One VM to Rule Them All

Christian Wimmer

VM Research Group, Oracle Labs
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.
Agenda

- Motivation and project overview
- Implementation of a Simple Language (SL)
  - Truffle nodes
  - Specializations using Truffle DSL
  - Function calls
  - Compilation
  - Object layout
- Additional benefits of Truffle
  - Substrate VM: low-footprint standalone executables
  - Tools: language-agnostic debugging infrastructure
One Language to Rule Them All?
Let’s ask a search engine…

JavaScript: One language to rule them all | VentureBeat
venturebeat.com/2011/.../javascript-one-language-to-rule-them-....
by Peter Yared - in 23 Google+ circles
Jul 29, 2011 - Why code in two different scripting languages, one on the client and one on the server? It's time for one language to rule them all. Peter Yared ...

[PDF] Python: One Script (Language) to rule them all - Ian Darwin
www.darwinsys.com/python/python4unix.pdf
Another Language? ▶ Python was invented in 1991 by Guido van. Rossum. ▶ Named after the comedy troupe, not the snake. ▶ Simple. ▶ They all say that!

Q & Stuff: One Language to Rule Them All - Java
qstuff.blogspot.com/2005/10/one-language-to-rule-them-all-java.html
Oct 10, 2005 - One Language to Rule Them All - Java. For a long time I'd been hoping to add a scripting language to LibQ, to use in any of my (or other ...

Dart: one language to rule them all - MixIT 2013 - Slideshare
fr.slideshare.net/sdeleuze/dart-mixit2013en
DartSébastien Deleuze - @sdeleuzeMix-IT 2013One language to rule them all ...
One Language to Rule Them All?
Let's ask Stack Overflow…

Stack Overflow is a question and answer site for professional and enthusiast programmers. It's 100% free, no registration required.

Why can't there be an “ultimate” programming language?

closed as not constructive by Tim, Bo Persson, Devon_C_Miller, Mark, Graviton Jan 17 at 5:58
Relative Speed of Programming Languages
From the Computer Language Benchmarks Game, ~1y ago

Goal:
One VM to for all languages means interoperability and being able to choose the best language for the task!
"Write Your Own Language"

**Current situation**

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

Language agnostic dynamic compiler

Common API between language implementation and optimization system

Integrate with Java applications

Low-footprint VM, also suitable for embedding

Your language here!

- JavaScript
- Ruby
- Python
- R
- ...
Speculate and Optimize …
... and Deoptimize and Reoptimize!

Deoptimization to AST Interpreter → Node Rewriting to Update Profiling Feedback → Recompilation using Partial Evaluation
One VM to Rule Them All

Thomas Würthinger*  Christian Wimmer*  Andreas Wöß†  Lukas Stadler†
Gilles Dubosq†  Christian Humer†  Gregor Richards§  Doug Simon*  Mario Wolczko*

*Oracle Labs  †Institute for System Software, Johannes Kepler University Linz, Austria  §S3 Lab, Purdue University
{thomas.wuerthinger, christian.wimmer, doug.simon, mario.wolczko}@oracle.com
{woess, stadler, dubosq, christian.humer}@ssw.jku.at  gr@purdue.edu

Abstract

Building high-performance virtual machines is a complex and expensive undertaking; many popular languages still have low-performance implementations. We describe a new approach to virtual machine (VM) construction that amortizes much of the effort in initial construction by allowing new languages to be implemented with modest additional effort. The approach relies on abstract syntax tree (AST) interpretation where a node can rewrite itself to a more specialized or more general node, together with an optimizing compiler that exploits the structure of the interpreter. The compiler uses speculative assumptions and deoptimization in order to produce efficient machine code. Our initial experience suggests that high performance is attainable while preserving a modular and layered architecture, and that new high-performance language implementations can be obtained by writing little more than a stylized interpreter.

as Microsoft’s Common Language Runtime, the VM of the .NET framework [43]. These implementations can be characterized in the following way:

- Their performance on typical applications is within a small integer multiple (1-3x) of the best statically compiled code for most equivalent programs written in an unsafe language such as C.
- They are usually written in an unsafe, systems programming language (C or C++).
- Their implementation is highly complex.
- They implement a single language, or provide a bytecode interface that preferentially advantages a narrow set of languages to the detriment of other languages.

In contrast, there are numerous languages that are popular, have been around for about 20 years, and yet still have
Truffle Language Projects
Languages that we are aware of

- Ruby
  - Oracle Labs, experimental part of JRuby
  - Open source: https://github.com/jruby/jruby

- R
  - JKU Linz, Purdue University, Oracle Labs
  - Open source: https://bitbucket.org/allr/fastr

- Python
  - ZipPy: UC Irvine
  - Open source: https://bitbucket.org/ssllab/zippy/

- JavaScript
  - JKU Linz, Oracle Labs

- SOM (Smalltalk)
  - Stefan Marr
  - Open source: https://github.com/smarr/TruffleSOM
Performance Disclaimers

- All Truffle numbers reflect the current development snapshot
  - Subject to change at any time (hopefully improve)
  - You have to know a benchmark to understand why it is slow or fast

- We are not claiming to have complete language implementations
  - JavaScript: passes 100% of ECMAScript standard tests
  - Ruby: passing >45% of RubySpec language tests
    - About as complete as Topaz
  - R: early prototype
  - Python: early prototype

- Benchmarks that are not shown
  - may not run at all, or
  - may not run fast
Performance: JavaScript

Speedup relative to V8

<table>
<thead>
<tr>
<th>Function</th>
<th>V8</th>
<th>SpiderMonkey</th>
<th>Truffle</th>
</tr>
</thead>
<tbody>
<tr>
<td>richards</td>
<td>1.0</td>
<td>0.7</td>
<td>0.9</td>
</tr>
<tr>
<td>deltablue</td>
<td>1.0</td>
<td>0.8</td>
<td>1.3</td>
</tr>
<tr>
<td>crypto</td>
<td>1.0</td>
<td>0.6</td>
<td>1.3</td>
</tr>
<tr>
<td>raytrace</td>
<td>1.0</td>
<td>0.5</td>
<td>1.0</td>
</tr>
<tr>
<td>navier-stokes</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>splay</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
</tr>
<tr>
<td>earley-boyer</td>
<td>1.0</td>
<td>0.6</td>
<td>0.4</td>
</tr>
<tr>
<td>box2d</td>
<td>1.2</td>
<td>1.4</td>
<td>1.4</td>
</tr>
<tr>
<td>gbemu</td>
<td>1.0</td>
<td>0.7</td>
<td>0.7</td>
</tr>
</tbody>
</table>
Performance: Ruby

