ORACLE®

Oracle Labs Document 2014-0221

Debugging at Full Speed

Instrumenting Truffle-implemented Programs

Michael L. Van De Vanter
Oracle Labs, VM Research Group
July 30, 2014

2014 JVM Language Summit
Santa Clara, CA

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. Oracle reserves the right to alter its development plans and practices at any time, and 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.

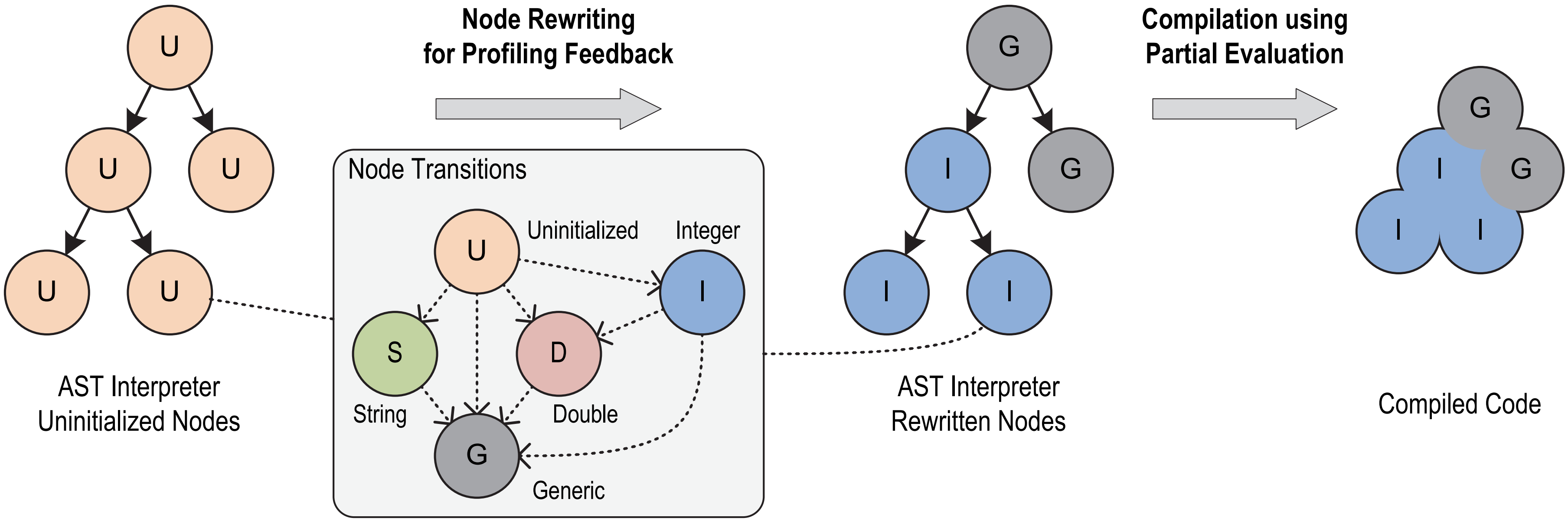
The Big Idea

- Extend the Truffle/Graal platform, which
  - enables high-performance implementations
  - of dynamic languages (Ruby, R, JavaScript, Python, Smalltalk)
  - with reduced implementation effort
- by building in *instrumentation* support that
  - enables flexible access to Truffle execution events
  - by many kinds of tools (debuggers, profilers, etc.)
  - with minimal runtime overhead
  - requiring modest implementation effort.

Instrumenting Truffle Programs

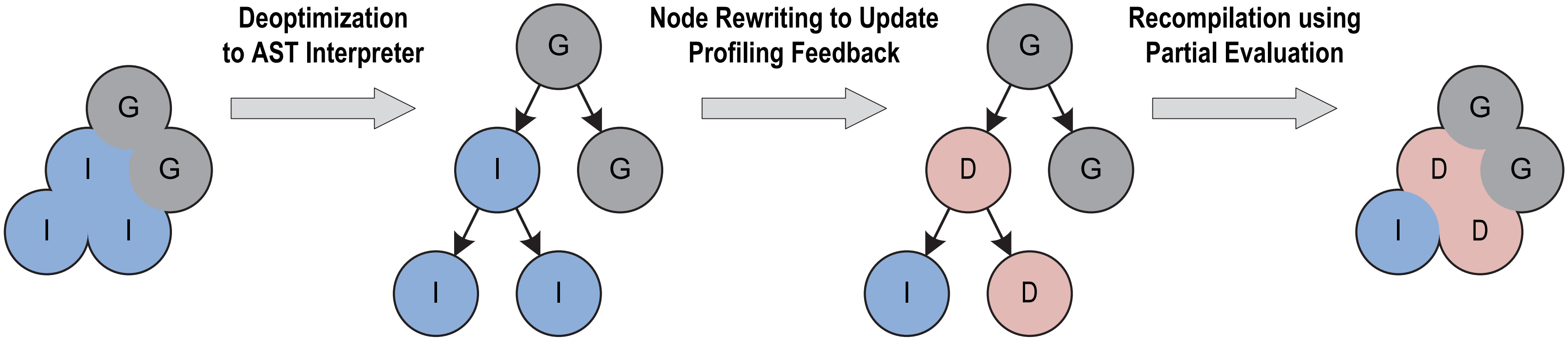
1. Truffle: Execution Overview
2. Tools: Opportunity, Challenge, Strategy
3. Proof of Concept: Ruby, Ruby debugger
4. Truffle: new Instrumentation API
5. Applications & Status

