicher
Pit Fender
Francois Farquet
Brandon Fish
Matthias Grimmer
Christian Häubl
Peter Hofer
Bastian Hossbach
Christian Humer
Tomáš Hůrka
Mick Jordan

Oracle (continued)
Vojin Jovanovic
Anantha Kandukuri
Harshad Kasture
Cansu Kaynak
Peter Kessler
Duncan MacGregor
Jiří Maršík
Kevin Menard
Miloslav Metelka
Tomáš Myšík
Petr Pišl
Oleg Pliss
Jakub Podlešák
Aleksandar Prokopec
Tom Rodriguez
Roland Schatz
Benjamin Schlegel
Chris Seaton
Jiří Sedláček
Doug Simon
Štěpán Šindelář
Zbyněk Šlajchrt
Boris Spasojevic
Lukas Stadler
Codrut Stancu

Oracle (continued)
Jan Štola
Tomáš Stupka
Farhan Tauheed
Jaroslav Tulach
Alexander Ulrich
Michael Van De Vanter
Aleksandar Vitorovic
Christian Wimmer
Christian Wirth
Paul Wögerer
Mario Wolczko
Andreas Wöß
Thomas Würthinger
Tomáš Zezula
Yudi Zheng

Red Hat
Andrew Dinn
Andrew Haley

Intel
Michael Berg

Twitter
Chris Thalinger

Oracle Interns
Brian Belleville
Ondrej Douda
Juan Fumero
Miguel Garcia
Hugo Guiroux
Shams Imam
Berkin Ilbeyi
Hugo Kapp
Alexey Karyakin
Stephen Kell
Andreas Kunft
Volker Lanting
Gero Leinemann
Julian Lettner
Joe Nash
Tristan Overney
Aleksandar Pejovic
David Piorkowski
Philipp Riedmann
Gregor Richards
Robert Seilbeck
Rifat Shariyar

Oracle Alumni
Erik Eckstein
Michael Haupt
Christos Kotselidis
David Leibs
Adam Welc
Till Westmann

JKU Linz
Hanspeter Mössenböck
Benoit Daloze
Josef Eisl
Thomas Feichtinger
Josef Haider
Christian Huber
David Leopoldseder
Stefan Marr
Manuel Rigger
Stefan Rumzucker
Bernhard Urban

TU Berlin:
Volker Markl
Andreas Kunft
Jens Meiners
Tilmann Rabl

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

LaBRI
Floréal Morandat

University of California, Irvine
Michael Franz
Yeoul Na
Mohaned Qunaibit
Gulfem Savrun Yeniceri
Wei Zhang

Purdue University
Jan Vitek
Tomas Kalibera
Petr Maj
Lei Zhao

T. U. Dortmund
Peter Marwedel
Helena Kotthaus
Ingo Korb

University of California, Davis
Duncan Temple Lang
Nicholas Ulle

University of Lugano, Switzerland
Walter Binder
Sun Haiyang

ORACLE

Part 1: The Graal Compiler

What is Graal?

- A high-performance optimizing JIT compiler for the Java HotSpot VM
  - Written in Java and benefitting from Java’s annotation and metaprogramming
- A modular platform to experiment with new compiler optimizations
- A customizable and targetable compiler that you can invoke from Java
  - Compile what you want, the way you want
- A platform for speculative optimization of managed languages
  - Especially dynamic programming languages benefit from speculation
- A platform for static analysis of Java bytecodes

Why use Graal for Your Research Project?

- Because your paper abstract will sound very convincing
  - "We implemented this novel optimization in a production quality compiler, and evaluate it with industry-standard benchmarks for Java, JavaScript, Ruby, R, and C"

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
  - Compiler-VM separation
- Written in Java to lower the entry barrier
  - Graal compiling and optimizing itself is also a good optimization opportunity

Getting Started

Get mx, our script to simplify building and execution

```
$ git clone https://github.com/graalvm/mx
$ export PATH=$PWD/mx:$PATH
$ export JAVA_HOME=path to downloaded labsjdk
```

Get and build the Graal source code:

```
$ git clone https://github.com/graalvm/graal.git
$ cd graal/compiler
$ mx build
```

Run the Java VM with Graal as the JIT compiler:

```
$ mx vm -XX:+UseJVMCICompiler -version
```

Generate Eclipse and NetBeans projects:

```
$ mx ideinit
```

Run the whitebox unit tests

```
$ mx unittest
```

Run a specific unit test in the Java debugger

```
$ mx -d unittest GraalTutorial#testStringHashCode
```

Examples in this tutorial assume that `mx` is on path

Download labsjdk (JDK 8 with JVMCI) from www.oracle.com/technetwork/oracle-labs/program-languages/downloads/

Operating Systems: Windows, Linux, MacOS, Solaris

Architectures: Intel 64-bit, Sparc, AArch64 (experimental)

Use the predefined Eclipse launch configuration to connect to the Graal VM

ORACLE

Java 9

- Graal communicates with the VM using JVMCI (JVM Compiler Interface)
  - Java interfaces to access classes, fields methods
  - Provider interfaces to install code into the VM

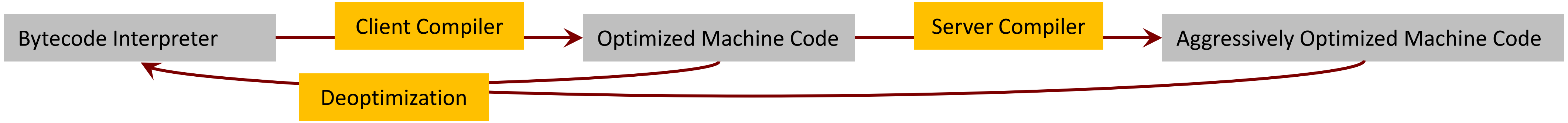
- JVMCI is part of OpenJDK starting with JDK 9
  - Graal will run on any standard OpenJDK / Oracle JDK
  - JDK 9 is still under development, changes related to Jigsaw break Graal occasionally

- Until Java 9 is released, using our JDK 8 version is simpler to use
  - Download the "labsjdk" from the Oracle Technical Network
    - www.oracle.com/technetwork/oracle-labs/program-languages/downloads/
  - Or build it yourself
    - http://hg.openjdk.java.net/graal/graal-jvmci-8

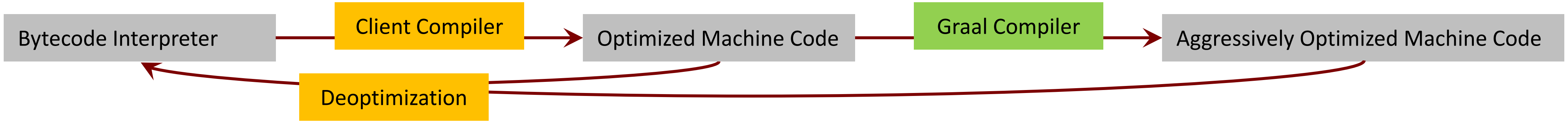
ORACLE

Mixed-Mode Execution

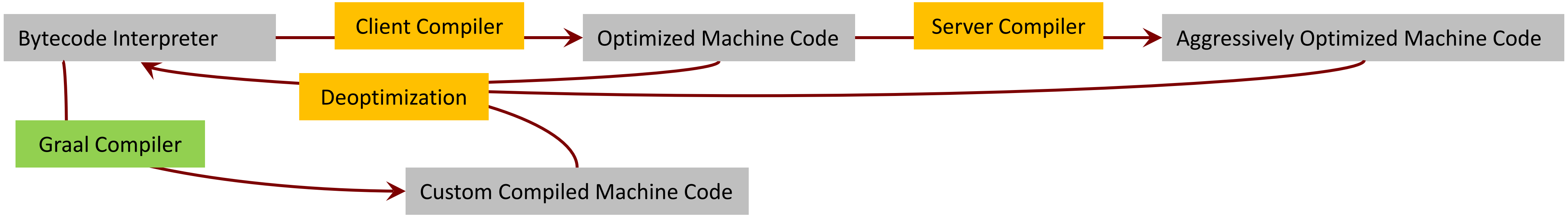
Default configuration of Java HotSpot VM in production:



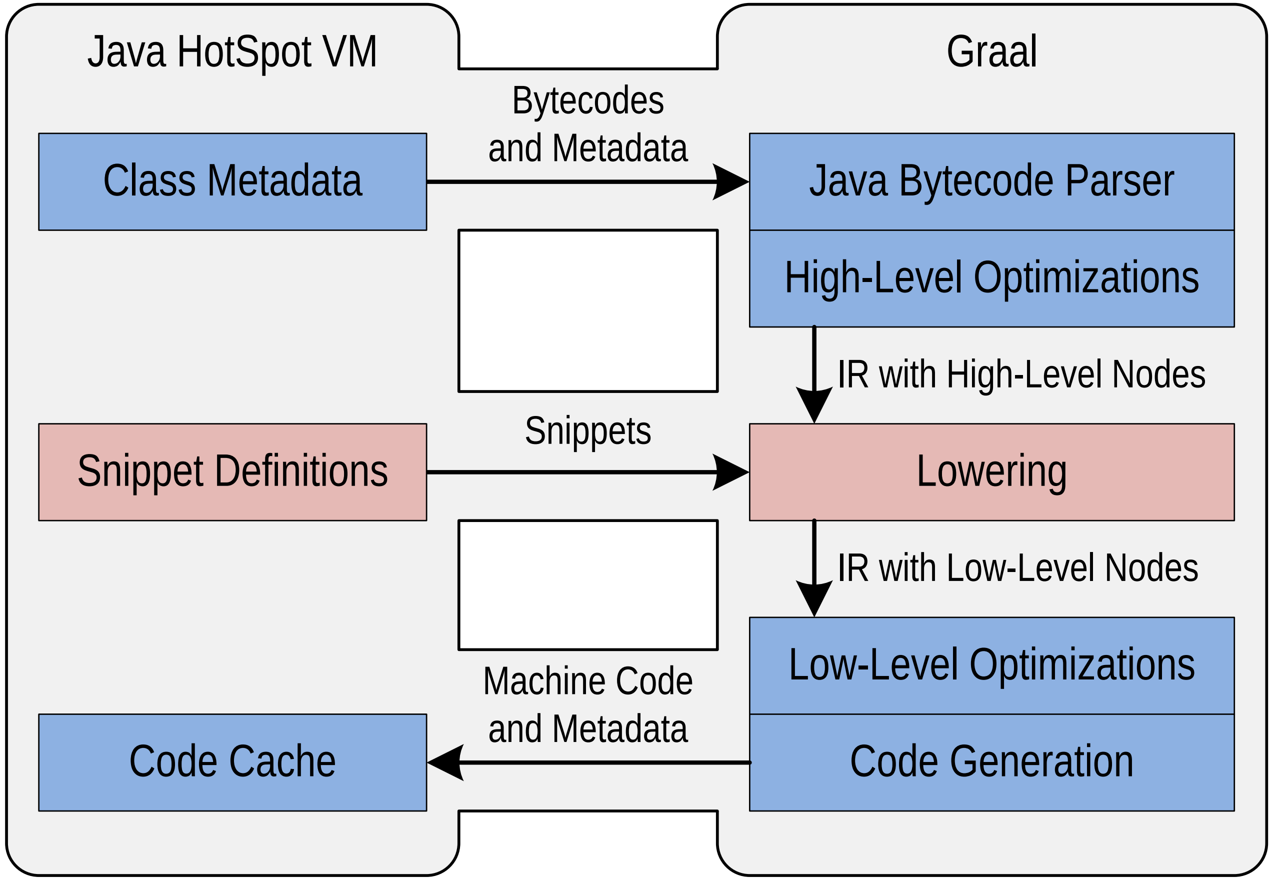
Graal VM in configuration "-XX:+UseJVMCICompiler": Graal replaces the server compiler



Graal VM in configuration "-XX:-UseJVMCICompiler": Graal used only for custom compilations



Compiler-VM Separation



Default Compilation Pipeline

- Java bytecode parser
- Front end: graph based intermediate representation (IR) in static single assignment (SSA) form
  - High Tier
    - Method inlining
    - Partial escape analysis
    - Lowering using snippets
  - Mid Tier
    - Memory optimizations
    - Lowering using snippets
  - Low Tier
- Back end: register based low-level IR (LIR)
  - Register allocation
  - Peephole optimizations
- Machine code generation

Source code reference: `GraalCompiler.compile()`

ORACLE

Graph-Based Intermediate Representation

Basic Properties

- Two interposed directed graphs
  - Control flow graph: Control flow edges point “downwards” in graph
  - Data flow graph: Data flow edges point “upwards” in graph
- Floating nodes
  - Nodes that can be scheduled freely are not part of the control flow graph
  - Avoids unnecessary restrictions of compiler optimizations
- Graph edges specified as annotated Java fields in node classes
  - Control flow edges: @Successor fields
  - Data flow edges: @Input fields
  - Reverse edges (i.e., predecessors, usages) automatically maintained by Graal
- Always in Static Single Assignment (SSA) form
- Only explicit and structured loops
  - Loop begin, end, and exit nodes
- Graph visualization tool: “Ideal Graph Visualizer”, start using “mx igv”

ORACLE

IR Example: Defining Nodes

```
public abstract class BinaryNode ... {
  @Input protected ValueNode x;
  @Input protected ValueNode y;
}
```

@Input fields: data flow

```
public class IfNode ... {
  @Successor BeginNode trueSuccessor;
  @Successor BeginNode falseSuccessor;
  @Input(InputType.Condition) LogicNode condition;
  protected double trueSuccessorProbability;
}
```

@Successor fields: control flow

Fields without annotation: normal data properties

```
public abstract class Node ... {
  public NodeClassIterable inputs() { ... }
  public NodeClassIterable successors() { ... }

  public NodeIterable<Node> usages() { ... }
  public Node predecessor() { ... }
}
```

Base class allows iteration of all inputs / successors

Base class maintains reverse edges: usages / predecessor

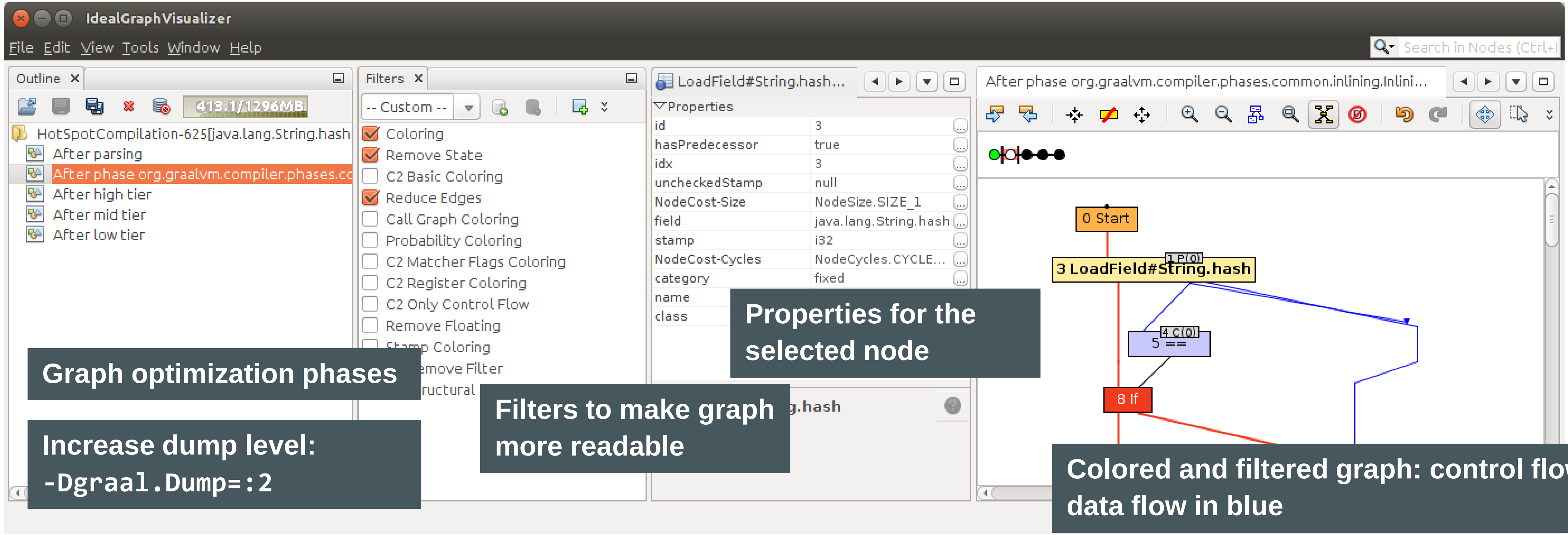
Design invariant: a node has at most one predecessor

IR Example: Ideal Graph Visualizer

Start the Graal VM with graph dumping enabled

```
$ mx igv &
$ mx unittest -Dgraal.Dump= -Dgraal.MethodFilter=String.hashCode GraalTutorial#testStringHashCode
```

Test that just compiles `String.hashCode()`



Graph optimization phases

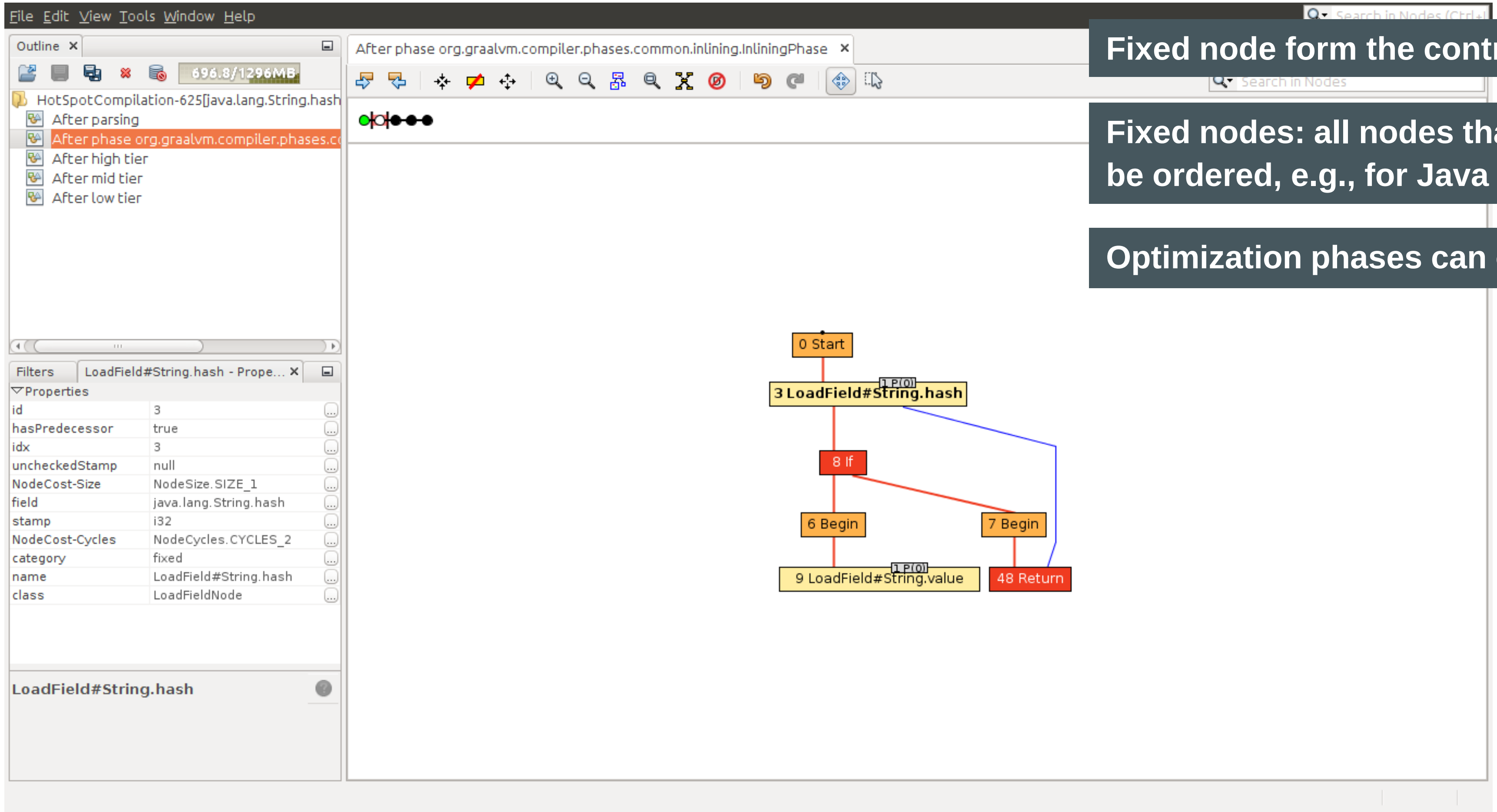
Increase dump level: `-Dgraal.Dump=:2`

Filters to make graph more readable

Properties for the selected node

Colored and filtered graph: control flow in red, data flow in blue

IR Example: Control Flow

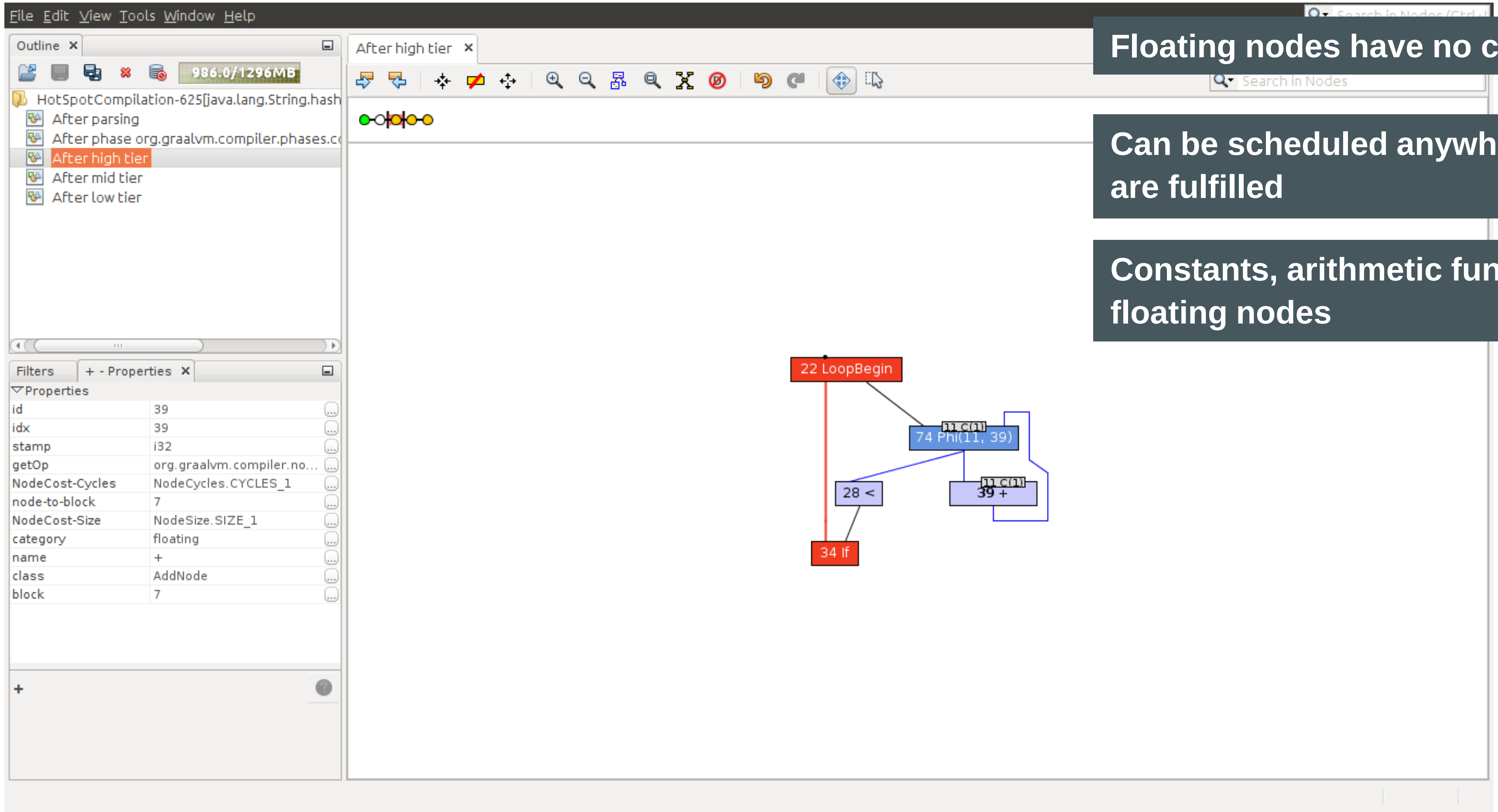


Fixed node form the control flow graph

Fixed nodes: all nodes that have side effects and need to be ordered, e.g., for Java exception semantics