Speedup relative to Ruby 2.1.0-p0

- jruby-1.7.9-server-indy
- rubytruffle-server
- jruby-head+truffle
- topaz-dev

Chart showing speedup relative to Ruby 2.1.0-p0 for various benchmarks.
Performance: R

Speedup compared to GNU R 3.0 Bytecode Interpreter
Performance: Python

Speedup relative to CPython 3.3
Acknowledgements

Oracle Labs
Danilo Ansaloni
Daniele Bonetta
Laurent Daynès
Erik Eckstein
Michael Haupt
Mick Jordan
Peter Kessler
Christos Kotselidis
David Leibs
Tom Rodriguez
Roland Schatz
Doug Simon
Lukas Stadler
Michael Van De Vanter
Christian Wimmer
Christian Wirth
Mario Wolczko
Thomas Würthinger
Laura Hill (Manager)

JKU Linz
Prof. Hanspeter Mössenböck
Gilles Duboscq
Matthias Grimmer
Christian Häubl
Josef Haider
Christian Humer
Christian Huber
Manuel Rigger
Bernhard Urban
Andreas Wöß

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

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

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

University of Manchester
Chris Seaton

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

Interns
Shams Imam
Stephen Kell
Gregor Richards
Rifat Shariyar

LaBRI
Floréal Morandat
Simple Language
SL: A Simple Language

- Language to demonstrate and showcase features of Truffle
  - Simple and clean implementation
  - Not the language for your next implementation project

- Language highlights
  - Dynamically typed
  - Strongly typed
    - No automatic type conversions
  - Arbitrary precision integer numbers
  - First class functions
  - Dynamic function redefinition

- Omitted language features
  - No floating point numbers
  - No object model: no run-time memory allocation
## Types

<table>
<thead>
<tr>
<th>SL Type</th>
<th>Values</th>
<th>Java Type in Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>Arbitrary precision integer numbers</td>
<td>long for values that fit within 64 bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>java.lang.BigInteger on overflow</td>
</tr>
<tr>
<td>Boolean</td>
<td>true, false</td>
<td>boolean</td>
</tr>
<tr>
<td>String</td>
<td>Unicode characters</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>Function</td>
<td>Reference to a function</td>
<td>SLFunction</td>
</tr>
<tr>
<td>Null</td>
<td>null</td>
<td>SLNull.SINGLETON</td>
</tr>
</tbody>
</table>

**Null is its own type because SL has no object type; could also be called "Undefined"**

**Best Practice: Use Java primitive types as much as possible to increase performance**

**Best Practice: Do not use the Java null value for the guest language null value**
Syntax

- C-like syntax
- Control flow statements
  - if, while, break, continue, return
- Operators
  - +, -, *, /, ==, !=, <, <=, >, >=, &&, ||, ( )
  - + is defined on String, performs String concatenation
  - && and || have short-circuit semantics
- Literals
  - Number, String, Function
- Builtin functions
  - println, readln: Standard I/O
  - nanoTime: to allow time measurements
  - defineFunction: dynamic function redefinition
Parsing

- Scanner and parser generated from grammar
  - Using Coco/R
  - Available from http://ssw.jku.at/coco/

- Refer to Coco/R documentation for details
  - This is not a tutorial about parsing

- Building a Truffle AST from a parse tree is usually simple

Best Practice: Use your favorite parser generator, or an existing parser for your language
SL Examples

Hello World:

```javascript
function main() {
    println("Hello World!");
}
```

Strings:

```javascript
function f(a, b) {
    return a + " < " + b + ": " + (a < b);
}

function main() {
    println(f(2, 4));
    println(f(2, "4"));
}
```

Simple loop:

```javascript
function main() {
    i = 0;
    sum = 0;
    while (i <= 10000) {
        sum = sum + i;
        i = i + 1;
    }
    return sum;
}
```

First class functions:

```javascript
function add(a, b) { return a + b; }
function sub(a, b) { return a - b; }

function foo(f) {
    println(f(40, 2));
}

function main() {
    foo(add);
    foo(sub);
}
```

Function definition and redefinition:

```javascript
function f(a, b) {
    return a + " < " + b + ": " + (a < b);
}

function main() {
    println(f(2, 4));
    println(f(2, "4"));
}
```

```javascript
function foo() { println(f(40, 2)); }

function main() {
    function f(a, b) { return a + b; }
    foo();

    function f(a, b) { return a - b; }
    foo();
}
```
Hello World:

```
function main() {
    println("Hello World!");
}
```

Strings:

```
function f(a, b) {
    return a + " < " + b + ": " + (a < b);
}
function main() {
    println(f(2, 4));
    println(f(2, "4"));
}
```

Simple loop:

```
function main() {
    i = 0;
    sum = 0;
    while (i <= 10000) {
        sum = sum + i;
        i = i + 1;
    }
    return sum;
}
```

First class functions:

```
function add(a, b) { return a + b; }
function sub(a, b) { return a - b; }

function foo(f) {
    println(f(40, 2));
}
function main() {
    foo(add);
    foo(sub);
}
```

Function definition and redefinition:

```
function foo() { println(f(40, 2)); }

function main() {
    defineFunction("function f(a, b) { return a + b; }" kl);
    foo();

    defineFunction("function f(a, b) { return a - b; }" kl);
    foo();
}
```

Strings:

```
Hello World!
2 < 4: true
Type error
```

```
50005000
```

```
42
38
```

```
42
38
```
Getting Started

Get and build the source code:

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

Run SL example program:

```bash
$ ./mx.sh sl graal/com.oracle.truffle.sl.test/tests/HelloWorld.sl
```

Generate Eclipse and NetBeans projects:

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

Import (at least) the following projects to work with SL:

![Package Explorer](image)

Use the "server" configuration when `mx` asks you

`mx` is our script to simplify building and execution
Simple Tree Nodes
Truffle Nodes and Trees

- **Class Node**: base class of all Truffle tree nodes
  - Management of parent and children
  - Replacement of this node with a (new) node
  - Copy a node
  - No execute() methods: define your own in subclasses

