ORACLE®

Graal

Christian Wimmer

VM Research Group, Oracle Labs

ORACLE

Safe Harbor Statement

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

ORACLE

Tutorial Outline

- 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
- 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
- Custom compilations with Graal: integration of the compiler with an application or library
- Graal as a compiler for dynamic programming languages in the Truffle framework

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

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 and build the source code:

```
$ hg clone http://hg.openjdk.java.net/graal/graal
$ cd graal
$ ./mx.sh build
```

Run the Graal VM:

```
$ ./mx.sh vm -version
```

Generate Eclipse and NetBeans projects:

```
$ ./mx.sh ideinit
```

Run the whitebox unit tests

```
$ ./mx.sh unittest
```

Run a specific unit test in the Java debugger

```
$ ./mx.sh –d unittest GraalTutorial#testStringHashCode
```

`mx` is our script to simplify building and execution

Configuration "graal" for JIT compilations with Graal

Configuration "server" for unittest, static analysis, custom compilations from application

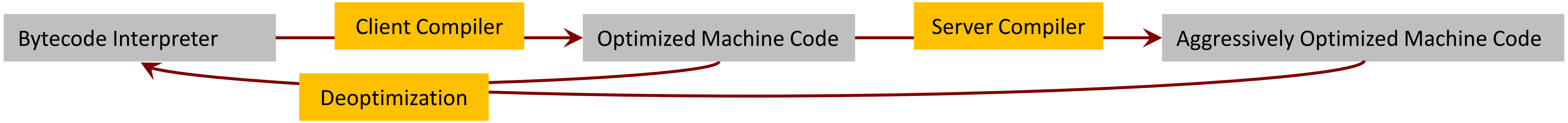
Operating Systems: Windows, Linux, MacOS, Solaris

Architectures: Intel 64-bit, Sparc (experimental)

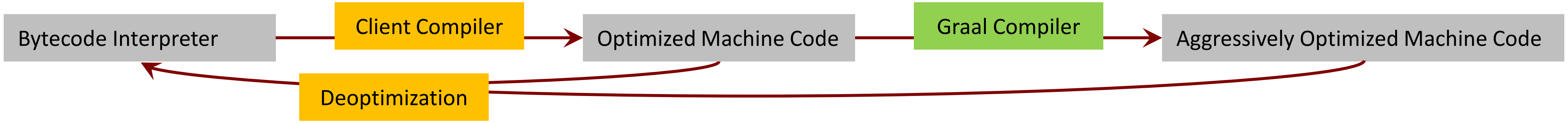
Use the predefined Eclipse launch configuration to connect to the Graal VM

Mixed-Mode Execution

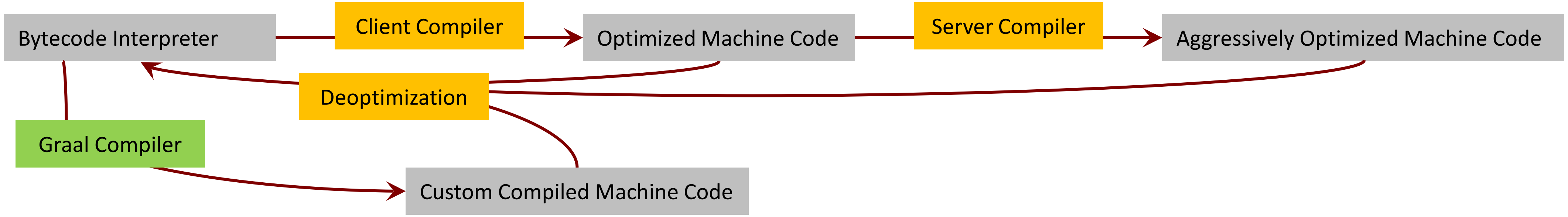
Default configuration of Java HotSpot VM in production:



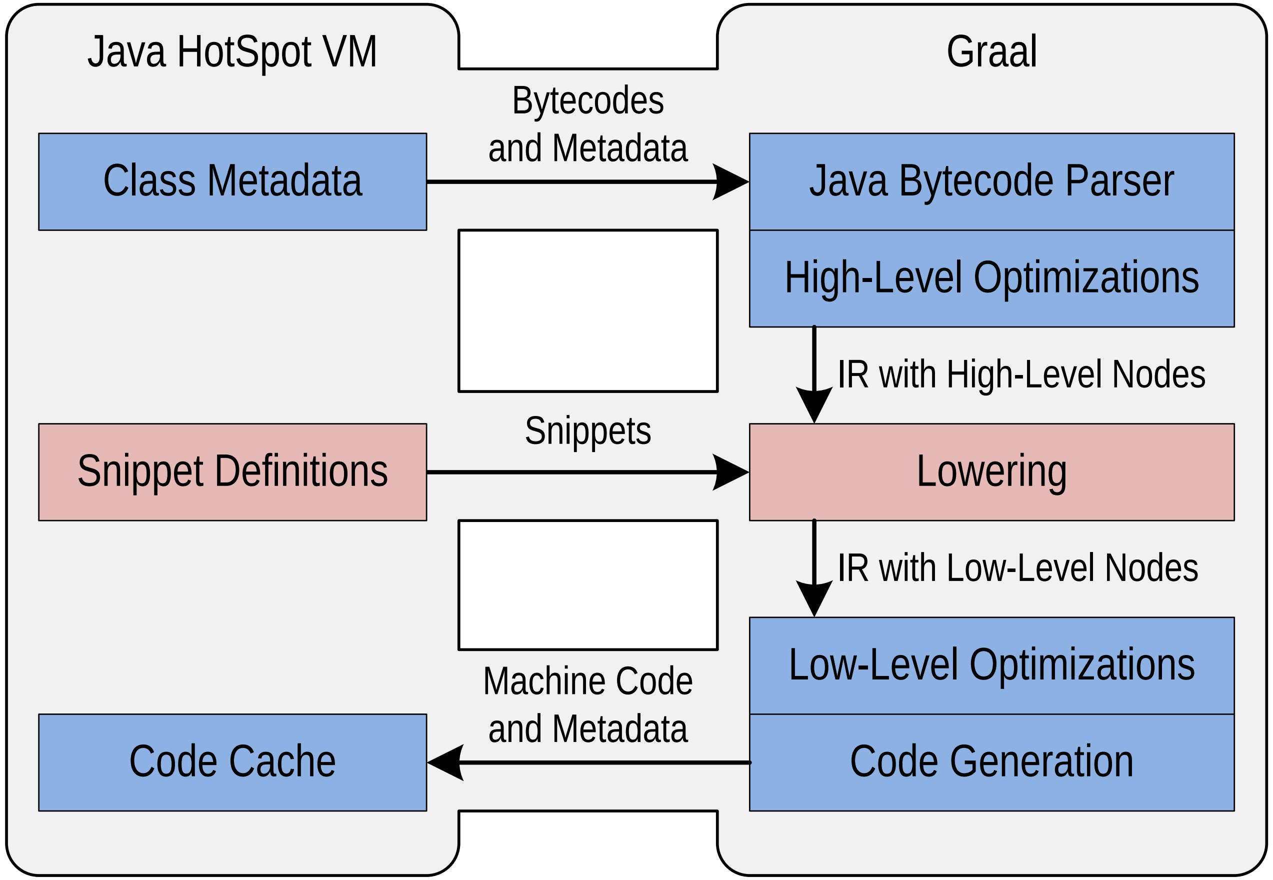
Graal VM in configuration "graal": Graal replaces the server compiler



Graal VM in configuration "server": Graal used only for custom compilations



Compiler-VM Separation



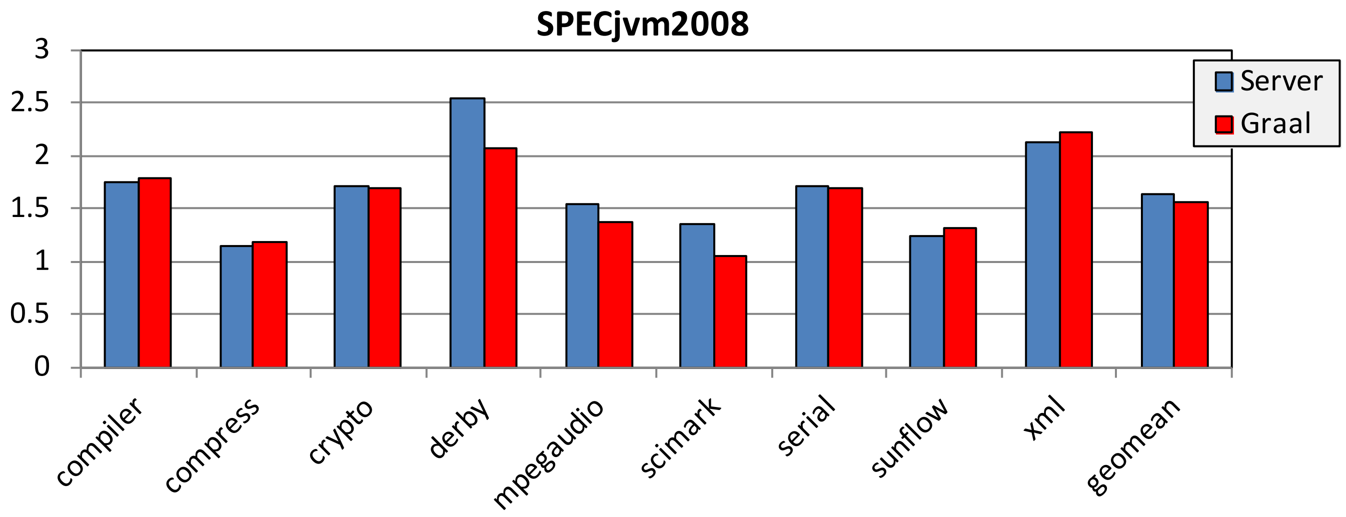
Default Compilation Pipeline

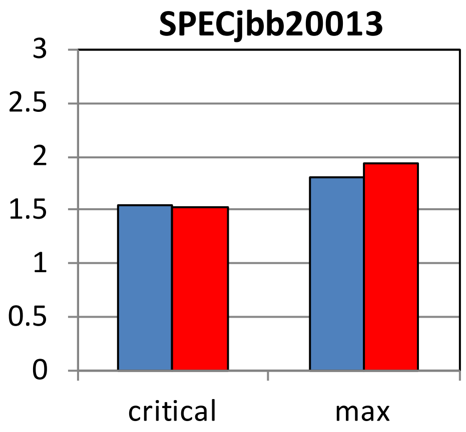
- Java bytecode parser
- Front end: graph based intermediate representation (IR) in static single assignment (SSA) form
  - High Tier
    - Method inlining
    - Partial escape analysis
    - Lowering using snippets
  - Mid Tier
    - Memory optimizations
    - Lowering using snippets
  - Low Tier
-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

Source code reference: `GraalCompiler.compile()`

ORACLE

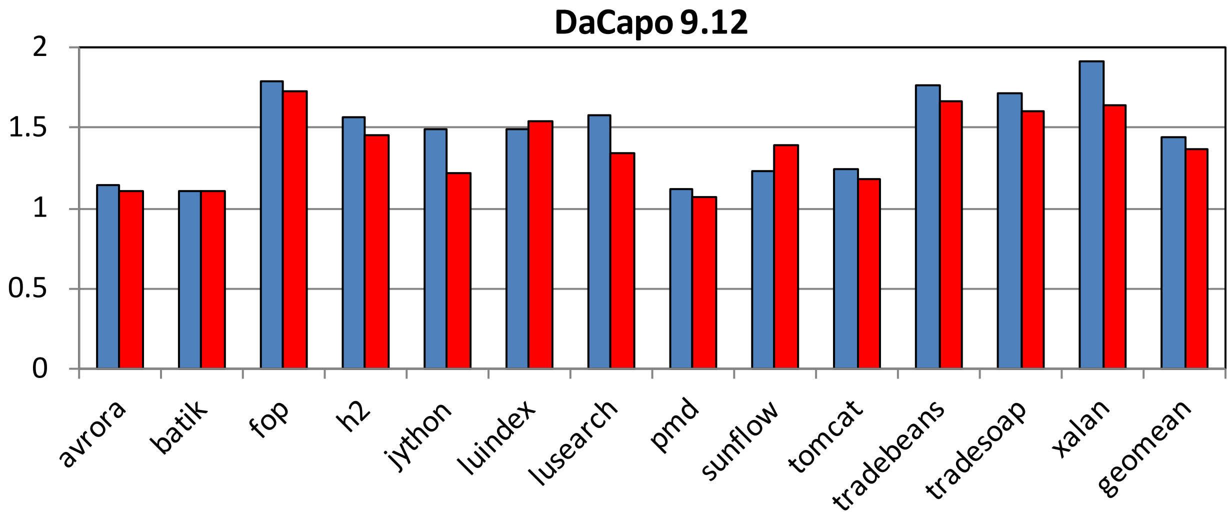
Graal Benchmark Results

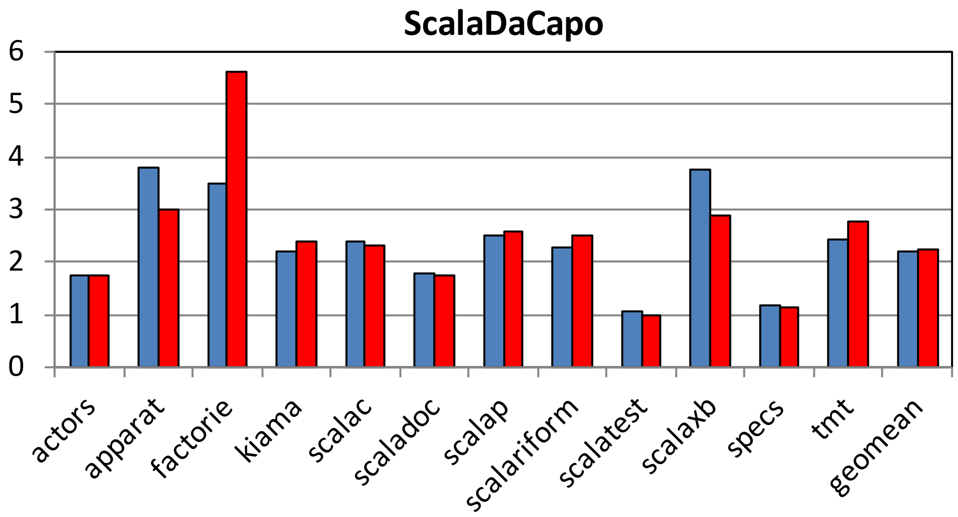




Higher is better, normalized to Client compiler.

Results are not SPEC compliant, but follow the rules for research use.





Acknowledgements

Oracle Labs
Danilo Ansaloni
Stefan Anzinger
Daniele Bonetta
Matthias Brantner
Laurent Daynès
Gilles Duboscq
Michael Haupt
Mick Jordan
Peter Kessler
Hyunjin Lee
David Leibs
Kevin Menard
Tom Rodriguez
Roland Schatz
Chris Seaton
Doug Simon
Lukas Stadler
Michael Van De Vanter

Oracle Labs (continued)
Adam Welc
Till Westmann
Christian Wimmer
Christian Wirth
Paul Wögerer
Mario Wolczko
Andreas Wöß
Thomas Würthinger

Oracle Labs Interns
Shams Imam
Stephen Kell
Gero Leinemann
Julian Lettner
Gregor Richards
Robert Seilbeck
Rifat Shariyar

Oracle Labs Alumni
Erik Eckstein
Christos Kotselidis

JKU Linz
Prof. Hanspeter Mössenböck
Benoit Daloze
Josef Eisl
Matthias Grimmer
Christian Häubl
Josef Haider
Christian Humer
Christian Huber
Manuel Rigger
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
Codrut Stancu
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

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.sh 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

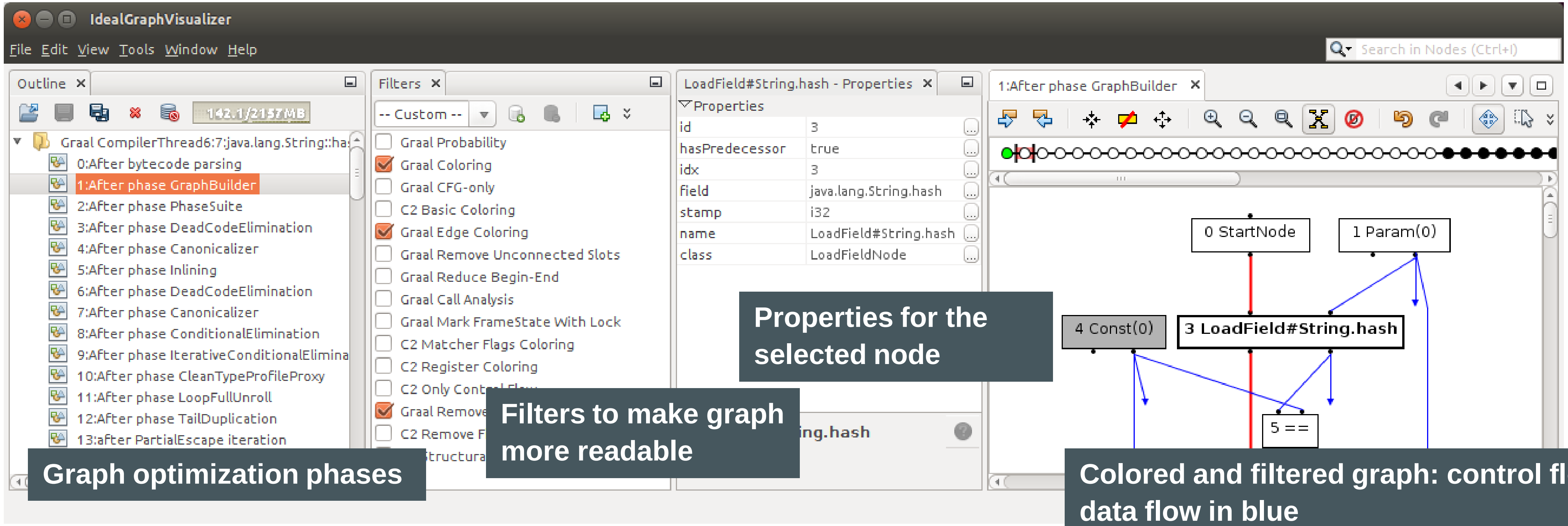
ORACLE

IR Example: Ideal Graph Visualizer

Start the Graal VM with graph dumping enabled

```
$ ./mx.sh igv &
$ ./mx.sh unittest -G:Dump= -G:MethodFilter=String.hashCode GraalTutorial#testStringHashCode
```