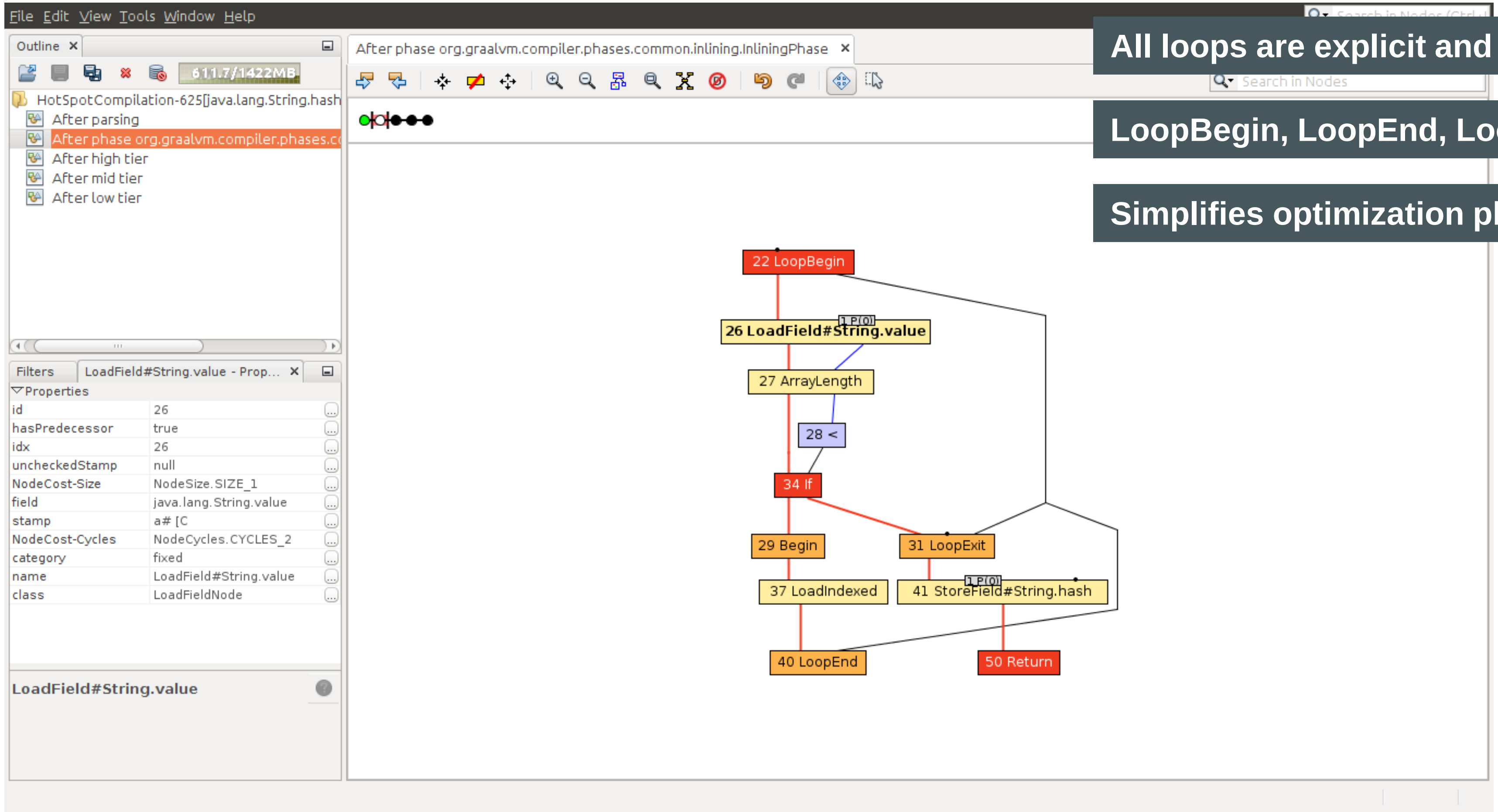
Optimization phases can convert fixed to floating nodes

IR Example: Floating Nodes



- **Floating nodes have no control flow dependency**
- **Can be scheduled anywhere as long as data dependencies are fulfilled**
- **Constants, arithmetic functions, phi functions, … are floating nodes**

IR Example: Loops



All loops are explicit and structured

LoopBegin, LoopEnd, LoopExit nodes

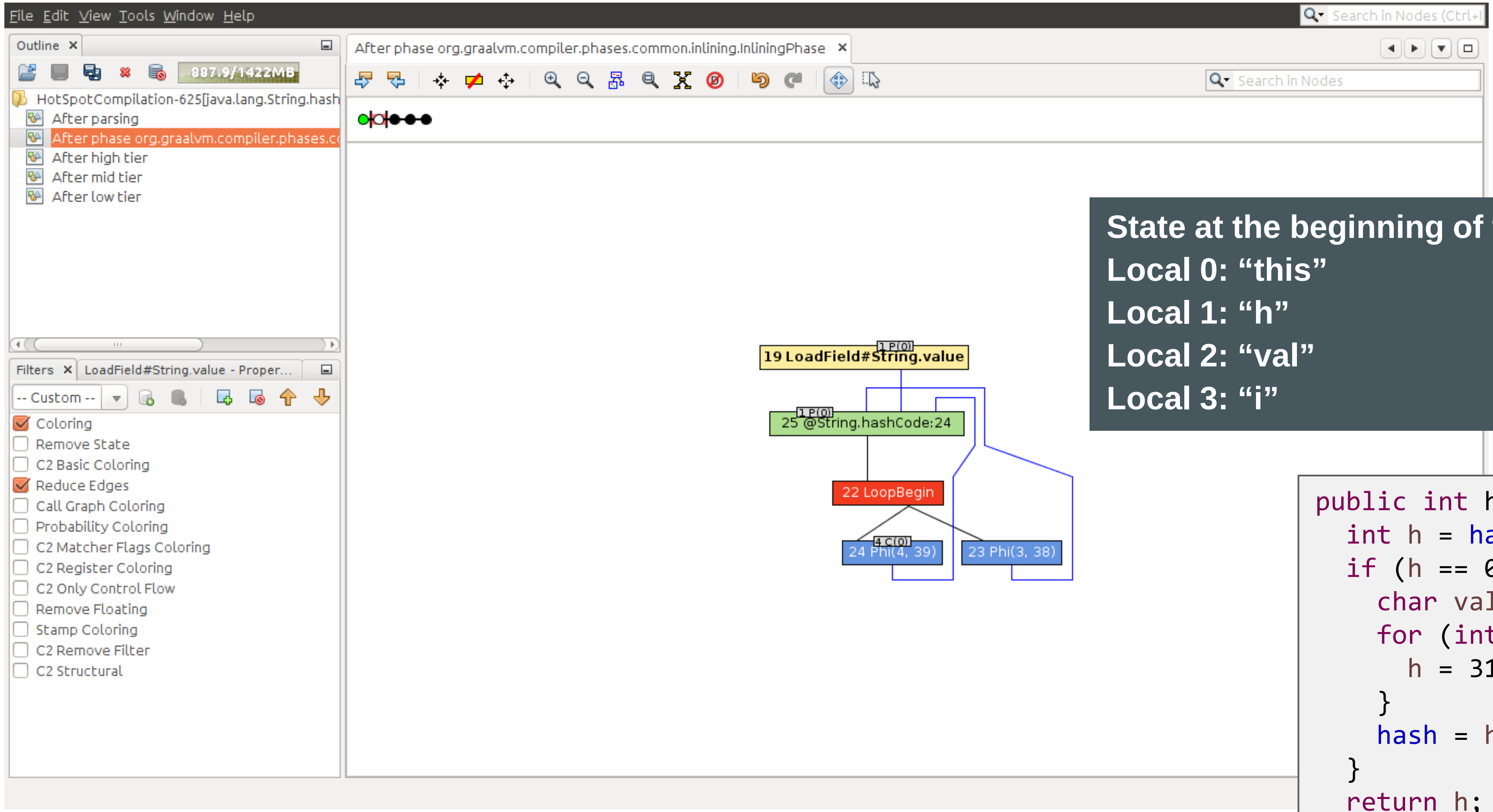
Simplifies optimization phases

FrameState

- Speculative optimizations require deoptimization
  - Restore Java interpreter state at safepoints
  - Graal tracks the interpreter state throughout the whole compilation
    - FrameState nodes capture the state of Java local variables and Java expression stack
    - And: method + bytecode index

- Method inlining produces nested frame states
  - `FrameState` of callee has `@Input outerFrameState`
  - Points to `FrameState` of caller

IR Example: Frame States



State at the beginning of the loop:
Local 0: “this”
Local 1: “h”
Local 2: “val”
Local 3: “i”

```
public int hashCode() {
  int h = hash;
  if (h == 0 && value.length > 0) {
    char val[] = value;
    for (int i = 0; i < value.length; i++) {
      h = 31 * h + val[i];
    }
    hash = h;
  }
  return h;
}
```

Important Optimizations

- Constant folding, arithmetic optimizations, strength reduction, ...
  - `CanonicalizerPhase`
  - Nodes implement the interface `Canonicalizeable`
  - Executed often in the compilation pipeline
  - Incremental canonicalizer only looks at new / changed nodes to save time
- Global Value Numbering
  - Automatically done based on node equality

ORACLE

A Simple Optimization Phase

```
public class LockEliminationPhase extends Phase {

  @Override
  protected void run(StructuredGraph graph) {
    for (MonitorExitNode monitorExitNode : graph.getNodes(MonitorExitNode.TYPE)) {

      FixedNode next = monitorExitNode.next();
      if ((next instanceof MonitorEnterNode || next instanceof RawMonitorEnterNode)) {
        AccessMonitorNode monitorEnterNode = (AccessMonitorNode) next;

        if (GraphUtil.unproxify(monitorEnterNode.object()) == GraphUtil.unproxify(monitorExitNode.object())) {

          MonitorIdNode enterId = monitorEnterNode.getMonitorId();
          MonitorIdNode exitId = monitorExitNode.getMonitorId();
          if (enterId != exitId) {
            enterId.replaceAndDelete(exitId);
          }
          GraphUtil.removeFixedWithUnusedInputs(monitorEnterNode);
          GraphUtil.removeFixedWithUnusedInputs(monitorExitNode);
        }
      }
    }
  }
}
```

Eliminate unnecessary release-reacquire of a monitor when no instructions are between

Iterate all nodes of a certain class

Modify the graph

ORACLE

Type System (Stamps)

- Every node has a `Stamp` that describes the possible values of the node
  - The kind of the value (object, integer, float)
  - But with additional details if available
  - Stamps form a lattice with `meet` (= union) and `join` (= intersection) operations

- `ObjectStamp`
  - Declared type: the node produces a value of this type, or any subclass
  - Exact type: the node produces a value of this type (exactly, not a subclass)
  - Value is never null (or always null)
- `IntegerStamp`
  - Number of bits used
  - Minimum and maximum value
  - Bits that are always set, bits that are never set
- `FloatStamp`

ORACLE

Speculative Optimizations

Motivating Example for Speculative Optimizations

- Inlining of virtual methods
  - Most methods in Java are dynamically bound
  - Class Hierarchy Analysis
  - Inline when only one suitable method exists
- Compilation of foo() when only A loaded
  - Method getX() is inlined
  - Same machine code as direct field access
  - No dynamic type check
- Later loading of class B
  - Discard machine code of foo()
  - Recompile later without inlining
- Deoptimization
  - Switch to interpreter in the middle of foo()
  - Reconstruct interpreter stack frames
  - Expensive, but rare situation
  - Most classes already loaded at first compile

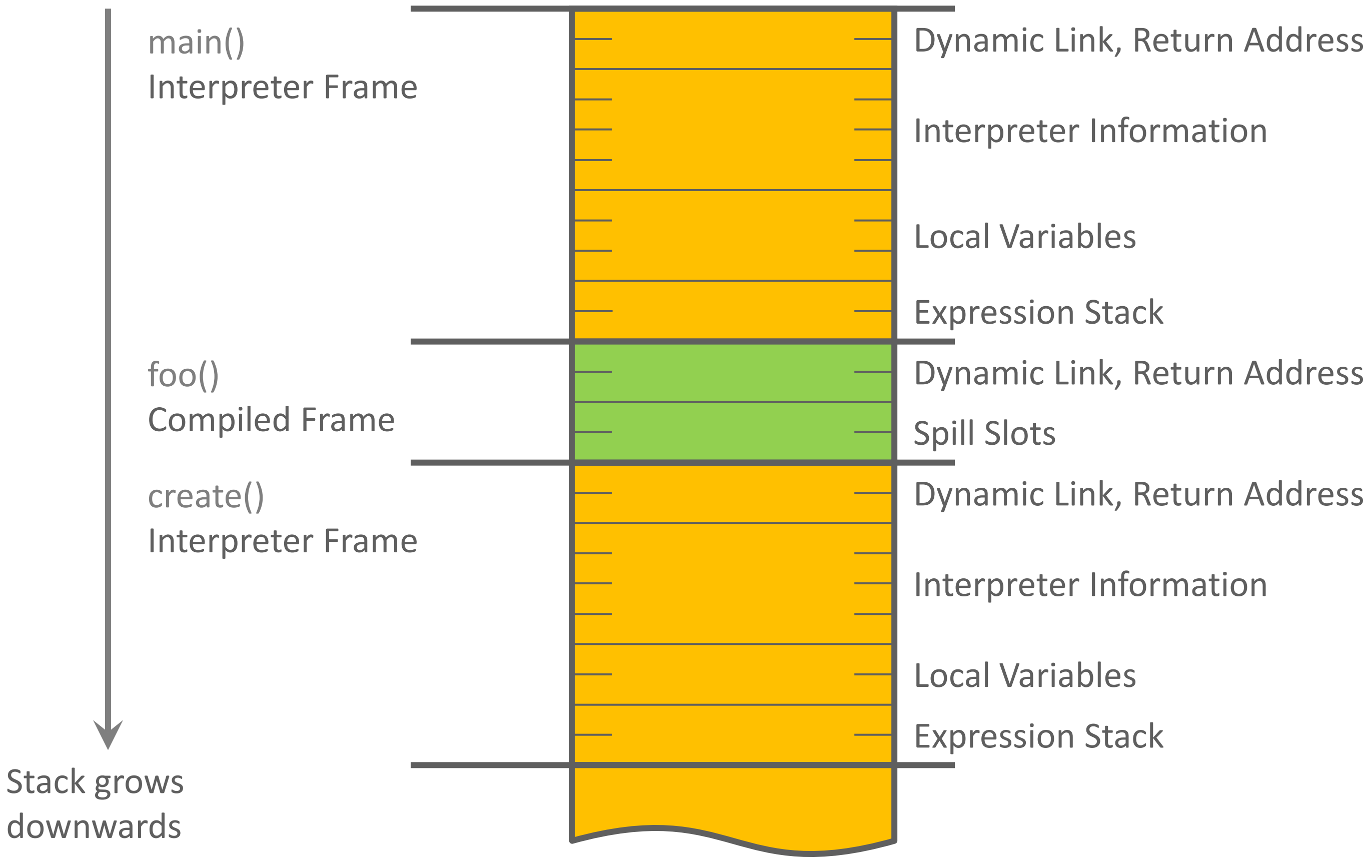
```
void foo() {
  A a = create();
  a.getX();
}
```

```
class A {
  int x;

  int getX() {
    return x;
  }
}
```

```
class B extends A {
  int getX() {
    return ...
  }
}
```

ORACLE

Deoptimization

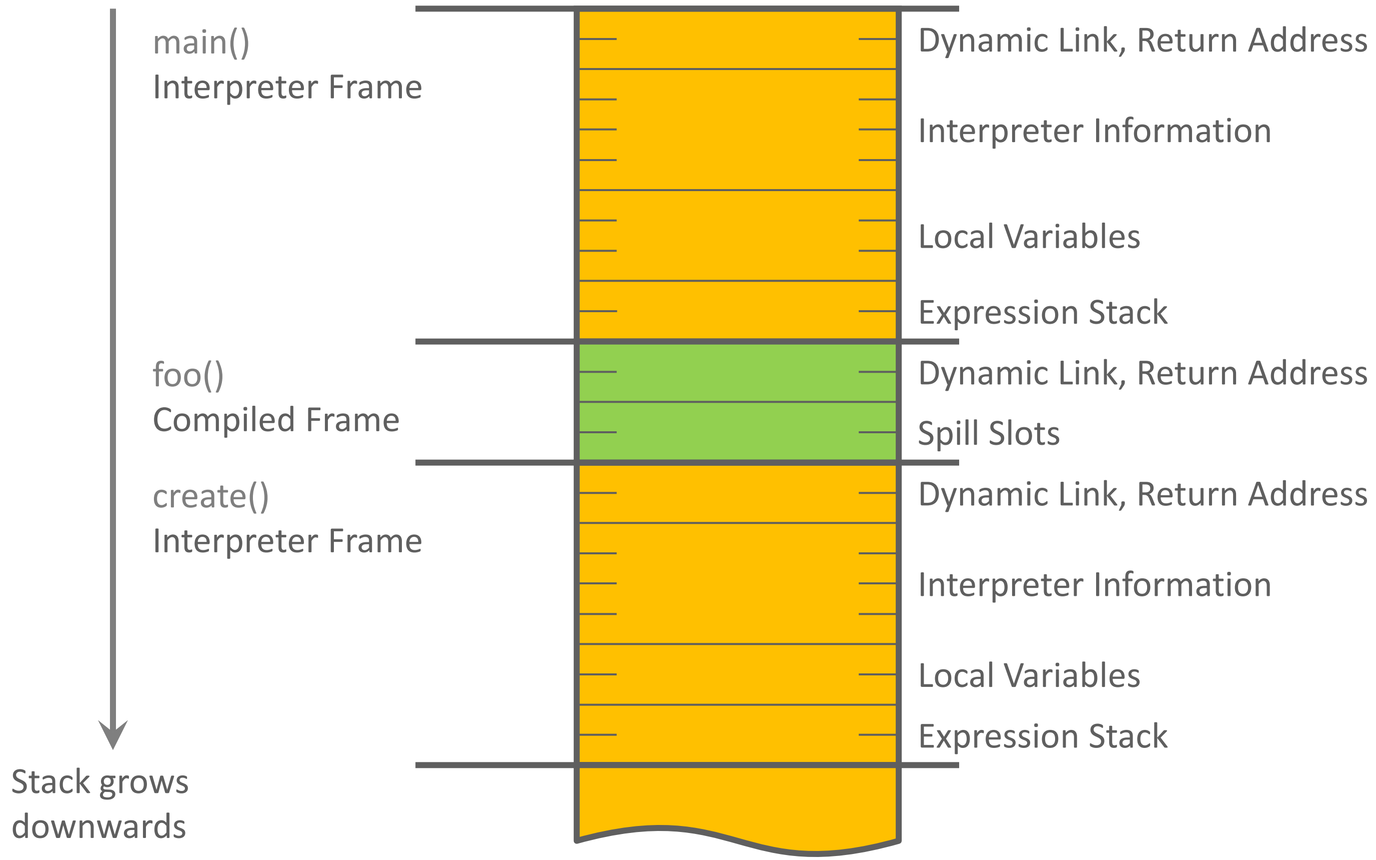


Machine code for foo():

```
enter
call create
move [eax + 8] -> esi
leave
return
```