- **Class NodeUtil** provides useful utility methods

```java
public abstract class Node implements Cloneable {
    protected Node(SourceSection sourceSection) { ... }

    public final Node getParent() { ... }
    public final Iterable<Node> getChildren() { ... }

    protected final <T extends Node> T adoptChild(T newChild) { ... }
    protected final <T extends Node> T[] adoptChildren(T[] newChildren) { ... }

    public final <T extends Node> T replace(T newNode) { ... }
    public Node copy() { ... }
}
```
If Statement

```java
public final class SLIfNode extends SLStatementNode {
    @Child private SLEvaluationNode conditionNode;
    @Child private SLStatementNode thenPartNode;
    @Child private SLStatementNode elsePartNode;

    public SLIfNode(SLEvaluationNode conditionNode,
                    SLStatementNode thenPartNode, SLStatementNode elsePartNode) {
        this.conditionNode = adoptChild(conditionNode);
        this.thenPartNode = adoptChild(thenPartNode);
        this.elsePartNode = adoptChild(elsePartNode);
    }

    public void executeVoid(VirtualFrame frame) {
        if (conditionNode.executeBoolean(frame)) {
            thenPartNode.executeVoid(frame);
        } else {
            elsePartNode.executeVoid(frame);
        }
    }
}
```

**Rule:** A field for a child node must be annotated with `@Child` and must not be `final`.

**Rule:** `adoptChild()` must be called when assigning a child node field.
public final class SLBlockNode extends SLStatementNode {

    @Children private final SLStatementNode[] bodyNodes;

    public SLBlockNode(SLStatementNode[] bodyNodes) {
        this.bodyNodes = adoptChildren(bodyNodes);
    }

    @ExplodeLoop
    public void executeVoid(VirtualFrame frame) {
        for (SLStatementNode statement : bodyNodes) {
            statement.executeVoid(frame);
        }
    }
}

Rule: A field for multiple child nodes must be annotated with @Children and a final array

Rule: adoptChildren() must be called when assigning a children node field

Rule: The iteration of the children must be annotated with @ExplodeLoop
Best practice: Use Java exceptions for inter-node control flow

Rule: Exceptions used to model control flow extend ControlFlowException
Example for Inter-Node Control Flow

SL source code:

```java
function main() {
    foo();
    if (1 < 2) {
        bar();
        return 1;
    }
}
```

AST:

```java
SLFunctionBodyNode
  bodyNode = SLBlockNode
  bodyNodes[0] = SLInvokeNode
  bodyNodes[1] = SLIfNode
    conditionNode = ...
    thenPartNode = SLBlockNode
      bodyNodes[0] = SLInvokeNode
      bodyNodes[1] = SLReturnNode
      valueNode = SLLongLiteralNode
    elsePartNode = ...
```
Specialization and Node Rewriting
Addition: A First Try (1)

```java
public final class SLAddNode extends SLExpressionNode {
    @Child private SLExpressionNode leftNode;
    @Child private SLExpressionNode rightNode;

    @Override
    public Object executeGeneric(VirtualFrame frame) {
        Object left = leftNode.executeGeneric(frame);
        Object right = rightNode.executeGeneric(frame);

        if (left instanceof Long && right instanceof Long) {
            try {
                return ExactMath.addExact((Long) left, (Long) right);
            } catch (ArithmeticException ex) {
            }
        }

        if (left instanceof Long) {
            left = BigInteger.valueOf((Long) left);
        }
        if (right instanceof Long) {
            right = BigInteger.valueOf((Long) right);
        }
        if (left instanceof BigInteger && right instanceof BigInteger) {
            return ((BigInteger) left).add(((BigInteger) right));
        }

        if (left instanceof String || right instanceof String) {
            return left.toString() + right.toString();
        }

        throw new UnsupportedSpecializationException(this, ...);
    }
}
```

Warning: If you ever write such code for a Truffle node, you did something wrong!
Addition: A First Try (2)

Problems of the code on the previous slide

- Type checks and type casts
  - Many `instanceof` checks to select correct operation
- Boxing
  - Primitive `long` value is boxed into `Long`
  - Boxing consumes time and memory
- Complex control flow, including exception handling
  - Error prone code
- Slow
  - A lot of code to execute just to perform an addition

- Solution: node rewriting
  - Factor out every type into a separate node class
  - Truffle DSL generates the boilerplate code for you
Addition with Truffle DSL

```java
@NodeChildren({@NodeChild("leftNode"), @NodeChild("rightNode")})
public abstract class SLBinaryNode extends SLExpressionNode {
}

public abstract class SLAddNode extends SLBinaryNode {

    @Specialization(rewriteOn = ArithmeticException.class)
    protected final long add(long left, long right) {
        return ExactMath.addExact(left, right);
    }

    @Specialization
    protected final BigInteger add(BigInteger left, BigInteger right) {
        return left.add(right);
    }

    @Specialization(guards = "isString")
    protected final String add(Object left, Object right) {
        return left.toString() + right.toString();
    }

    protected final boolean isString(Object a, Object b) {
        return a instanceof String || b instanceof String;
    }
}

For all other specializations, guards are implicit based on method signature
```
Code Generated by Truffle DSL (1)

Generated class hierarchy:

The parser creates a `SLAddUninitializedNode`, using the `SLAddNodeFactory`

SLAddPolymorphicNode is a performance optimization

Generated factory class:

```java
@GeneratedBy(SLAddNode.class)
public final class SLAddNodeFactory implements NodeFactory<SLAddNode> {

    public SLAddNode createNode(Object... arguments) { ... }

    public static SLAddNode create(SLEvaluationContext context, Object leftNode, SLEvaluationContext rightNode) { ... }

    ...
}
```
Type Transitions vs. Node Rewriting

Possible type transitions:

Node rewriting for addition:

The linearization of the type lattice simplifies the node rewriting logic.
private final class SLAddLongNode extends SLAddBaseNode {
    public long executeLong(VirtualFrame frameValue) throws UnexpectedResultException {
        long leftNodeValue;
        try {
            leftNodeValue = leftNode.executeLong(frameValue);
        } catch (UnexpectedResultException ex) {
            Object rightNodeValue = rightNode.executeGeneric(frameValue);
            return SLTYPES.expectLong(executeAndSpecialize0(1, frameValue, ex.getResult(), rightNodeValue));
        }
        long rightNodeValue;
        try {
            rightNodeValue = this.rightNode.executeLong(frameValue);
        } catch (UnexpectedResultException ex) {
            return SLTYPES.expectLong(executeAndSpecialize0(1, frameValue, leftNodeValue, ex.getResult()));
        }
        try {
            return super.add(leftNodeValue, rightNodeValue);
        } catch (ArithmeticException ex) {
            return SLTYPES.expectLong(executeAndSpecialize0(1, frameValue, leftNodeValue, rightNodeValue));
        }
    }

    public Object executeGeneric(VirtualFrame frameValue) {
        try {
            return executeLong(frameValue);
        } catch (UnexpectedResultException ex) {
            return ex.getResult();
        }
    }
}
Type System Definition in Truffle DSL