Truffle – node specialization



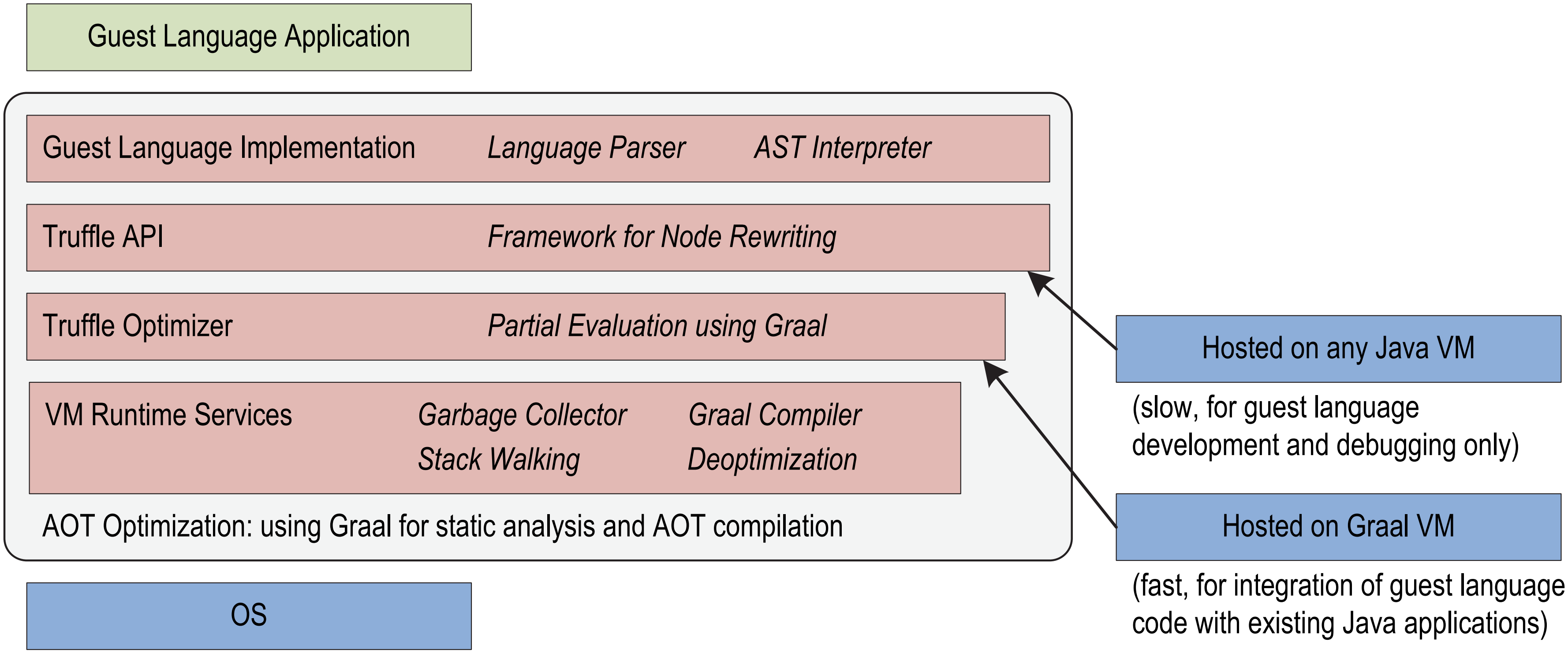
T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In Proceedings of Onward!, 2013.

Truffle – deoptimization and rewriting



T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In Proceedings of Onward!, 2013.