ORACLE

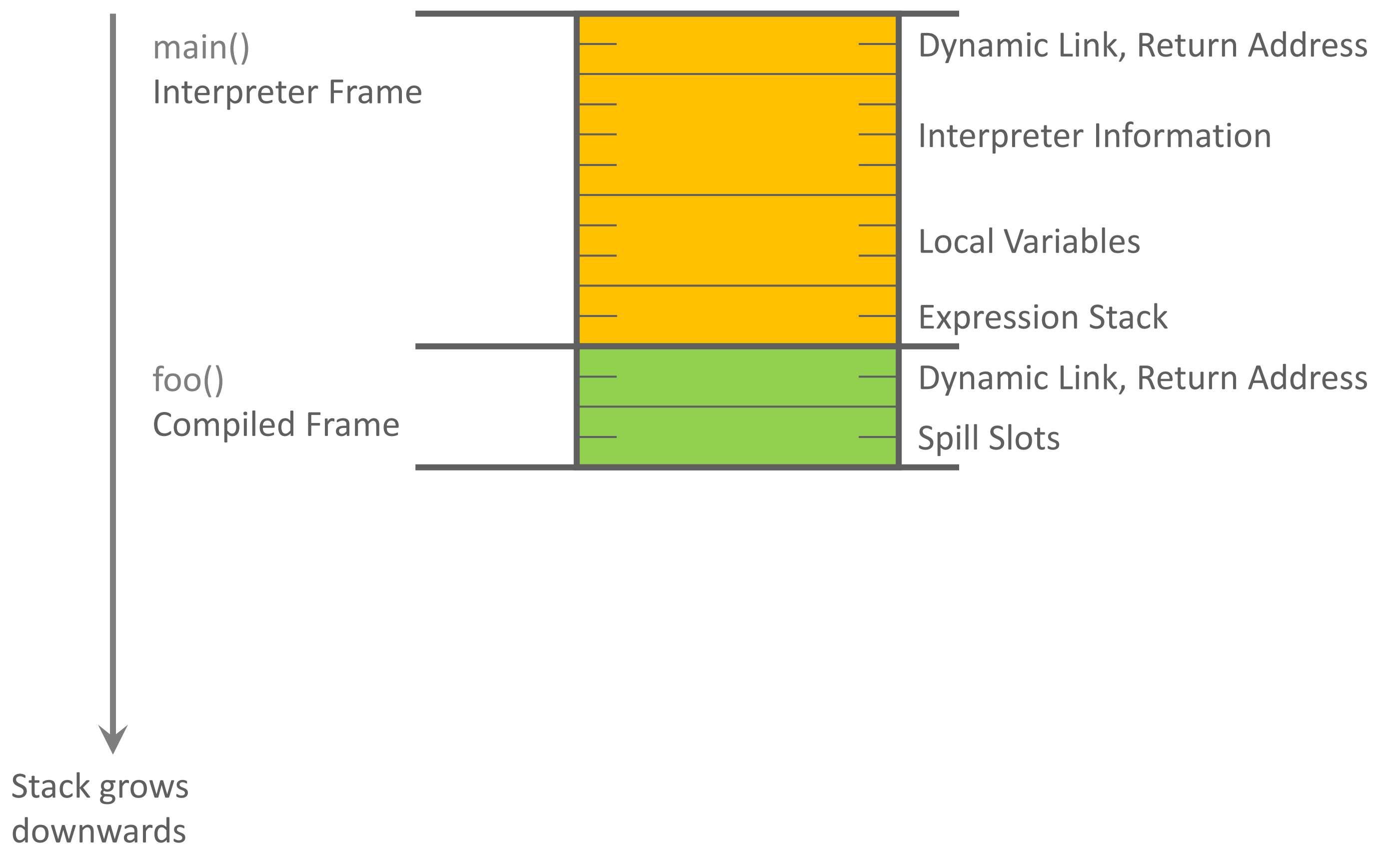
Deoptimization



Machine code for foo():

```
jump Interpreter
call create
call Deoptimization
leave
return
```

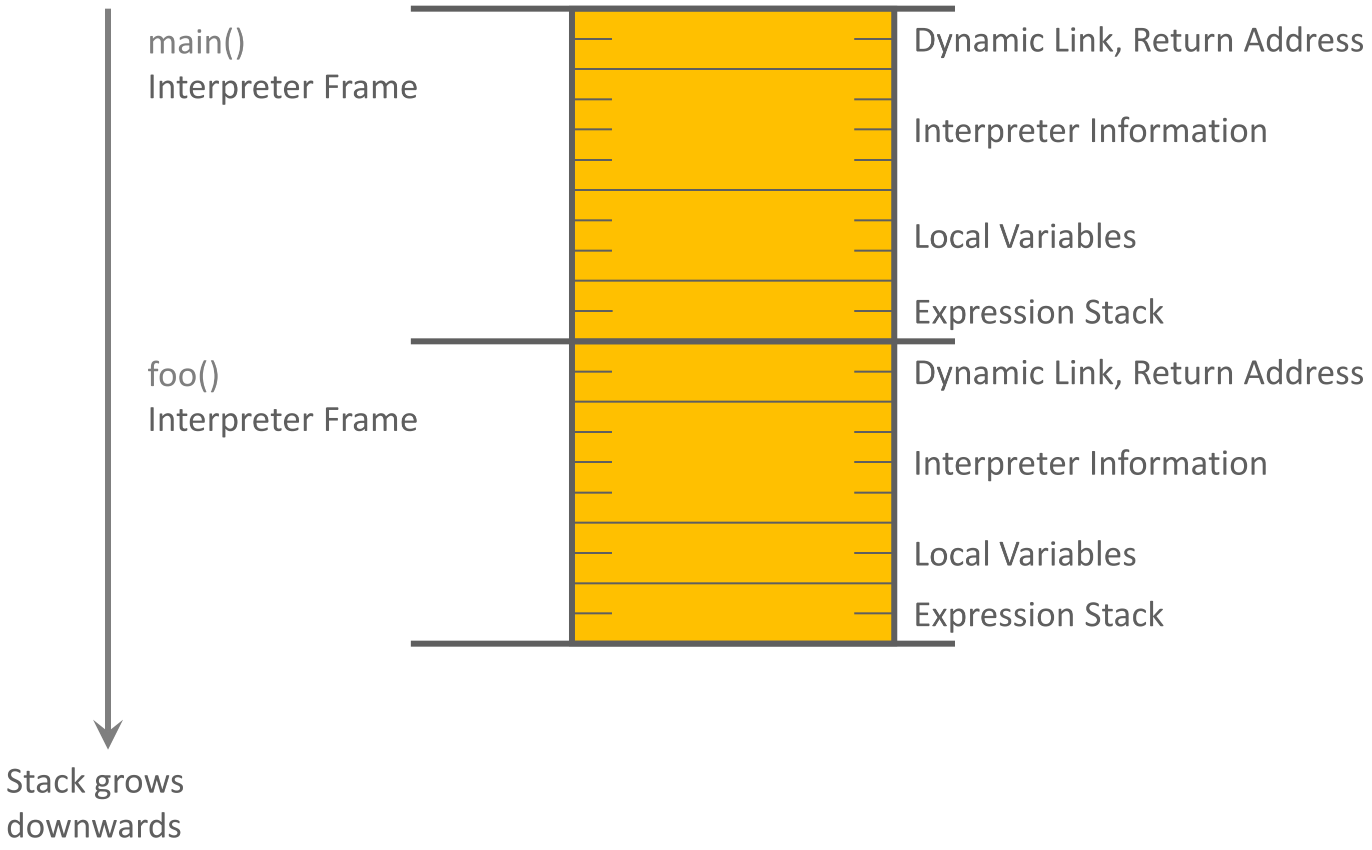
Deoptimization



Machine code for foo():

```
jump Interpreter
call create
call Deoptimization
leave
return
```

Deoptimization



Machine code for foo():

```
jump Interpreter
call create
call Deoptimization
leave
return
```

Example: Speculative Optimization

Java source code:

```
int f1;
int f2;

void speculativeOptimization(boolean flag) {
  f1 = 41;
  if (flag) {
    f2 = 42;
    return;
  }
  f2 = 43;
}
```

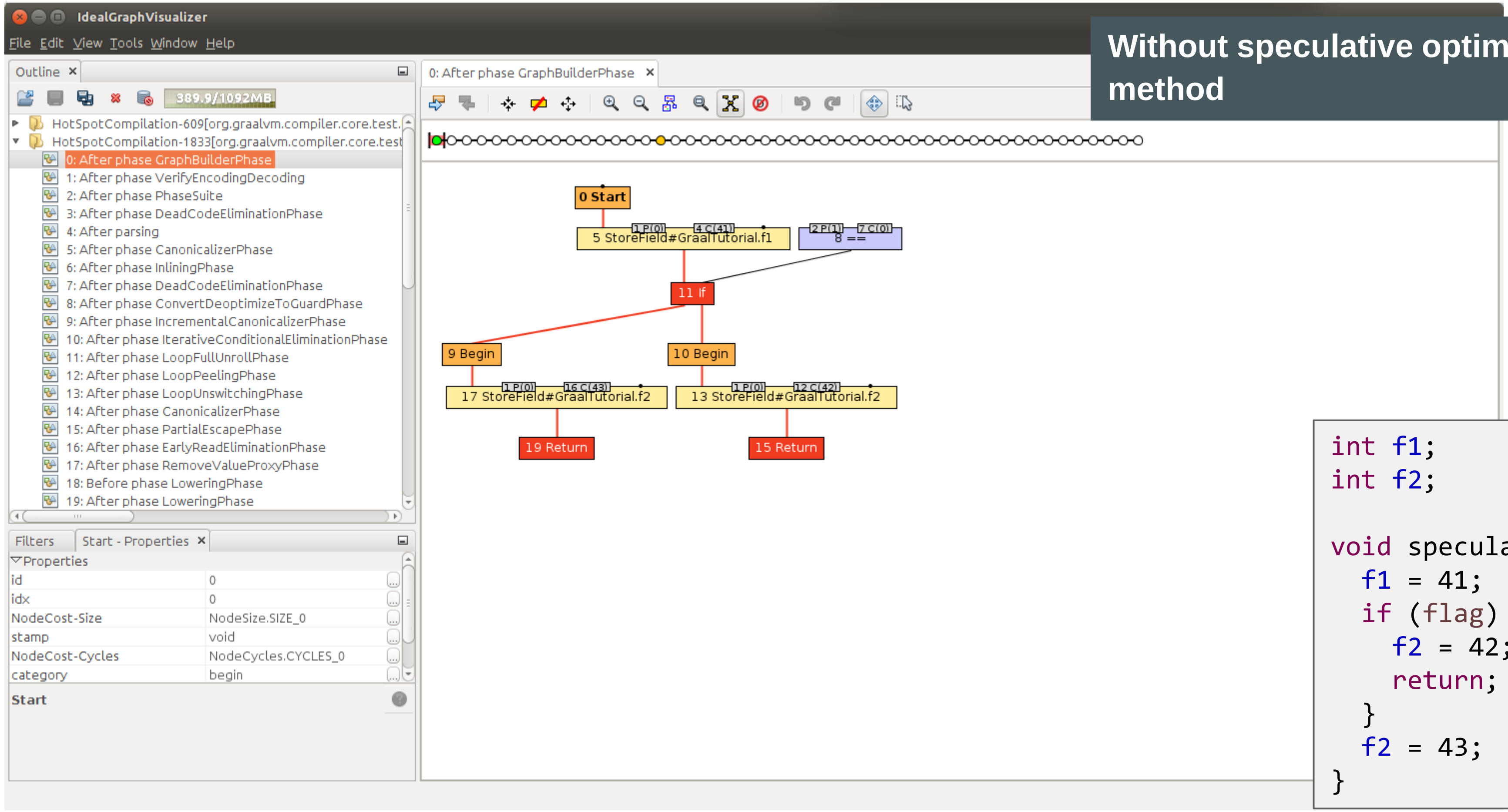
Assumption: method `speculativeOptimization` is always called with parameter `flag` set to `false`

Command line to run example:

```
mx igv &
mx unittest -Dgraal.Dump=:2 -Dgraal.MethodFilter=GraalTutorial.* GraalTutorial#testSpeculativeOptimization
```

The test case dumps two graphs: first with speculation, then without speculation

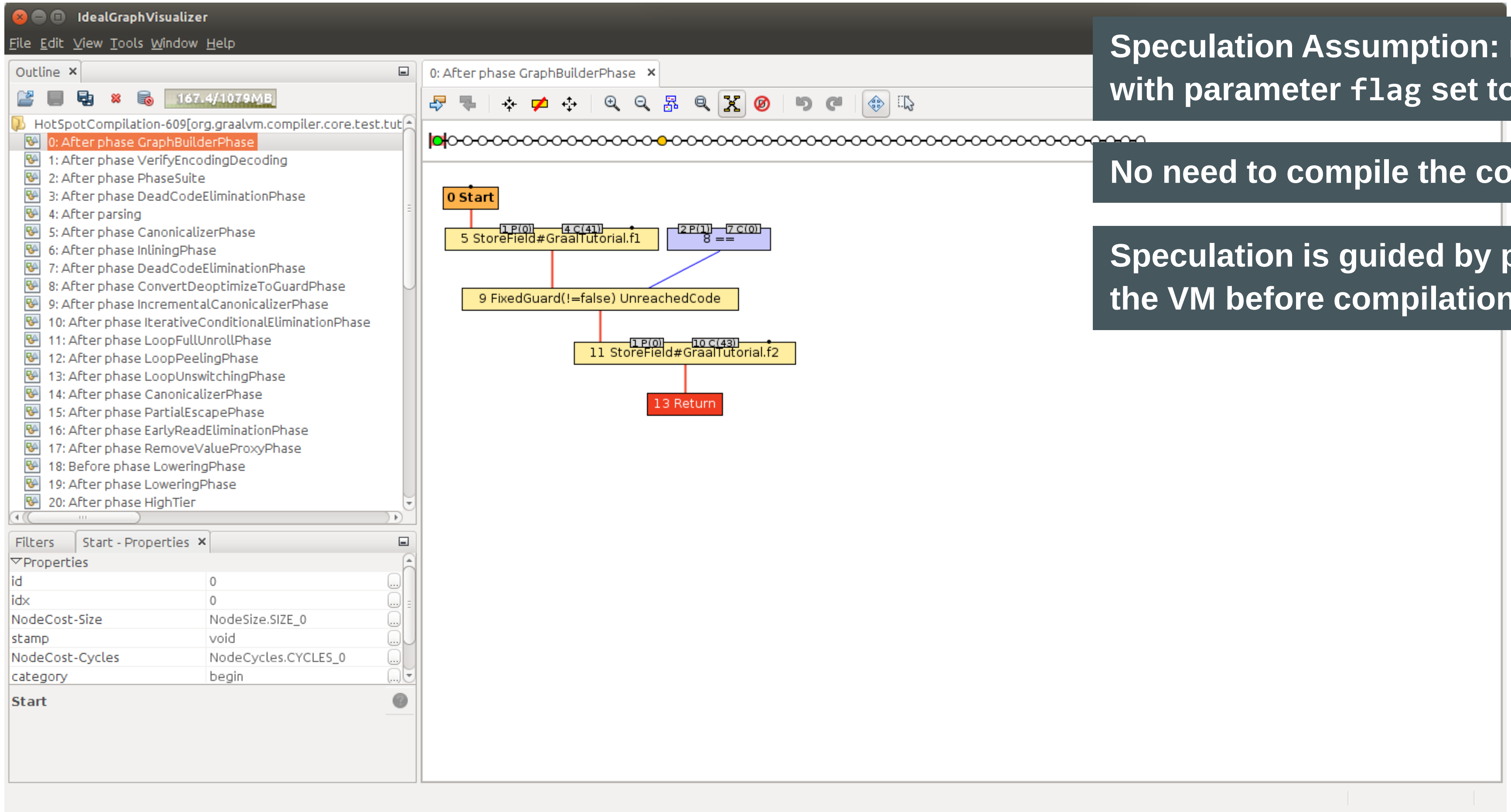
After Parsing without Speculation



Without speculative optimizations: graph covers the whole method

```java
int f1;
int f2;

void speculativeOptimization(boolean flag) {
  f1 = 41;
  if (flag) {
    f2 = 42;
    return;
  }
  f2 = 43;
}
```

After Parsing with Speculation

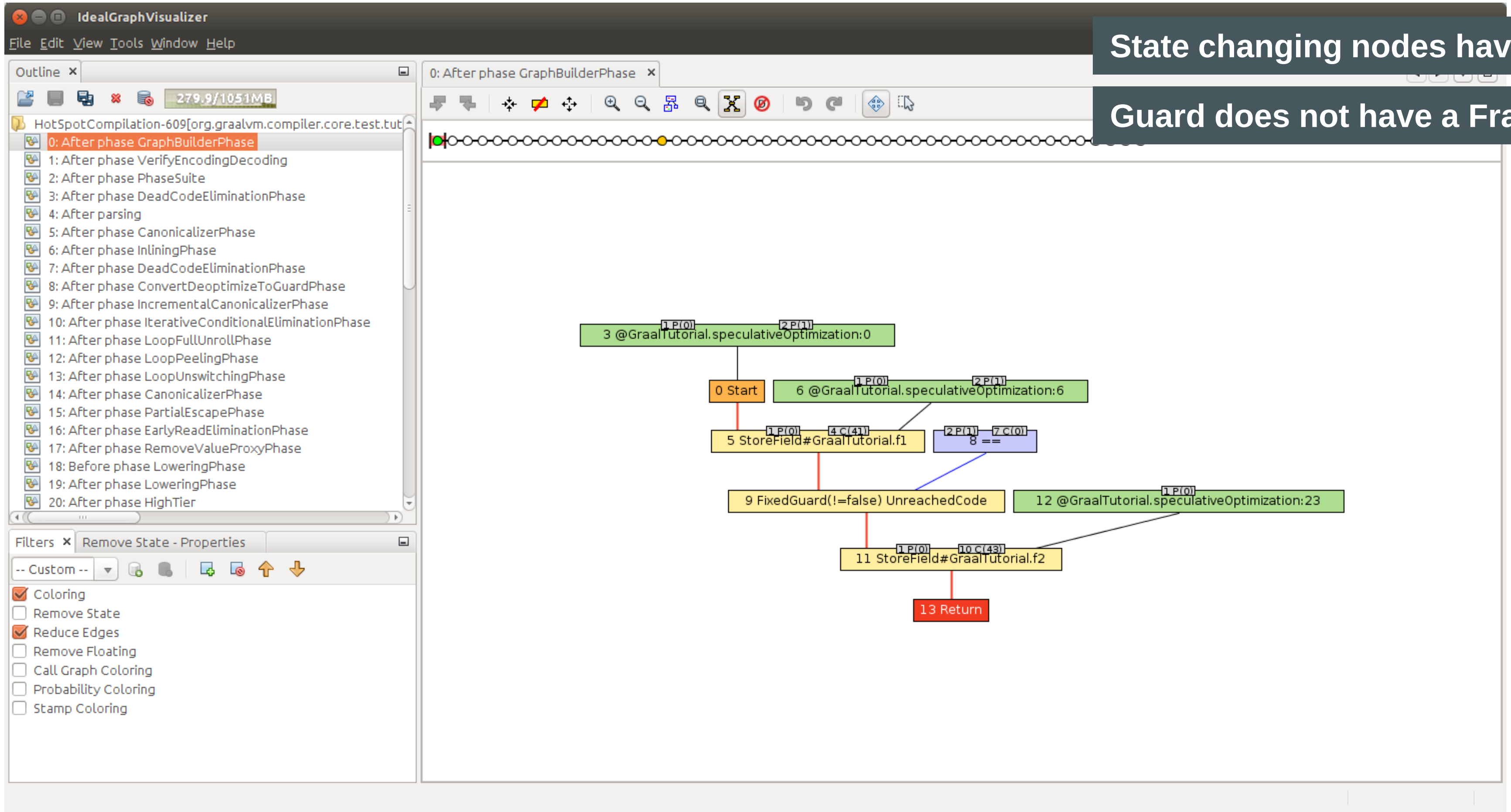


Speculation Assumption: method `test` is always called with parameter `flag` set to `false`

No need to compile the code inside the `if` block

Speculation is guided by profiling information collected by the VM before compilation