@TypeSystem({"long.class", BigInteger.class, boolean.class, String.class, SLFunction.class, SLNull.class})
public abstract class SLTypes {
    @ImplicitCast
    public BigInteger castBigInteger(long value) {
        return BigInteger.valueOf(value);
    }
}

@TypeSystemReference(SLTypes.class)
public abstract class SLExpressionNode extends SLStatementNode {
    public abstract Object executeGeneric(VirtualFrame frame);
    public long executeLong(VirtualFrame frame) throws UnexpectedResultException {
        return SLTypesGen.SLTYPES.expectLong(executeGeneric(frame));
    }
    public BigInteger executeBigInteger(VirtualFrame frame) ...
    public boolean executeBoolean(VirtualFrame frame) ...
    public String executeString(VirtualFrame frame) ...
    public SLFunction executeFunction(VirtualFrame frame) ...
    public SLNull executeNull(VirtualFrame frame) ...
}

Rule: One execute() method per type, in addition to the abstract executeGeneric() method
**UnexpectedResultException**

- Type-specialized `execute()` methods have specialized return type
  - Allows primitive return types, to avoid boxing
  - Allows to use the result without type casts
  - Speculation types are stable and the specialization fits

- But what to do when speculation was too optimistic?
  - Need to return a value with a type more general than the return type
  - Solution: return the value "boxed" in an `UnexpectedResultException`

- Exception handler performs node rewriting
  - Exception is thrown only once, so no performance bottleneck
Source Position Information

- Every node can specify a pointer to a source location
  - For error messages, tool support, and sometime language features
  - Created by the parser

- Source
  - A file, or any other source of code

- SourceManager
  - Keeps all sources in one place
  - Allows, e.g., lookup by file name

- SourceSection
  - A range of characters in a Source

- The Source of a node is preserved during node rewriting
Frames and Local Variables
Frame Layout

- In the interpreter, a frame is an object on the heap
  - Allocated in the function prologue
  - Passed around as parameter to execute() methods
- The compiler eliminates the allocation
  - No object allocation and object access
  - Guest language local variables have the same performance as Java local variables
- FrameDescriptor: describes the layout of a frame
  - A mapping from identifiers (usually variable names) to typed slots
  - Every slot has a unique index into the frame object
  - Created and filled during parsing
- Frame
  - Created for every invoked guest language function
- Arguments
  - Data that is passed around between functions
  - Allocated in caller, accessed in callee function
  - Language implementations have their own subclasses
Frame Management

- Truffle API only exposes frame interfaces
  - Implementation class depends on the optimizing system

- VirtualFrame
  - What you usually use: automatically optimized by the compiler
  - Must never be assigned to a field, or escape out of an interpreted function

- MaterializedFrame
  - A frame that can be stored without restrictions
  - Example: frame of a closure that needs to be passed to other function

- PackedFrame
  - A handle to a VirtualFrame

- Allocation of frames
  - Factory methods in the class TruffleRuntime
Frames support all Java primitive types, and Object

SL types String, SLFunction, and SLNull are stored as Object in the frame

Rule: Never allocate frames yourself, and never make your own frame implementations
Local Variables

@NodeField(name = "slot", type = FrameSlot.class)
public abstract class SLReadLocalVariableNode extends SLExpressionNode {

    protected abstract FrameSlot getSlot();

    @Specialization(rewriteOn = FrameSlotTypeException.class)
    protected final long readLong(VirtualFrame frame) throws FrameSlotTypeException {
        return frame.getLong(getSlot());
    }

    ...

    @Specialization(order = 1, rewriteOn = FrameSlotTypeException.class)
    protected final Object readObject(VirtualFrame frame) throws FrameSlotTypeException {
        return frame.getObject(getSlot());
    }

    @Specialization(order = 2)
    protected final Object read(VirtualFrame frame) {
        return frame.getValue(getSlot());
    }
}
Local Variables

```java
@NodeChild("valueNode")
@NodeField(name = "slot", type = FrameSlot.class)
public abstract class SLWriteLocalVariableNode extends SLExpressionNode {
    protected abstract FrameSlot getSlot();

    @Specialization(guards = "isLongKind")
    protected final long write(VirtualFrame frame, long value) {
        frame.setLong(getSlot(), value);
        return value;
    }

    @Specialization
    protected final Object write(VirtualFrame frame, Object value) {
        if (getSlot().getKind() != FrameSlotKind.Object) {
            getSlot().setKind(FrameSlotKind.Object);
        }
        frame.setObject(getSlot(), value);
        return value;
    }

    protected final boolean isLongKind() {
        return isKind(FrameSlotKind.Long);
    }

    private boolean isKind(FrameSlotKind kind) {
        if (getSlot().getKind() == kind) {
            return true;
        } else if (getSlot().getKind() == FrameSlotKind.Illegal) {
            getSlot().setKind(kind);
            return true;
        } else {
            return false;
        }
    }
}
```
Frame Management

Runtime provides optimized frame implementations

Function Call

Call Graph

VirtualFrame

pack()

PackedFrame

Arguments

VirtualFrame

Allocate runtime-specific frame object

Truffle Framework code

VirtualFrame

Language implementation code

VirtualFrame

materialize()

VirtualFrame

pack()

PackedFrame

Arguments

VirtualFrame

getCaller()

VirtualFrame

unpack()

VirtualFrame

getcharler(

VirtualFrame

unpack()

VirtualFrame

Stack Walking

Function AST Interpretation

Function Call

Runtime provides optimized frame implementations

Pass, e.g., to closure
Compilation
Compilation

- Automatic partial evaluation of AST
  - Automatically triggered by function execution count

- Compilation assumes that the AST is stable
  - All @Child and @Children fields treated like final fields

- Later node rewriting invalidates the machine code
  - Transfer back to the interpreter: "Deoptimization"
  - Complex logic for node rewriting not part of compiled code
  - Essential for excellent peak performance

- Compiler optimizations eliminate the interpreter overhead
  - No more dispatch between nodes
  - No more allocation of VirtualFrame objects
  - No more exceptions for inter-node control flow
Compilation

SL source code:

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

Machine code for loop:

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

Run this example:

```
./mx.sh sl -G:-TruffleBackgroundCompilation
graal/com.oracle.truffle.sl.test/tests/LoopPrint.sl
```

-G:-TruffleBackgroundCompilation forces compilation in the main execution thread

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

Add the flag -G:Dump= to dump compiled functions to IGV
Visualization Tools: IGV
Visualization Tools: IGV
Function Calls
Polymorphic Inline Caches

- Function lookups are expensive
  - At least in a real language, in SL lookups are only a few field loads
- Checking whether a function is the correct one is cheap
  - Always a single comparison

- Inline Cache
  - Cache the result of the previous lookup and check if it still correct