Test that just compiles `String.hashCode()`



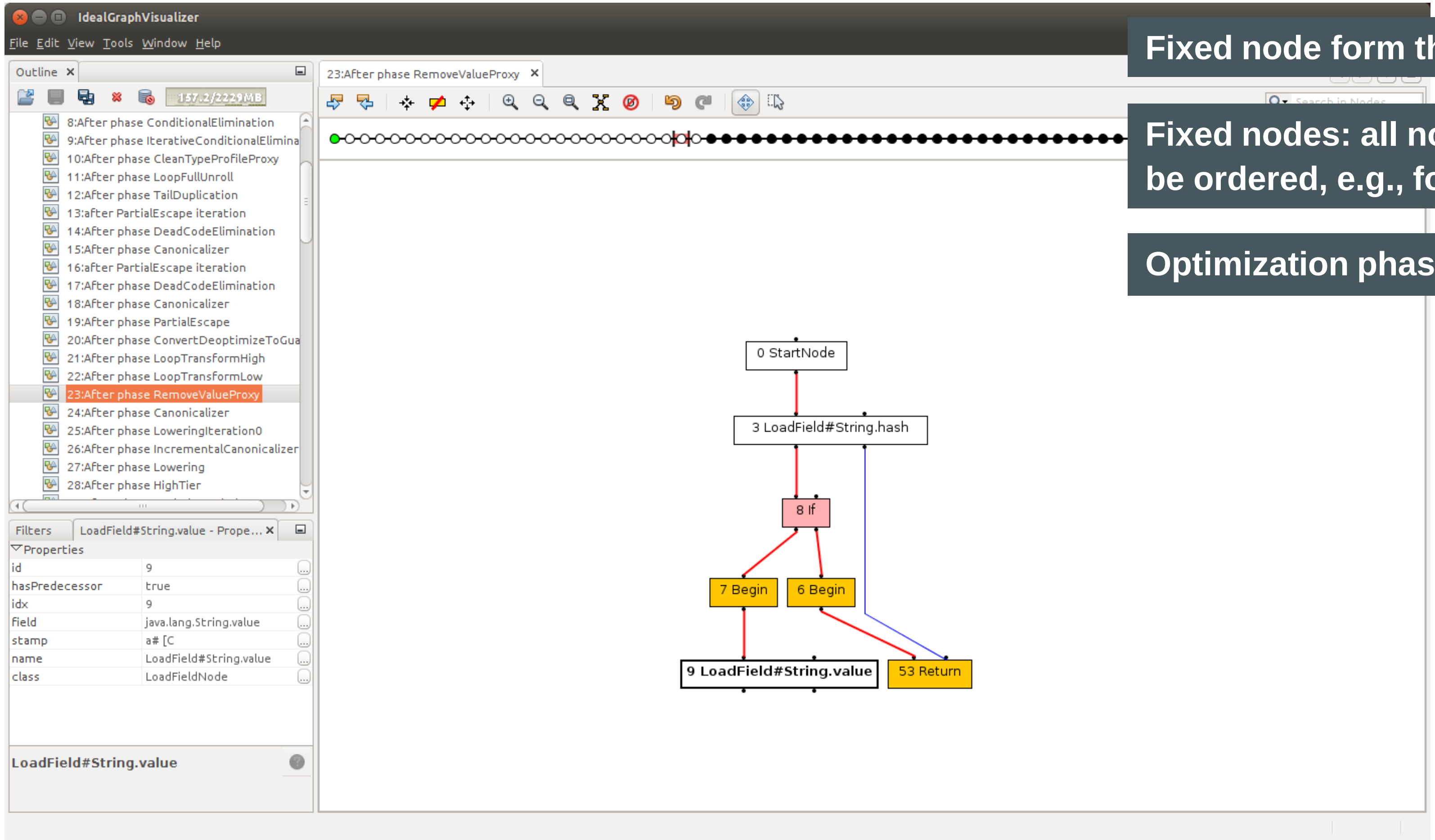
Properties for the selected node

Filters to make graph more readable

Graph optimization phases

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

IR Example: Control Flow

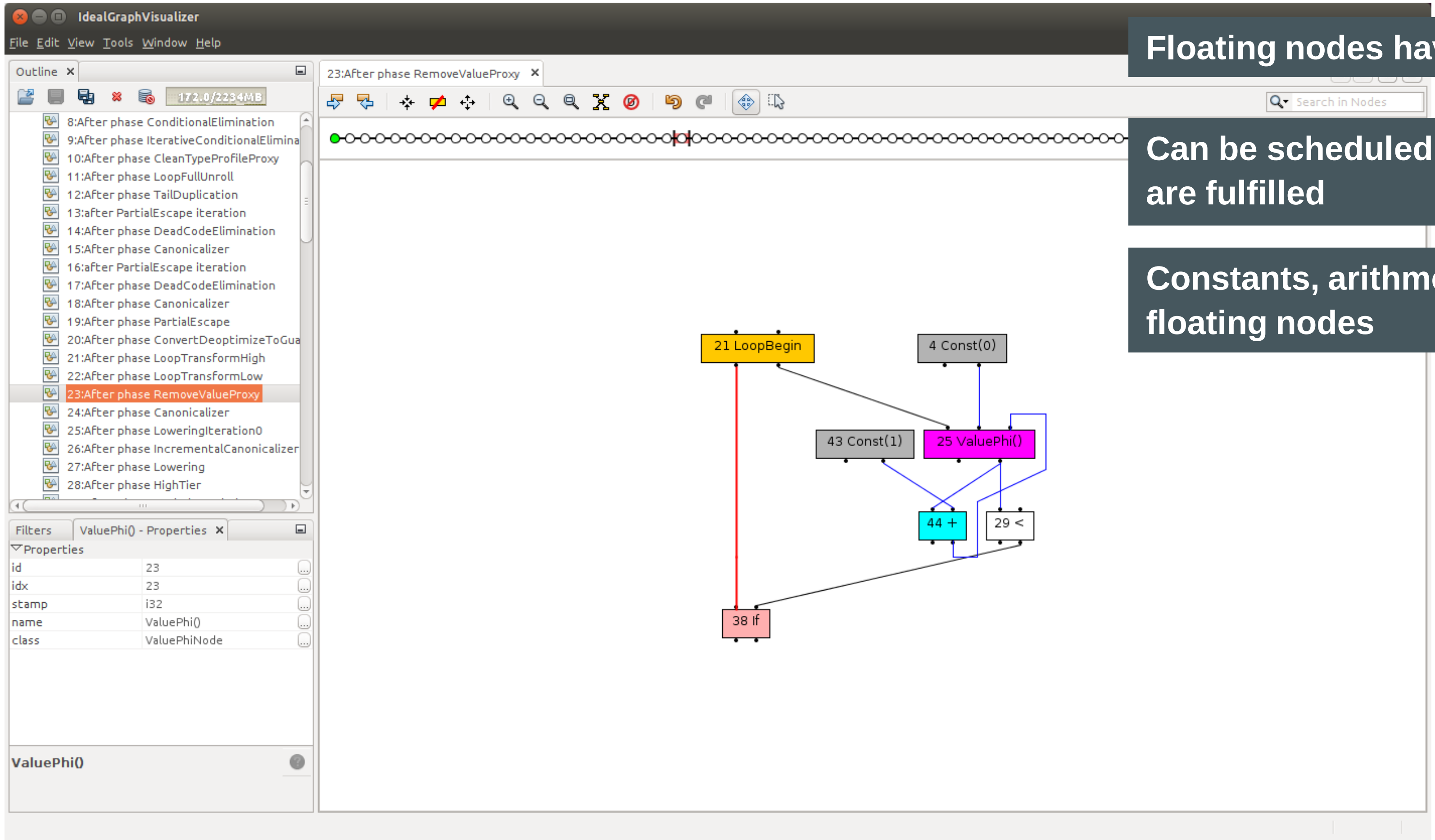


Fixed node form the control flow graph

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

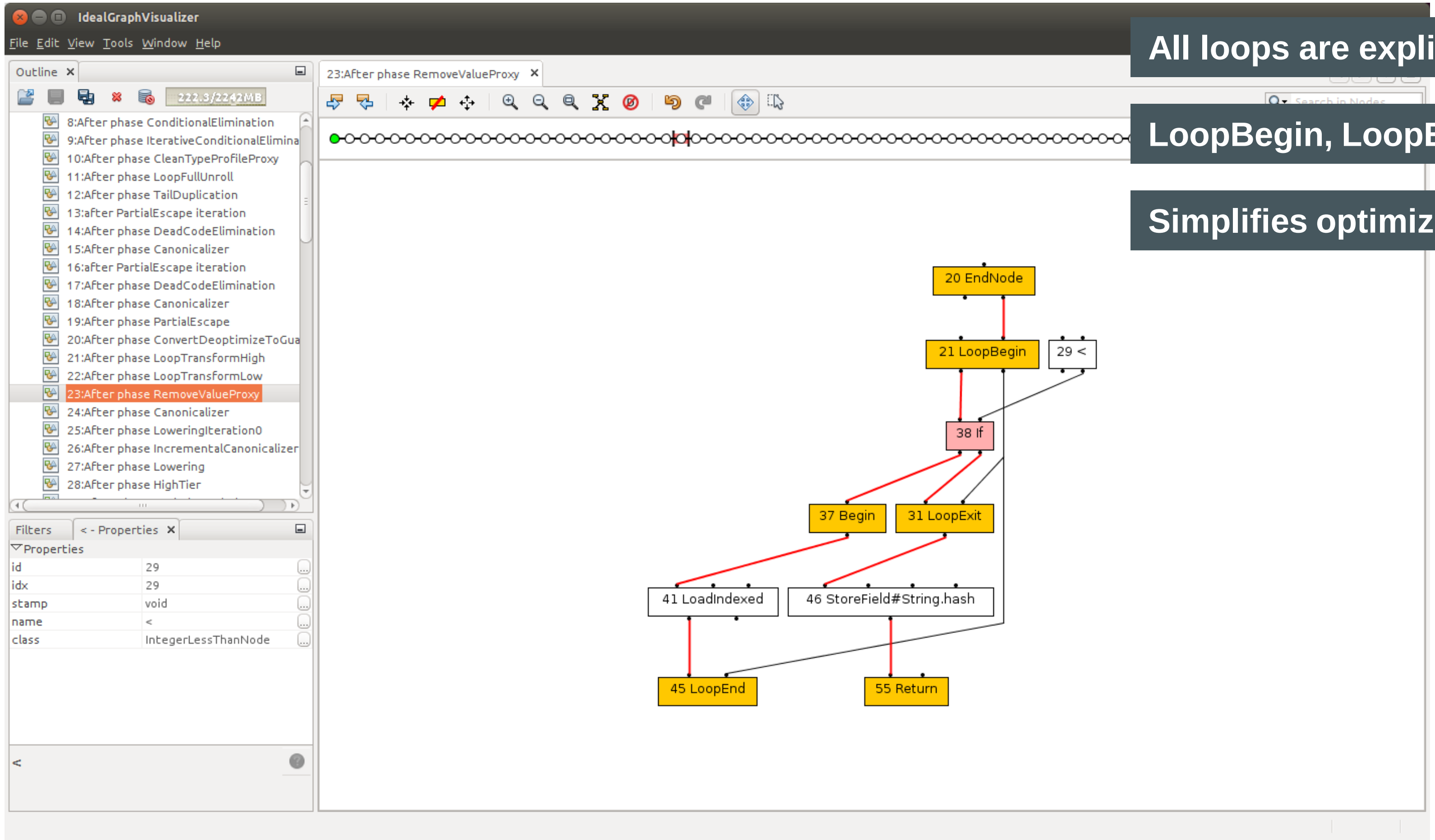


Floating nodes have no control flow dependency

Can be scheduled anywhere as long as data dependencies are fulfilled

Constants, arithmetic functions, phi functions, … are floating nodes

IR Example: Loops



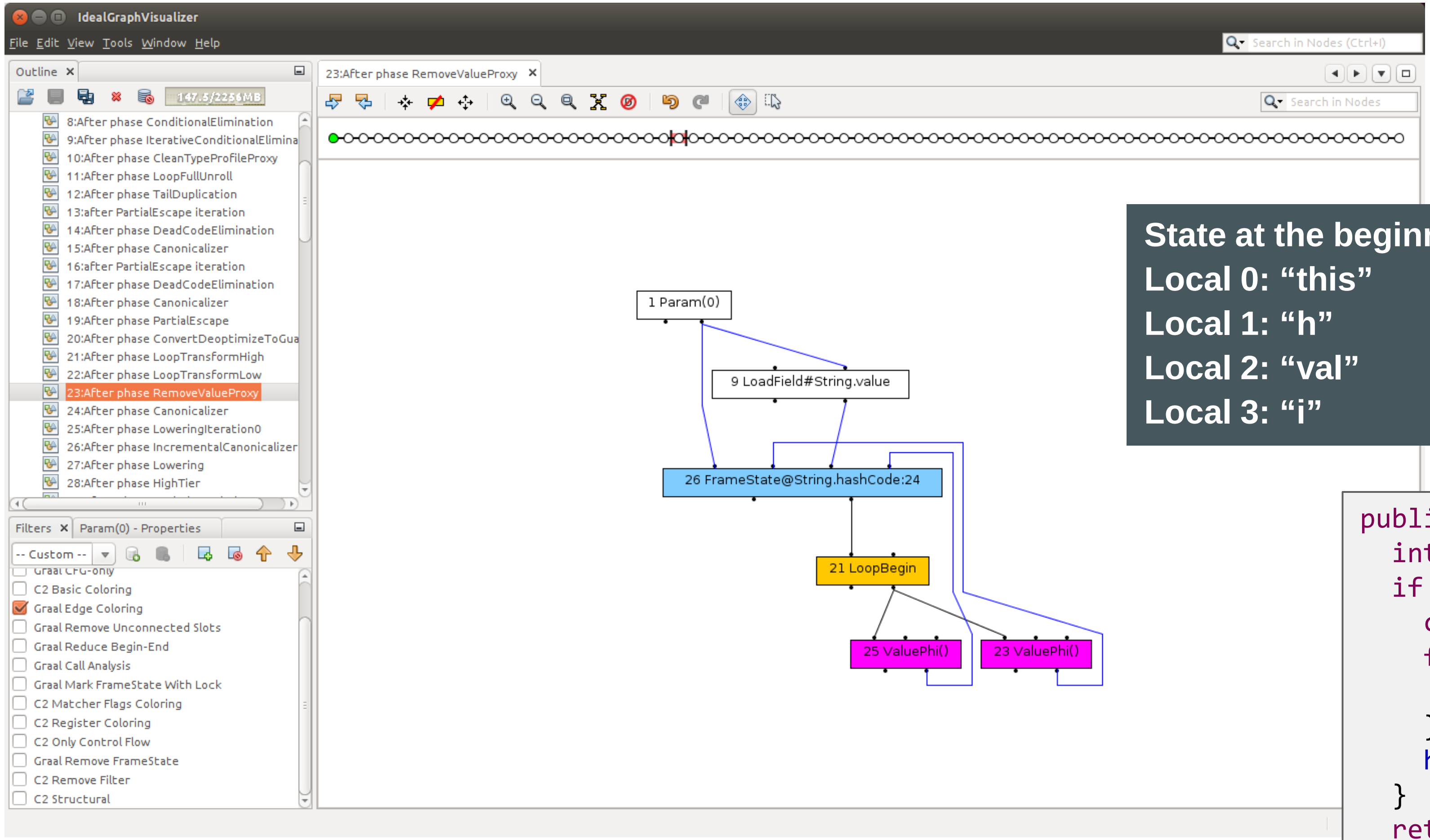
- **All loops are explicit and structured**
- **LoopBegin, LoopEnd, LoopExit nodes**
- **Simplifies optimization phases**

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

A Simple Optimization Phase

```
public class LockEliminationPhase extends Phase {

  @Override
  protected void run(StructuredGraph graph) {
    for (MonitorExitNode node : graph.getNodes(MonitorExitNode.class)) {
      FixedNode next = node.next();
      if (next instanceof MonitorEnterNode) {
        MonitorEnterNode monitorEnterNode = (MonitorEnterNode) next;
        if (monitorEnterNode.object() == node.object()) {
          GraphUtil.removeFixedWithUnusedInputs(monitorEnterNode);
          GraphUtil.removeFixedWithUnusedInputs(node);
        }
      }
    }
  }
}
```

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

Iterate all nodes of a certain class

Modify the graph

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

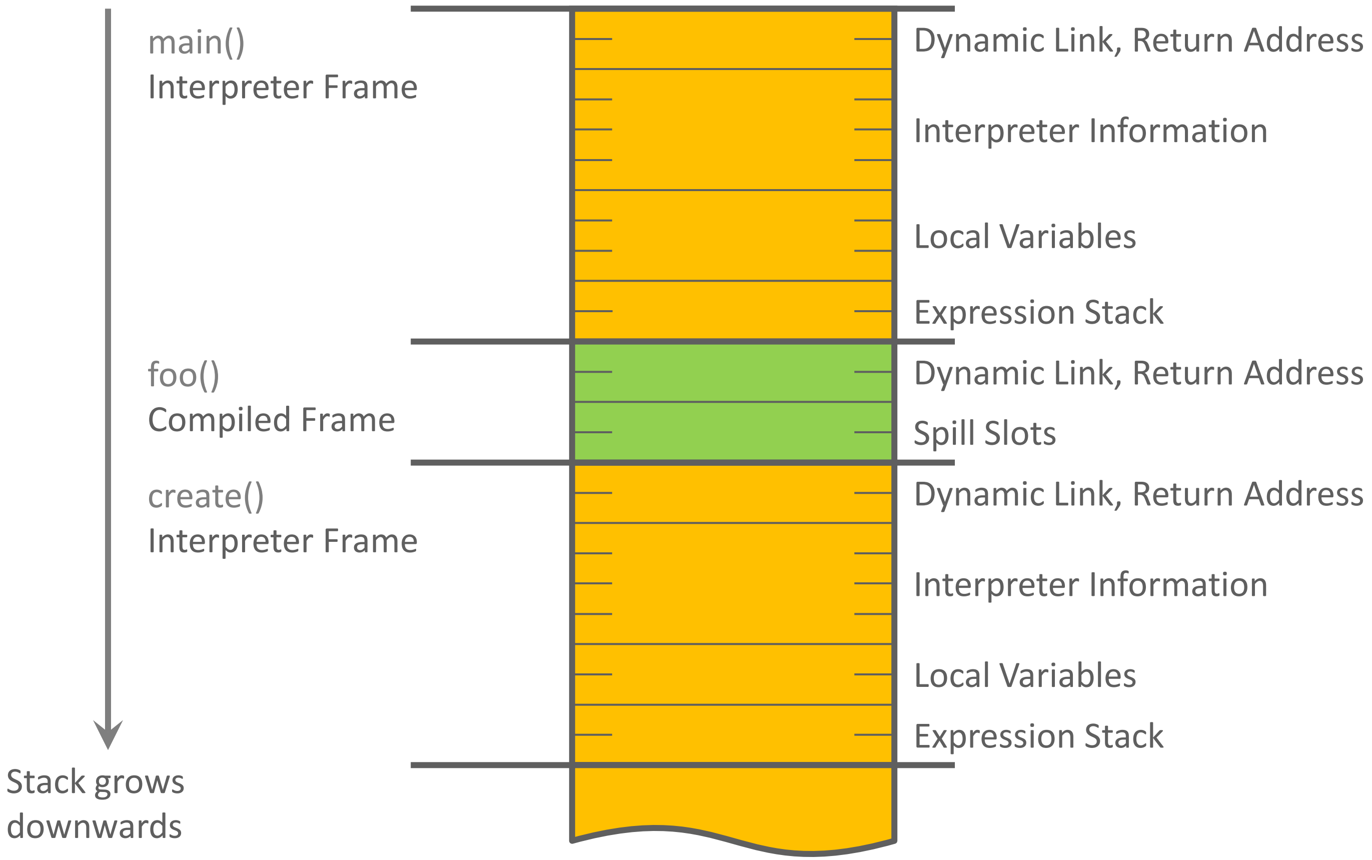
- Inlining of virtual methods
  - Most methods in Java are dynamically bound
  - Class Hierarchy Analysis
  - Inline when only one suitable method exists
- Compilation of foo() when only A loaded
  - Method getX() is inlined
  - Same machine code as direct field access
  - No dynamic type check
- Later 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