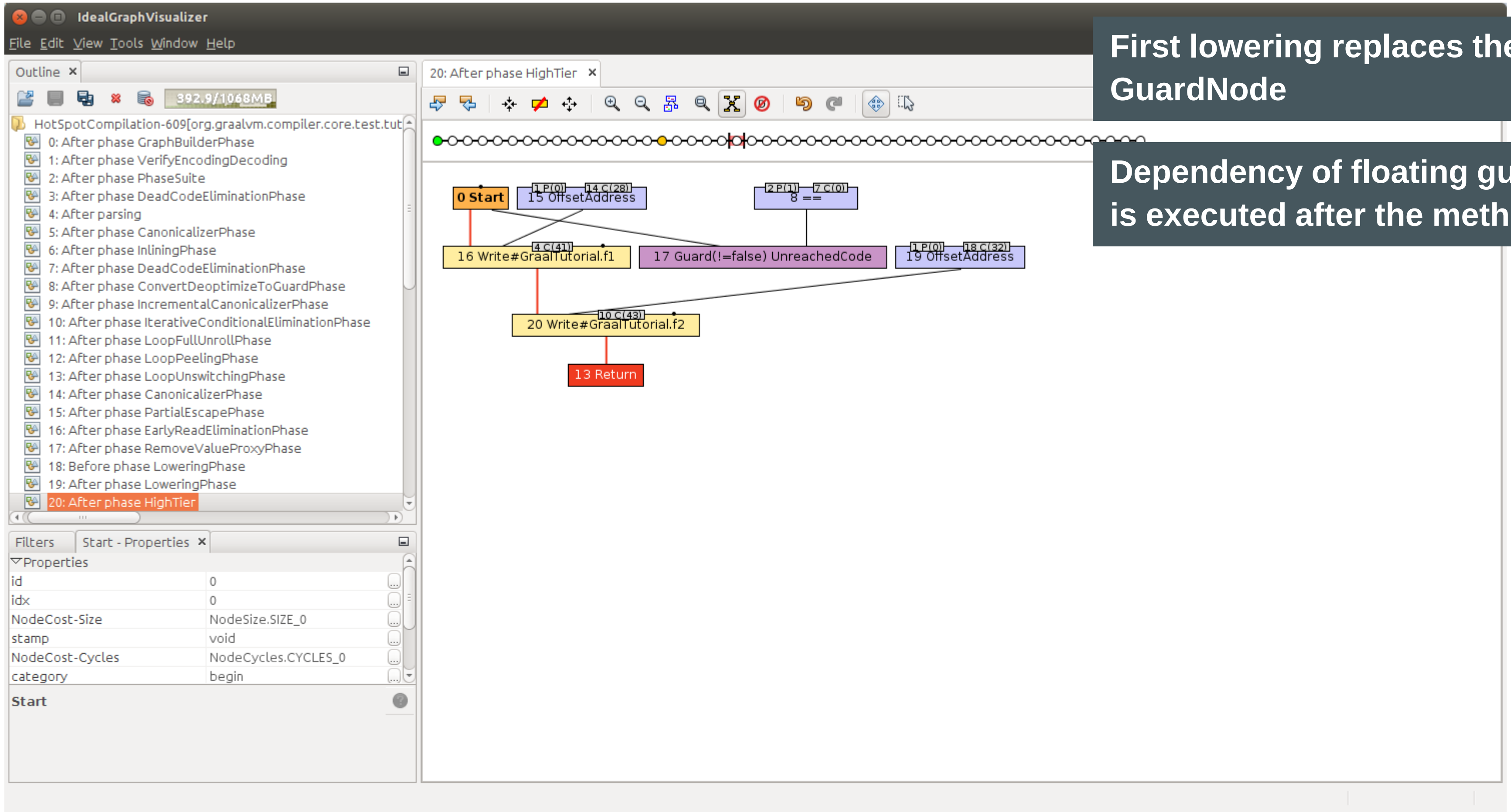
Frame states after Parsing



State changing nodes have a FrameState

Guard does not have a FrameState

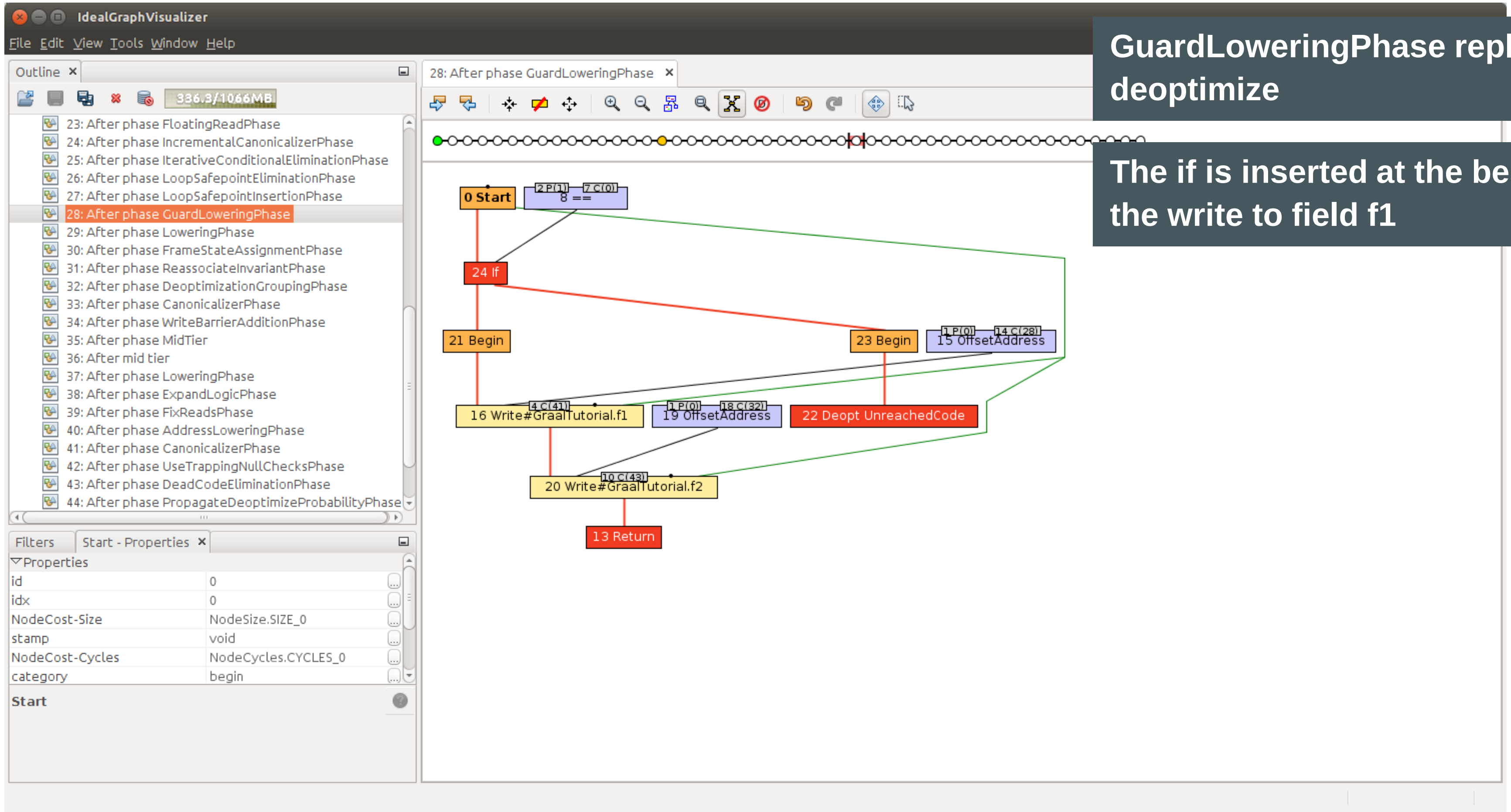
After Lowering: Guard is Floating



First lowering replaces the FixedGuardNode with a floating GuardNode

Dependency of floating guard on StartNode ensures guard is executed after the method start

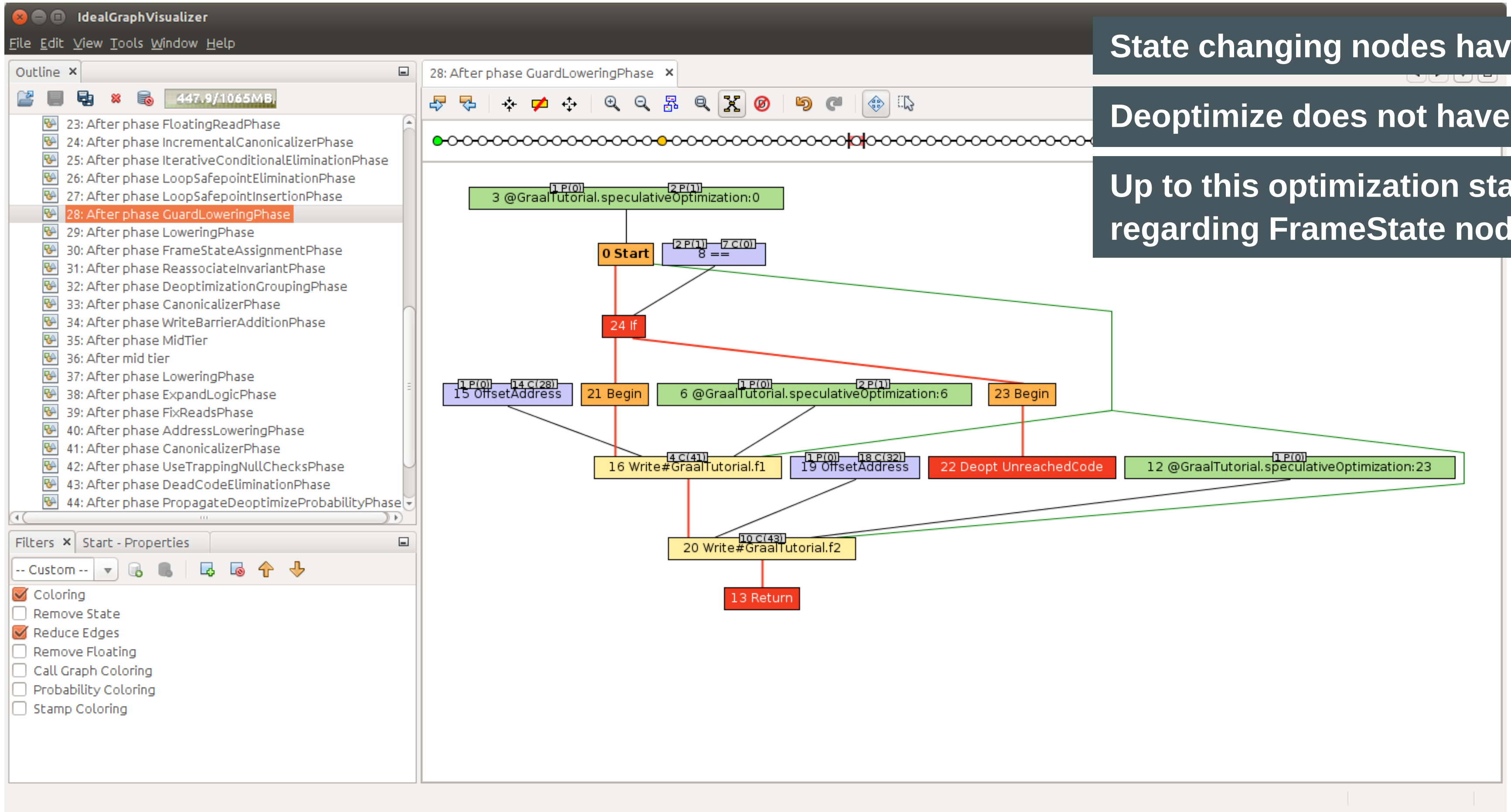
After Replacing Guard with If-Deoptimize



GuardLoweringPhase replaces GuardNode with if-deoptimize

The if is inserted at the best (earliest) position – it is before the write to field f1

Frame States are Still Unchanged

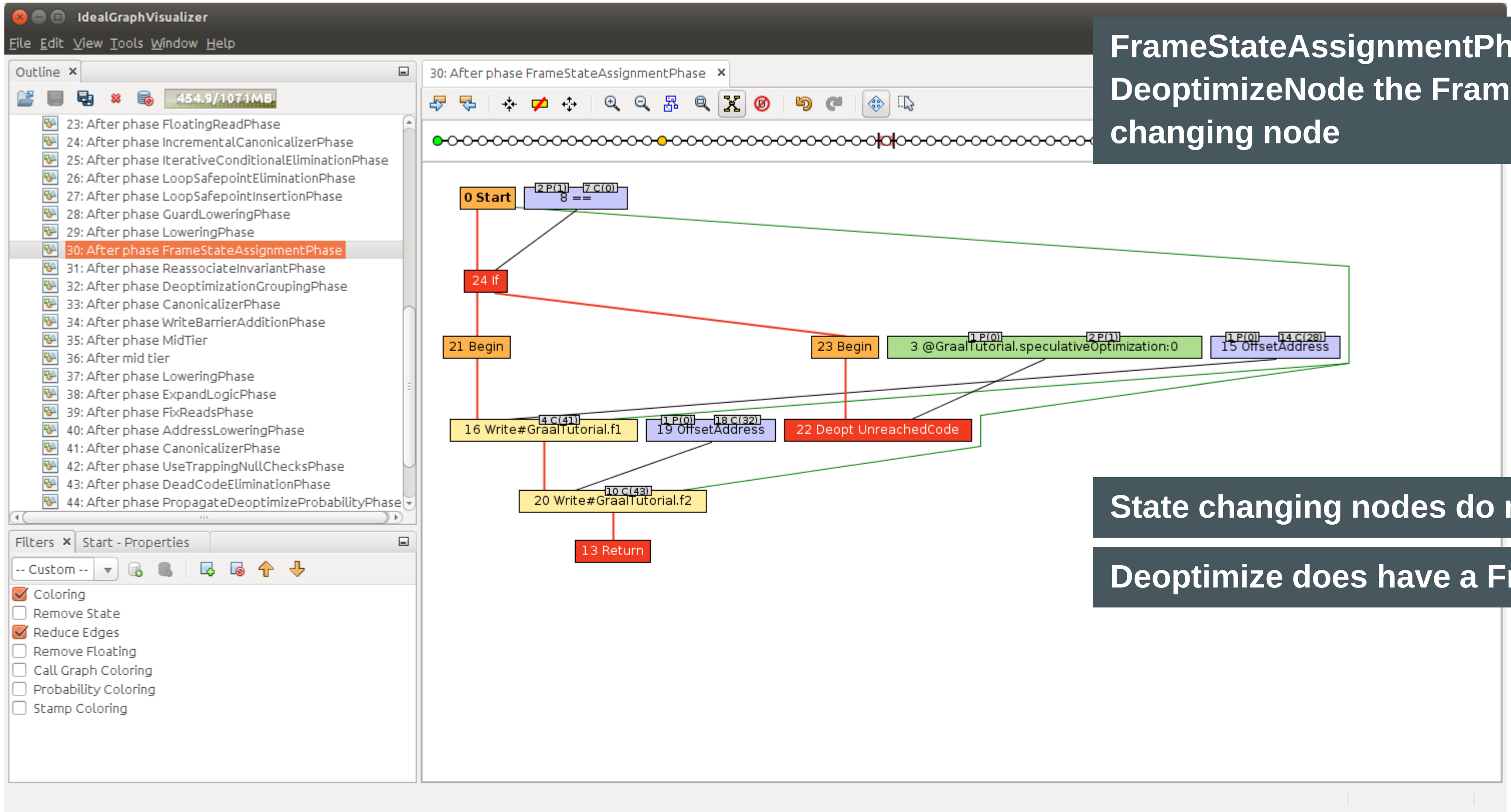


State changing nodes have a FrameState

Deoptimize does not have a FrameState

Up to this optimization stage, nothing has changed regarding FrameState nodes

After FrameStateAssignmentPhase

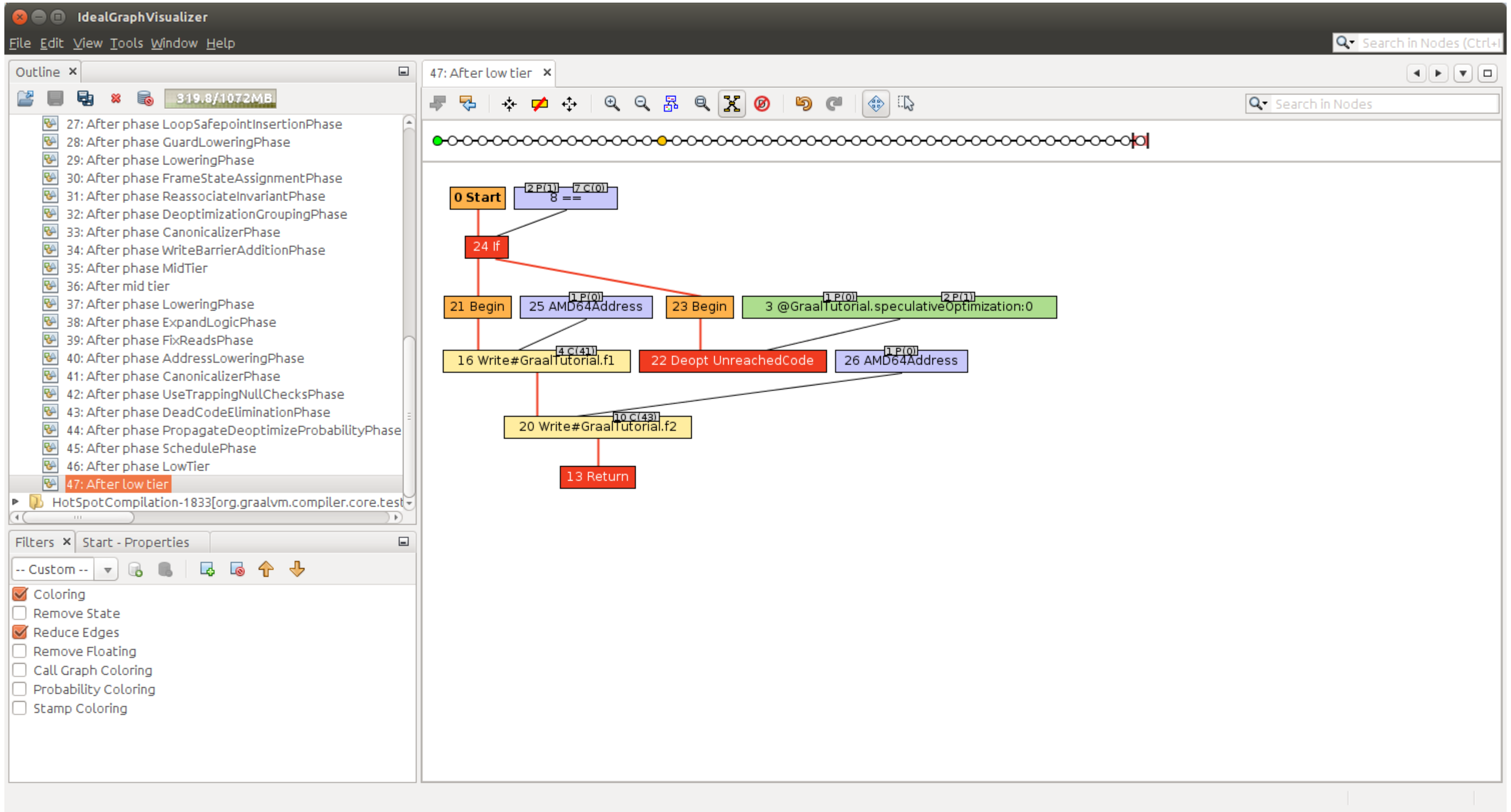


FrameStateAssignmentPhase assigns every DeoptimizeNode the FrameState of the preceding state changing node

State changing nodes do not have a FrameState

Deoptimize does have a FrameState

Final Graph After Optimizations



Frame States: Two Stages of Compilation

	First Stage: Guard Optimizations	Second Stage: Side-effects Optimizations
`FrameState` is on ...	... nodes with side effects	... nodes that deoptimize
Nodes with side effects ...	... cannot be moved within the graph	... can be moved
Nodes that deoptimize ...	... can be moved within the graph	... cannot be moved
	New guards can be introduced anywhere at any time. Redundant guards can be eliminated. Most optimizations are performed in this stage.	Nodes with side effects can be reordered or combined.
`StructuredGraph.guardsStage =`	`GuardsStage.FLOATING_GUARDS`	`GuardsStage.AFTER_FSA`
Graph is in this stage ...	... before `GuardLoweringPhase`	... after `FrameStateAssignmentPhase`

Implementation note: Between `GuardLoweringPhase` and `FrameStateAssignmentPhase`, the graph is in stage `GuardsStage.FIXED_DEOPTS`. This stage has no benefit for optimization, because it has the restrictions of both major stages.