- Polymorphic Inline Cache
  - Cache multiple previous lookups, up to a certain limit

- Inline cache miss needs to perform the slow lookup

- Implementation using tree rewriting
  - One node per cached value
  - Build chain of multiple cache nodes
Polymorphic Inline Cache

Example of cache with length 2

After Parsing 1 Function 2 Functions >2 Functions

SLInvokeNode -> SLUninitializedDispatch
function arguments

SLInvokeNode -> SLDirectDispatch

SLInvokeNode -> SLDirectDispatch

SLInvokeNode -> SLGenericDispatch
public final class SLInvokeNode extends SLExpressionNode {

    @Child protected SLExpressionNode functionNode;
    @Children protected final SLExpressionNode[] argumentNodes;
    @Child protected SLAbstractDispatchNode dispatchNode;

    @ExplodeLoop
    public Object executeGeneric(VirtualFrame frame) {
        SLFunction function = functionNode.executeFunction(frame);

        Object[] argumentValues = new Object[argumentNodes.length];
        for (int i = 0; i < argumentNodes.length; i++) {
            argumentValues[i] = argumentNodes[i].executeGeneric(frame);
        }
        SLArguments arguments = new SLArguments(argumentValues);

        return dispatchNode.executeDispatch(frame, function, arguments);
    }
}
Uninitialized Dispatch Node

```java
final class SLUninitializedDispatchNode extends SLAbstractDispatchNode {

    protected Object executeDispatch(VirtualFrame frame,
          SLFunction function, SLArguments arguments) {

        int depth = ...
        SLInvokeNode invokeNode = ...

        SLAbstractDispatchNode replacement;
        if (depth < INLINE_CACHE_SIZE) {
            SLAbstractDispatchNode next = new SLUninitializedDispatchNode();
            replacement = new SLDirectDispatchNode(next, function);
            replace(replacement);
        } else {
            replacement = new SLGenericDispatchNode();
            invokeNode.dispatchNode.replace(replacement);
        }

        return replacement.executeDispatch(frame, function, arguments);
    }
}
```

Separation of concerns: this node builds the inline cache chain
Inline Cache Node

```java
final class SLDirectDispatchNode extends SLAbstractDispatchNode {

    private final SLFunction cachedFunction;
    @Child private CallNode callCachedTargetNode;
    @Child private SLAbstractDispatchNode nextNode;

    protected SLDirectDispatchNode(SLAbstractDispatchNode next, SLFunction cachedFunction) {
        this.cachedFunction = cachedFunction;
        this.callCachedTargetNode = adoptChild(...);
        this.nextNode = adoptChild(next);
    }

    protected Object executeDispatch(VirtualFrame frame, SLFunction function, SLArguments arguments) {
        if (this.cachedFunction == function) {
            return callCachedTargetNode.call(frame.pack(), arguments);
        } else {
            return nextNode.executeDispatch(frame, function, arguments);
        }
    }
}

Separation of concerns: this node performs the inline cache check and optimized dispatch
```

Rule: the cachedFunction must be a final field
Generic Dispatch Node

```java
final class SLGenericDispatchNode extends SLAbstractDispatchNode {

    protected Object executeDispatch(VirtualFrame frame,
        SLFunction function, SLArguments arguments) {

        return function.getCallTarget().call(frame.pack(), arguments);
    }
}
```

Separation of concerns: this is the always succeeding, but slow, fallback node
Example: Function Calls (1)

Stack trace of a function call

Truffle framework code triggers compilation, function inlining, …
Truffle framework code triggers compilation, function inlining, …
Function Arguments

- Function arguments are not type-specialized
  - Passed in `Object[]` array
- Function prologue writes them to local variables
  - `SLReadArgumentNode` in the function prologue
  - Local variable accesses are type-specialized, so only one unboxing

Example SL code:

```javascript
function add(a, b) {
    return a + b;
}

function main() {
    add(2, 3);
}
```

Specialized AST for function `add()`:

```javascript
SLRootNode
    bodyNode = SLFunctionBodyNode
        bodyNode = SLBlockNode
            bodyNodes[0] = SLWriteLocalVariableLongNode(name = "a")
                valueNode = SLReadArgumentNode(index = 0)
            bodyNodes[1] = SLWriteLocalVariableLongNode(name = "b")
                valueNode = SLReadArgumentNode(index = 1)
            bodyNodes[2] = SLReturnNode
                valueNode = SLAddLongNode
                    leftNode = SLReadLocalVariableLongNode(name = "a")
                    rightNode = SLReadLocalVariableLongNode(name = "b")
```
Function Inlining

- Function inlining is one of the most important optimizations
  - Replace a call with a copy of the callee

- Function inlining in Truffle operates on the AST level
  - The whole node tree of a function is duplicated
  - Call node is replaced with the root of the duplicated tree

- Benefits
  - Inlined tree is specialized separately
    - Result: context sensitive profiling information
  - All later optimizations see the big combined tree, without further work
    - Partial evaluation operates on the inlined tree

- Language-independent inlining logic in class CallNode
  - SLRootNode overrides methods that provide the AST copy that is inlined
  - SLRootNode keeps copy of the uninitialized (= non-specialized) AST
Function Inlining (1)

Frequently executed call
Function Inlining (2)
Function Inlining Nodes

```java
final class SLDirectDispatchNode extends SLAbstractDispatchNode {
    protected SLDirectDispatchNode(... SLFunction cachedFunction) {
        this.callCachedTargetNode = adoptChild(
            CallNode.create(cachedFunction.getCallTarget()));
        ...
    }
}

public final class SLRootNode extends RootNode {
    @Child private SLExpressionNode bodyNode;
    private final SLExpressionNode uninitializedBodyNode;

    public SLRootNode(... SLExpressionNode bodyNode) {
        this.uninitializedBodyNode = NodeUtil.cloneNode(bodyNode);
        this.bodyNode = adoptChild(bodyNode);
    }

    public RootNode inline() {
        return new SLRootNode(... NodeUtil.cloneNode(uninitializedBodyNode));
    }
    public int getInlineNodeCount() {
        return NodeUtil.countNodes(uninitializedBodyNode);
    }
    public boolean isInlinable() {
        return true;
    }
}
```
Function Inlining Example

SL source code:

```plaintext
function add(a, b) {
    return a + b;
}
function foo() {
    add(1, 2);
    add("x", "y") {

```

AST before function inlining:

function `add()` called by both calls

```
SLReturnNode
    value = SLAddGenericNode
    left = SLReadLocalVariableObjectNode
    right = SLReadLocalVariableObjectNode
```