GraalVM Architecture



Guest Language Application

Guest Language Implementation *Language Parser* *AST Interpreter*

Truffle API *Framework for Node Rewriting*

Truffle Optimizer *Partial Evaluation using Graal*

VM Runtime Services *Garbage Collector* *Graal Compiler* *Stack Walking* *Deoptimization*

AOT Optimization: using Graal for static analysis and AOT compilation

OS

Hosted on any Java VM

(slow, for guest language development and debugging only)

Hosted on Graal VM

(fast, for integration of guest language code with existing Java applications)

T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In Proceedings of Onward!, 2013.

Instrumenting Truffle Programs

1. Truffle: Execution Overview
2. Tools: Opportunity, Challenge, Strategy
3. Proof of Concept: Ruby, Ruby debugger
4. Truffle: new Instrumentation API
5. Applications & Status

Opportunity: Tool Access to Truffle Runtime

- Language implementations relatively *accessible*
    - All Java
    - Written as *interpreters*
    - Execution state represented explicitly: stack, frames, …
- Truffle/Graal *optimizes* and *deoptimizes*
    - Many aggressive optimizations down to native code
    - For its own purposes can *deoptimize*
    - Reconstruct execution state completely

Challenge: Blend with Truffle

- Define language-agnostic services
  - Automate as much tool support as possible
- Leverage Truffle/Graal optimizations
  - E.g. breakpoint conditions should fully optimize
  - Unused framework should compile away
- Minimize disruption
  - Language implementation code
  - Truffle optimizations
- Simple API for tool clients

Strategy: Interpose with Node Wrappers

- Insert additional nodes into Truffle ASTs (wrappers)
  - Attachment points for tool interposition
- Semantically transparent during execution
  - Largely orthogonal to language implementation
- Can be composed
- Focus on language execution semantics
  - No access (for now) to low-level state, e.g. memory

Instrumenting Truffle Programs

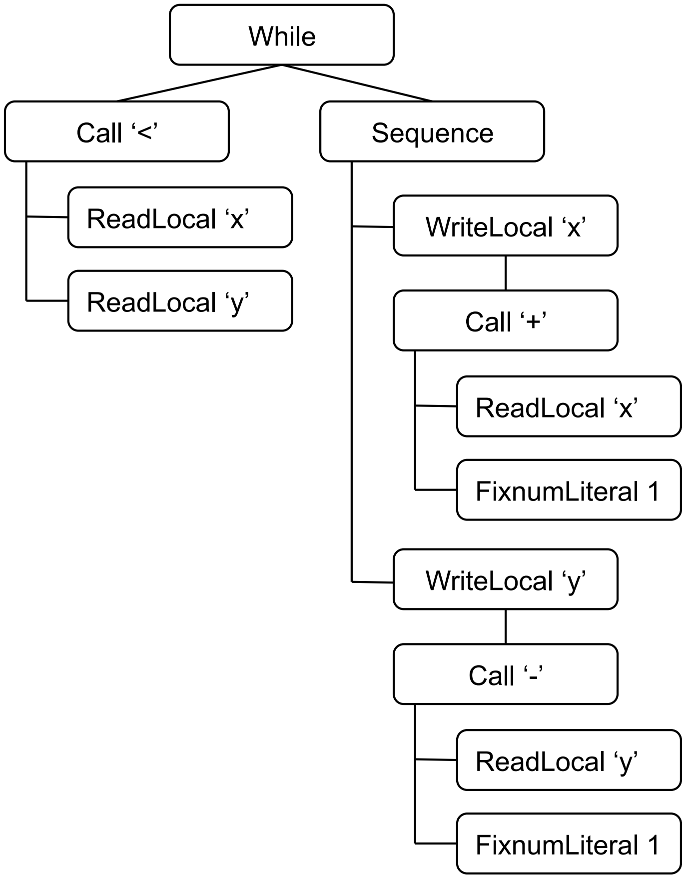
1. Truffle: Execution Overview
2. Tools: Opportunity, Challenge, Strategy
3. Proof of Concept: Ruby, Ruby debugger
4. Truffle: new Instrumentation API
5. Applications & Status