ORACLE

Optimizations on Floating Guards

- Redundant guards are eliminated
  - Automatically done by global value numbering
  - Example: multiple bounds checks on the same array
- Guards are moved out of loops
  - Automatically done by scheduling
  - GuardLoweringPhase assigns every guard a dependency on the reverse postdominator of the original fixed location
    - The block whose execution guarantees that the original fixed location will be reached too
  - For guards in loops (but not within a if inside the loop), this is a block before the loop
- Speculative optimizations can move guards further up
  - This needs a feedback cycle with the interpreter: if the guard actually triggers deoptimization, subsequent recompilation must not move the guard again

ORACLE

JVMCI

JVMCI Interfaces

- Interfaces for everything coming from a .class file
  - `JavaType`, `JavaMethod`, `JavaField`, `ConstantPool`, `Signature`, …
- Provider interfaces
  - `MetaAccessProvider`, `CodeCacheProvider`, `ConstantReflectionProvider`, …
- VM implements the interfaces, Graal uses the interfaces

- `CompilationResult` is produced by Graal
  - Machine code in `byte[]` array
  - Pointer map information for garbage collection
  - Information about local variables for deoptimization
  - Information about speculations performed during compilation

Dynamic Class Loading

- From the Java specification: Classes are loaded and initialized as late as possible
  - Code that is never executed can reference a non-existing class, method, or field
  - Invoking a method does not make the whole method executed
  - Result: Even a frequently executed (= compiled) method can have parts that reference non-existing elements
  - The compiler must not trigger class loading or initialization, and must not throw linker errors
- JVMCI distinguishes between unresolved and resolved elements
  - Interfaces for unresolved elements: `JavaType`, `JavaMethod`, `JavaField`
    - Only basic information: name, field kind, method signature
  - Interfaces for resolved elements: `ResolvedJavaType`, `ResolvedJavaMethod`, `ResolvedJavaField`
    - All the information that Java reflection gives you, and more
- Graal as a JIT compiler does not trigger class loading
  - Replace accesses to unresolved elements with deoptimization, let interpreter then do the loading and linking
- Graal as a static analysis framework can trigger class loading

ORACLE

Important Provider Interfaces

```
public interface MetaAccessProvider {
  ResolvedJavaType lookupJavaType(Class<?> clazz);
  ResolvedJavaMethod lookupJavaMethod(Executable reflectionMethod);
  ResolvedJavaField lookupJavaField(Field reflectionField);
  ...
}
```

Convert Java reflection objects to Graal API

```
public interface ConstantReflectionProvider {
  Boolean constantEquals(Constant x, Constant y);
  Integer readArrayLength(JavaConstant array);
  ...
}
```

Look into constants – note that the VM can deny the request, maybe it does not even have the information

It breaks the compiler-VM separation to get the raw object encapsulated in a Constant – so there is no method for it

```
public interface CodeCacheProvider {
  InstalledCode installCode(ResolvedJavaMethod method, CompiledCode compiledCode,
                  InstalledCode installedCode, SpeculationLog log, boolean isDefault);

  void invalidateInstalledCode(InstalledCode installedCode);

  TargetDescription getTarget();
  ...
}
```

Install compiled code into the VM

ORACLE

Example: Get Bytecodes of a Method

```
/* Entry point object to the Graal API from the hosting VM. */
RuntimeProvider runtimeProvider = Graal.getRequiredCapability(RuntimeProvider.class);

/* The default backend (architecture, VM configuration) that the hosting VM is running on. */
Backend backend = runtimeProvider.getHostBackend();

/* Access to all of the Graal API providers, as implemented by the hosting VM. */
Providers providers = backend.getProviders();

/* The provider that allows converting reflection objects to Graal API. */
MetaAccessProvider metaAccess = providers.getMetaAccess();

Method reflectionMethod = String.class.getDeclaredMethod("hashCode");
ResolvedJavaMethod method = metaAccess.lookupJavaMethod(reflectionMethod);

/* ResolvedJavaMethod provides all information that you want about a method, for example, the bytecodes. */
byte[] bytecodes = method.getCode();

/* BytecodeDisassembler shows you how to iterate bytecodes, how to access type information, and more. */
String disassembly = new BytecodeDisassembler().disassemble(method);
```

Command line to run example:

```
mx unittest GraalTutorial#testGetBytecodes
```

Compiler Intrinsics

Compiler Intrinsics

- Implemented using an invocation plugin
  - A graph builder plugin for a single fixed method
  - Invoked by bytecode parser
- Use cases
  - Use a special hardware instruction instead of calling a Java method
  - Replace a runtime call into the VM with low-level Java code
- Implementation steps
  - Define a node for the intrinsic functionality
  - Instantiate the node in a graph builder plugin
  - Define a LIR instruction for your functionality
  - Generate this LIR instruction in the LIRLowerable.generate() method of your node
  - Generate machine code in your LIRInstruction.emitCode() method

Example: Intrinsification of Integer.reverseBytes ()

Java source code:

```
static int intrinsicIntegerReverseBytes(int val) {
  return Integer.reverseBytes(val);
}
```

Java implementation of reverseBytes() uses bit operations

x86 provides an instruction: bswap

Command line to run example:

```
mx igv &
mx c1visualizer &
mx unittest -Dgraal.Dump= -Dgraal.MethodFilter=GraalTutorial.* GraalTutorial#testIntrinsicIntegerReverseBytes
```

C1Visualizer shows the LIR and generated machine code

Load the generated .cfg file with C1Visualzier

ORACLE

Node and Invocation Plugin

```
public final class ReverseBytesNode extends UnaryNode implements LIRLowerable {
  public ReverseBytesNode(ValueNode value) { ... }

  @Override
  public ValueNode canonical(CanonicalizerTool tool, ValueNode forValue) {
    if (forValue.isConstant()) {
      return ConstantNode.forInt(Integer.reverseBytes(forValue.asJavaConstant().asInt()));
    }
    return this;
  }

  @Override
  public void generate(NodeLIRBuilderTool gen) {
    Value result = gen.getLIRGeneratorTool().emitByteSwap(gen.operand(getValue()));
    gen.setResult(this, result);
  }
}
```

Node for intrinsified operation

Constant folding: call the original method

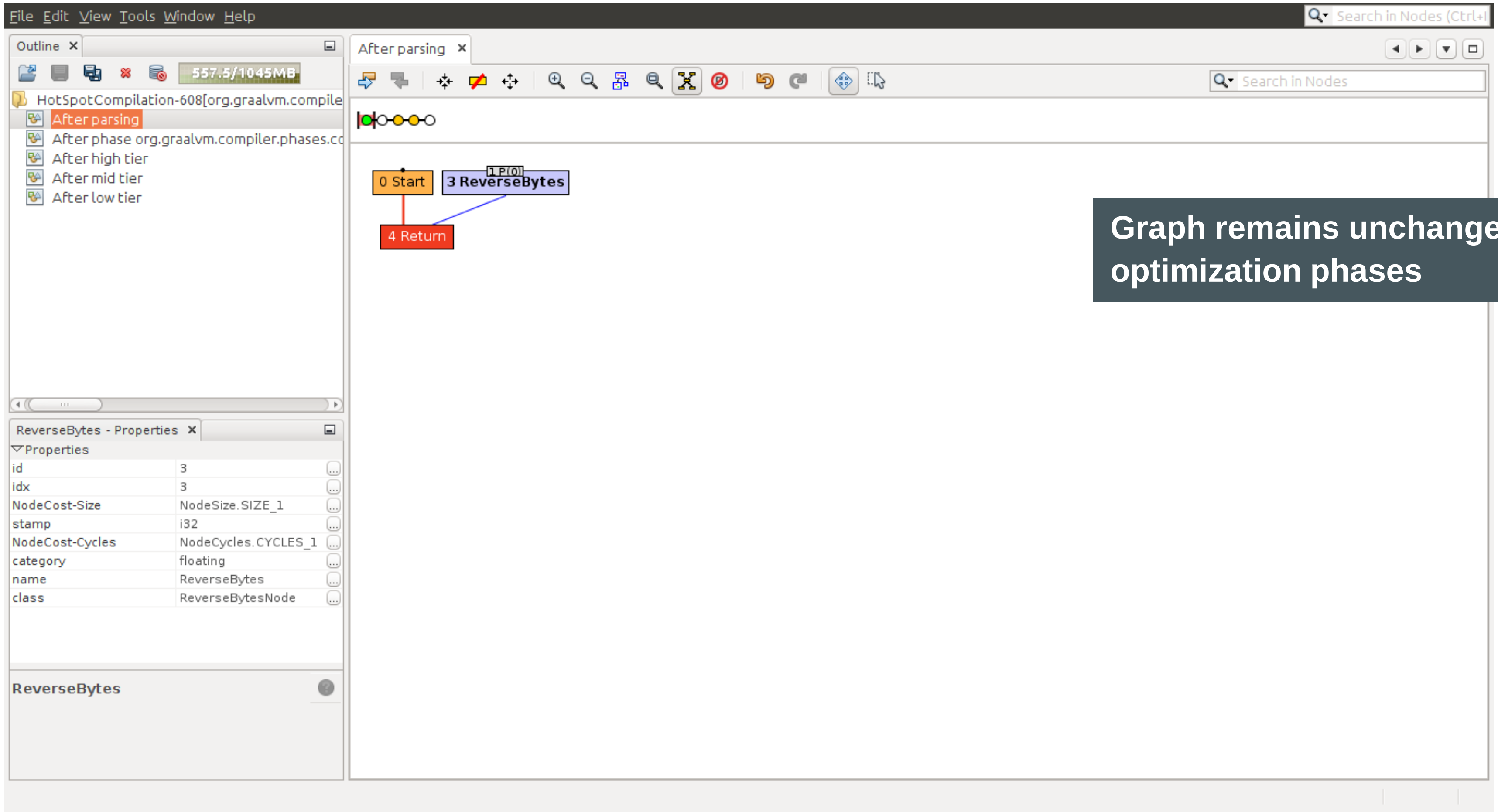
LIR Generation

```
Registration r = new Registration(plugins, Integer.class);
r.register1("reverseBytes", int.class, new InvocationPlugin() {
  @Override
  public boolean apply(GraphBuilderContext b, ResolvedJavaMethod targetMethod, Receiver receiver, ValueNode value) {
    b.push(JavaKind.Int, b.append(new ReverseBytesNode(value)));
    return true;
  }
});
```

Class and method that isintrinsified

Invoked by bytecode parser, create the node

After Parsing



Graph remains unchanged throughout all further optimization phases

ORACLE®

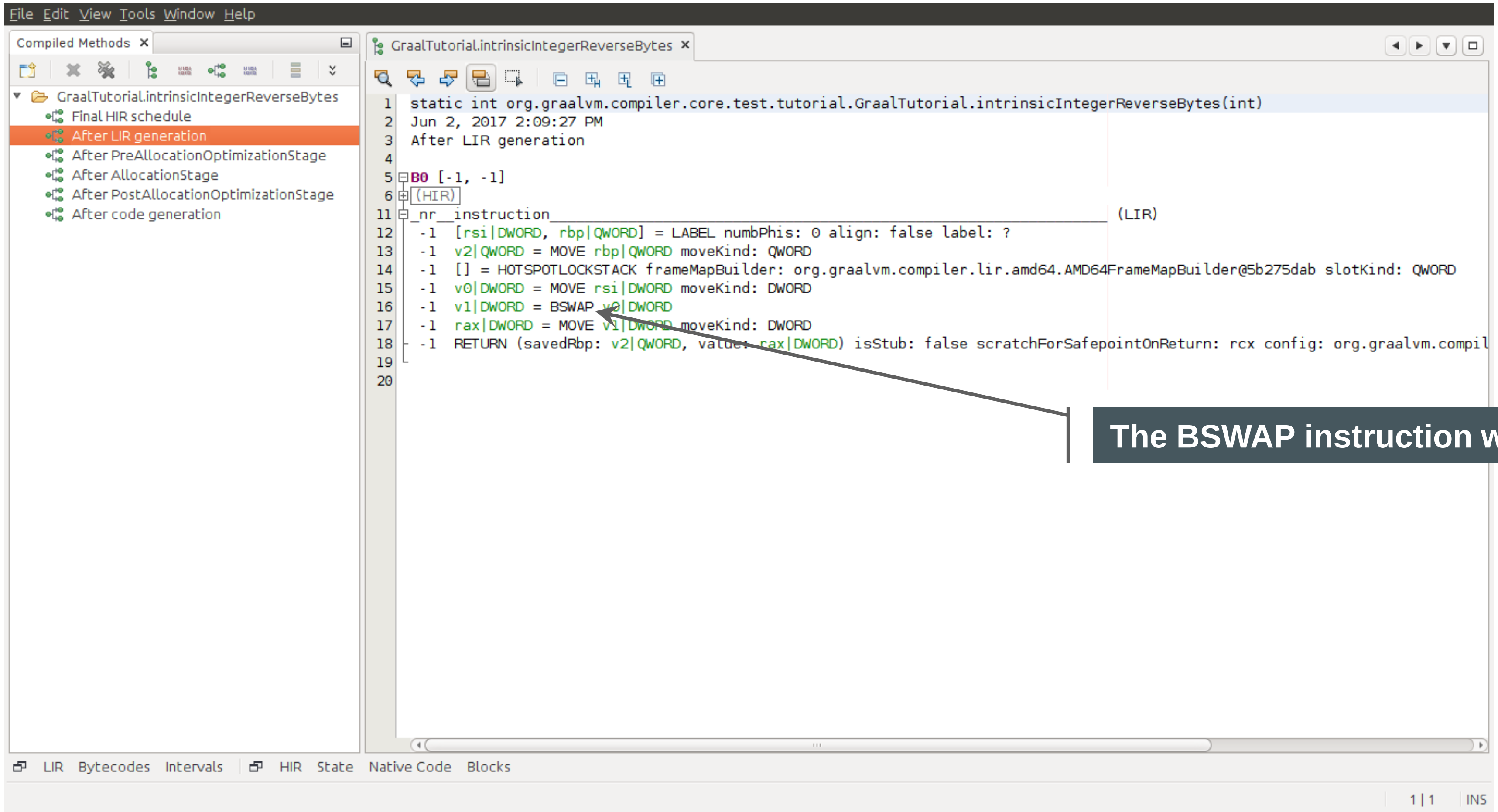
LIR Instruction

```java
@Opcode("BSWAP")
public final class AMD64ByteSwapOp extends AMD64LIRInstruction {
  public static final LIRInstructionClass<AMD64ByteSwapOp> TYPE = LIRInstructionClass.create(AMD64ByteSwapOp.class);

  @Use protected Value input;
  @Def({OperandFlag.REG, OperandFlag.HINT}) protected Value result;

  public AMD64ByteSwapOp(Value result, Value input) {
    super(TYPE);
    this.result = result;
    this.input = input;
  }

  @Override
  public void emitCode(CompilationResultBuilder crb, AMD64MacroAssembler masm) {
    AMD64Move.move(crb, masm, result, input);
    switch ((AMD64Kind) input.getPlatformKind()) {
      case DWORD: masm.bswapl(ValueUtil.asRegister(result)); break;
      case QWORD: masm.bswapq(ValueUtil.asRegister(result)); break;
      default: throw GraalError.shouldNotReachHere();
    }
  }
}
```

LIR uses annotation to specify input, output, or temporary registers for an instruction

Finally the call to the assembler to emit the bits

LIR Before Register Allocation



The BSWAP instruction we are looking for

NodePlugin, GraphBuilderConfiguration

- `InvocationPlugin` is for a single, known method
- `NodePlugin` can intrinsify any invoke, field access, array access, ...
  - Overwrite the appropriate method
- Plugins are configured as part of the graph builder configuration
  - `GraphBuilderConfiguration` instance passed in to bytecode parser

ORACLE

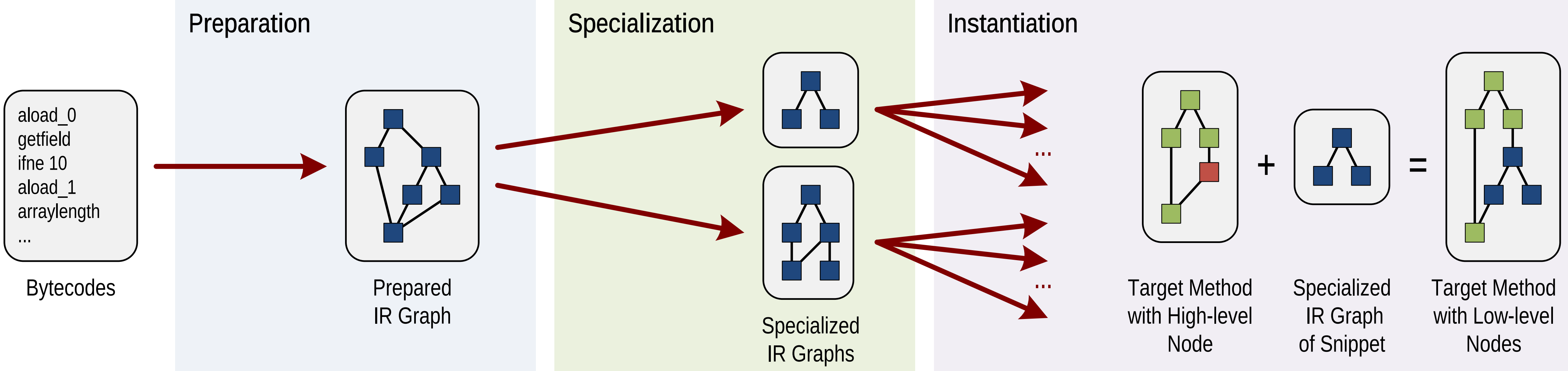
Snippets

The Lowering Problem

- How do you express the low-level semantics of a high-level operation?
- Manually building low-level IR graphs
  - Tedious and error prone
- Manually generating machine code
  - Tedious and error prone
  - Probably too low level (no more compiler optimizations possible after lowering)

- Solution: Snippets
  - Express the semantics of high-level Java operations in low-level Java code
    - `Word` type representing a machine word allows raw memory access
  - Simplistic view: replace a high-level node with an inlined method
  - To make it work in practice, a few more things are necessary

Snippet Lifecycle



	Preparation	Specialization	Instantiation
Frequency:	Once	Few Times	Many Times
Steps:	Java Bytecode Parsing Exhaustive Method Inlining Node Intrinsification Constant Folding, Canonicalization	Graph Duplication Constant Parameter Replacement Node Intrinsification Constant Folding, Canonicalization	Graph Duplication Graph Inlining in Target Method Constant Folding, Canonicalization

Example: Snippets for Lowering

Java source code:

```
static int identityHashCodeUsage (Object obj) {
  return System.identityHashCode(obj);
}
```

Command line to run example:

```
mx igv &
mx unittest -Dgraal.Dump=:2 -Dgraal.DebugStubsAndSnippets=true GraalTutorial#testIdentityHashCodeUsage
```

Snippet: Fast Access to Identity Hash Code

```
@Snippet
static int identityHashCodeSnippet(Object x) {
  if (probability(NOT_FREQUENT_PROBABILITY, x == null)) {
    return 0;
  }

  Word mark = loadWordFromObject(x, markOffset());

  final Word biasedLock = mark.and(
                      biasedLockMaskInPlace());
  if (probability(FAST_PATH_PROBABILITY,
                      biasedLock.equal(WordFactory.unsigned(
                      unlockedMask())))) {
    int hash = (int) mark.unsignedShiftRight(
                     identityHashCodeShift()).rawValue();
    if (probability(FAST_PATH_PROBABILITY,
                hash != uninitializedIdentityHashCodeValue())) {
      return hash;
    }
  }

  return identityHashCode(IDENTITY_HASHCODE, x);
}
```

The snippet is in class `HashCodeSnippets`

Node intrinsic

Constant folding during snippet parsing

Machine-word sized value

Node Intrinsics

```
final class BranchProbabilityNode extends ... {

  BranchProbabilityNode(ValueNode probability, ValueNode condition) { ... }

  @NodeIntrinsic
  static native boolean probability(double probability, boolean condition);
```

Calling the node intrinsic reflectively instantiates the node using the matching constructor

```
class ForeignCallNode extends ... {

  static boolean intrinsify(GraphBuilderContext b, ResolvedJavaMethod targetMethod,
          @InjectedNodeParameter Stamp returnStamp,  @InjectedNodeParameter ForeignCallsProvider foreignCalls,
          ForeignCallDescriptor descriptor,  ValueNode... arguments) {
    ...
  }
}


@NodeIntrinsic(ForeignCallNode.class)
public static native int identityHashCode(@ConstantNodeParameter ForeignCallDescriptor descriptor, Object object);
```

Factory method is more flexible than constructor

Parameter must be constant during snippet specialization

Snippet Instantiation

```
SnippetInfo identityHashCodeSnippet = snippet(HashCodeSnippets.class, "identityHashCodeSnippet",
                                              HotSpotReplacementsUtil.MARK_WORD_LOCATION);

void lower(IdentityHashCodeNode node, LoweringTool tool) {
  StructuredGraph graph = node.graph();

  Arguments args = new Arguments(identityHashCodeSnippet, graph.getGuardsStage(), tool.getLoweringStage());
  args.add("thisObj", node.object);


  SnippetTemplate template = template(args);
  template.instantiate(providers.getMetaAccess(), node, SnippetTemplate.DEFAULT_REPLACER, args);
}
```

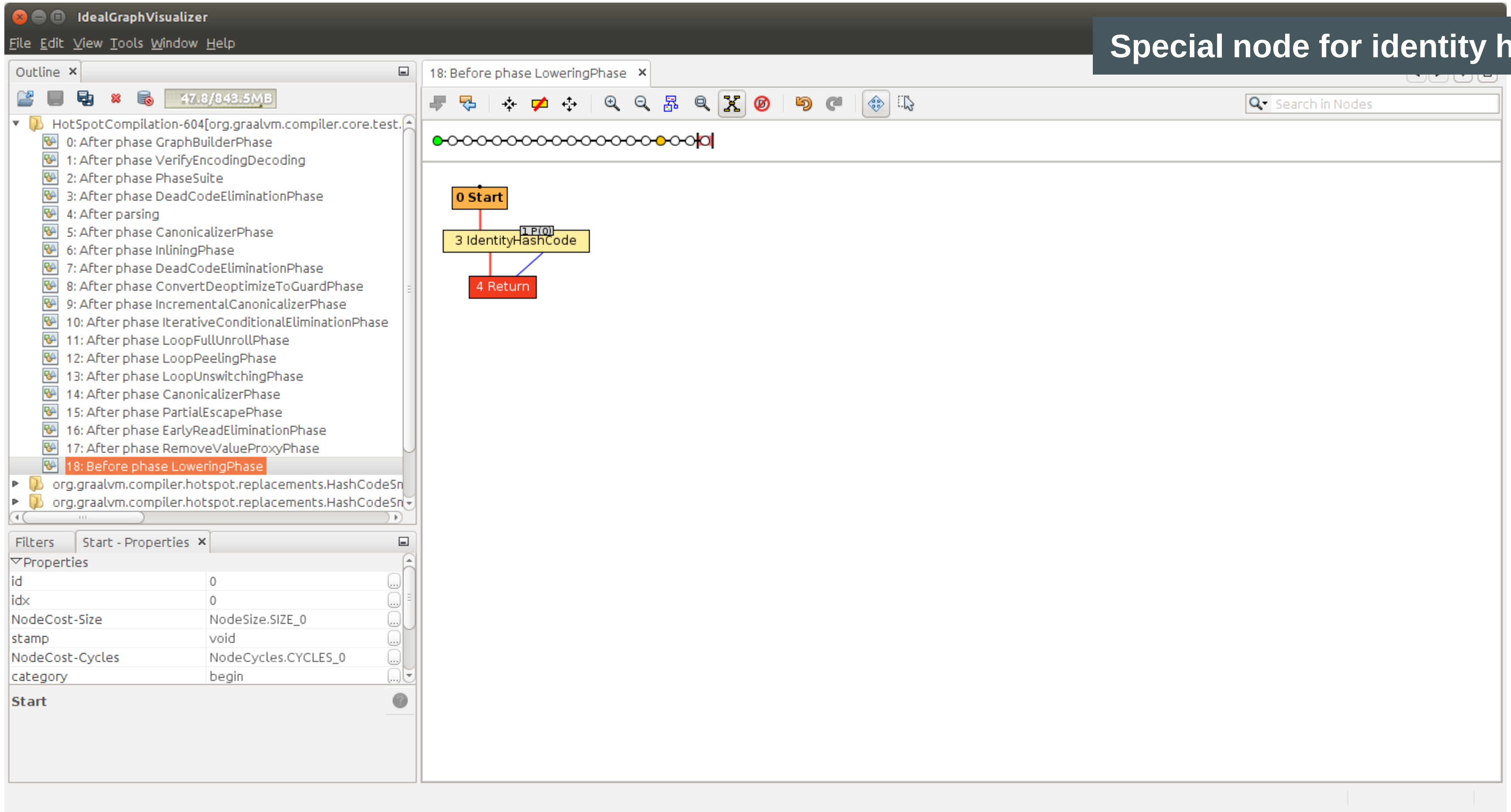
Memory locations that are private to the snippet