AST after function inlining:

function `add()` inlined for first call

```
SLReturnNode
    value = SLAddLongNode
    left = SLReadLocalVariableLongNode
    right = SLReadLocalVariableLongNode
```

function `add()` inlined for second call

```
SLReturnNode
    value = SLAddStringNode
    left = SLReadLocalVariableObjectNode
    right = SLReadLocalVariableObjectNode
```
Compilation with Inlined Function

SL source code without call:

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

SL source code with call:

```javascript
function add(a, b) {
  return a + b;
}

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

Machine code for loop without call:

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

Machine code for loop with inlined call:

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

- Default behavior of compilation: Inline all reachable Java methods
- Truffle API provides class CompilerDirectives to influence compilation
  - @CompilationFinal
    - Treat a field as final during compilation
  - transferToInterpreter()
    - Never compile part of a Java method
  - transferToInterpreterAndInvalidate()
    - Invalidate machine code when reached
    - Implicitly done by Node.replace()
  - @SlowPath
    - Hint that this method is not important for performance, i.e., hint to not inline it
  - inInterpreter()
    - For profiling code that runs only in the interpreter
  - Assumption
    - Invalidate machine code from outside
    - Avoid checking a condition over and over in compiled code
Guards

```java
public final class BranchProfile {
    @CompilationFinal private boolean visited;

    public void enter() {
        if (!visited) {
            CompilerDirectives.transferToInterpreterAndInvalidate();
            visited = true;
        }
    }
}
```

```java
public final class SLIfNode extends SLStatementNode {
    private final BranchProfile thenTaken = new BranchProfile();
    private final BranchProfile elseTaken = new BranchProfile();

    public void executeVoid(VirtualFrame frame) {
        if (conditionNode.executeBoolean(frame)) {
            thenTaken.enter();
            thenPartNode.executeVoid(frame);
        } else {
            elseTaken.enter();
            elsePartNode.executeVoid(frame);
        }
    }
}
```

**Best practice:** Profiling in the interpreter allows the compiler to generate better code.

*transferToInterpreter*() does nothing when running in interpreter.
Slow Path Annotation

```java
public abstract class SLPrintlnBuiltin extends SLBuiltinNode {

    @Specialization
    public final Object println(Object value) {
        doPrint(getContext().getOutput(), value);
        return value;
    }

    @SlowPath
    private static void doPrint(PrintStream out, Object value) {
        out.println(value);
    }
}
```

Why @SlowPath? Inlining something as big as `println()` would lead to code explosion.

When compiling, the output stream is a constant.
Function Redefinition (1)

- **Problem**
  - In SL, functions can be redefined at any time
  - This invalidates optimized call dispatch, and function inlining
  - Checking for redefinition before each call would be a huge overhead

- **Solution**
  - Every `SLFunction` has an `Assumption`
  - Assumption is invalidated when the function is redefined
    - This invalidates optimized machine code

- **Result**
  - No overhead when calling a function
Assumptions

Create an assumption:

```java
Assumption assumption = Truffle.getRuntime().createAssumption();
```

Check an assumption:

```java
void foo() {
    assumption.check();
    // Some code that is only valid if assumption is true.
}
```

Respond to an invalidated assumption:

```java
void bar() {
    try {
        foo();
    } catch (InvalidAssumptionException ex) {
        // Perform node rewriting, or other slow-path code to respond to change.
    }
}
```

Invalidate an assumption:

```java
assumption.invalidate();
```
public abstract class SLDefineFunctionBuiltin extends SLBuiltinNode {

    @Specialization
    public final String defineFunction(String code) {
        doDefineFunction(getContext(), code);
        return code;
    }

    @SlowPath
    private static void doDefineFunction(SLContext context, String code) {
        Source source = context.getSourceManager().get("[defineFunction]", code);
        Parser.parseSL(context, source);
    }
}

SL semantics: Functions can be defined and redefined at any time

Why @SlowPath? Inlining something as big as the parser would lead to code explosion
public final class SLFunction {

    private RootCallTarget callTarget;
    private Assumption callTargetStable;

    protected void setCallTarget(RootCallTarget callTarget) {
        this.callTarget = callTarget;

        if (callTargetStable != null) {
            callTargetStable.invalidate();
        }
        callTargetStable = Truffle.getRuntime().createAssumption(name);
    }

    public RootCallTarget getCallTarget() {
        return callTarget;
    }

    public Assumption getCallTargetStable() {
        return callTargetStable;
    }
}

The utility class CyclicAssumption simplifies this code
Function Redefinition (4)

```java
final class SLDirectDispatchNode extends SLAbstractDispatchNode {

    private final SLFunction cachedFunction;
    @Child private CallNode callCachedTargetNode;
    private final Assumption cachedTargetStable;

    protected SLDirectDispatchNode(... SLFunction cachedFunction) {
        this.cachedFunction = cachedFunction;
        this.callCachedTargetNode = adoptChild(...);
        this.cachedTargetStable = cachedFunction.getCallTargetStable();
    }

    protected Object executeDispatch(VirtualFrame frame, SLFunction function, SLArguments arguments) {
        if (this.cachedFunction == function) {
            try {
                cachedTargetStable.check();
                return callCachedTargetNode.call(frame.pack(), arguments);
            }
            catch (InvalidAssumptionException ex) {
                replace(nextNode);
            }
        }
        return nextNode.executeDispatch(frame, function, arguments);
    }
}
```

No compiled code for check()

Exception handler is not compiled
public final class SLWhileNode extends SLStatementNode {
    @Child private SLExpressionNode conditionNode;
    @Child private SLStatementNode bodyNode;
    private final BranchProfile continueTaken = new BranchProfile();
    private final BranchProfile breakTaken = new BranchProfile();

    public void executeVoid(VirtualFrame frame) {
        int count = 0;
        try {
            while (conditionNode.executeBoolean(frame)) {
                try {
                    bodyNode.executeVoid(frame);
                    if (CompilerDirectives.inInterpreter()) {
                        count++;
                    }
                } catch (SLContinueException ex) {
                    continueTaken.enter();
                }
                catch (SLBreakException ex) {
                    breakTaken.enter();
                }
            }
            finally {
                if (CompilerDirectives.inInterpreter()) {
                    getRootNode().reportLoopCount(count);
                }
            }
        } finally {
            if (CompilerDirectives.inInterpreter()) {
                getRootNode().reportLoopCount(count);
            }
        }
    }
}

Best practice: Profiling in the interpreter allows the compiler to generate better code
Compilation and function inlining heuristics of Truffle use the loop count
Compiler Assertions

- You work hard to help the compiler
- How do you check that you succeeded?

- `CompilerAsserts.compilationConstant()`
  - Checks that the passed in value is a compile-time constant
  - Compiler fails with a compilation error if the value is not a constant
  - When the assertion holds, no code is generated to produce the value

- `CompilerAsserts.neverPartOfCompilation()`
  - Checks that this code is never reached in a compiled method
  - Compiler fails with a compilation error if code is reachable
  - Useful at the beginning of helper methods that are big or rewrite nodes
  - All code dominated by the assertion is never compiled

- Assertions are checked after aggressive compiler optimizations
  - Method inlining, constant folding, dead code elimination, escape analysis, …
Trace the Compilation (1)