Tools: The Debugger Challenge

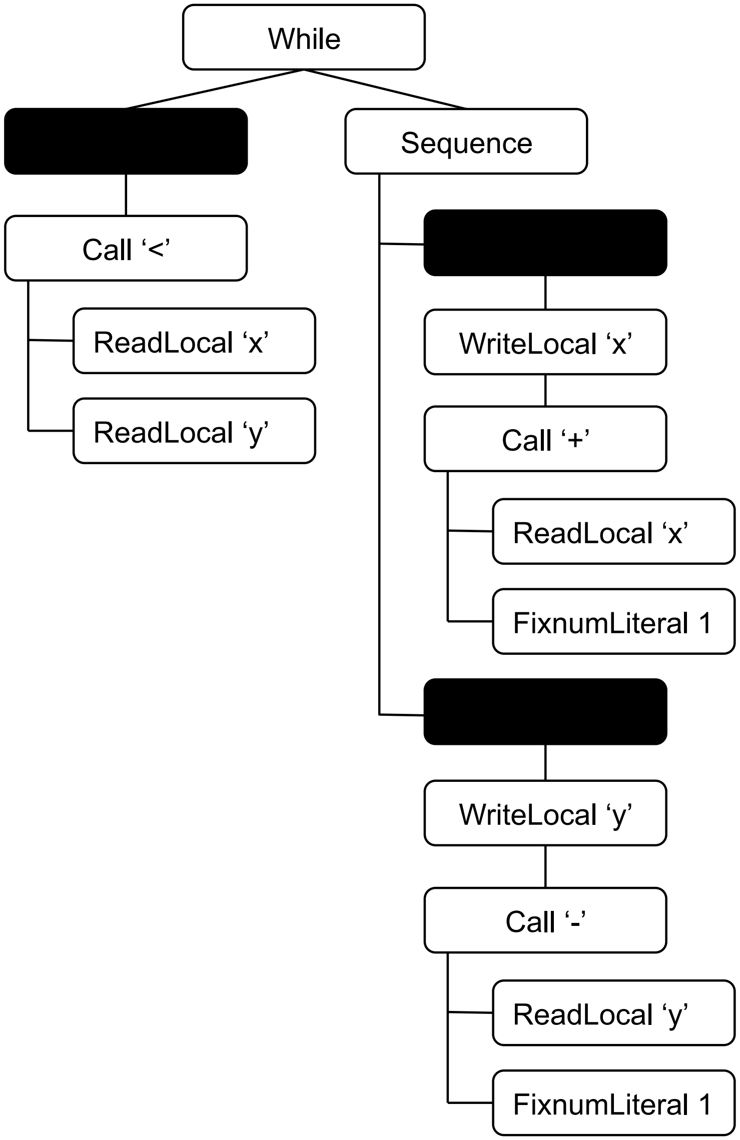
- They are difficult to build
  - Often tightly coupled to native platforms
  - Cross many levels of system abstractions
  - Contend with incomplete access to execution state
- They present significant productivity tradeoffs
  - Performance – disabled optimizations
  - Functionality – inhibited language features
  - Complexity – language implementation requirements
  - Inconvenience – nonstandard context (debug flags)

Wrapper Nodes

```
while x < y
  x += 1
  y -= 1
end
```



- While
  - Call ‘<’
    - ReadLocal ‘x’
    - ReadLocal ‘y’
  - Sequence
    - WriteLocal ‘x’
      - Call ‘+’
        - ReadLocal ‘x’
        - FixnumLiteral 1
    - WriteLocal ‘y’
      - Call ‘-’
        - ReadLocal ‘y’
        - FixnumLiteral 1



While

Sequence

Call ‘<’

ReadLocal ‘x’

ReadLocal ‘y’

WriteLocal ‘x’

Call ‘+’

ReadLocal ‘x’