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

```java
class A {
  int x;

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

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

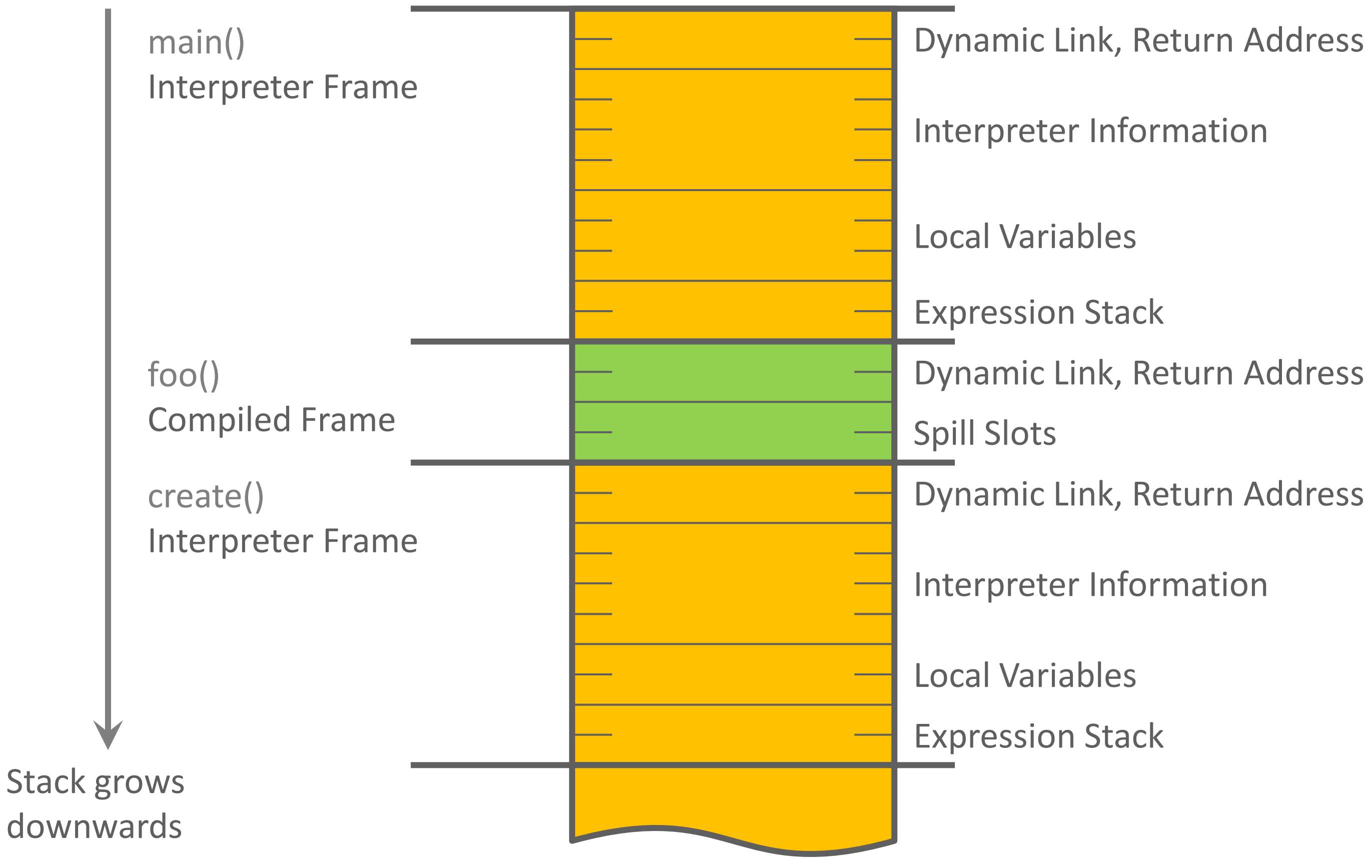
Deoptimization



Machine code for foo():

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

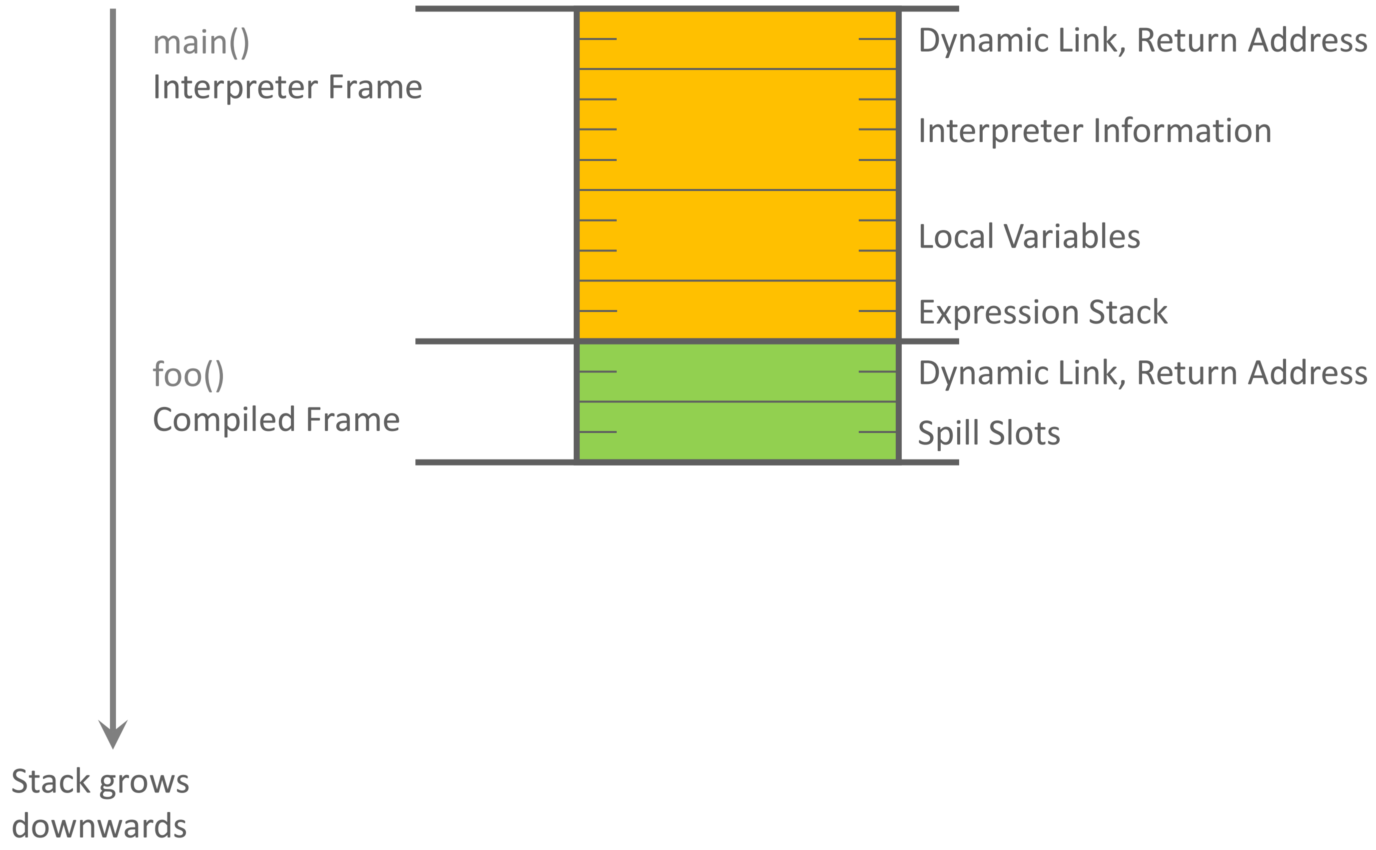
Deoptimization



Machine code for foo():

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

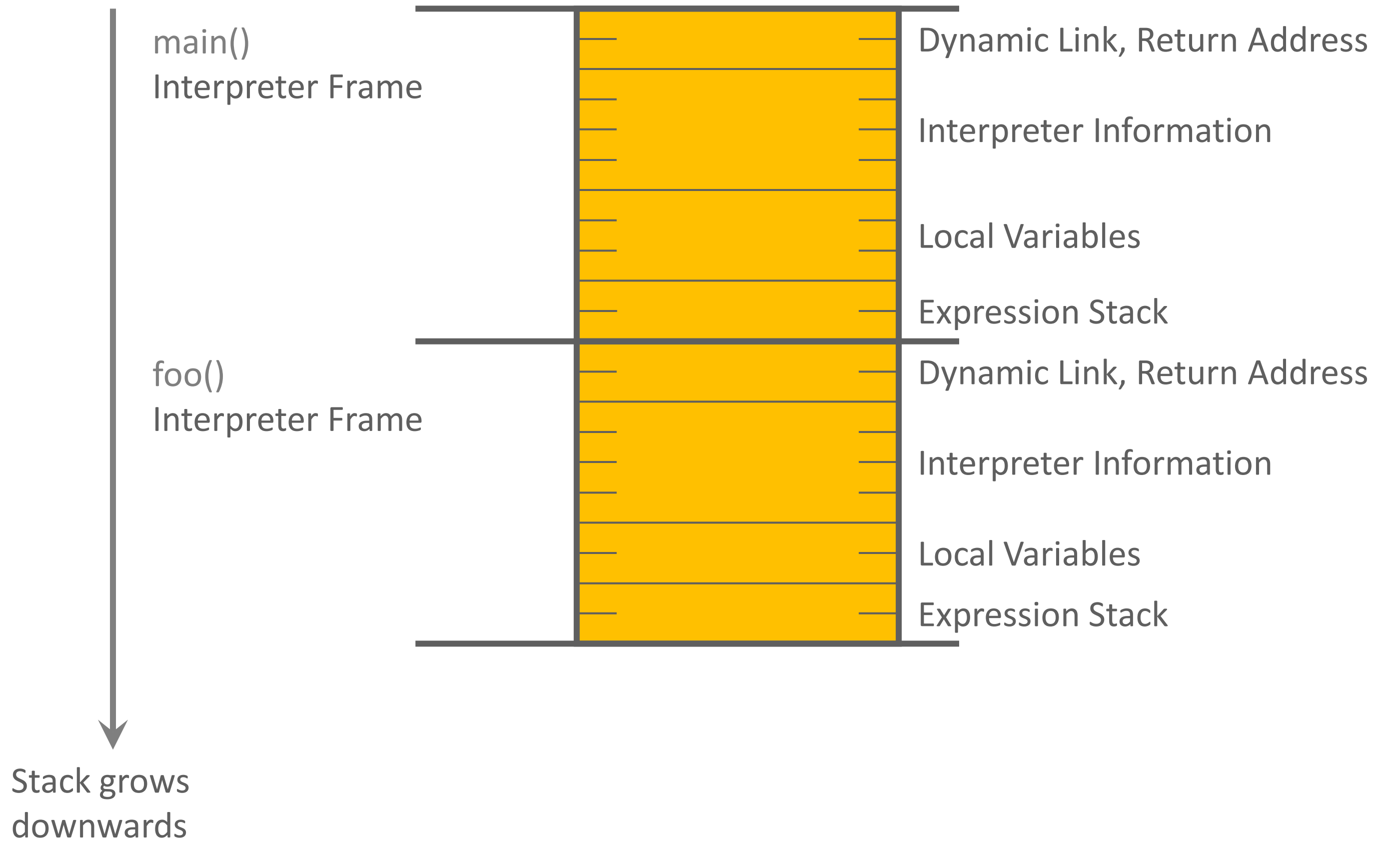

Deoptimization



Machine code for foo():

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

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;
}
```

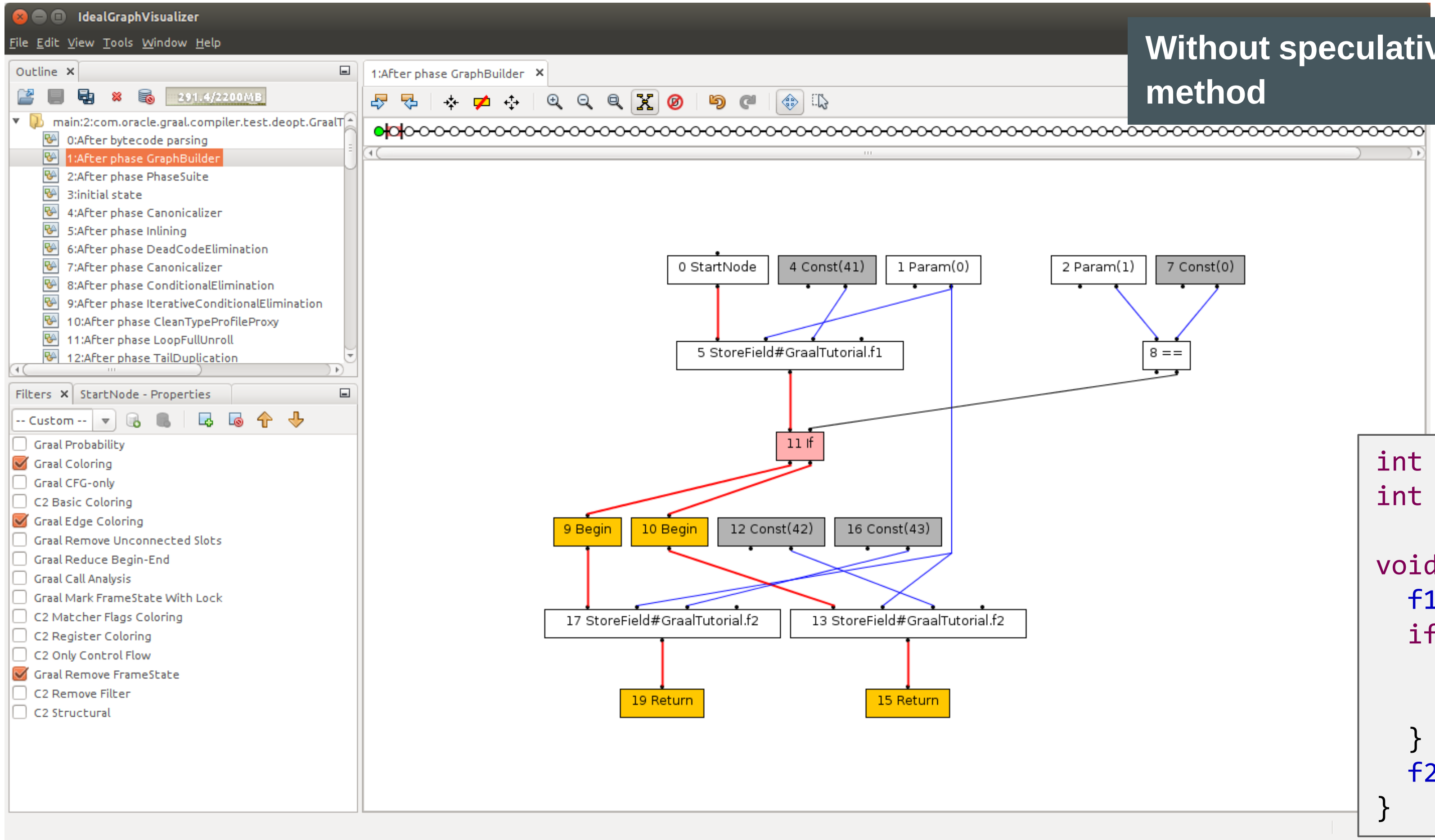
Assumption: method `speculativeOptimization` is always called with parameter `flag` set to `false`

Command line to run example:

```
./mx.sh igv &
./mx.sh unittest -G:Dump= -G:MethodFilter=GraalTutorial.speculativeOptimization GraalTutorial#testSpeculativeOptimization
```