Node argument: formal parameter of snippet is replaced with this node

Snippet preparation and specialization

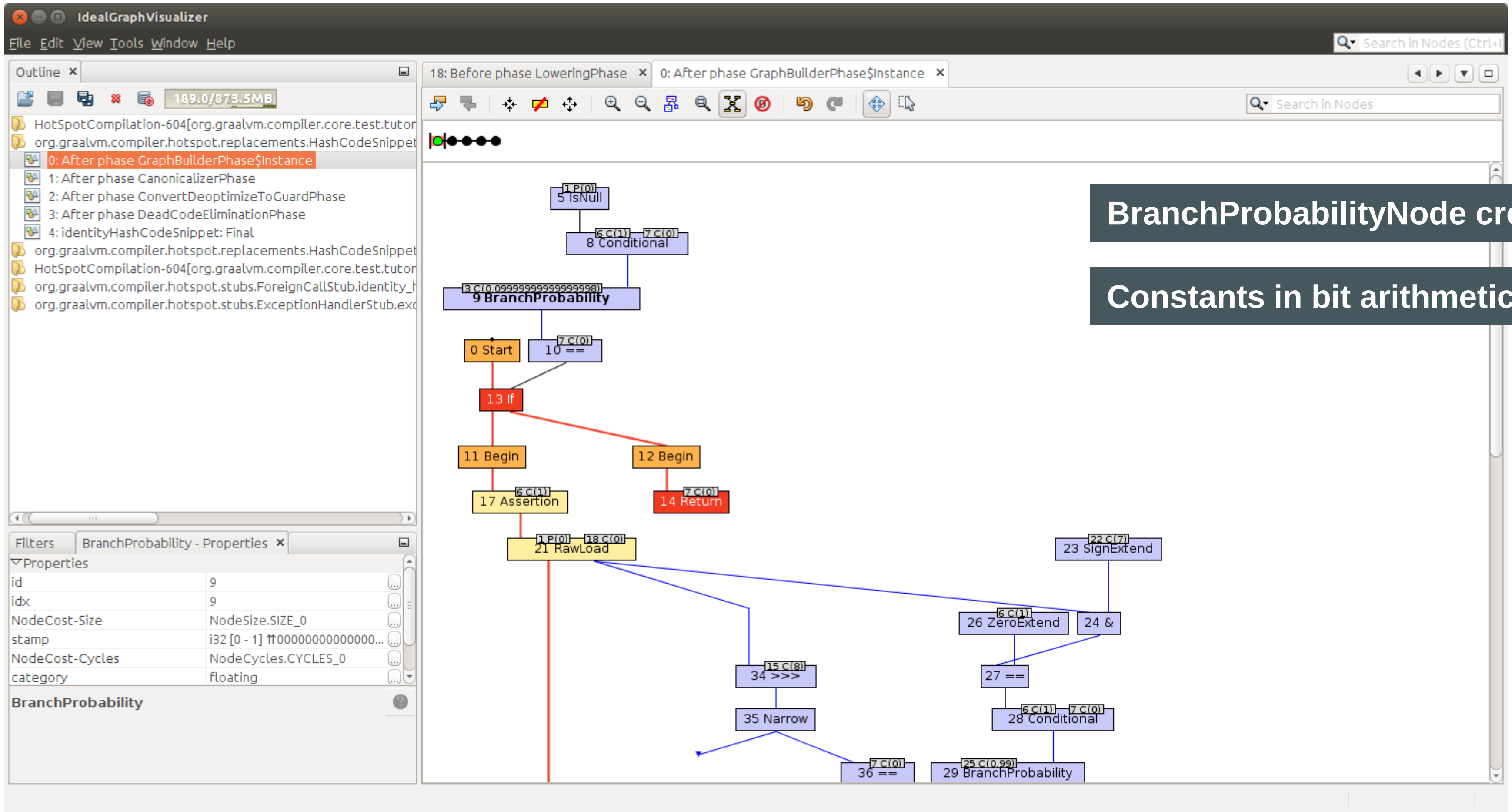
Snippet instantiation

Method Before Lowering



Special node for identity hash code access

Snippet After Parsing

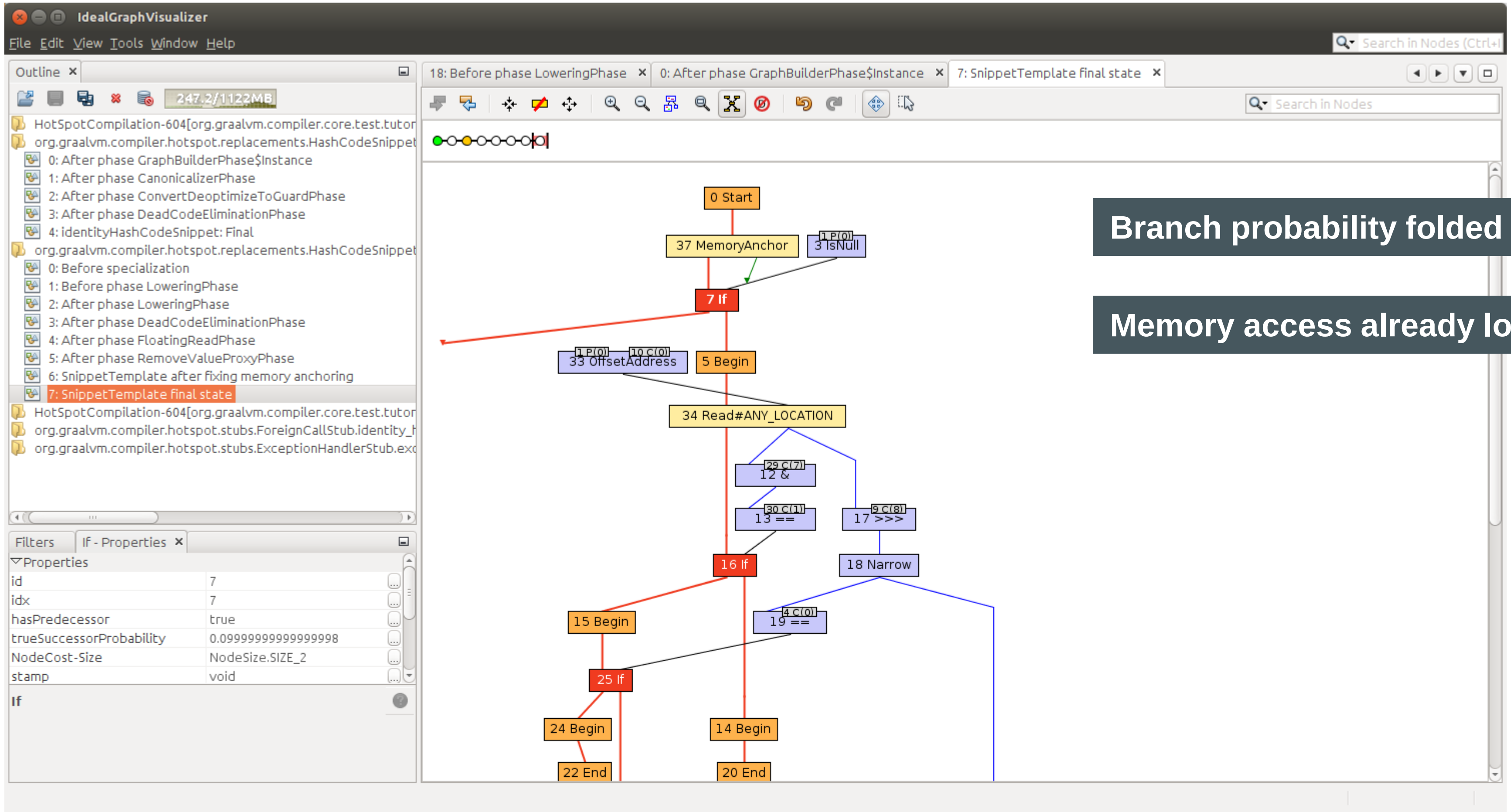


BranchProbabilityNode created by node intrinisc

Constants in bit arithmetic are from folded methods

ORACLE

Snippet After Specialization

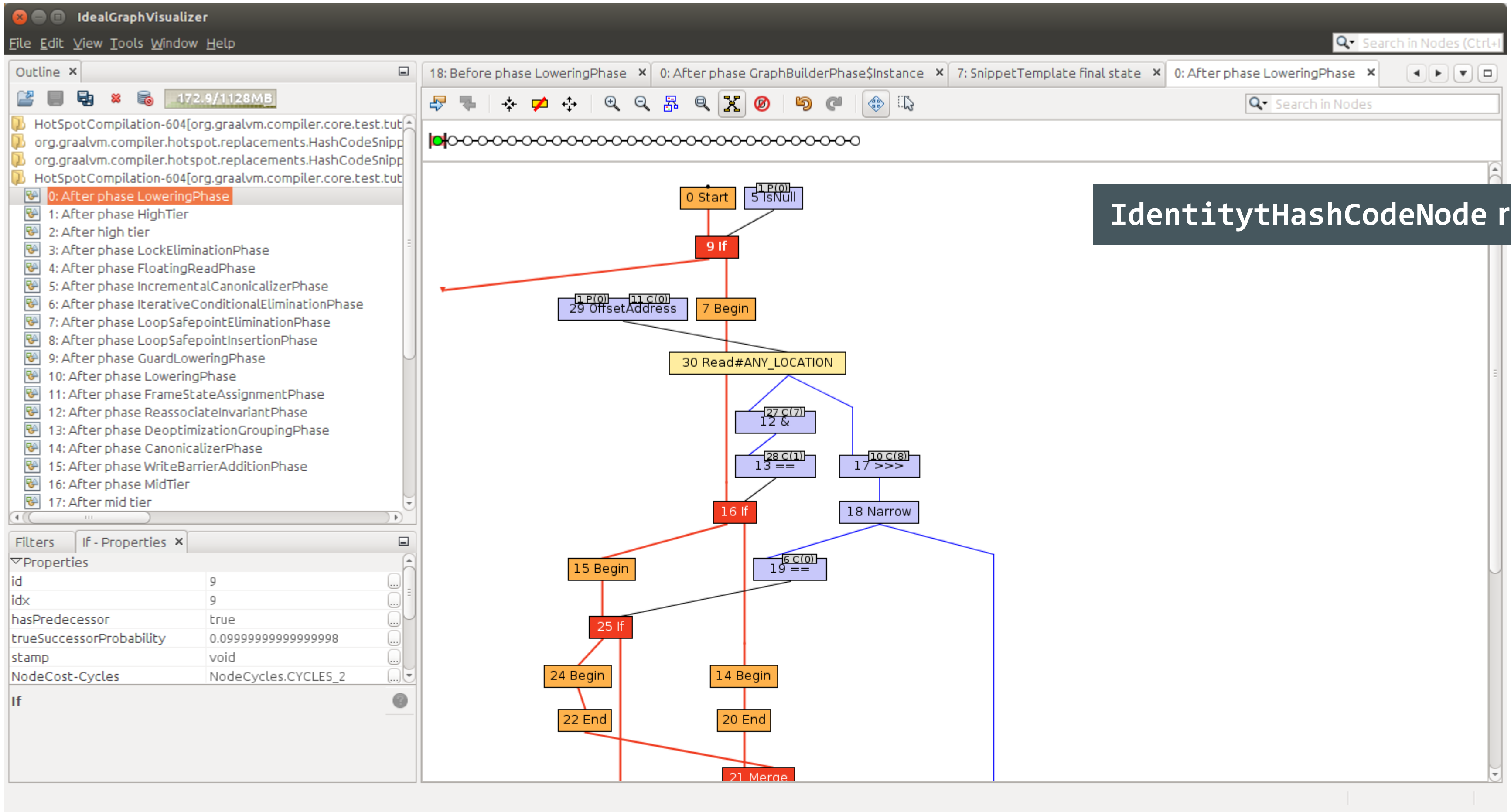


Branch probability folded into `IfNode`

Memory access already lowered

ORACLE

Method After Lowering



IdentitytHashCodeNode replaced with snippet graph

Static Analysis using Graal

Graal as a Static Analysis Framework

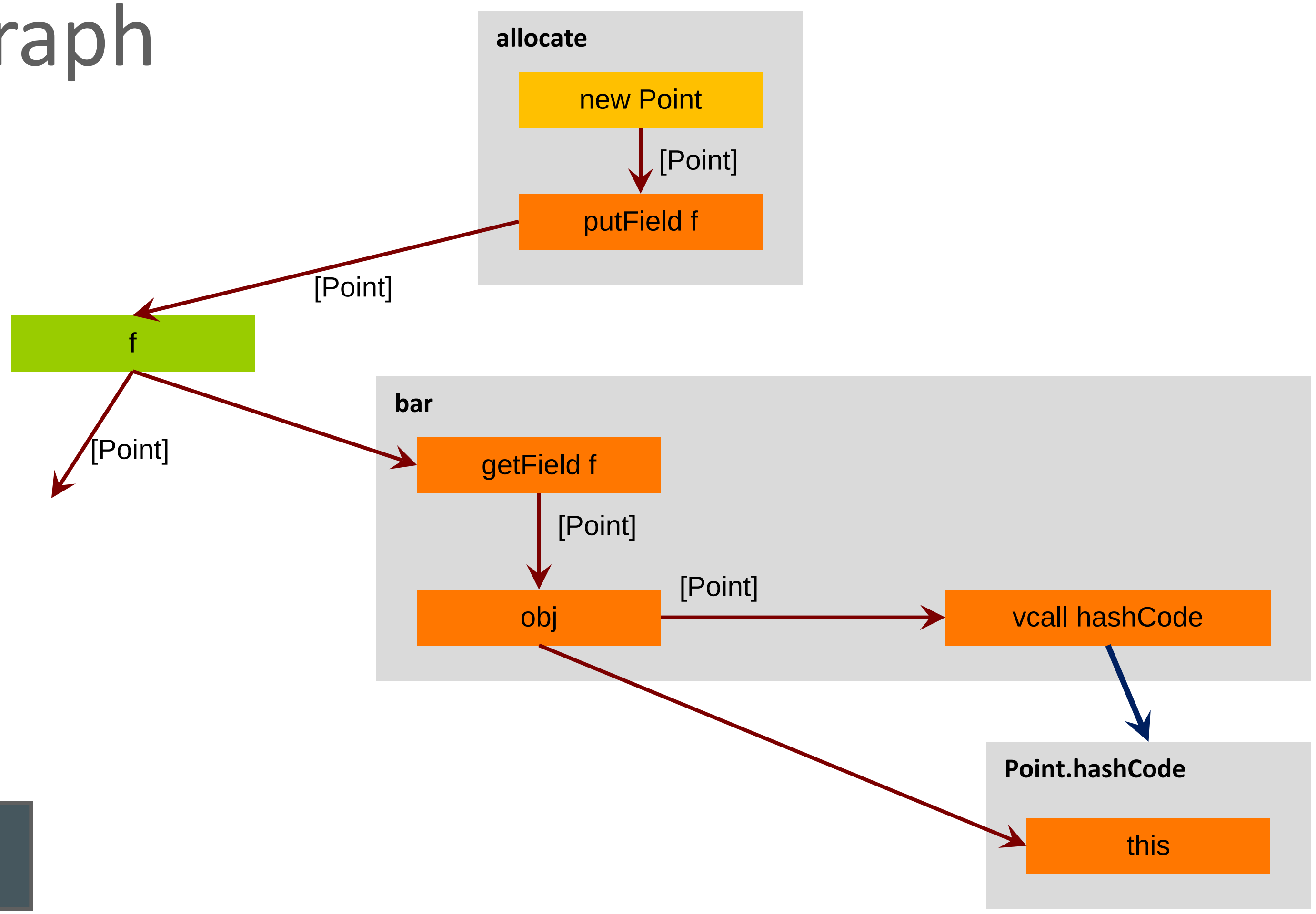
- Graal and the hosting Java VM provide
  - Class loading (parse the class file)
  - Access the bytecodes of a method
  - Access to the Java type hierarchy, type checks
  - Build a high-level IR graph in SSA form
  - Linking / method resolution of method calls
- Static analysis and compilation use same intermediate representation
  - Simplifies applying the static analysis results for optimizations

Example: A Simple Static Analysis

- Implemented just for this tutorial, not complete enough for production use
- Goals
  - 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
- Context insensitive
  - One set of types for each field
  - One set of types for each method parameter / method return

Example Type Flow Graph

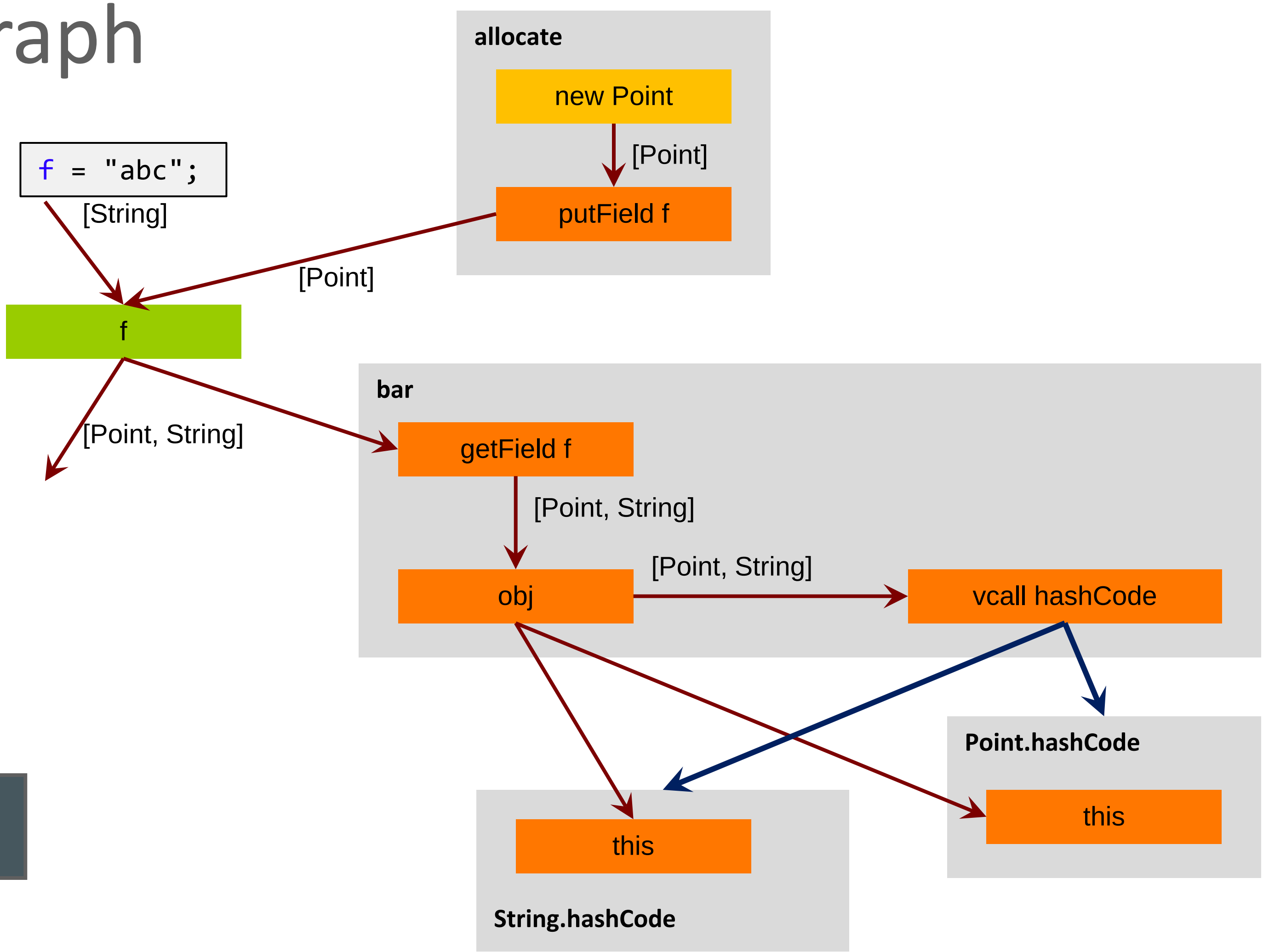
```
Object f;

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