```
./mx.sh sl -G:-TruffleBackgroundCompilation -G:+TraceTruffleExpansion
graal/com.oracle.truffle.sl.test/tests/LoopCall.sl
```

```
== running on Graal Truffle Runtime
OptimizedCallTarget.executeHelper
  RootCallTarget<OptimizedCallTarget>.getRootNode
RootNode<SLRootNode>.getFrameDescriptor
RootCallTarget<OptimizedCallTarget>.getRootNode
SLRootNode.execute
  SLFunctionBodyNode.executeGeneric
    SLBlockNode.executeVoid
      SLExpressionNode<SLWriteLocalVariableLongNode>.executeVoid
        SLExpressionNode<SLReadArgumentNode>.executeLong
        SLExpressionNode<SLWriteLocalVariableLongNode>.executeVoid
          SLExpressionNode<SLReadArgumentNode>.executeLong
          SLExpressionNode<SLWriteLocalVariableLongNode>.executeVoid
          SLExpressionNode<SLReadArgumentNode>.executeLong
          SLReturnNode.executeVoid
            SLAddLongNode.executeGeneric
              SLReadLocalVariableLongNode.executeLong
              SLReadLocalVariableLongNode.executeLong
[truffle] optimized root add
77533eef |Nodes 11 |Time 529( 289+240 )ms |Nodes 34/ 130 |CodeSize 456
```

Useful to start finding performance problems, if you do not really have a clue what is wrong. You can then look at the tree, method by method, see what code is expanded, and optimize it with that information.
Trace the Compilation (2)

```shell
./mx.sh sl -G:-TruffleBackgroundCompilation -G:+TraceTruffleExpansion -G:+TraceTruffleExpansionSource graal/com.oracle.truffle.sl.test/tests/LoopCall.sl
```

Java Stack Trace Console

```
== running on Graal Truffle Runtime
OptimizedCallTarget.executeHelper (OptimizedCallTarget.java:215)
  (OptimizedCallTarget.java:215) RootCallTarget<OptimizedCallTarget>.getRootNode (RootCallTarget.java:48)
  (OptimizedCallTarget.java:215) RootNode<SLRootNode>.getFrameDescriptor (RootNode.java:139)
  (OptimizedCallTarget.java:215) RootCallTarget<OptimizedCallTarget>.getRootNode (RootCallTarget.java:48)
  (OptimizedCallTarget.java:215) SLRootNode.execute (SLRootNode.java:61)
    (SLRootNode.java:61) SLFunctionBodyNode.executeGeneric (SLFunctionBodyNode.java:64)
      (SLFunctionBodyNode.java:64) SBlockNode.executeVoid (SBlockNode.java:63)
        (SBlockNode.java:66) SExpressionNode<SLWriteLocalVariableLongNode>.executeVoid (SExpressionNode.java:52)
          (SExpressionNode.java:52) SExpressionNode<SLReadArgumentNode>.executeLong (SExpressionNode.java:66)
          (SExpressionNode.java:52) SExpressionNode<SLReadLongNode>.executeLong (SExpressionNode.java:66)
          (SExpressionNode.java:52) SExpressionNode<SLReadLocalVariableNodeFactory>.executeLong (SExpressionNode.java:188)
        (SBlockNodeFactory.java:344) SReadLocalVariableLongNode.executeLong (SReadLocalVariableNodeFactory.java:188)
      (SBlockNodeFactory.java) SReadLocalVariableNodeFactory.executeLong (SReadLocalVariableNodeFactory.java:188)
  4e377130 |Nodes 11 |Time 561(209+351)ms |Nodes
```

Makes the output "clickable" in the Eclipse Console view
Print the Function Inlining Tree

```bash
./mx.sh sl -G:-TruffleBackgroundCompilation -G:+TraceTruffleInliningTree graal/com.oracle.truffle.sl.test/tests/Inlining.sl
```

```
== running on Graal Truffle Runtime
Inlining tree for: root g
root d
  root c
    root b
    root a
root e
  root c
    root b
    root a
root f
  root c
    root b
    root a
[truffle] optimized root g
60412166 |Nodes 92 |Time 105( 102+3 )ms |Nodes 8/ 3 |CodeSize 64
```
# Print CallTarget Profile

```
./mx.sh sl -G:-TruffleBackgroundCompilation -G:+TruffleCallTargetProfiling
graal/com.oracle.truffle.sl.test/tests/Inlining.sl
```

<table>
<thead>
<tr>
<th>Call Target</th>
<th>Call Count</th>
<th>Calls Sites</th>
<th>Inlined / Not Inlined</th>
<th>Node Count</th>
<th>Inv</th>
</tr>
</thead>
<tbody>
<tr>
<td>root g</td>
<td>2200</td>
<td>12</td>
<td>0</td>
<td>92</td>
<td>0</td>
</tr>
<tr>
<td>root f</td>
<td>1300</td>
<td>3</td>
<td>0</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>root e</td>
<td>1300</td>
<td>3</td>
<td>0</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>root c</td>
<td>1200</td>
<td>2</td>
<td>0</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>root b</td>
<td>1100</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>root a</td>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>root d</td>
<td>999</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>0 int</td>
</tr>
<tr>
<td>root main</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>26</td>
<td>0 int</td>
</tr>
<tr>
<td>Total</td>
<td>9100</td>
<td>21</td>
<td>2</td>
<td>209</td>
<td>0</td>
</tr>
</tbody>
</table>
Print Histogram of Nodes