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

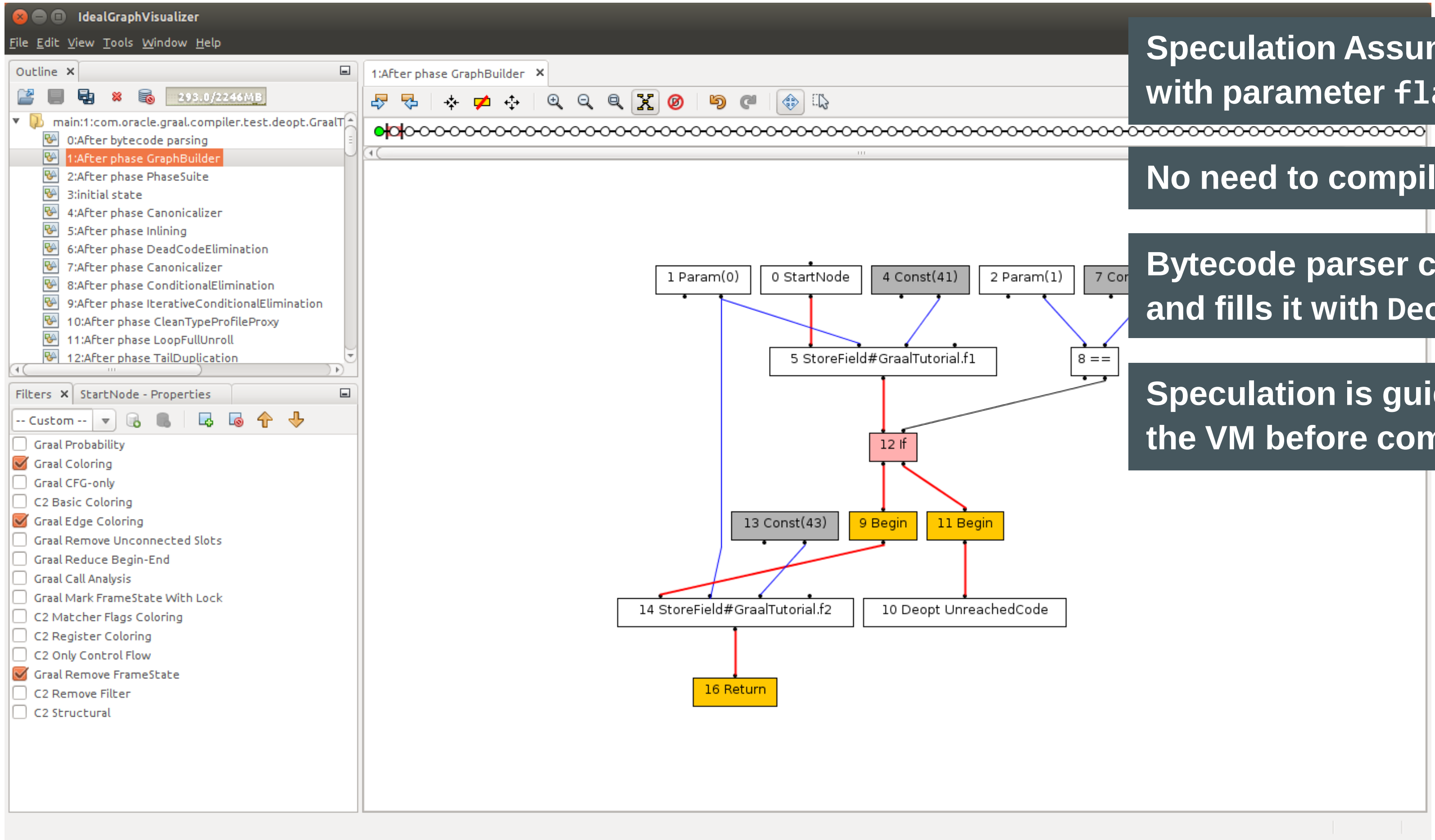
After Parsing without Speculation



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`

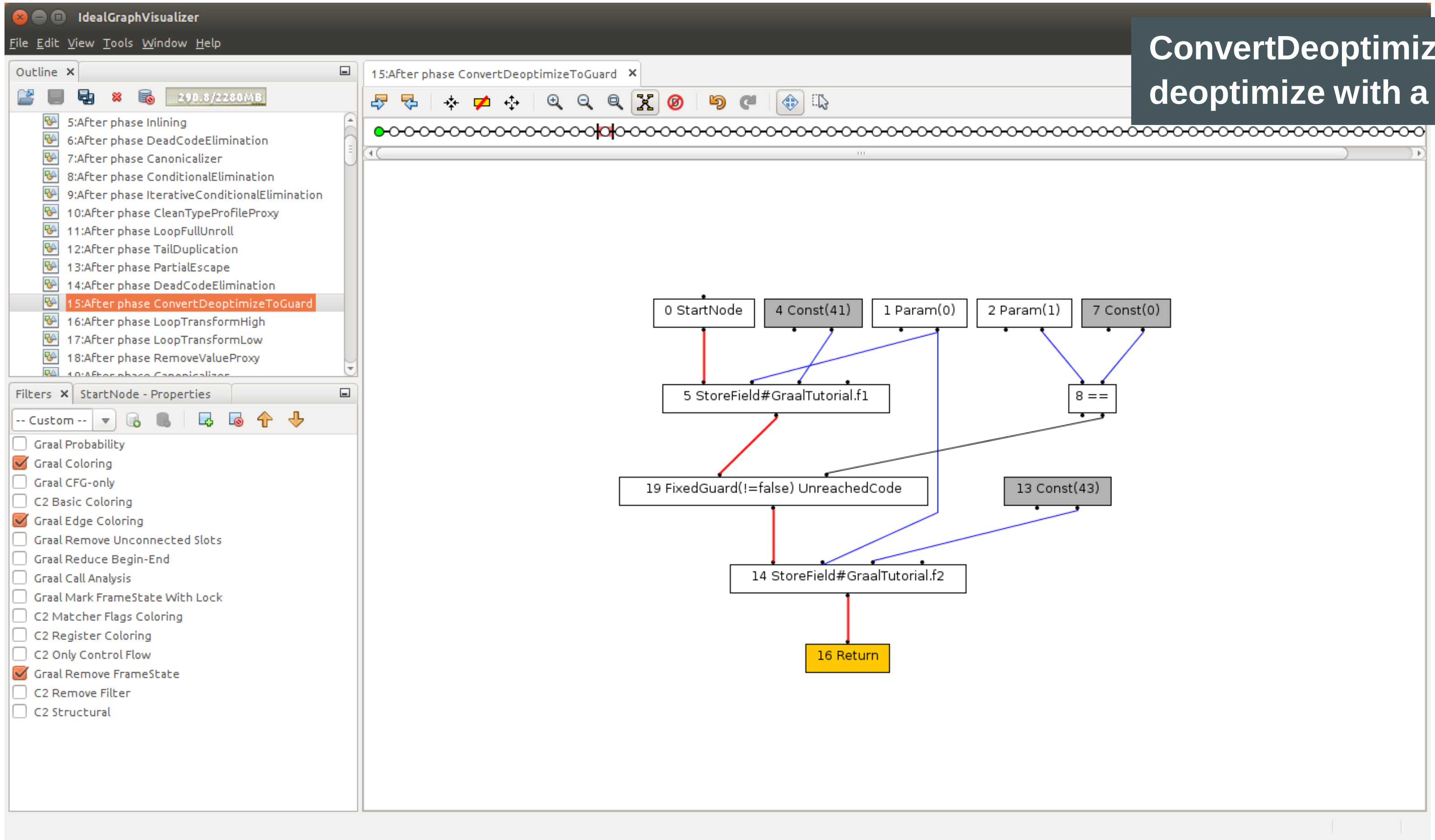
No need to compile the code inside the `if` block

Bytecode parser creates the `if` block, but stops parsing and fills it with `DeoptimizeNode`

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

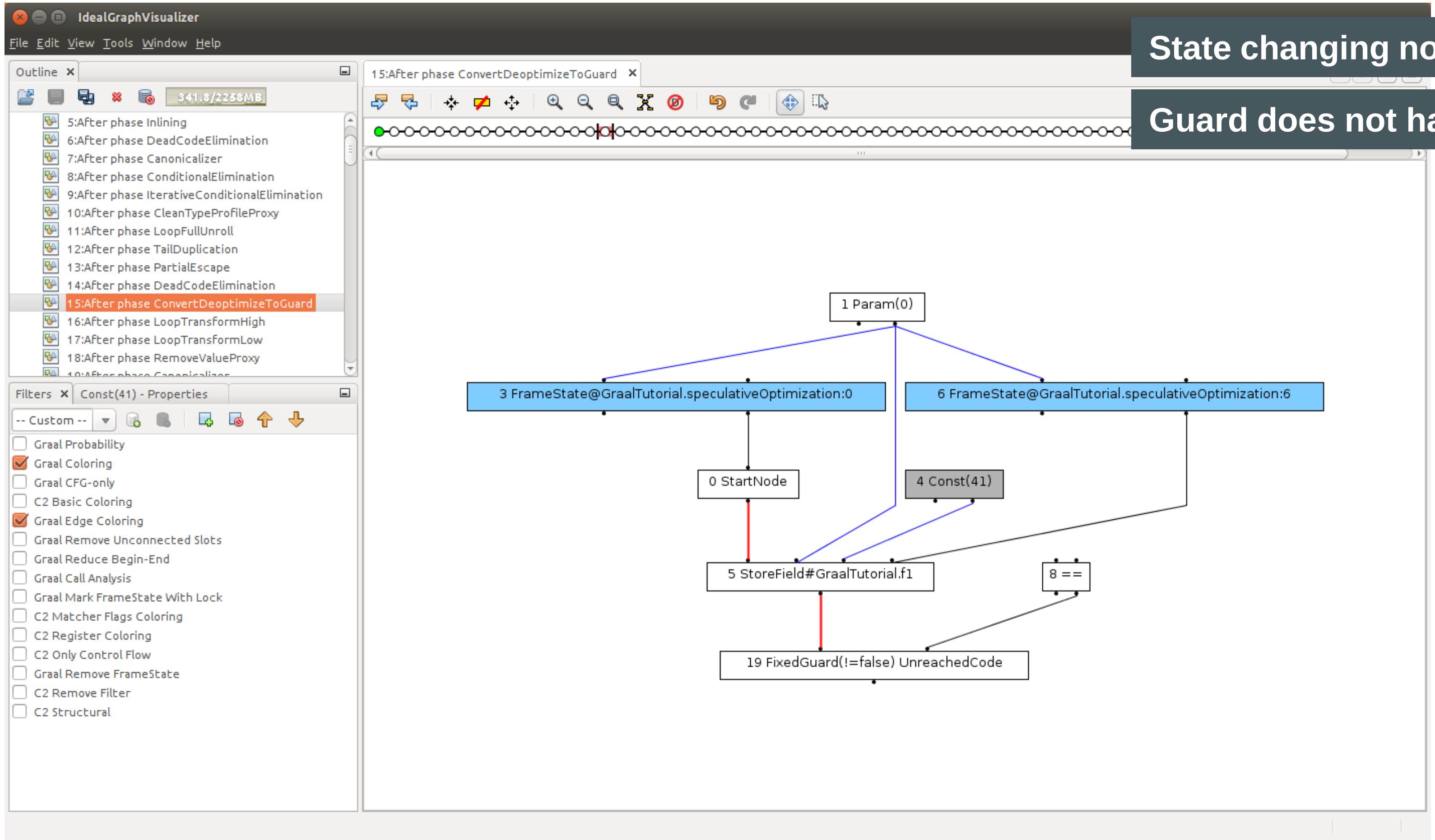
ORACLE

After Converting Deoptimize to Fixed Guard



ConvertDeoptimizeToGuardPhase replaces the if-deoptimize with a single FixedGuardNode

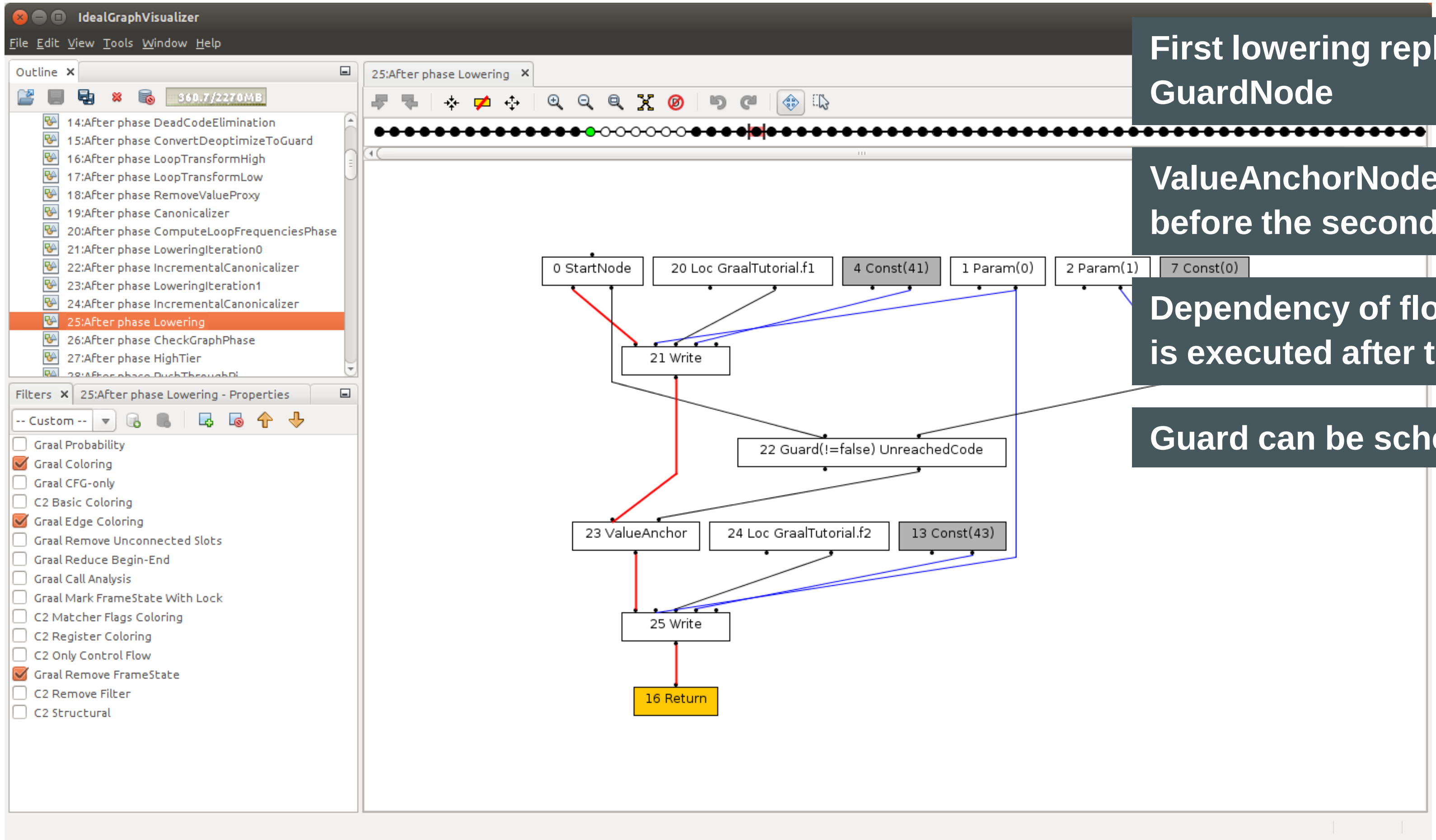
Frame states after Parsing



State changing nodes have a FrameState

Guard does not have a FrameState

After Lowering: Guard is Floating



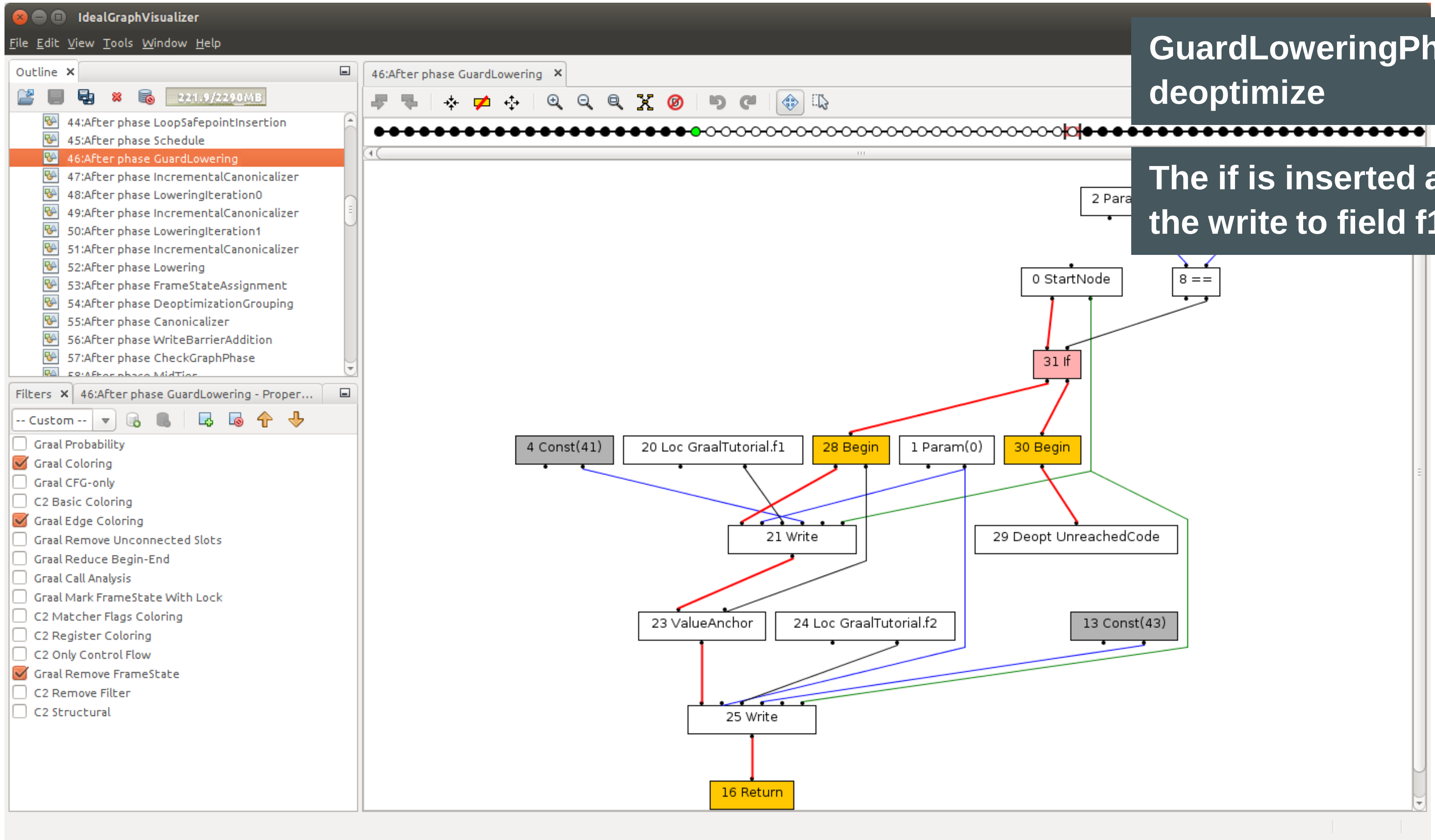
First lowering replaces the FixedGuardNode with a floating GuardNode

ValueAnchorNode ensures the floating guard is executed before the second write

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

Guard can be scheduled within these constraints

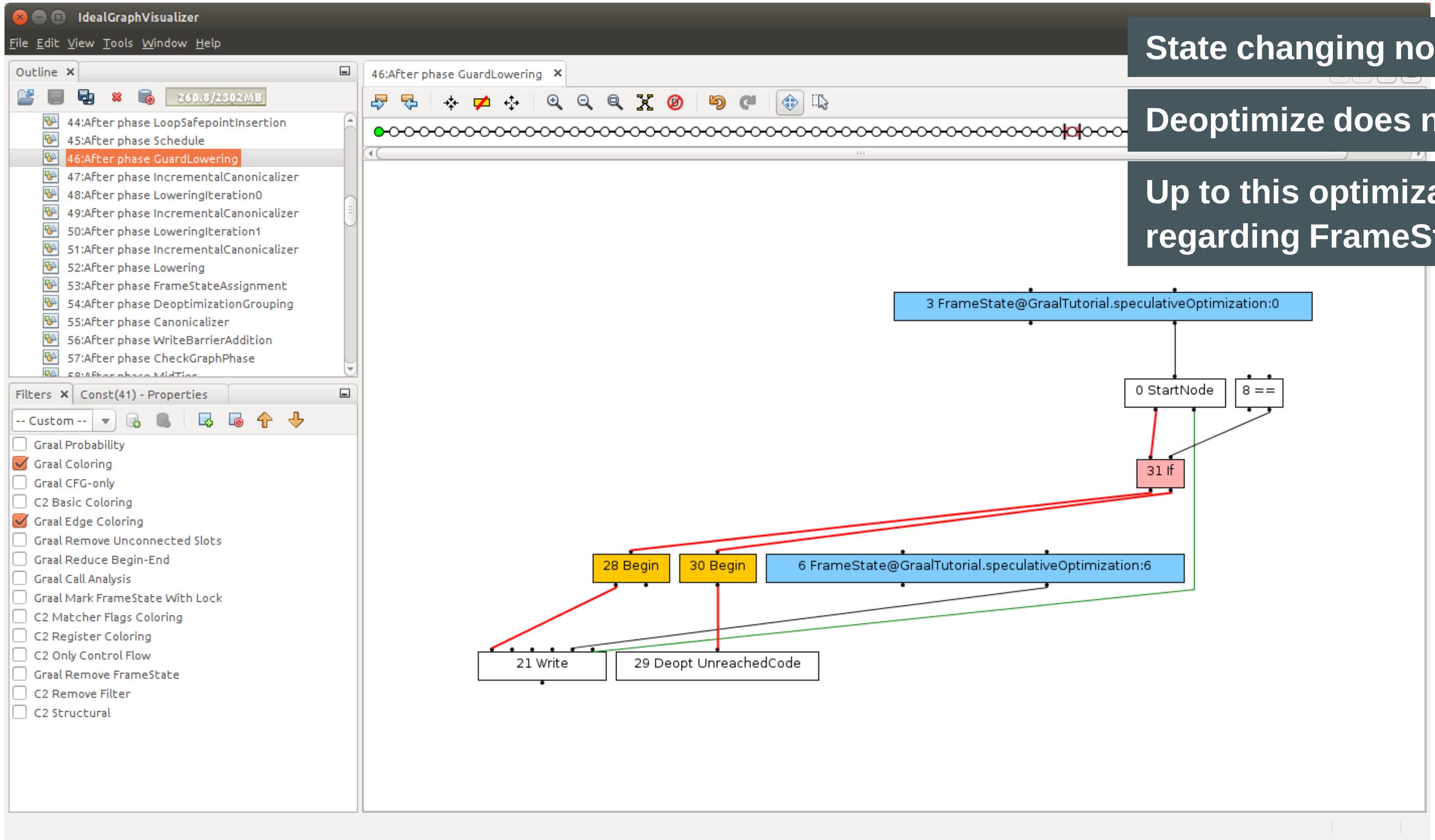
After Replacing Guard with If-Deoptimize