```
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

Building the Graal Graph

Code from `MethodState.process()`

```
StructuredGraph graph = new StructuredGraph.Builder(getInitialOptions()).method(method).build();

try (Scope scope = Debug.scope("graph building", graph)) {

  Plugins plugins = new Plugins(new InvocationPlugins());
  GraphBuilderConfiguration config= GraphBuilderConfiguration.getDefault(plugins).withEagerResolving(true);


  config = config.withBytecodeExceptionMode(OmitAll);


  OptimisticOptimizations optimisticOpts = NONE;


  GraphBuilderPhase.Instance graphBuilder = new GraphBuilderPhase.Instance(metaAccess, stampProvider, null, null,
                                          config, optimisticOpts, null);
  graphBuilder.apply(graph);

} catch (Throwable ex) {
  Debug.handle(ex);
}

TypeFlowBuilder typeFlowBuilder = new TypeFlowBuilder(graph);
typeFlowBuilder.apply();
```

Support for graph dumping to IGV

We want all types to be resolved, i.e., classes loaded

For simplicity we ignore exception handlers

Disable speculation and optimistic optimizations

Parse bytecodes

Convert Graal graph to our type flow graph

ORACLE

Building the Type Flow Graph

```
class TypeFlowBuilder extends StatelessPostOrderNodeIterator {

  private final NodeMap<TypeFlow> typeFlows;

  public void apply() {
    for (Node n : graph.getNodes()) {
      if (n instanceof ParameterNode) {
        ParameterNode node = (ParameterNode) n;
        registerFlow(node, methodState.formalParameters[(node.index())]);
      }
    }
    super.apply();
  }

  protected void node(FixedNode n) {
    if (n instanceof NewInstanceNode) {
      NewInstanceNode node = (NewInstanceNode) n;
      TypeFlow flow = new TypeFlow();
      flow.addTypes(Collections.singleton(type));
      registerFlow(node, flow);
      flow.addUse(results.getAllInstantiatedTypes());

    } else if (n instanceof LoadFieldNode) {
      LoadFieldNode node = (LoadFieldNode) n;
      registerFlow(node, results.lookupField(node.field()));
```

Graal class for iterating fixed nodes in reverse postorder

Graal class to store additional temporary data for nodes

Iterate all graph nodes, not ordered

Register the flow for a node in the `typeFlows` map

Called for all fixed graph nodes in reverse postorder

Type flow for an allocation: just the allocated type

Type flow for a field load: the types assigned to the field

Linking Method Invocations

Code from `InvokeTypeFlow.process()`

```
if (callTarget.invokeKind().isDirect()) {
  /* Static and special calls: link the statically known callee method. */
  linkCallee(callTarget.targetMethod());

} else {
  /* Virtual and interface call: Iterate all receiver types. */
  for (ResolvedJavaType type : getTypes()) {
    /*
     * Resolve the method call for one exact receiver type. The method linking
     * semantics of Java are complicated, but fortunatley we can use the linker of
     * the hosting Java VM. The Graal API exposes this functionality.
     */
    ResolvedJavaMethod method = type.resolveConcreteMethod(callTarget.targetMethod(),
                                        callTarget.invoke().getContextType());
    linkCallee(method);
  }
}
```

New receiver types found by the static analysis are added to this set – this method is then executed again

Custom Compilations with Graal

Custom Compilations with Graal

- Applications can call Graal like a library to perform custom compilations
  - With application-specific optimization phases
  - With application-specific compiler intrinsics
  - Reusing all standard Graal optimization phases
  - Reusing lowerings provided by the hosting VM

- Example use cases
  - Perform partial evaluation
    - Staged execution
    - Specialize for a fixed number of loop iterations
  - Custom method inlining
  - Use special hardware instructions

ORACLE

Example: Custom Compilation

```
public class InvokeGraal {
  protected final Backend backend;
  protected final Providers providers;
  protected final MetaAccessProvider metaAccess;
  protected final CodeCacheProvider codeCache;
  protected final TargetDescription target;

  public InvokeGraal() {
    /* Ask the hosting Java VM for the entry point object to the Graal API. */
    RuntimeProvider runtimeProvider = Graal.getRequiredCapability(RuntimeProvider.class);
    /* The default backend (architecture, VM configuration) that the hosting VM is running on. */
    backend = runtimeProvider.getHostBackend();
    /* Access to all of the Graal API providers, as implemented by the hosting VM. */
    providers = backend.getProviders();
    /* Some frequently used providers and configuration objects. */
    metaAccess = providers.getMetaAccess();
    codeCache = providers.getCodeCache();
    target = codeCache.getTarget();
  }

  protected InstalledCode compileAndInstallMethod(ResolvedJavaMethod method) ...
```

See next slide

Custom compilation of String.hashCode()

```
$ mx igv &
$ mx unittest –Dgraal.Dump= -Dgraal.MethodFilter=String.hashCode GraalTutorial#testStringHashCode
```

Example: Custom Compilation

You can manually construct Graal IR and compile it

Add your custom optimization phases to the suites

```
ResolvedJavaMethod method = ...
StructuredGraph graph = new StructuredGraph.Builder(getInitialOptions(), AllowAssumptions.YES)
                            .method(method).compilationId(compilationId).build();
/* The phases used to build the graph. Usually this is just the GraphBuilderPhase. If
 * the graph already contains nodes, it is ignored. */
PhaseSuite<HighTierContext> graphBuilderSuite = backend.getSuites().getDefaultGraphBuilderSuite();
/* The optimization phases that are applied to the graph. This is the main configuration
 * point for Graal. Add or remove phases to customize your compilation. */
Suites suites = backend.getSuites().getDefaultSuites(options);

/* The low-level phases that are applied to the low-level representation. */
LIRSuites lirSuites = backend.getSuites().getDefaultLIRSuites(options);
/* We want Graal to perform all speculative optimistic optimizations, using the
 * profiling information that comes with the method (collected by the interpreter) for speculation. */
OptimisticOptimizations optimisticOpts = OptimisticOptimizations.ALL;
ProfilingInfo profilingInfo = graph.getProfilingInfo(method);
/* The default class and configuration for compilation results. */
CompilationResult compilationResult = new CompilationResult();
CompilationResultBuilderFactory factory = CompilationResultBuilderFactory.Default;
/* Invoke the whole Graal compilation pipeline. */
GraalCompiler.compileGraph(graph, method, providers, backend, graphBuilderSuite, optimisticOpts, profilingInfo, suites,
lirSuites, compilationResult, factory);
/* Install the compilation result into the VM, i.e., copy the byte[] array that contains
 * the machine code into an actual executable memory location. */
InstalledCode installedCode = return backend.addInstalledCode(method, asCompilationRequest(compilationId), compilationResult);
/* Invoke the installed code with your arguments. */
installedCode.executeVarargs([...]);
```

Part 2: GraalVM

Truffle

A Language Implementation Framework that uses Graal for Custom Compilation

“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

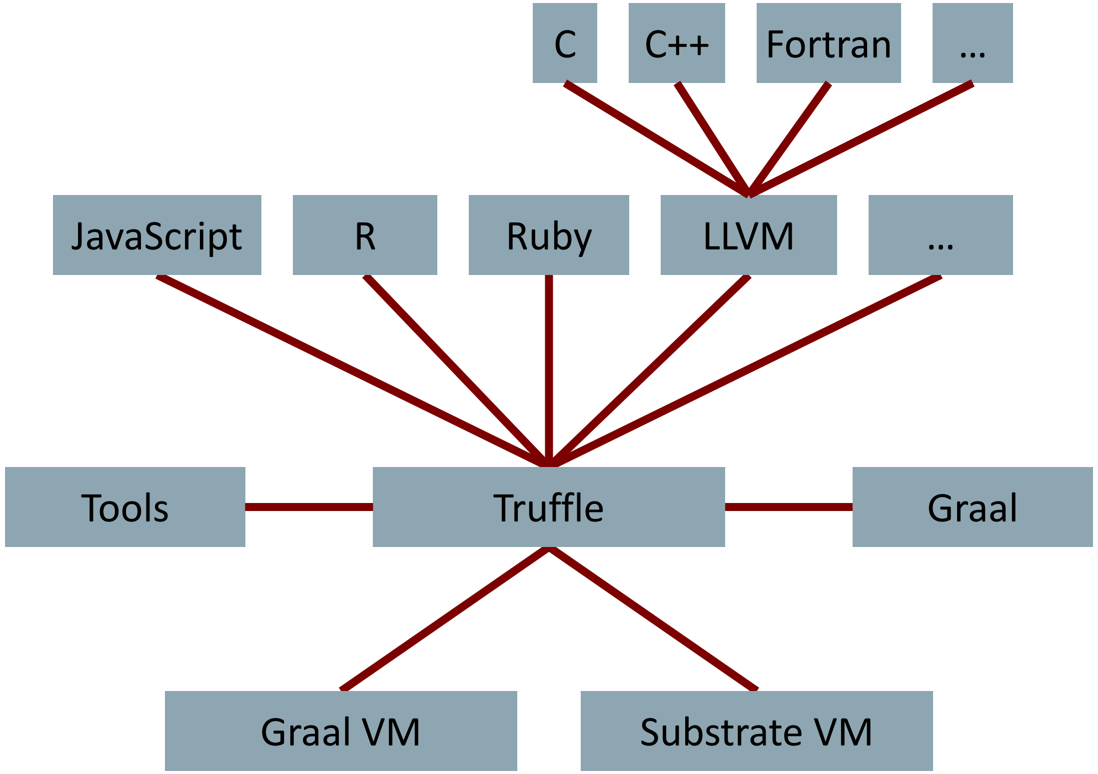
People start using it

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

Overall System Structure

Interpreter for every language

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



Language agnostic dynamic compiler

Integrate with Java applications

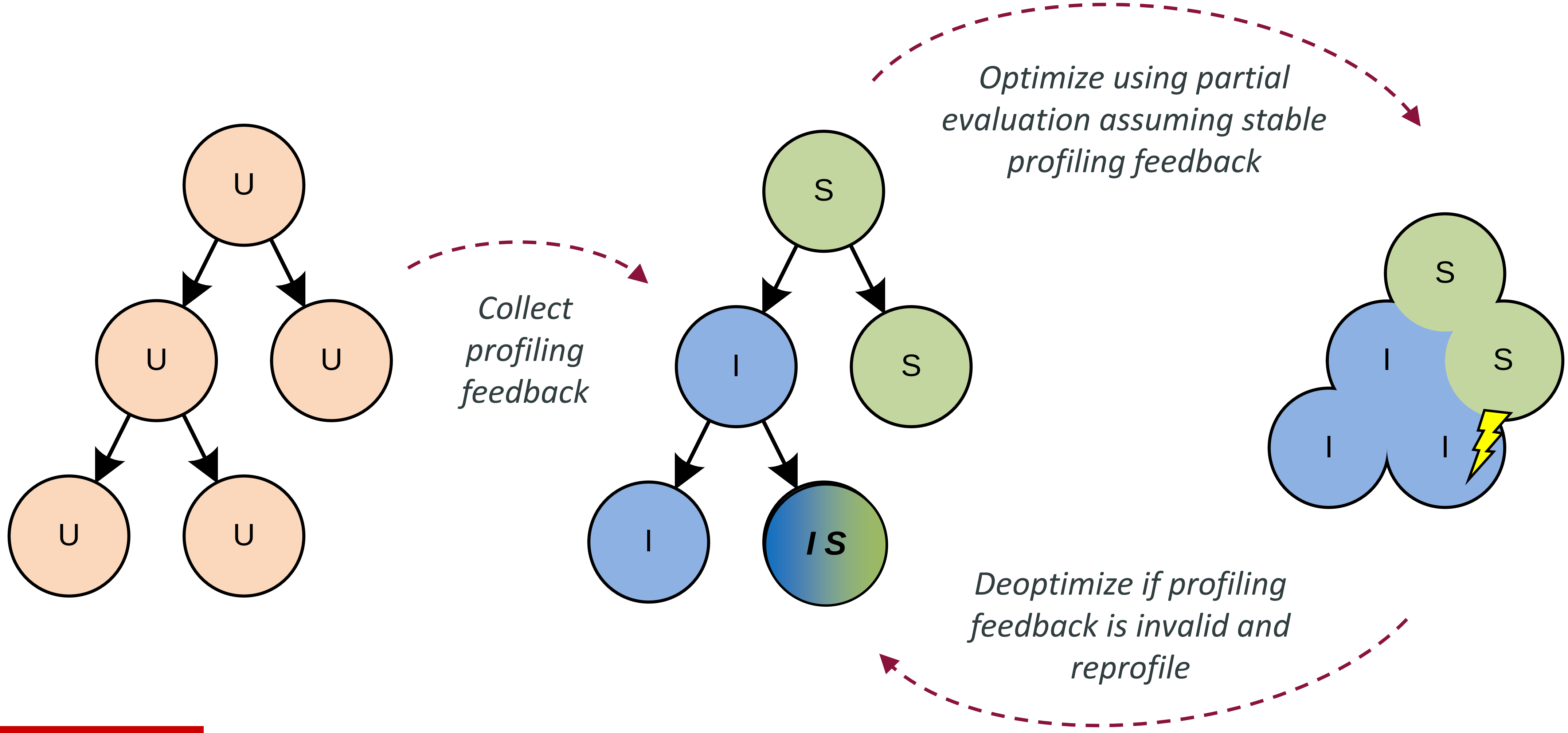
Low-footprint VM, also suitable for embedding

Lets talk about JavaScript...

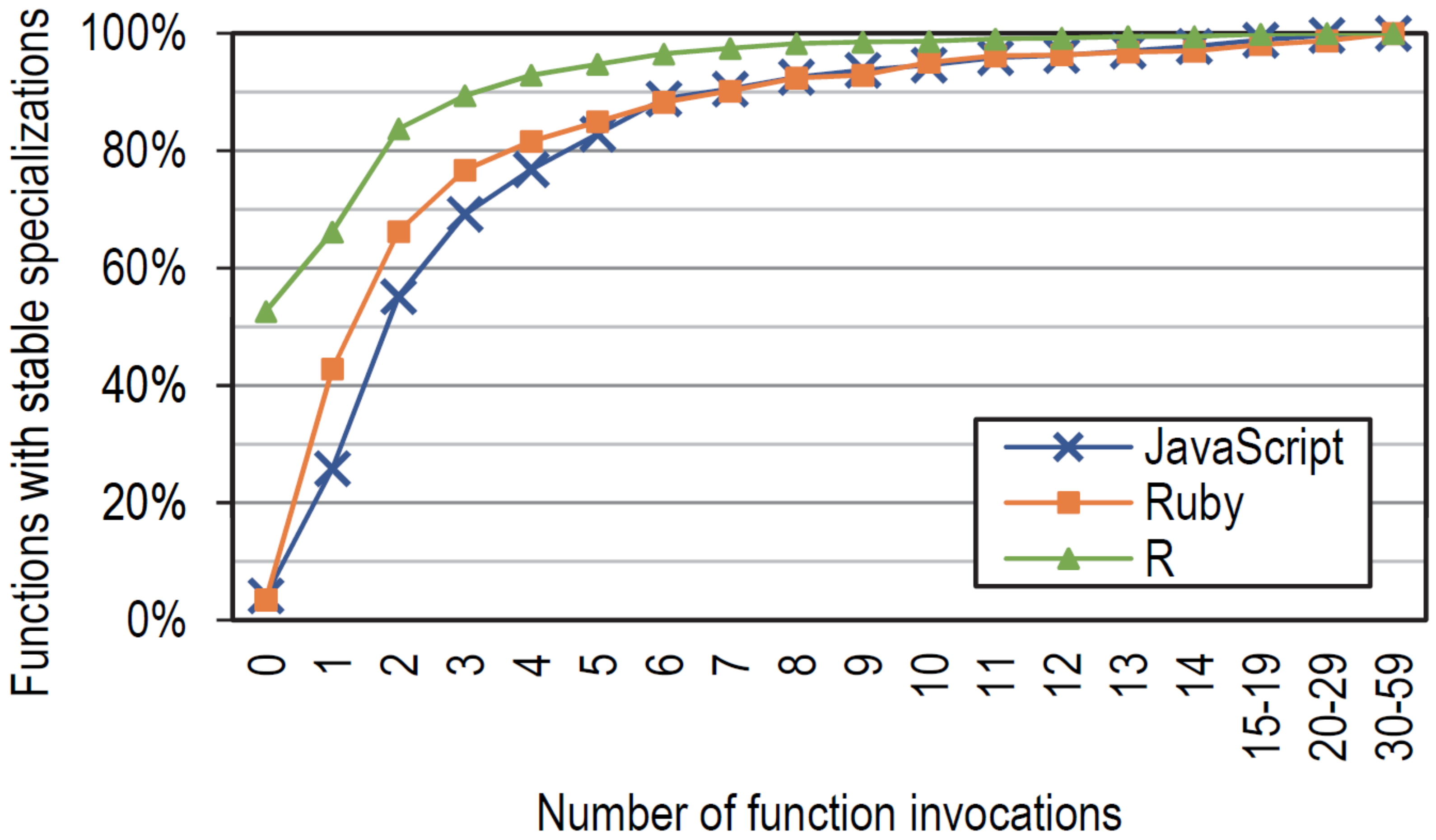
```
function negate(a) {
   return -a
}
```

```
> negate(42)
-42

> negate(-"42")
"-42"

> negate({})
NaN

> negate([])
-0
```

The Truffle Idea



Stability



Functions with stable specializations
100%
80%
60%
40%
20%
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

Partial Evaluation and Deoptimization with Truffle

Introduction to Partial Evaluation

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

class AddNode extends Node {
    final Node left, right;

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

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

```java
class Arg extends Node {
    final int index;
    Arg(int i) {this.index = i;}

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

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

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

Introduction to 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);
}
```

```
partiallyEvaluate(interpret, sample)
```

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

ORACLE

Introduction to 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 interpretSample(int[] args) {
    return sample.execute(args);
}
```

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

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

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

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

ORACLE

Truffle Core Features

- Initiate Partial Evaluation

  + Transition from Java to Partial evaluated code

- Speculation with Internal Invalidation (guards)

- Speculation with External Invalidation (assumptions)

- Explicit Boundaries for Partial Evaluation
- ...

ORACLE

Initiate Partial Evaluation

```
class Function extends RootNode {
    @Child Node child;

    Object execute(VirtualFrame frame) {
        return child.execute(frame)
    }
}

public static void main(String[] args) {
    CallTarget target = Truffle.getRuntime().createCallTarget(new Function());

    for (int i = 0; i < 10000; i++) {
        // after a few calls partially evaluates on a background thread
        // installs partially evaluated code when ready
        target.call();
    }
}
```

Speculation with Internal Invalidation

```
class NegateNode extends Node {

    @CompilationFinal boolean objectSeen = false;

    Object execute(Object v) {
        if (v instanceof Double) {
            return -((double) v);
        } else {
            if (!objectSeen) {
                transferToInterpreter();
                objectSeen = true;
            }
            // slow-case handling of all
            // other types
            return objectNegate(v);
        }
    }
}
```

Compiler sees: `objectSeen = false`

```
if (v instanceof Double) {
    return -((double) v);
} else {
    deoptimize;
}
```

Compiler sees: `objectSeen = true`

```
if (v instanceof Double) {
    return -((double) v);
} else {
    return objectNegate(v);
}
```

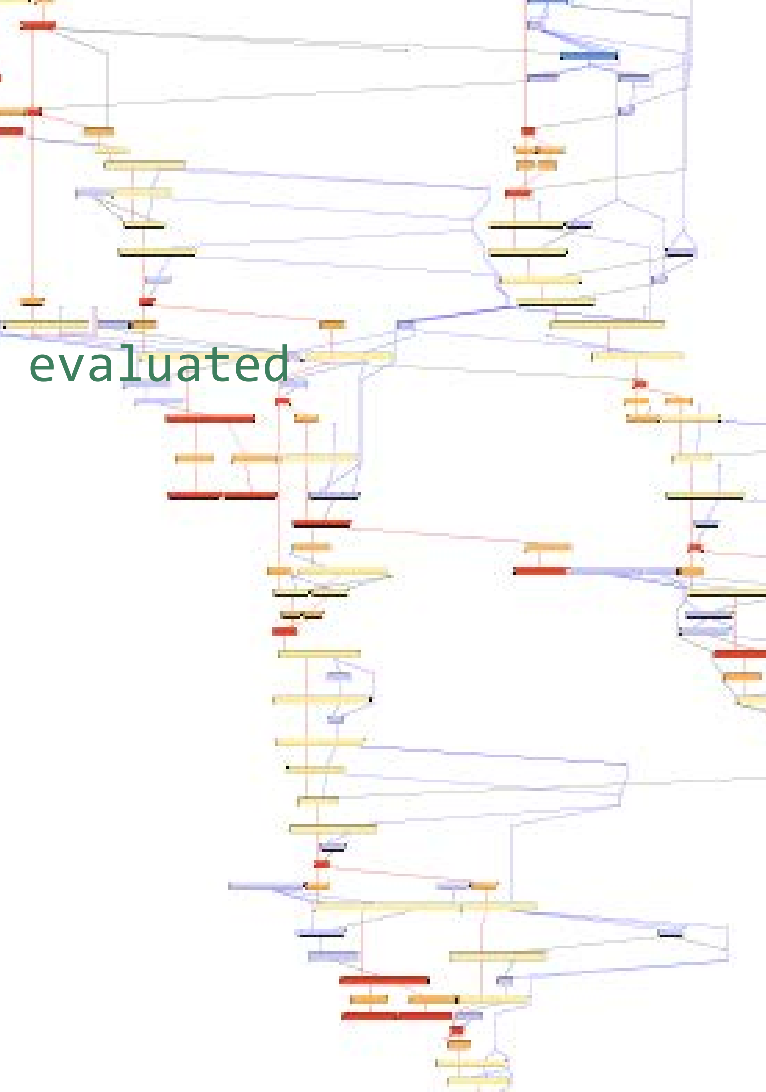
Speculation with External Invalidation