```
./mx.sh sl -G:-TruffleBackgroundCompilation -G:+TraceTruffleCompilationDetails
graal/com.oracle.truffle.sl.test/tests/Inlining.sl
```

<table>
<thead>
<tr>
<th>Expanded Truffle Nodes has 11 unique elements and 99 total elements:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLRootNode</td>
</tr>
<tr>
<td>SLFunctionBodyNode</td>
</tr>
<tr>
<td>SLReturnNode</td>
</tr>
<tr>
<td>SLInvokeNode</td>
</tr>
<tr>
<td>InlinedCallNode</td>
</tr>
<tr>
<td>SLFunctionLiteralNode</td>
</tr>
<tr>
<td>SLDirectDispatchNode</td>
</tr>
<tr>
<td>OptimizedAssumption</td>
</tr>
<tr>
<td>SLLongLiteralNode</td>
</tr>
<tr>
<td>SLAddLongNode</td>
</tr>
<tr>
<td>OptimizedCallTarget</td>
</tr>
</tbody>
</table>

[truffle] optimized root g 69e85433 | Nodes 92 | Time 129( 125+4 )ms | Nodes 8/ 3 | CodeSize 64
Truffle Mindset

- Do not optimize interpreter performance
  - Only optimize compiled code performance

- Collect profiling information in interpreter
  - Yes, it makes the interpreter slower
  - But it makes your compiled code faster

- Do not specialize nodes in the parser, e.g., via static analysis
  - Trust the specialization at run time

- Keep node implementations small and simple
  - Split complex control flow into multiple nodes, use node rewriting

- Use final fields
  - Compiler can aggressively optimize them
  - Example: An if on a final field is optimized away by the compiler
  - Try using @CompilationFinal if the Java final is too restrictive

- Use microbenchmarks to assess and track performance of specializations
  - Ensure and assert that you end up in the expected specialization
Truffle Mindset: Frames

- Use VirtualFrame, and ensure it does not escape
  - Sometimes, you get strangely looking error messages about escaping frames
  - Graal must be able to inline all methods that get the VirtualFrame parameter
    - Call must be statically bound during compilation
    - Calls to static or private methods are always statically bound
    - Virtual calls and interface calls work if either
      - The receiver has a known exact type, e.g., comes from a final field
      - The method is not overridden in a subclass

- Important rules on passing around a VirtualFrame
  - Never assign it to a field
  - Never pass it to a recursive method
    - Graal cannot inline a call to a recursive method

- Use a MaterializedFrame if a VirtualFrame is too restrictive
  - But keep in mind that access is probably slower
Object Layout
Object Layout

- SL has no dynamically allocated objects
- But most other languages do…

- Problem when implementing a dynamic programming language
  - Java does not allow objects to grow
  - How to support adding properties to an object dynamically?

- Non-solution
  - Use `java.util.HashMap`
  - High memory overhead, slow access, no specialization possible

- Solution
  - Define a Java class with a few primitive and object fields
  - Extension array if the few inline fields are not sufficient
  - Use shapes (also called hidden classes) to map property names to indices
  - Use polymorphic inline caches to optimize access
Dynamic Object

```java
public class YourLanguageObject {
    ObjectLayout objectLayout;

    long primitiveStorageLocation1;
    ...
    long primitiveStorageLocationN;

    Object objectStorageLocation1;
    ...
    Object objectStorageLocationN;

    long[] primitiveStorageExtension; // Allocated only when necessary.
    Object[] objectStorageExtension; // Allocated only when necessary.
}
```

```java
public class ObjectLayout {
    ObjectLayout parent;

    String propertyName;
    Type propertyType;
    int propertyIndex;
}
```
var x = {};
x.foo = 0;
x.bar = 0;
// + subtree A
Object Layout Transitions (2)

```javascript
var x = {};  
x.foo = 0;   
x.bar = 0;   
// + subtree A

var y = {};  
y.foo = 0.5; 
y.bar = "foo"; 
// + subtree B
```
Object Layout Transitions (3)

```plaintext
var x = {}; 
x.foo = 0; 
x.bar = 0; 
// + subtree A

var y = {}; 
y.foo = 0.5; 
y.bar = "foo"; 
// + subtree B

x.foo += 0.2 
// + subtree C
```
Substrate VM
Substrate VM

- **Goal**
  - Run Truffle languages without the overhead of a Java VM

- **Approach**
  - Ahead-of-time compile the Java bytecodes to machine code
  - Build standard Linux / MacOS executable
Substrate VM Execution Model

Static Analysis

- Truffle Language
- JDK
- Substrate VM

Reachable methods, fields, and classes

Ahead-of-Time Compilation

- Machine Code
- Initial Heap
- DWARF Info
- ELF / MachO Binary

Application running without compilation or Java class loading

All Java classes from Truffle language (or any application), JDK, and Substrate VM
Startup Performance
Running Ruby "Hello World"

Execution time: `time -f "%e"`
Memory footprint: `time -f "%M"`

<table>
<thead>
<tr>
<th></th>
<th>MRI</th>
<th>JRuby</th>
<th>Truffle on JVM</th>
<th>Truffle on SVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>msec</td>
<td>13</td>
<td>353</td>
<td>688</td>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>MRI</th>
<th>JRuby</th>
<th>Truffle on JVM</th>
<th>Truffle on SVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>MByte</td>
<td>5</td>
<td>35</td>
<td>53</td>
<td>∞</td>
</tr>
</tbody>
</table>
Debugging Tools
“Write Your Own Language and Debugger”

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
- People want supporting tools
  - Write a debugger (really?)
  - Use `printf` (most likely)

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 has a debugger
System Structure Extended for Debugging

Language agnostic dynamic compiler

Graal → Truffle

Common API between language implementation and debugging framework and optimization system

Integrate with Java applications

Low-footprint VM, also suitable for embedding

Your language here!

- JavaScript
- Ruby
- Python
- R
- ...
Proxy Nodes as Evaluation “Hooks”

Unmodified Truffle AST

Debuggable Truffle AST

The language implementer:
- Decides which nodes to proxy
- Tags nodes with desired debugging behavior
- Adds other language-specific behavior as needed

Proxies intercept execution if needed, e.g., to “break”
Marks these nodes as “statements” for debugging
Proxied nodes indexed by source location
Proxy Nodes

- Debugging Proxies are
  - Legitimate Truffle AST nodes
  - Transparent to execution semantics of program
    - Default behavior is to pass through all invocations
  - Independent of other platform services (mostly)
  - Compiled to no-ops when not active
  - Compiled into fast-path when active
    - Useful for breakpoint conditions in long-running code
  - Reconstructed through deoptimization, simplifying user interaction
  - A generalization of the approach used for the Self Debugger†
    - Set a breakpoint by modifying program, then reoptimize

Summary
Your Language?

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

graal-dev@openjdk.java.net

$ hg clone http://hg.openjdk.java.net/graal/graal
$ cd graal
$ ./mx --vm server build
$ ./mx ideinit
$ ./mx --vm server sl

- Truffle API Resources
  https://wiki.openjdk.java.net/display/Graal/Truffle+FAQ+and+Guidelines

- Truffle API License: GPLv2 with Classpath Exception