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

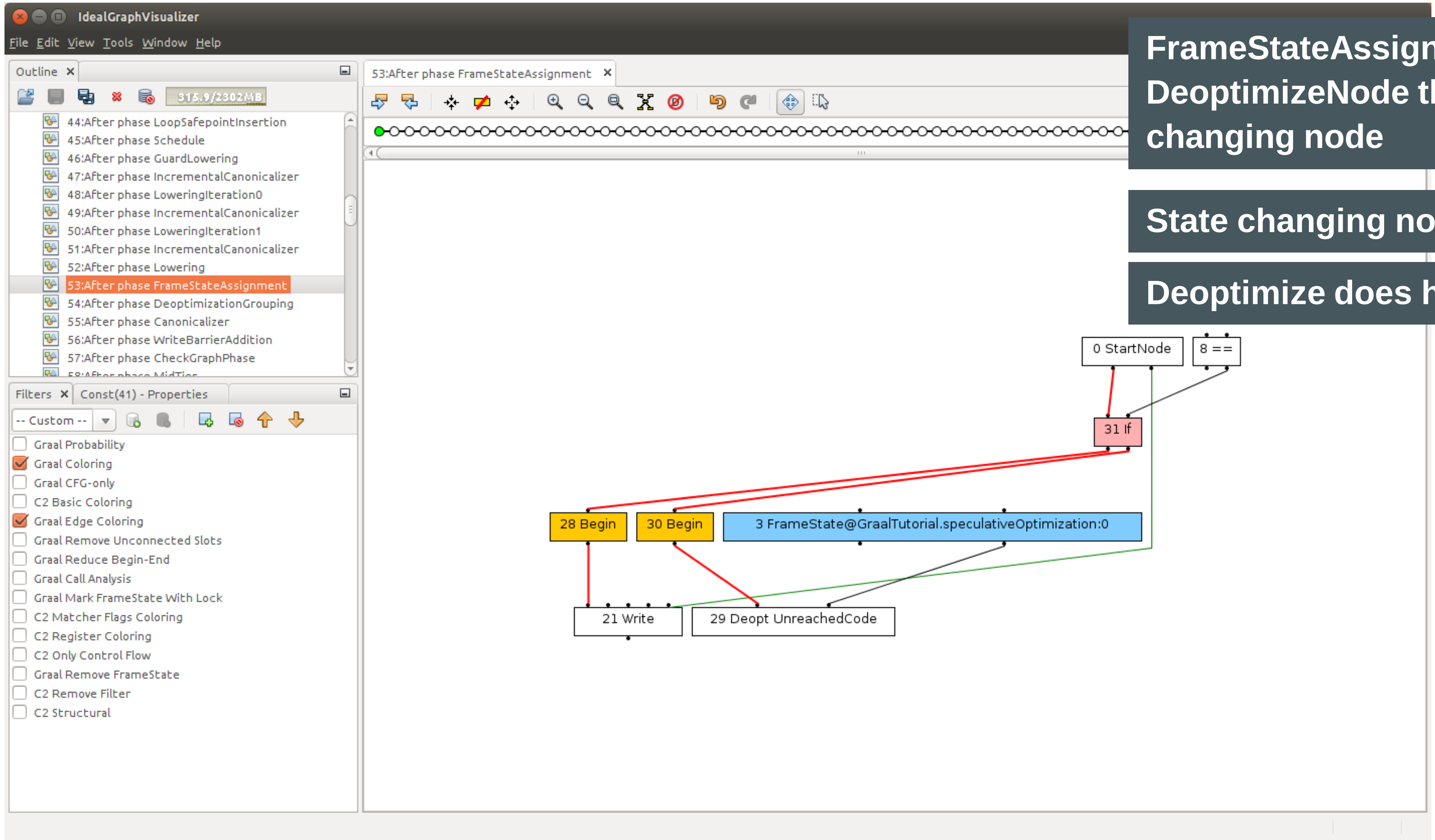


State changing nodes have a FrameState

Deoptimize does not have a FrameState

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

After FrameStateAssignmentPhase



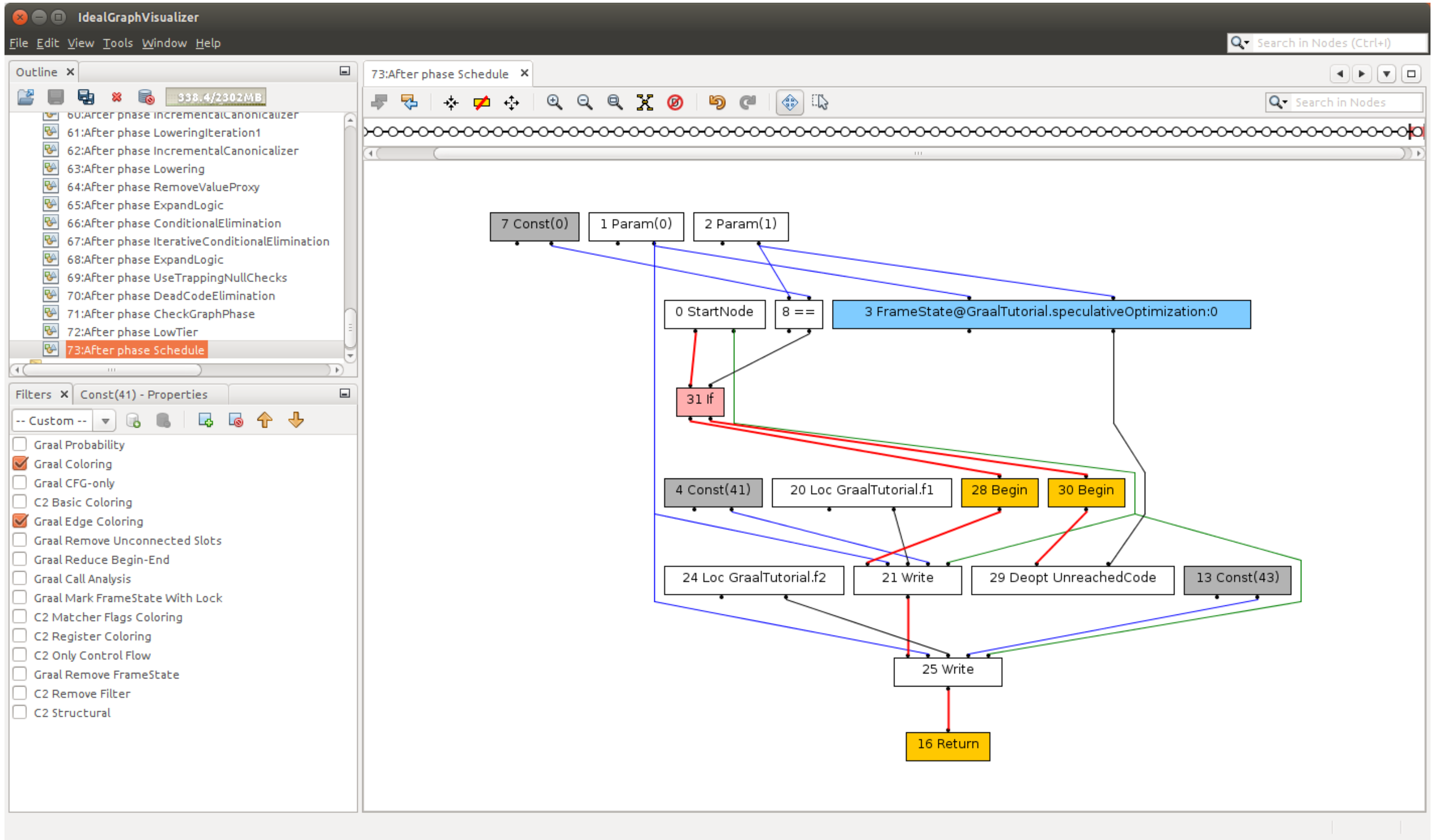
FrameStateAssignmentPhase assigns every DeoptimizeNode the FrameState of the preceding state changing node

State changing nodes do not have a FrameState

Deoptimize does have a FrameState

ORACLE

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.

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

Graal API

Graal API 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

- Graal API 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

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 addMethod(ResolvedJavaMethod method, CompilationResult compResult,
          SpeculationLog speculationLog, InstalledCode predefinedInstalledCode);
  InstalledCode setDefaultMethod(ResolvedJavaMethod method, CompilationResult compResult);
  TargetDescription getTarget();
  ...
}
```

Install compiled code into the VM

Example: Print 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 = ...
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. */
System.out.println(new BytecodeDisassembler().disassemble(method));
```

Command line to run example:

```
./mx.sh unittest GraalTutorial#testPrintBytecodes
```

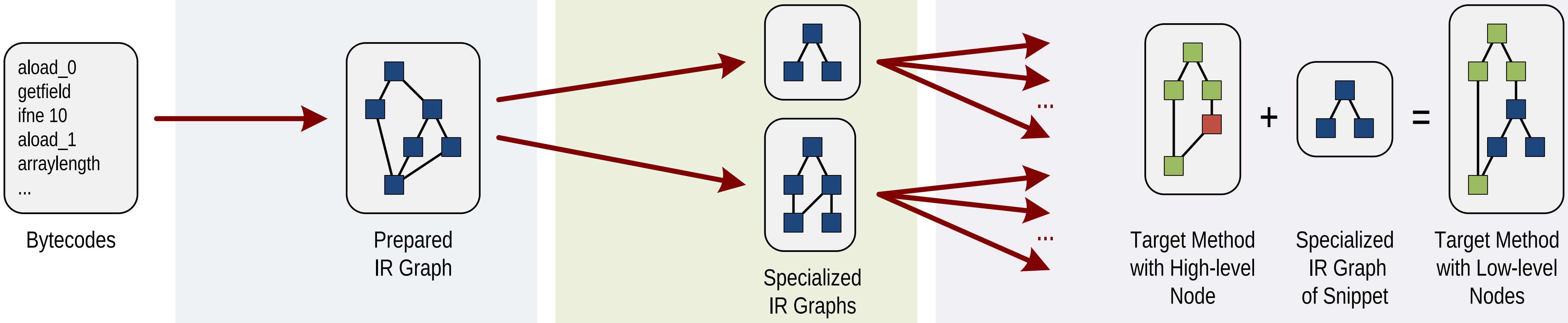
Snippets

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

Snippet Example: instanceOf with Profiling Information

```
@Snippet
static Object instanceofWithProfile(Object object,
      @ConstantParameter boolean nullSeen,
      @VarargsParameter Word[] profiledHubs,
      @VarargsParameter boolean[] hubIsPositive) {

  if (probability(NotFrequent, object == null)) {
    if (!nullSeen) {
      deoptimize(OptimizedTypeCheckViolated);
      throw shouldNotReachHere();
    }
    isNullCounter.increment();
    return false;
  }
  Anchor afterNullCheck = anchor();
  Word objectHub = loadHub(object, afterNullCheck);

  explodeLoop();
  for (int i = 0; i < profiledHubs.length; i++) {
    if (profiledHubs[i].equal(objectHub)) {
      profileHitCounter.increment();
      return hubIsPositive[i];
    }
  }
  deoptimize(OptimizedTypeCheckViolated);
  throw shouldNotReachHere();
}
```

Constant folding during specialization

Loop unrolling during specialization

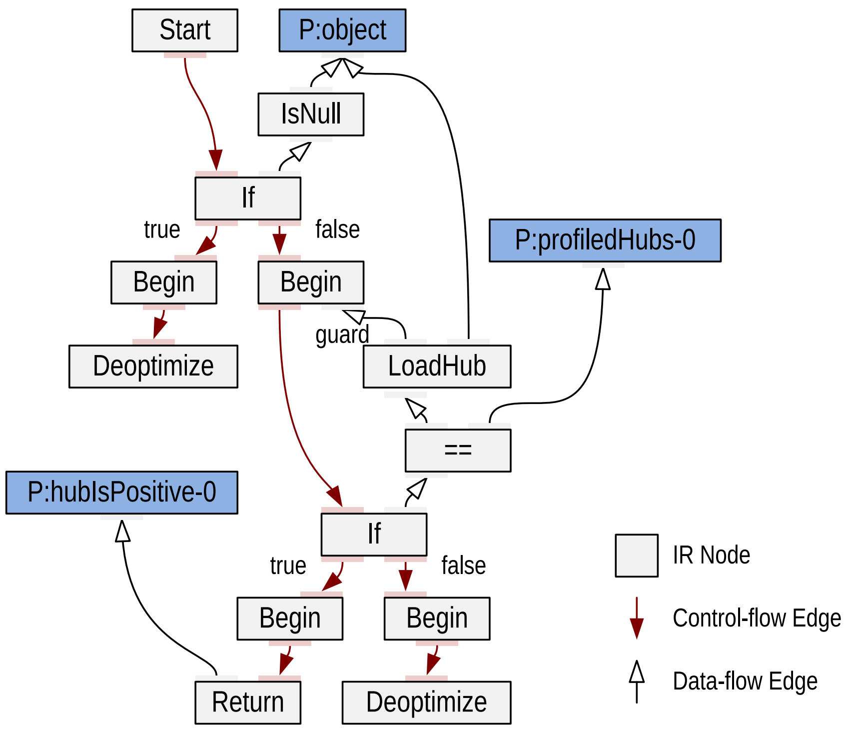
Node intrinsic

Debug / profiling code eliminated by constant folding and dead code elimination

Loop unrolling during specialization

Snippet Example: Specialization for One Type

```
@Snippet
static Object instanceofWithProfile(Object object,
      @ConstantParameter boolean nullSeen,
      @VarargsParameter Word[] profiledHubs,
      @VarargsParameter boolean[] hubIsPositive) {

  if (probability(NotFrequent, object == null)) {
    if (!nullSeen) {
      deoptimize(OptimizedTypeCheckViolated);
      throw shouldNotReachHere();
    }
    isNullCounter.increment();
    return false;
  }
  Anchor afterNullCheck = anchor();
  Word objectHub = loadHub(object, afterNullCheck);

  explodeLoop();
  for (int i = 0; i < profiledHubs.length; i++) {
    if (profiledHubs[i].equal(objectHub)) {
      profileHitCounter.increment();
      return hubIsPositive[i];
    }
  }
  deoptimize(OptimizedTypeCheckViolated);
  throw shouldNotReachHere();
}
```



Node Intrinsics

```
class LoadHubNode extends FloatingGuardedNode {

  @Input ValueNode object;

  LoadHubNode(ValueNode object, ValueNode guard) {
    super(guard);
    this.object = object;
  }
}

@NodeIntrinsic(LoadHubNode.class)
static native Word loadHub(Object object, Object guard);
```

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

```
class DeoptimizeNode extends ControlSinkNode {

  final Reason reason;

  DeoptimizeNode(Reason reason) {
    this.object = object;
  }
}

@NodeIntrinsic(DeoptimizeNode.class)
static native void deoptimize(
                       @ConstantNodeParameter Reason reason);
```

Constructor with non-Node parameter requires node intrinsic parameter to be a constant during snippet specialization

ORACLE

Snippet Instantiation

```
SnippetInfo instanceofWithProfile = snippet(InstanceOfSnippets.class, "instanceofWithProfile");

void lower(InstanceOfNode node) {
  ValueNode object = node.getObject();
  JavaTypeProfile profile = node.getProfile();

  if (profile.totalProbability() > threshold) {
    int numTypes = profile.getNumTypes();
    Word[] profiledHubs = new Word[numTypes];
    boolean hubIsPositive = new boolean[numTypes];
    for (int i = 0; i < numTypes; i++) {
      profiledHubs[i] = profile.getType(i).getHub();
      hubIsPositive[i] = profile.isPositive(i);
    }

    Args args = new Args(instanceofWithProfile);
    args.add(object);
    args.addConst(profile.getNullSeen());
    args.addVarargs(profiledHubs);
    args.addVarargs(hubIsPositive);

    SnippetTemplate s = template(args);
    s.instantiate(args, node);

  } else {
    // Use a different snippet.
  }
}
```

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

Constant argument for snippet specialization

Snippet preparation and specialization

Snippet instantiation

Example in IGV

- The previous slides are slightly simplified
  - In reality the snippet graph is a bit more complex
  - But the end result is the same

The snippets for lowering of instanceOf are in class `InstanceOfSnippets`

Java source code:

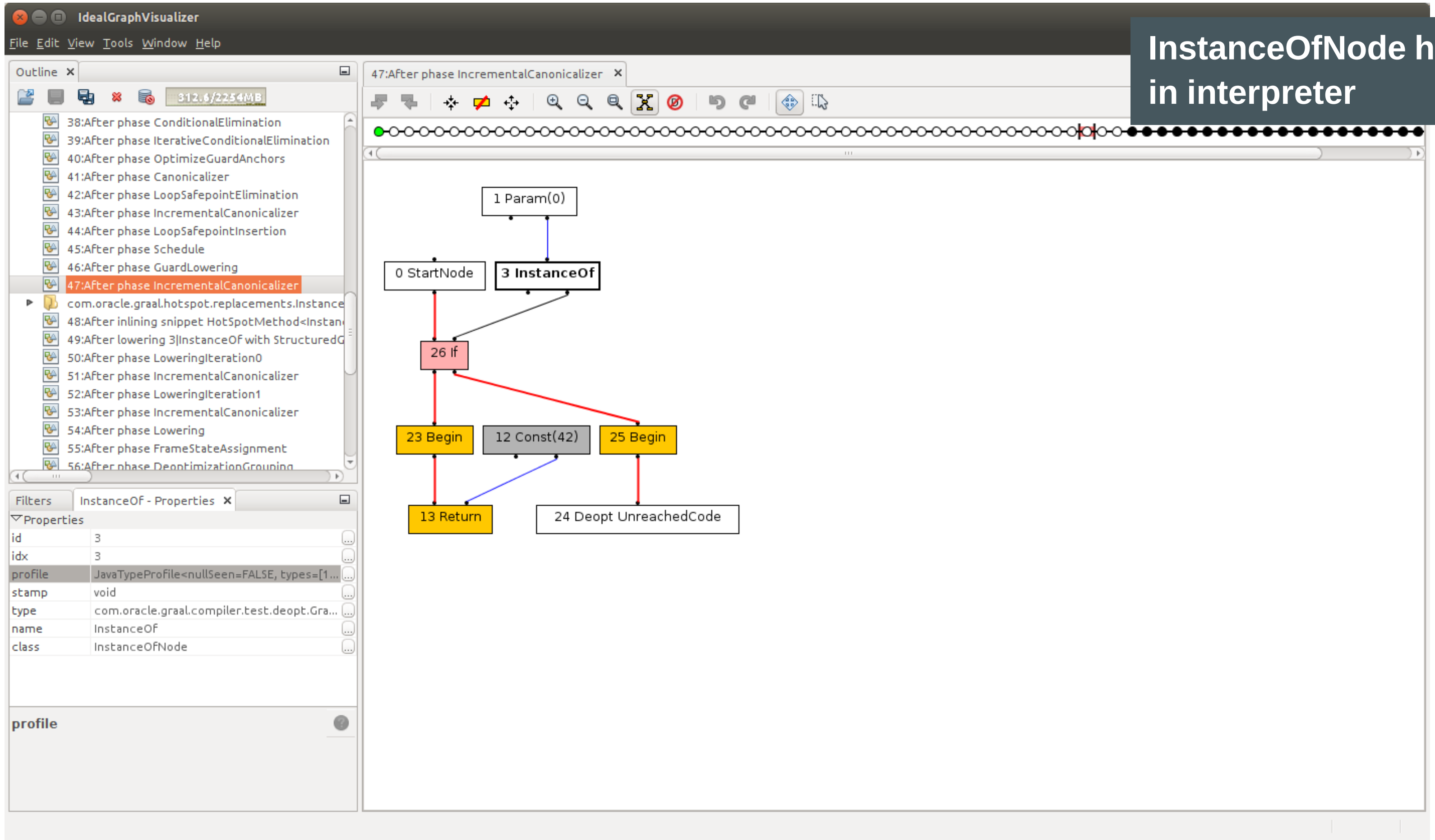
```
static class A { }
static class B extends A { }

static int instanceOfUsage(Object obj) {
  if (obj instanceof A) {
    return 42;
  } else {
    return 0;
  }
}
```

Assumption: method `instanceOfUsage` is always called with parameter `obj` having class `A`

Command line to run example:

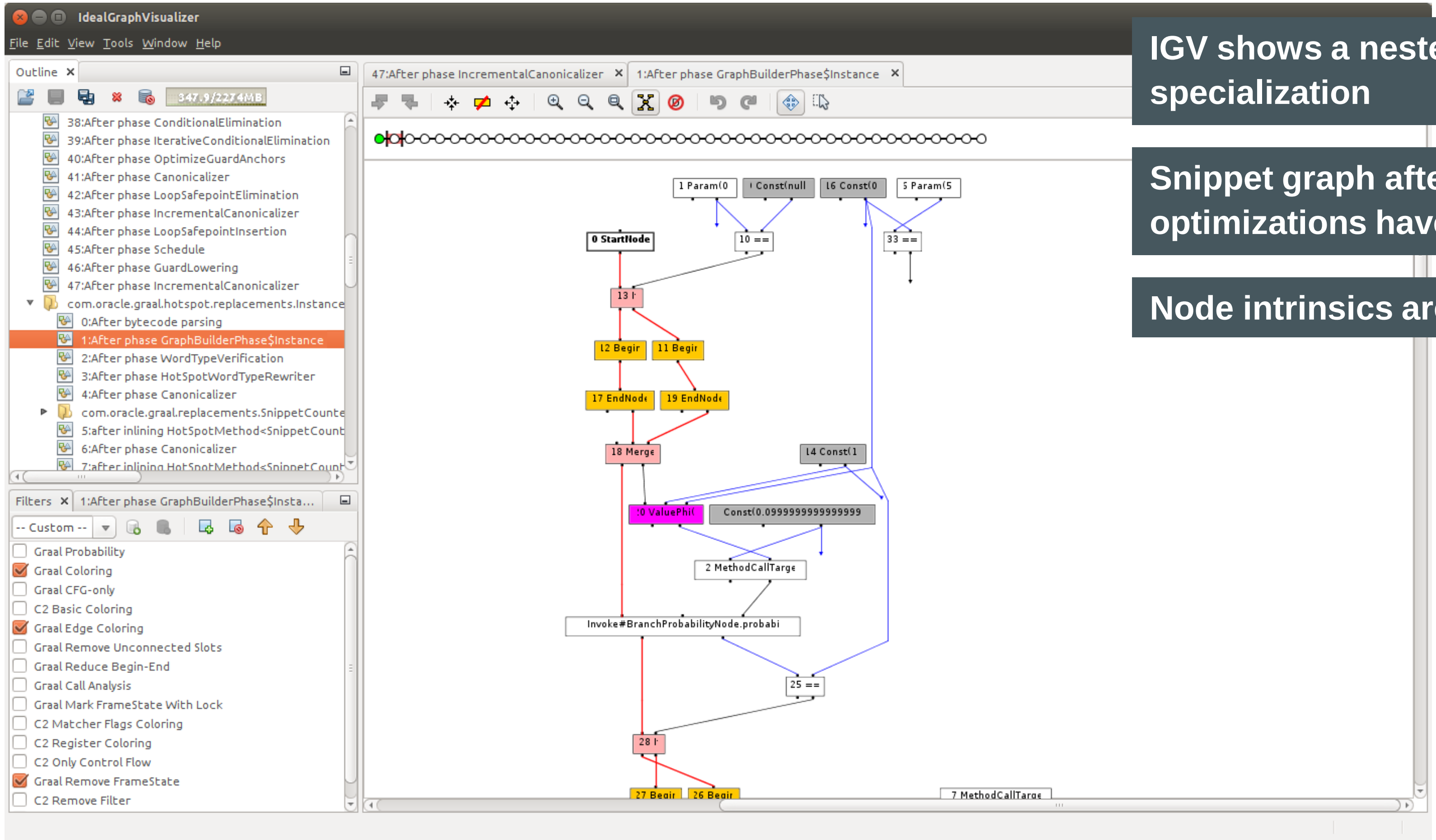
```
./mx.sh igv &
./mx.sh unittest -G:Dump= -G:MethodFilter=GraalTutorial.instanceOfUsage GraalTutorial#testInstanceOfUsage
```

Method Before Lowering



InstanceOfNode has profiling information: only type A seen in interpreter

Snippet After Parsing



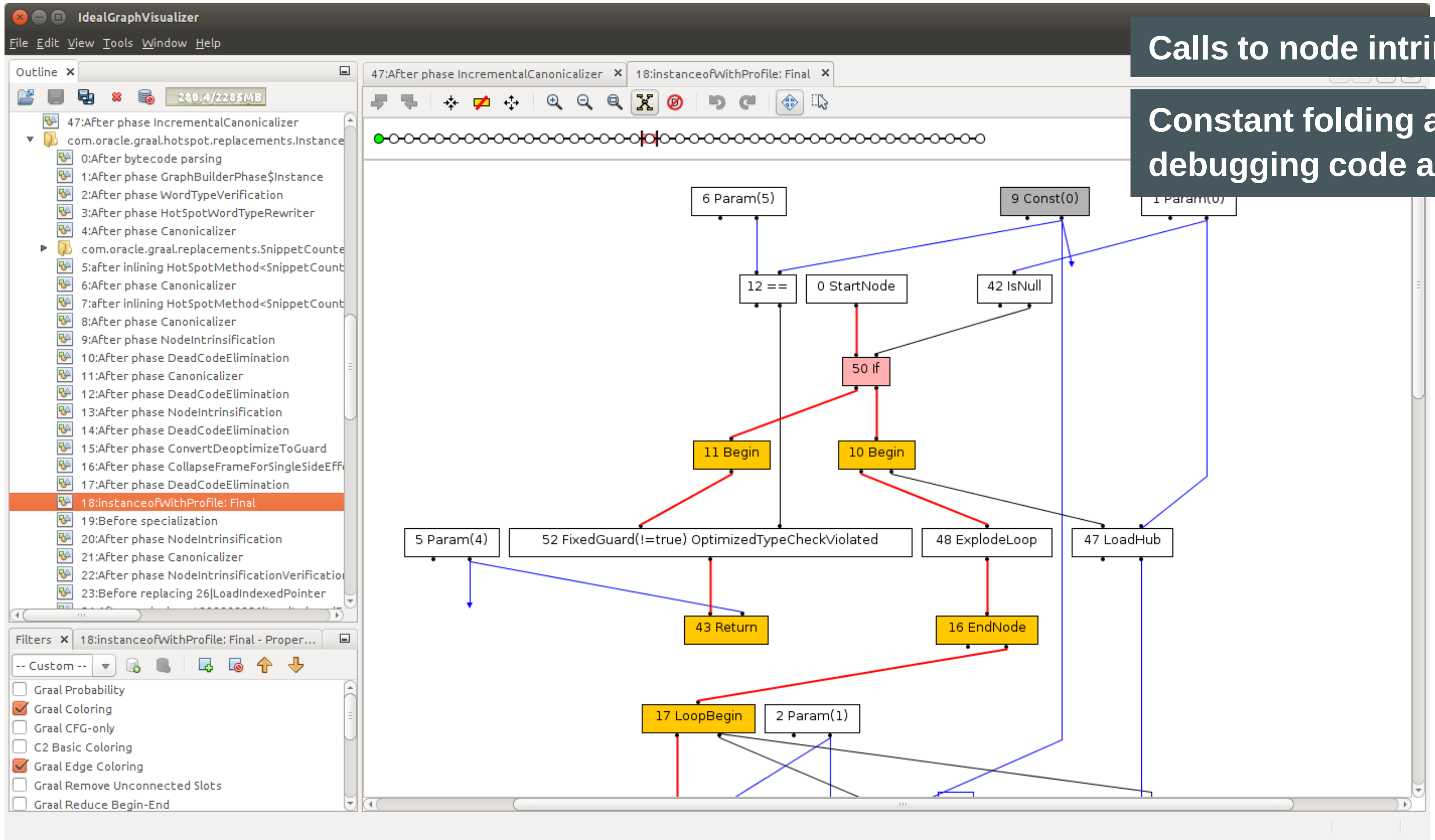
IGV shows a nested graph for snippet preparation and specialization

Snippet graph after bytecode parsing is big, because no optimizations have been performed yet

Node intrinsics are still method calls

ORACLE

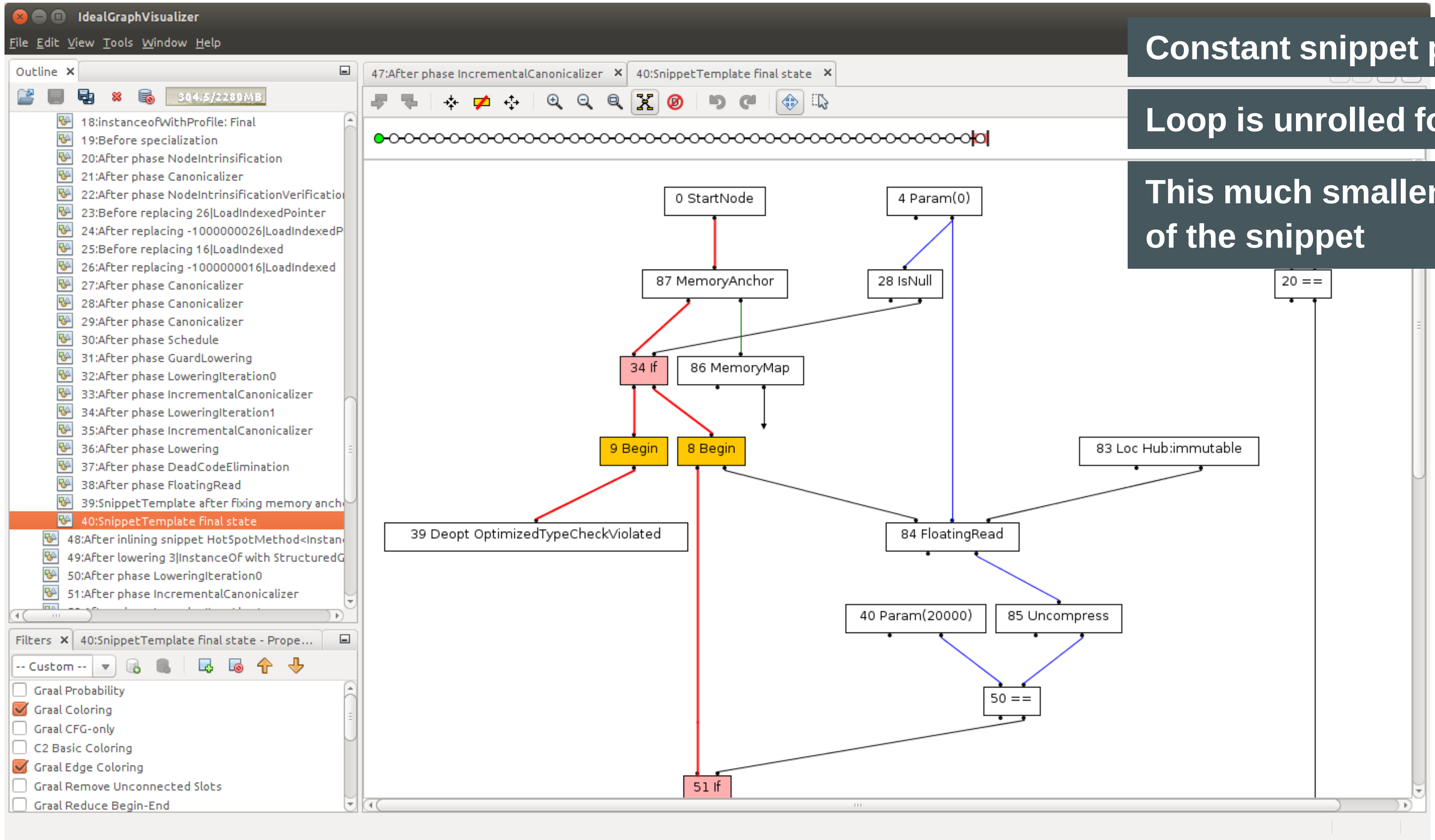
Snippet After Preparation



Calls to node intrinsics are replaced with actual nodes

Constant folding and dead code elimination removed debugging code and counters

Snippet After Specialization

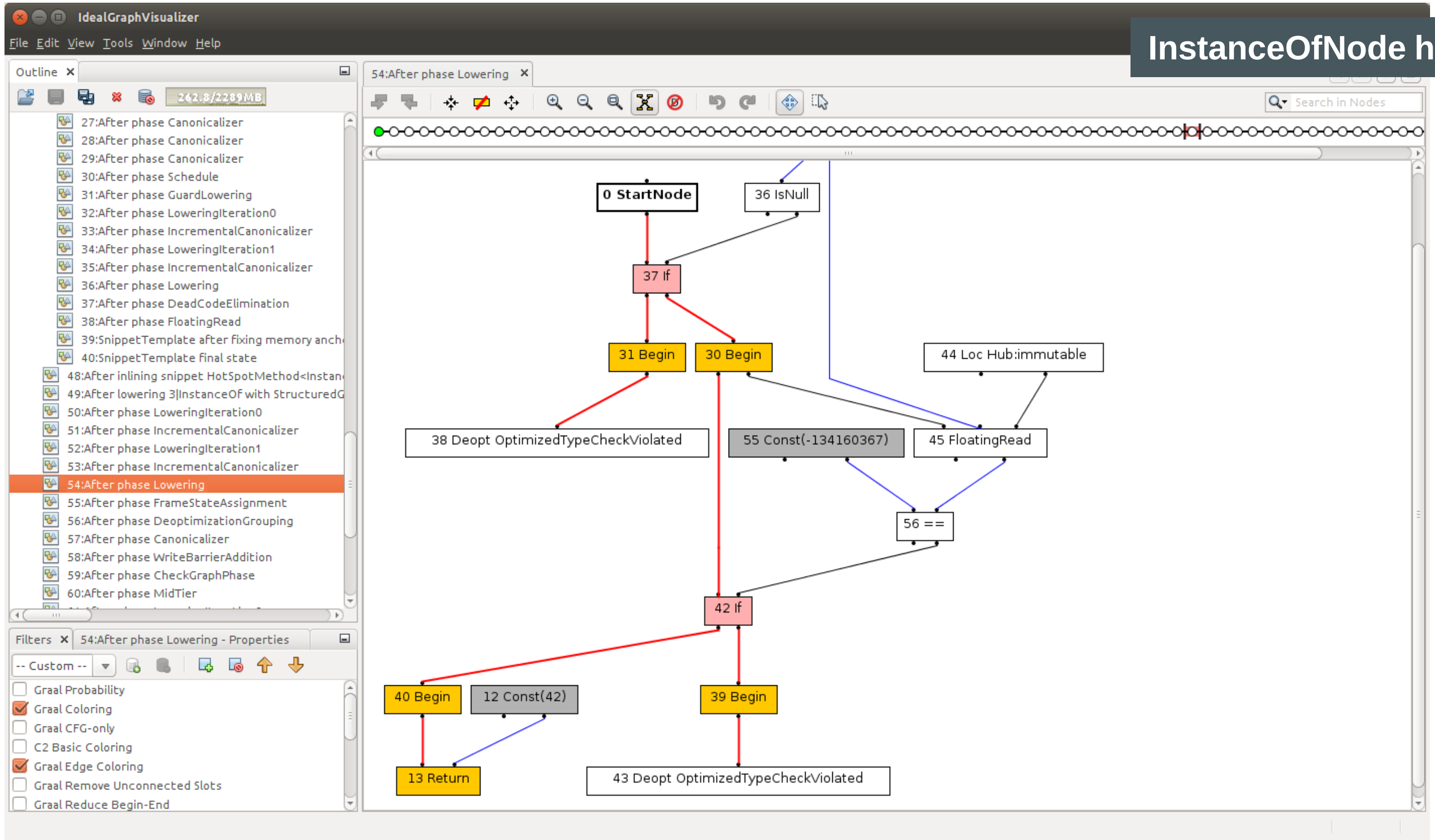


Constant snippet parameter is constant folded

Loop is unrolled for length 1

This much smaller graph is cached for future instantiations of the snippet

Method After Lowering



InstanceOfNode has been replaced with snippet graph

Compiler Intrinsics

Compiler Intrinsics

- Called “method substitution” in Graal
  - A lot mechanism and infrastructure shared with snippets

- 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
  - Define a method substitution for the Java method that should be intrinsified
    - Use a node intrinsic to create your node
  - 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

ORACLE

Example: Intrinsification of Math.sin()

Java source code:

```
static double intrinsicUsage(double val) {
  return Math.sin(val);
}
```

Java implementation of Math.sin() calls native code via JNI

x86 provides an FPU instruction: fsin

Command line to run example:

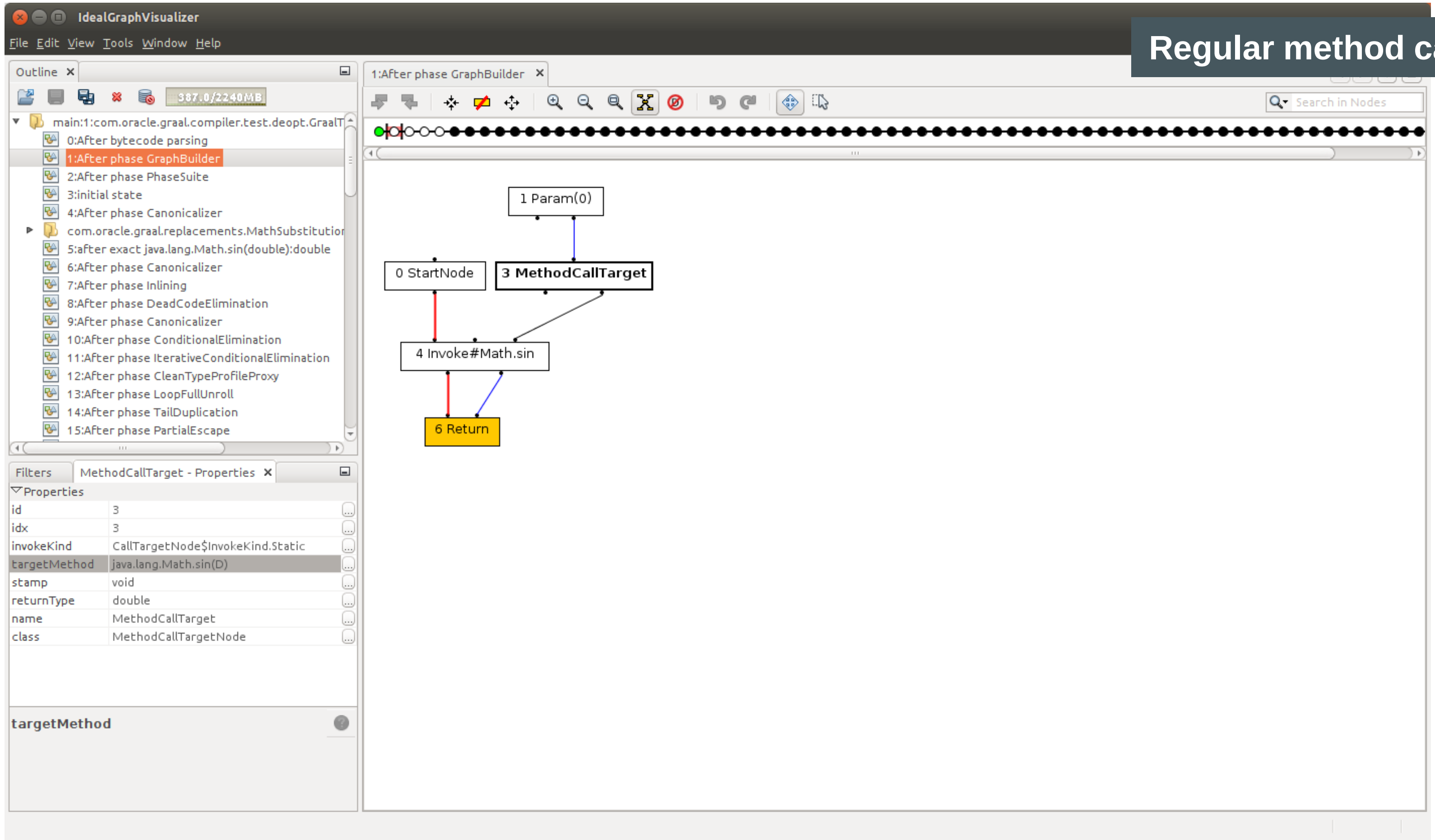
```
./mx.sh igv &
./mx.sh c1visualizer &
./mx.sh unittest -G:Dump= -G:MethodFilter=GraalTutorial.intrinsicUsage GraalTutorial#testIntrinsicUsage
```

C1Visualizer shows the LIR and generated machine code

Load the generated .cfg file with C1Visualzier

ORACLE

After Parsing



Regular method call to Math.sin()

Method Substitution

```
public class MathIntrinsicNode extends FloatingNode implements ArithmeticLIRLowerable {
  public enum Operation {LOG, LOG10, SIN, COS, TAN }

  @Input protected ValueNode value;
  protected final Operation operation;

  public MathIntrinsicNode(ValueNode value, Operation op) { ... }
  @NodeIntrinsic
  public static native double compute(double value, @ConstantNodeParameter Operation op);

  public void generate(NodeMappableLIRBuilder builder, ArithmeticLIRGenerator gen) { ... }
}
```

Node with node intrinsic shared several Math methods

LIR Generation

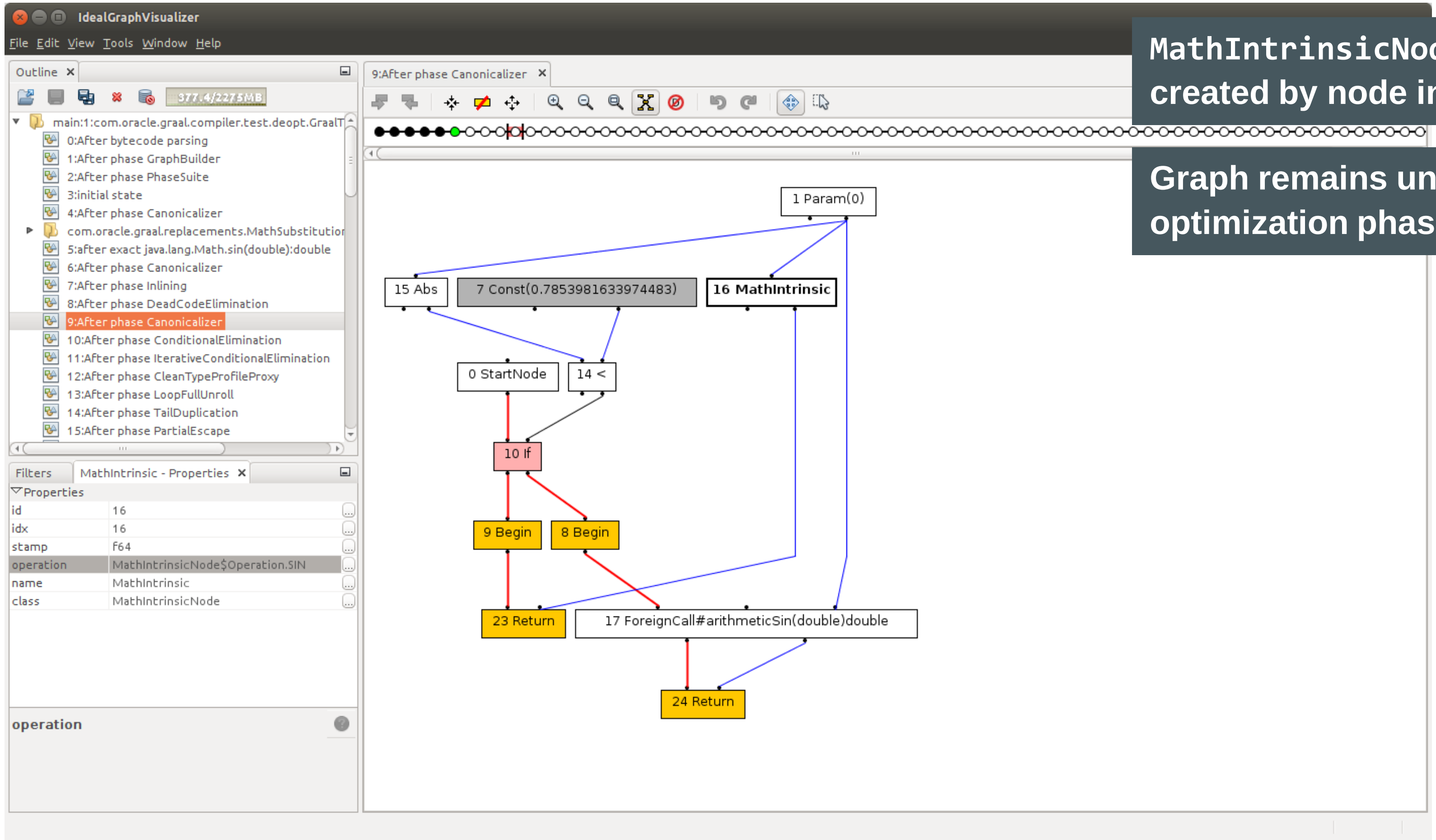
```
@ClassSubstitution(value = java.lang.Math.class)
public class MathSubstitutionsX86 {

  @MethodSubstitution(guard = UnsafeSubstitutions.GetAndSetGuard.class)
  public static double sin(double x) {
    if (abs(x) < PI_4) {
      return MathIntrinsicNode.compute(x, Operation.SIN);
    } else {
      return callDouble(ARITHMETIC_SIN, x);
    }
  }

  public static final ForeignCallDescriptor ARITHMETIC_SIN = new ForeignCallDescriptor("arithmeticSin", double.class, double.class);
}
```

Class that is substituted

The x86 instruction fsin can only be used for a small input values

Runtime call into the VM used for all other values

After Inlining the Substituted Method



`MathIntrinsicNode`, `AbsNode`, and `ForeignCallNode` are all created by node intrinsics

Graph remains unchanged throughout all further optimization phases

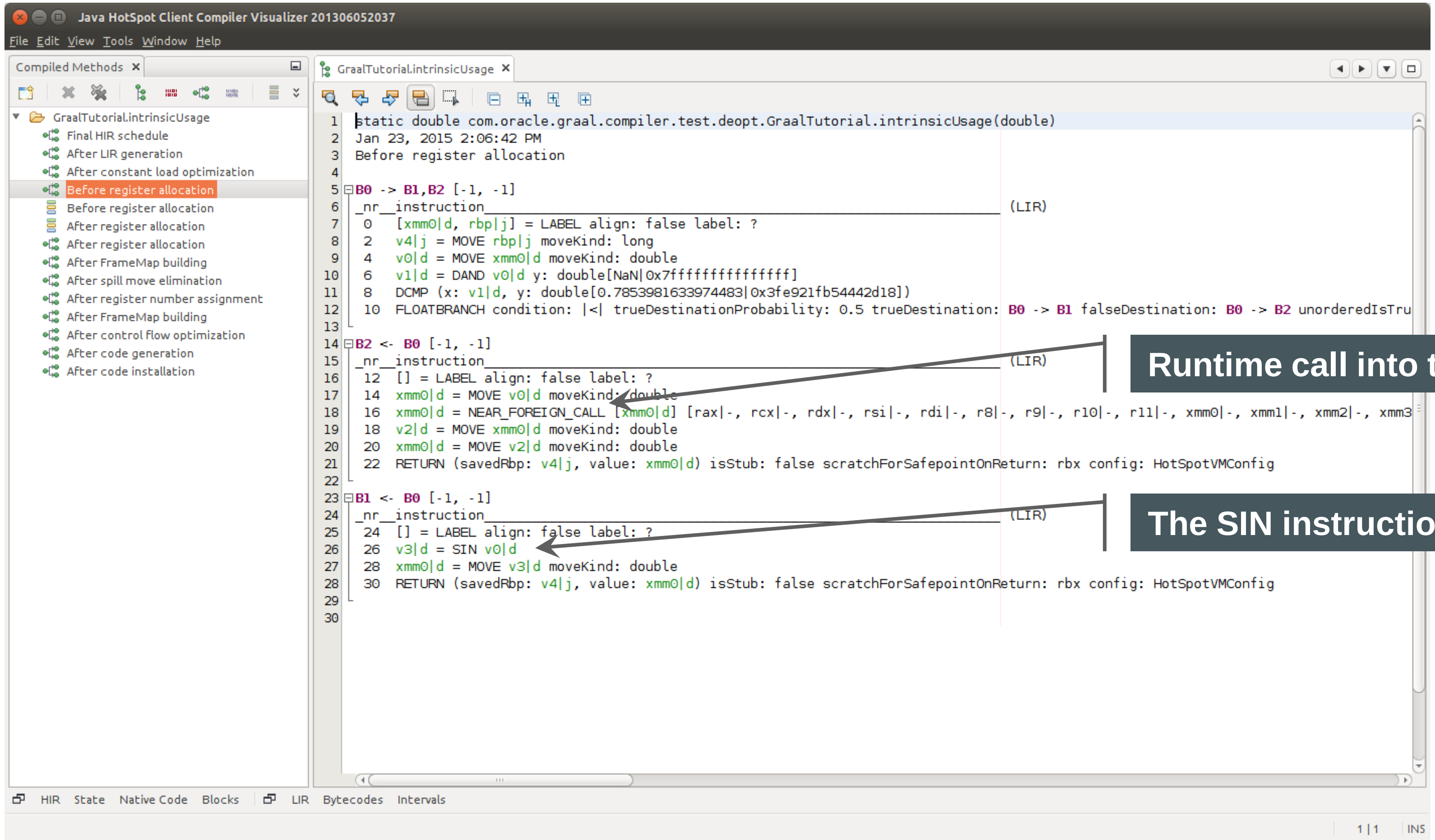
LIR Instruction

```
public class AMD64MathIntrinsicOp extends AMD64LIRInstruction {
  public enum IntrinsicOpcode  { SIN, COS, TAN, LOG, LOG10 }

  @Opcode private final IntrinsicOpcode opcode;
  @Def protected Value result;
  @Use protected Value input;

  public AMD64MathIntrinsicOp(IntrinsicOpcode opcode, Value result, Value input) {
    this.opcode = opcode;
    this.result = result;
    this.input = input;
  }

  @Override
  public void emitCode(CompilationResultBuilder crb, AMD64MacroAssembler masm) {
    switch (opcode) {
      case LOG:   masm.flog(asDoubleReg(result), asDoubleReg(input), false); break;
      case LOG10: masm.flog(asDoubleReg(result), asDoubleReg(input), true); break;
      case SIN:   masm.fsin(asDoubleReg(result), asDoubleReg(input)); break;
      case COS:   masm.fcos(asDoubleReg(result), asDoubleReg(input)); break;
      case TAN:   masm.ftan(asDoubleReg(result), asDoubleReg(input)); br
      default:    throw GraalInternalError.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



Runtime call into the VM (without JNI overhead)

The SIN instruction we are looking for

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

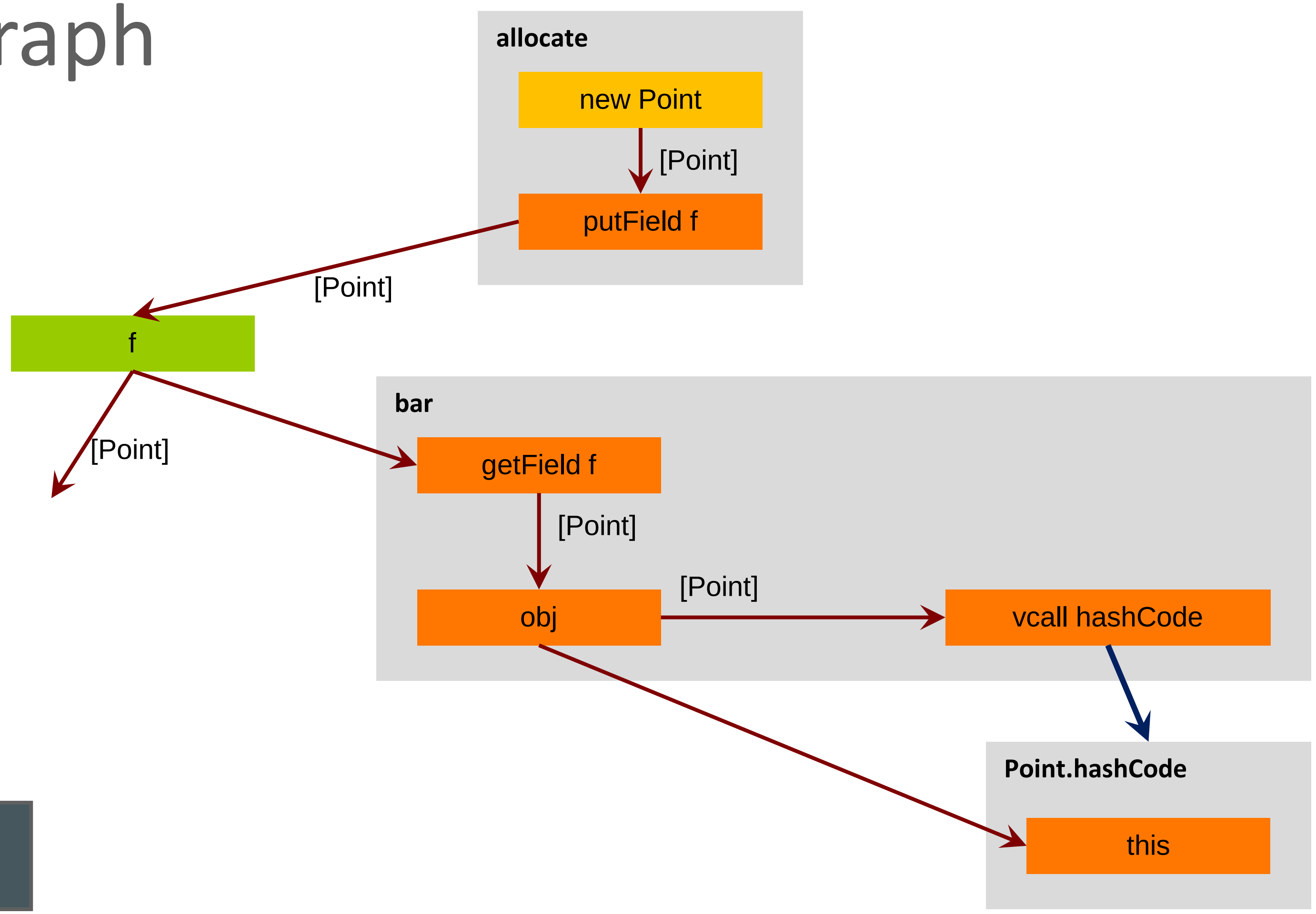
- Static analysis and compilation use same intermediate representation
  - Simplifies applying the static analysis results for optimizations

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

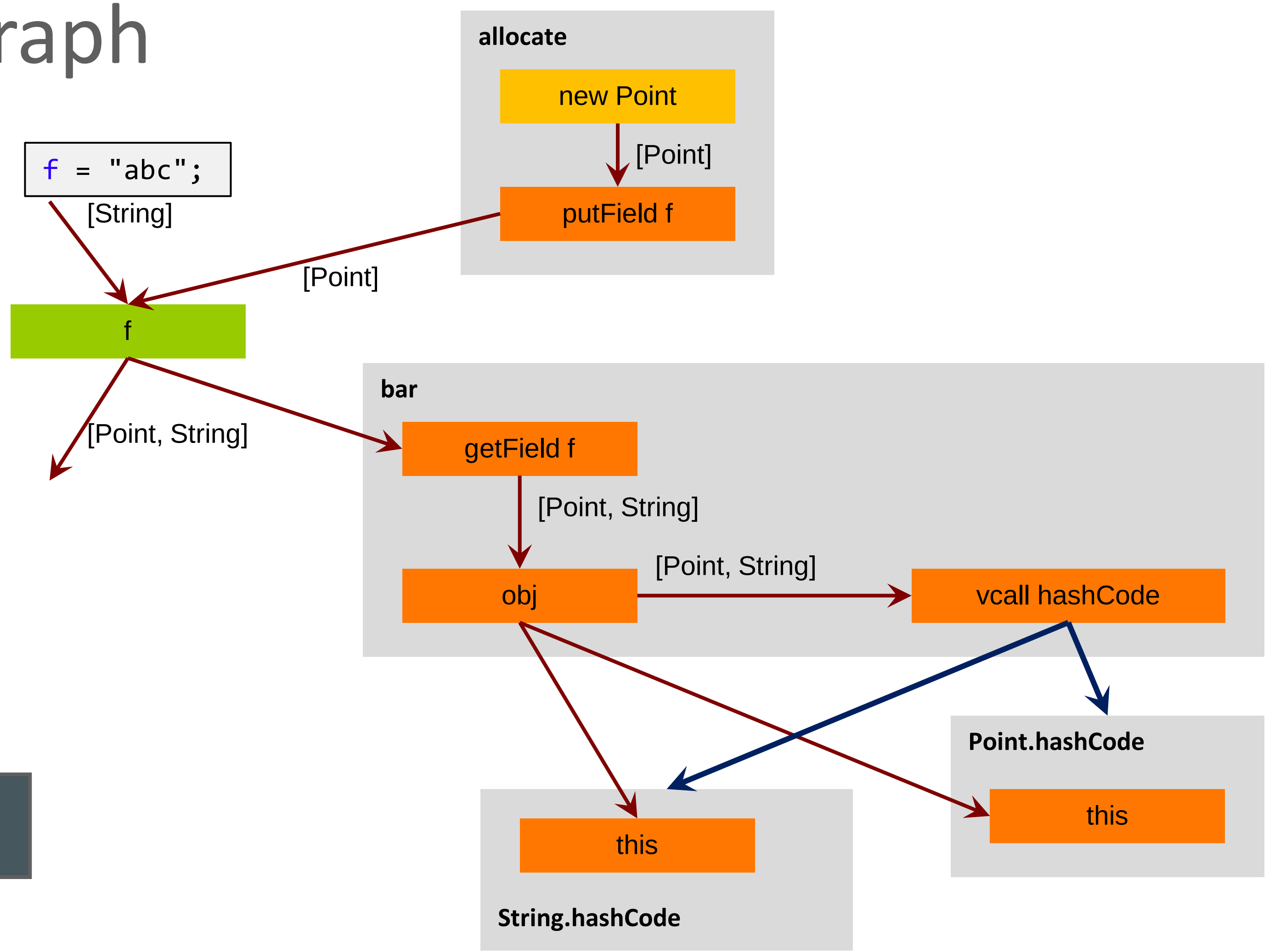
```
Object f;