```
@CompilationFinal static Assumption addNotDefined = new Assumption();

class AddNode extends Node {

    int execute(int left, int right) {
        if (addNotDefined.isValid()) {
            return left + right;
        }
        ... // complicated code to call user-defined add
    }
}

static void defineFunction(String name, Function f) {
    if (name.equals("+")) {
        addNotDefined.invalidate();
        ... // register user-defined add
    }
}
```

Explicit Boundaries for Partial Evaluation

```
Object parseJSON(Object value) {
    String s = objectToString(value);
    return parseJSONString(s);
}

@TruffleBoundary
Object parseJSONString(String value) {
    // complex JSON parsing code
}
```

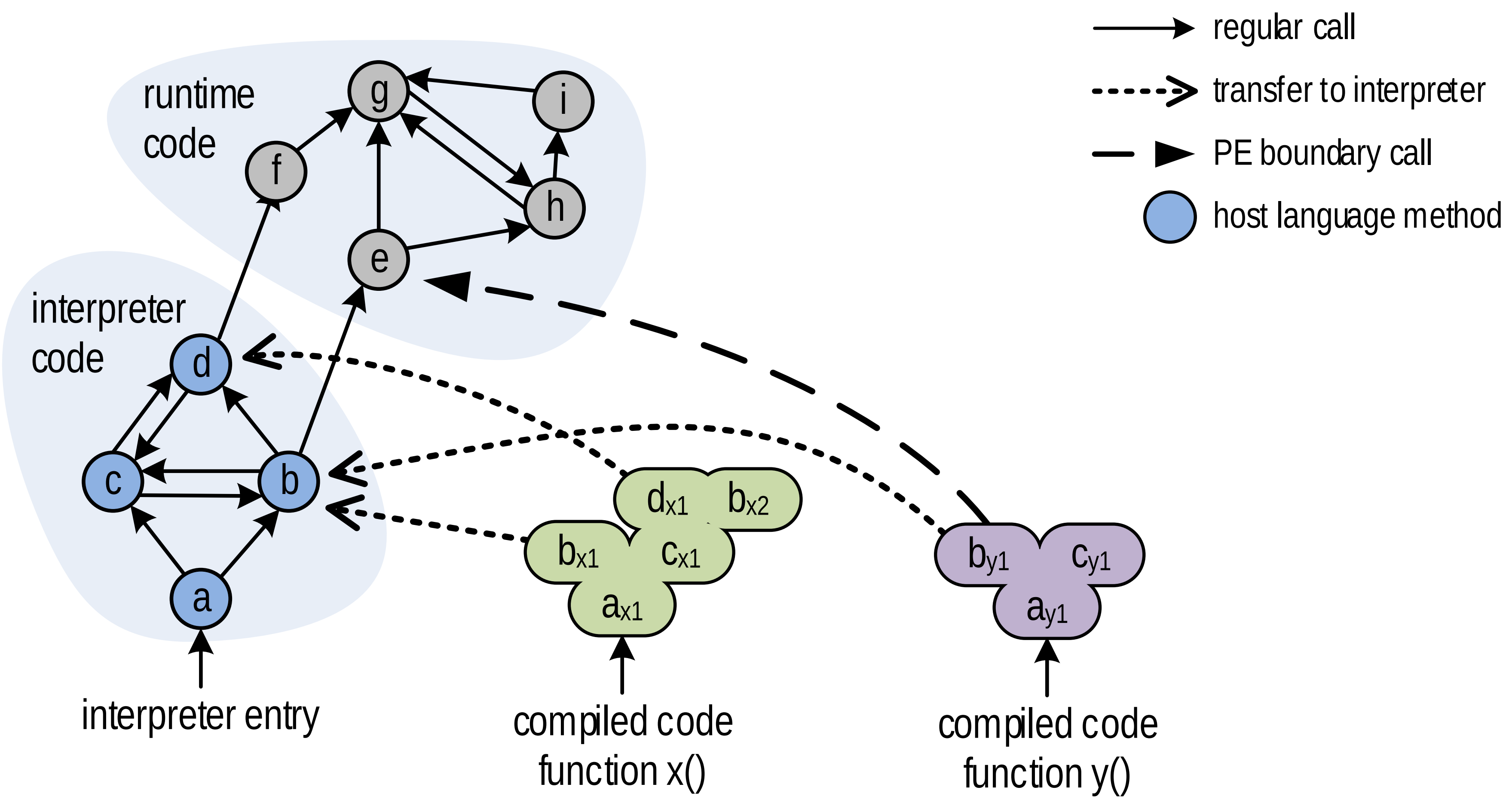
Explicit Boundaries for Partial Evaluation




```java
// no boundary, but partially evaluated
void println() {
    System.out.println()
}
```

=> Partially evaluated version can be significantly slower than Java if not handled with care!

Interpreter and Runtime Interactions



Example: Polymorphic Function Inline Caches

Monomorphic

```
function foo() {};

foo(); // foo
```

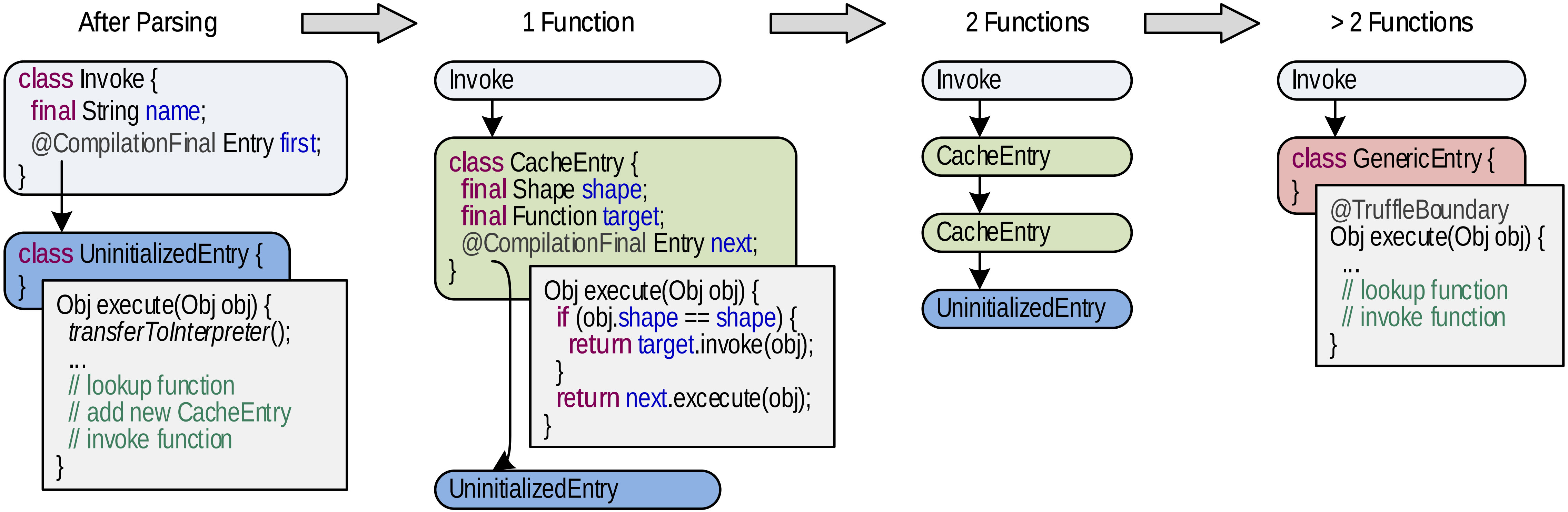
Polymorphic

```
function bar() {};
function baz() {};

functions = [foo, bar, baz];
for (f of functions) {
    f(); // either foo, bar or baz
}
```

Megamorphic

```
functions = [/*10 functions*/]
for (f of functions) {
    f(); // many
}
```

Example: Polymorphic Function Inline Caches



Example: Polymorphic Function Inline Caches

```
class Invoke extends Node {

    final String name;

    @Specialization(guards = "obj.shape == shape", limit = "2")
    Object doCached(Obj obj,
            @Cached("shape") Shape shape,
            @Cached("obj.lookup(name)") Function target) {
        return target.invoke(obj);
    }

    @TruffleBoundary
    @Specialization(replaces="doCached")
    Object doGeneric(Obj obj) {
        return obj.lookup(name).invoke(obj);
    }
}
```

ORACLE

Custom Graal Compilation in Truffle

- Custom method inlining
  - Unconditionally inline all Truffle node execution methods
  - See class `PartialEvaluator`, `TruffleCacheImpl`
- Custom escape analysis
  - Enforce that Truffle frames are escape analyzed
  - See class `NewFrameNode`
- Custom compiler intrinsics
  - See class `CompilerDirectivesSubstitutions`, `CompilerAssertsSubstitutions`
- Custom nodes for arithmetic operations with overflow check
  - See class `IntegerAddExactNode`, `IntegerSubExactNode`, `IntegerMulExactNode`
- Custom invalidation of compiled code when a Truffle Assumption is invalidated
  - See class `OptimizedAssumption`, `OptimizedAssumptionSubstitutions`

Example: Visualize Truffle Compilation

SL source code:

```
function loop(n) {
  i = 0;
  sum = 0;
  while (i <= n) {
    sum = sum + i;
    i = i + 1;
  }
  return sum;
}
```

Machine code for loop:

```
     mov  r14, 0
     mov  r13, 0
     jmp  L2
L1:  safepoint
     mov  rax, r13
     add  rax, r14
     jo   L3
     inc  r13
     mov  r14, rax
L2:  cmp  r13, rbp
     jle  L1
     ...
L3:  call transferToInterpreter
```

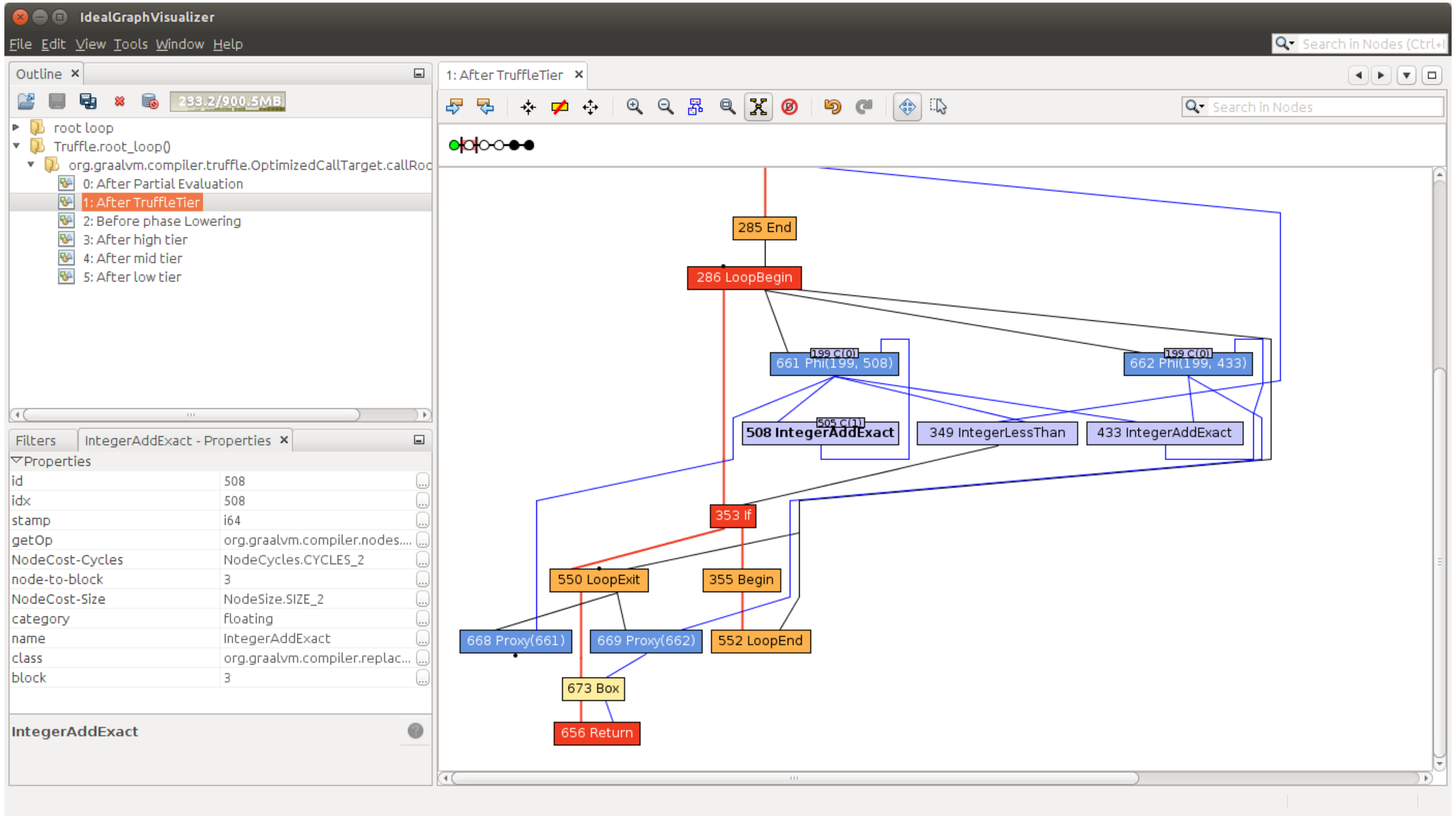
Run this example:

```
$ mx igv &
$ mx sl -Dgraal.Dump= -Dgraal.TruffleBackgroundCompilation=false ../truffle/src/com.oracle.truffle.sl.test/src/tests/SumPrint.sl
```

TruffleBackgroundCompilation=false forces compilation in the main thread

ORACLE

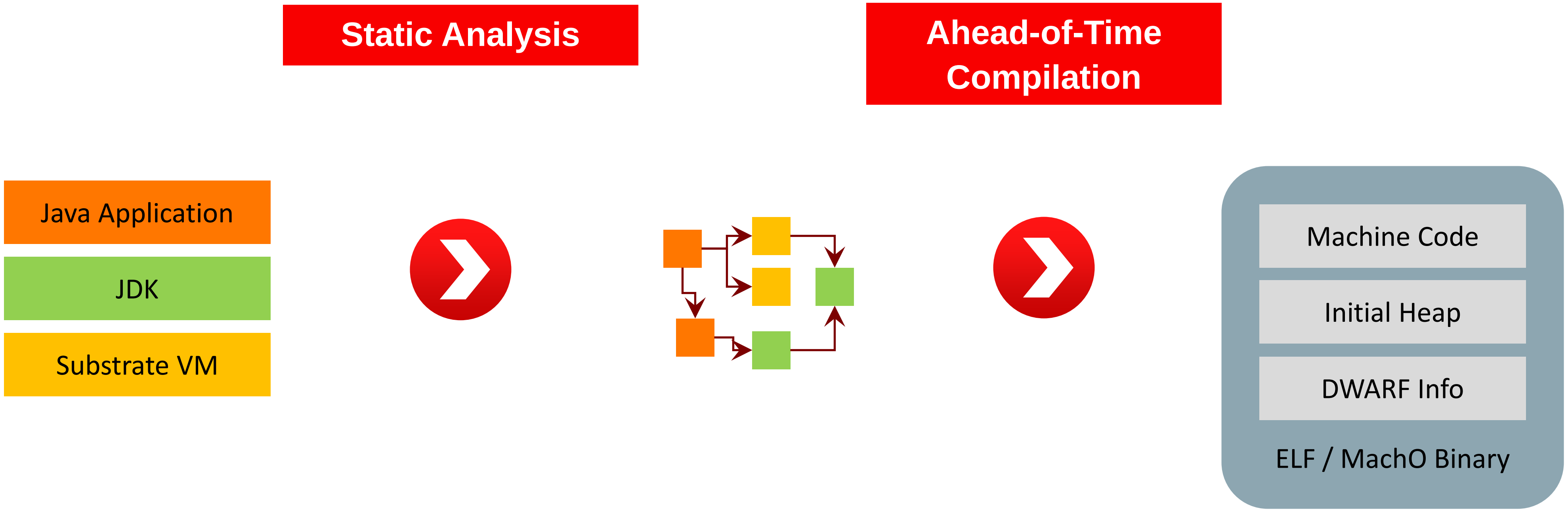
Graal Graph of Simple Language Method



Polyglot Native

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, JavaScript

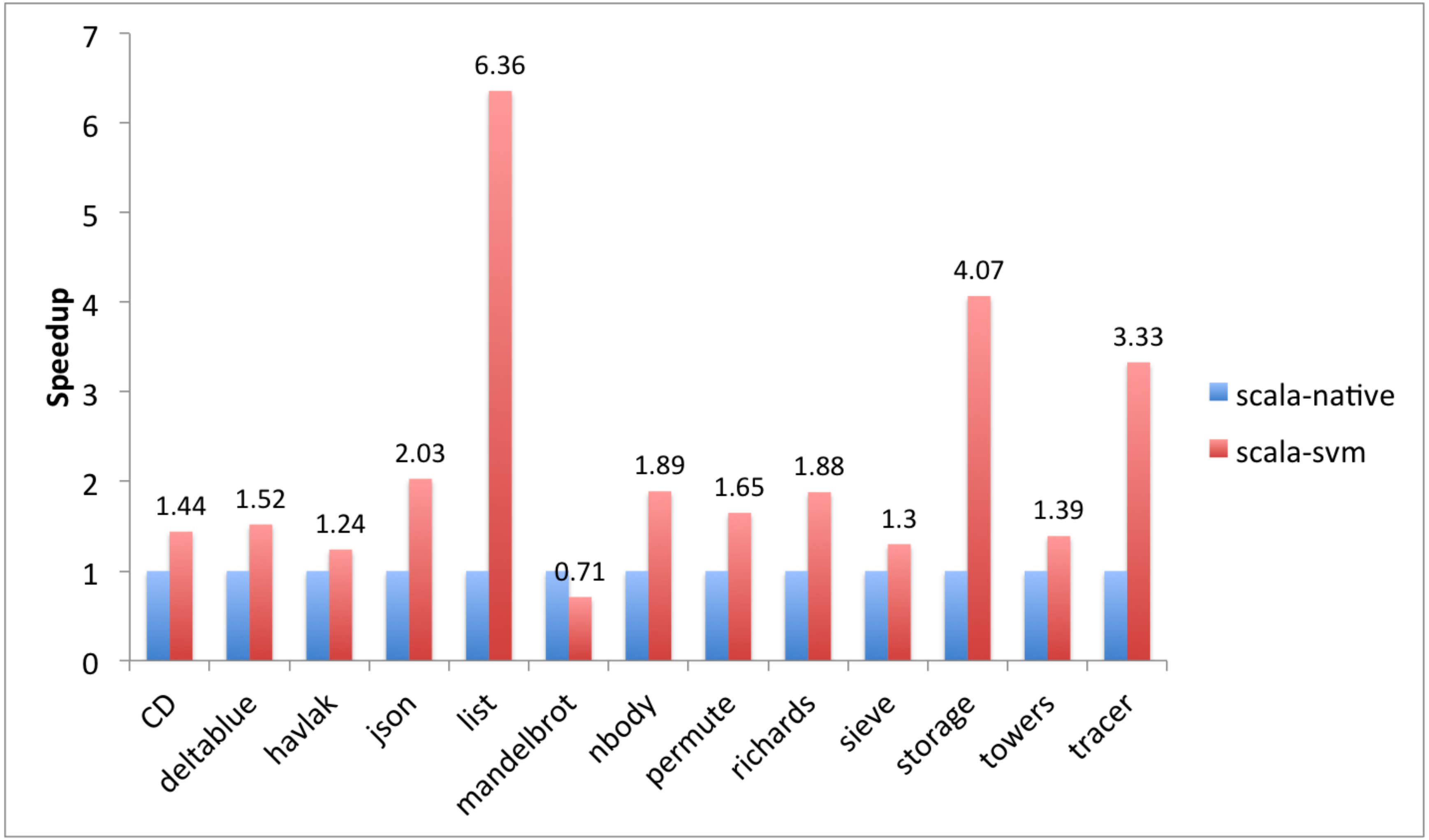
Language	Virtual Machine	Instructions	Time	Memory
C helloworld		100,000	< 10 ms	450 KByte
GNU helloworld 2.10		300,000	< 10 ms	800 KByte
Java	Java HotSpot VM	140,000,000	40 ms	24,000 KByte
Java	Substrate VM	220,000	< 10 ms	850 KByte
JavaScript	V8	10,000,000	<= 10 ms	18,000 KByte
JavaScript	Spidermonkey	77,000,000	20 – 30 ms	10,000 KByte
JavaScript	Nashorn on Java HotSpot VM	N/A	450 ms	56,000 KByte
JavaScript	Truffle on Java HotSpot VM	N/A	650 ms	120,000 KByte
JavaScript	**Truffle on Substrate VM**	**520,000**	**< 10 ms**	**4,200 KByte**

Substrate VM has a fully initialized JavaScript execution context in the boot image heap

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

ORACLE

Polyglot Native vs. Scala Native



ORACLE

SystemJava

- Call C code from Java
  - Need a convenient way to access preexisting C functions and structures
- Existing Java code integration
  - Leverage preexisting Java libraries
  - Example: JDK class library
- Call Java from C code
  - Entry points into JVM code

ORACLE

Word Type for Low-Level Memory Access

- Requirements
  - Support raw memory access and pointer arithmetic
  - Not an extension of the Java programming language
  - Pointer type modeled as a class to prevent mixing with, e.g., `long`
- 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;
}
```

ORACLE

Java Annotations for C Interoperability

Java	C
`@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

Managed Objects in Native Code

- Managed objects are different than native objects
  - In layout, as every object has a header
  - Memory location, they can, at any time, be moved by the garbage collector

- To avoid these issues, when passing objects to native code
  - Use handles when native code only holds a reference
  - Pin objects and ignore their header when native code reads the object

Summary

Graal VM Architecture

Sulong (LLVM)

Truffle Framework

Graal Compiler

JVM Compiler Interface (JVMCI) JEP 243

Java HotSpot Runtime

Integrated Cloud

Applications & Platform Services