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(method);

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

  GraphBuilderConfiguration config= GraphBuilderConfiguration.getEagerDefault();


  config = config.withOmitAllExceptionEdges(true);

  OptimisticOptimizations optOpts = OptimisticOptimizations.NONE;


  GraphBuilderPhase.Instance graphBuilder = new GraphBuilderPhase.Instance(metaAccess, config, optOpts);
  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

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

Substrate VM

Static Analysis and Ahead-of-Time Compilation using Graal

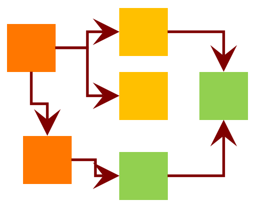
Static Analysis

Ahead-of-Time Compilation

Java Application

JDK

Substrate VM



Machine Code

Initial Heap

DWARF Info

ELF / MachO Binary

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

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.sh igv &
$ ./mx.sh unittest -G:Dump= -G:MethodFilter=String.hashCode GraalTutorial#testStringHashCode
```

ORACLE

Example: Custom Compilation

```
ResolvedJavaMethod method = ...
StructuredGraph graph = new StructuredGraph(method);
/* 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().createSuites();
/* The calling convention for the machine code. You should have a very g
 * before you switch to a different calling convention than the one that the VM provides by default. */
CallingConvention callingConvention = CodeUtil.getCallingConvention(codeCache, Type.JavaCallee, method, false);
/* We want Graal to perform all speculative optimisitic optimizations, using the
 * profiling information that comes with the method (collected by the interpreter) for speculation. */
OptimisticOptimizations optimisticOpts = OptimisticOptimizations.ALL;
ProfilingInfo profilingInfo = method.getProfilingInfo();
/* 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, callingConvention, method, providers, backend, target, null, graphBuilderSuite,
                           optimisticOpts, profilingInfo, null, suites, 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 = codeCache.addMethod(method, compilationResult, null, null);

/* Invoke the installed code with your arguments. */
installedCode.executeVarargs([...]);
```

You can manually construct Graal IR and compile it

Add your custom optimization phases to the suites

Truffle

A Language Implementation Framework that uses Graal for Custom Compilation

“Write Your Own Language”

Current situation	How it should be

Current situation

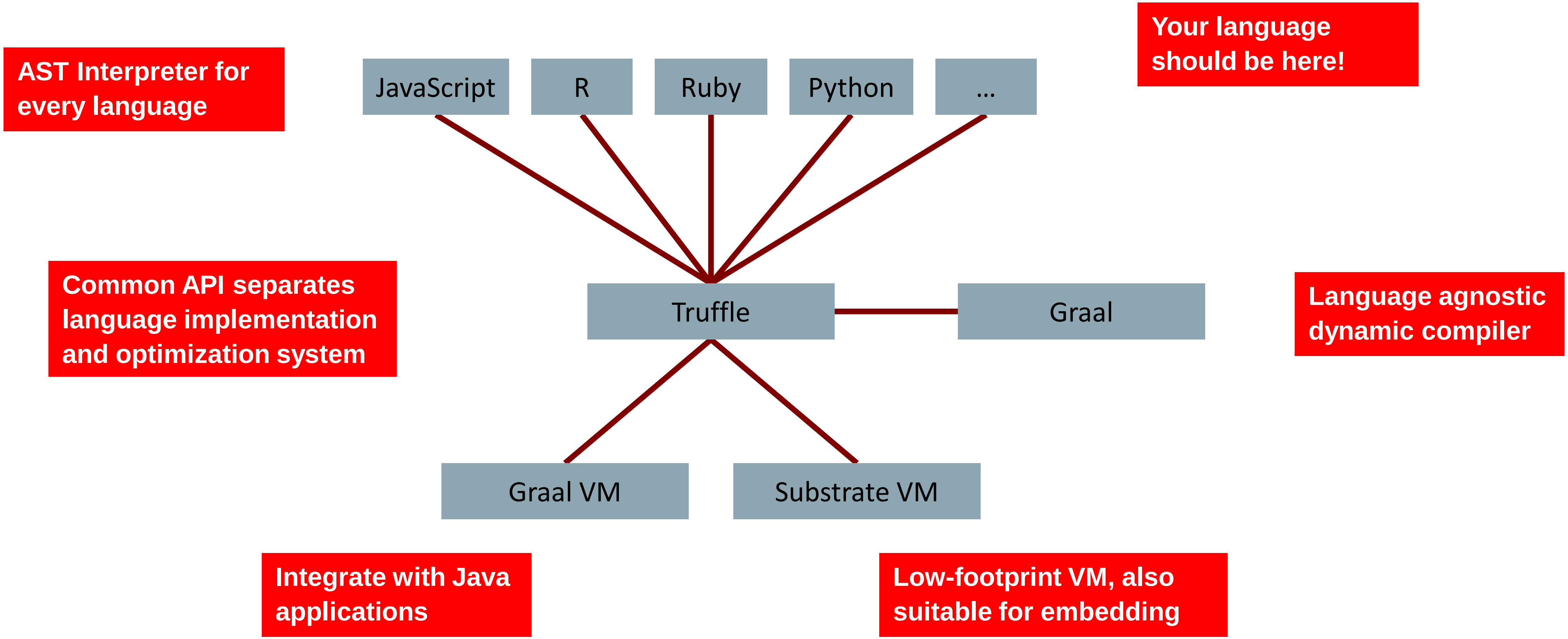
- Prototype a new language
  - Parser and language work to build syntax tree (AST), AST Interpreter
- Write a “real” VM
  - In C/C++, still using AST interpreter, spend a lot of time implementing runtime system, GC, …
- People start using it
- People complain about performance
  - Define a bytecode format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

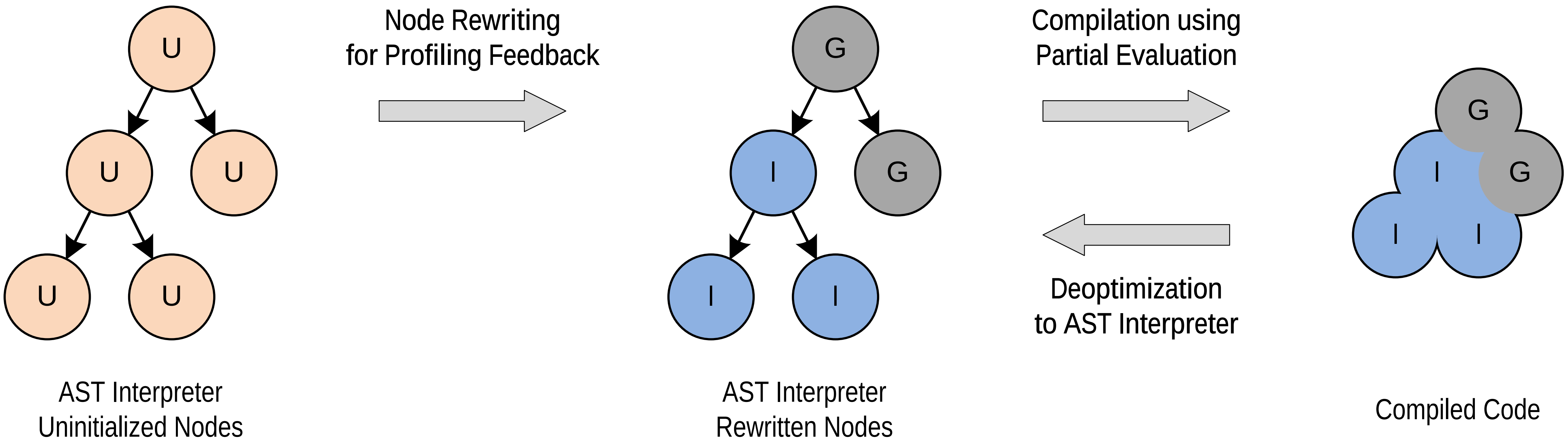
ORACLE

Truffle System Structure



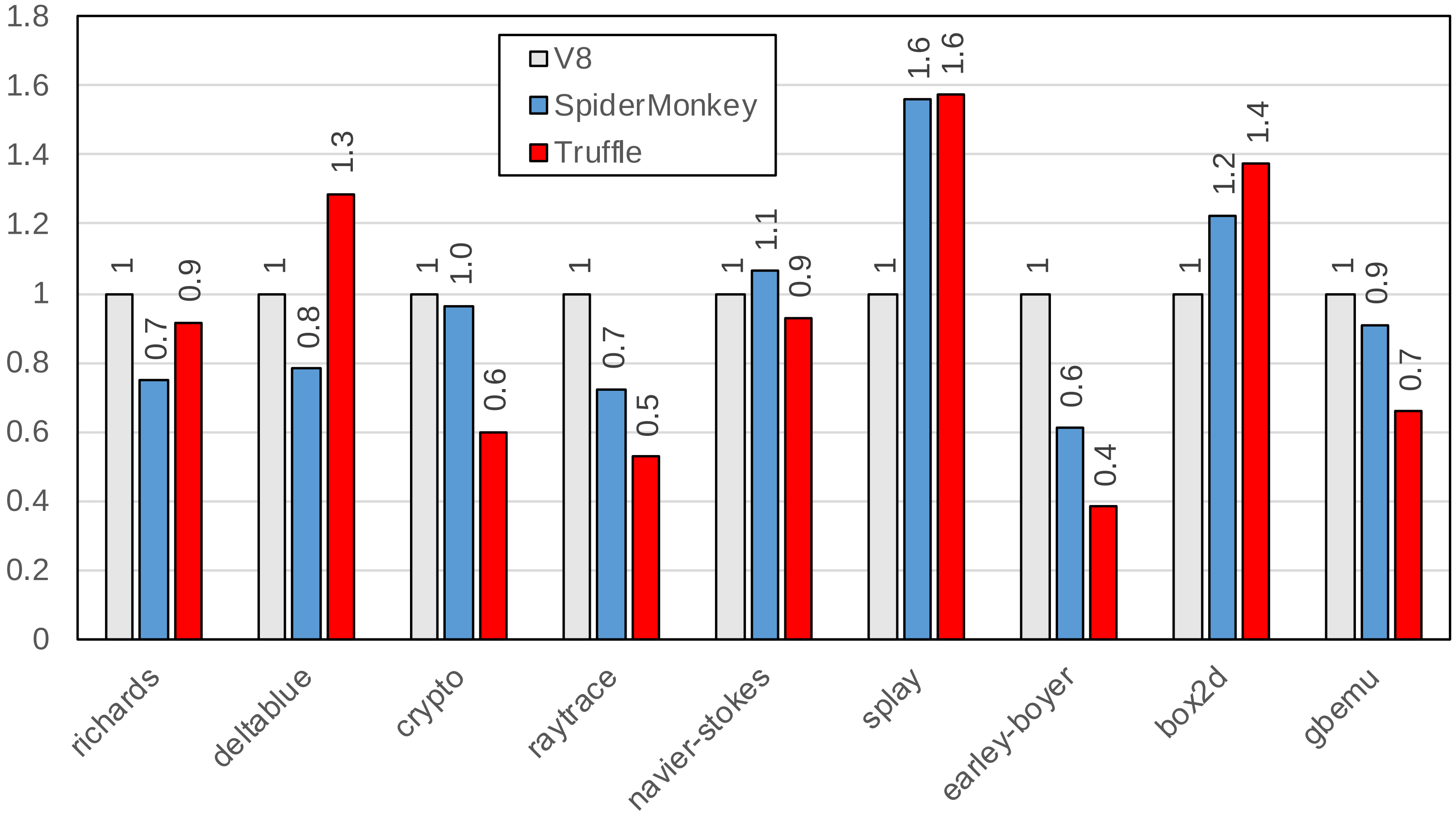
ORACLE

Truffle Approach



ORACLE

Performance: JavaScript



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`

ORACLE

Example: Visualize Truffle Compilation

SL source code:

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

Machine code for loop:

```
     ...
     movq      rcx, 0x0
     jmp       L2:
L1:  safepoint
     mov       rsi, rcx
     addq      rsi, 0x1
     jo        L3:
     mov       rcx, rsi
L2:  cmp       rax, rcx
     jnle      L1:
     ...
L3:  call      deoptimize
```

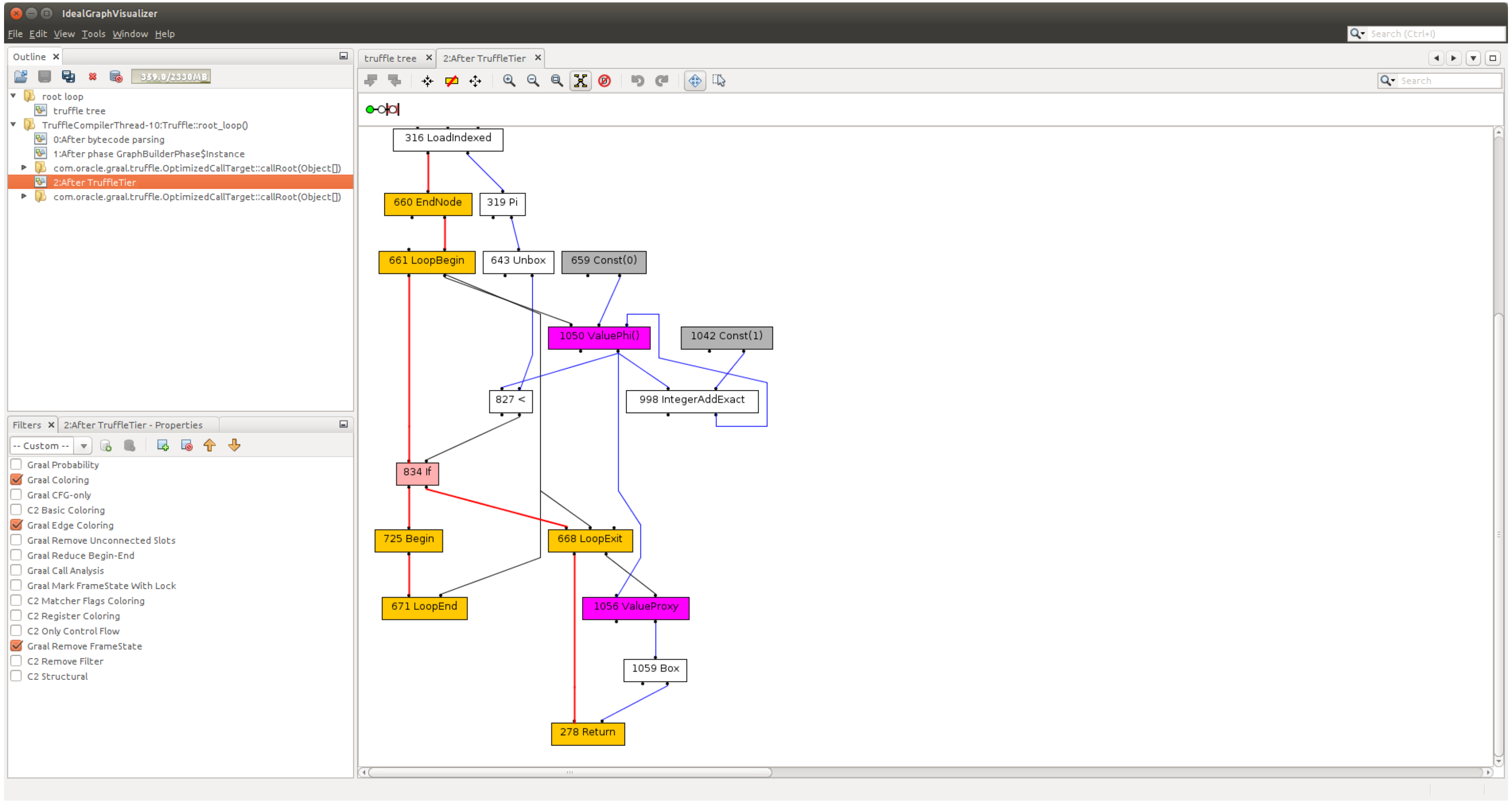
Run this example:

```
$ ./mx.sh igv &
$ ./mx.sh sl -G:Dump= -G:-TruffleBackgroundCompilation graal/com.oracle.truffle.sl.test/tests/LoopPrint.sl
```

-G:-TruffleBackgroundCompilation forces compilation in the main thread

-G:Dump= dumps compiled functions to IGV

Graal Graph of Simple Language Method



Summary

Your Usage of Graal?

http://openjdk.java.net/projects/graal/

graal-dev@openjdk.java.net

```
$ hg clone http://hg.openjdk.java.net/graal/graal
$ cd graal
$ ./mx build
$ ./mx ideinit
$ ./mx vm YourApplication
```

More Installation Instructions:
https://wiki.openjdk.java.net/display/Graal/Instructions

Graal License: GPLv2

ORACLE

Hardware and Software

Engineered to Work Together