FixnumLiteral 1

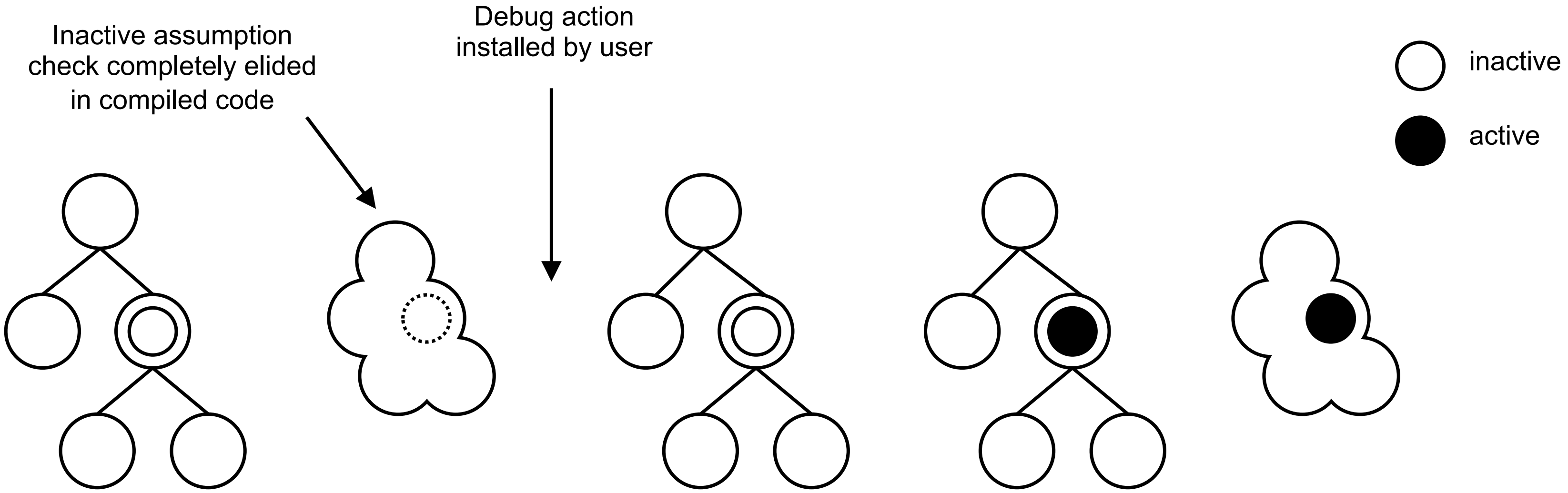
WriteLocal ‘y’

Call ‘-’

ReadLocal ‘y’

FixnumLiteral 1

Ruby Prototype (Chris Seaton)
Active/Inactive Actions



Inactive assumption check completely elided in compiled code

Debug action installed by user

○ inactive

● active

Compile: produces partially evaluated machine code from specialized AST.

Deoptimize: transfers control from the machine code back to the AST interpreter.

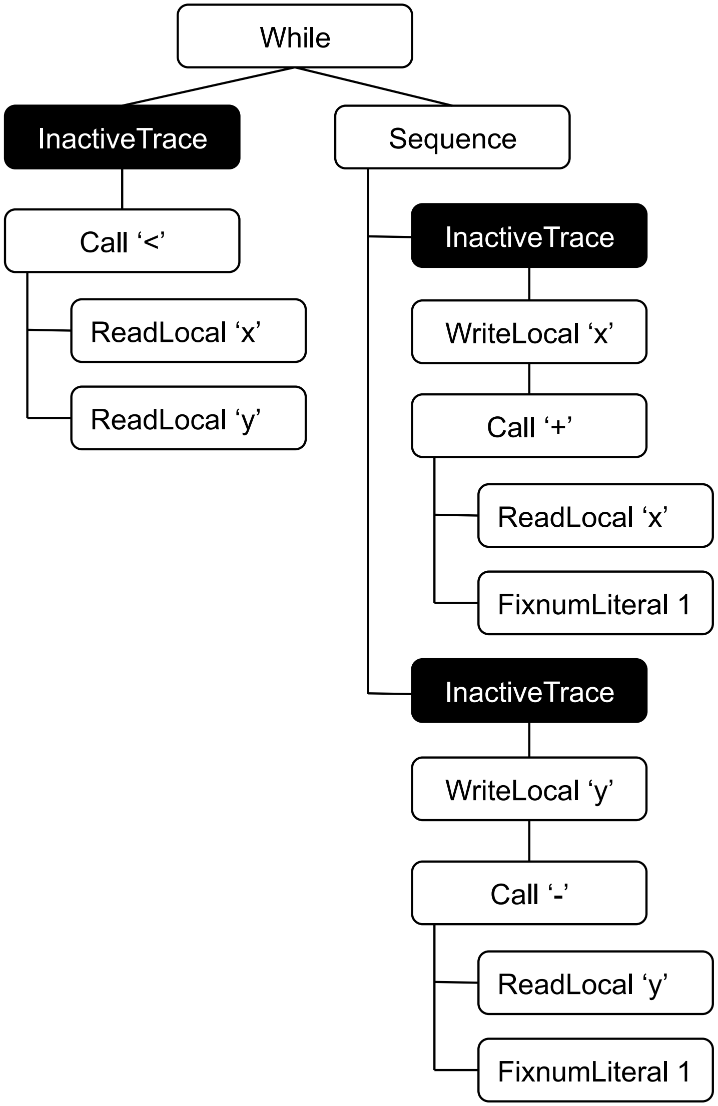
Replace: the inactive node with an active node to install the debug action

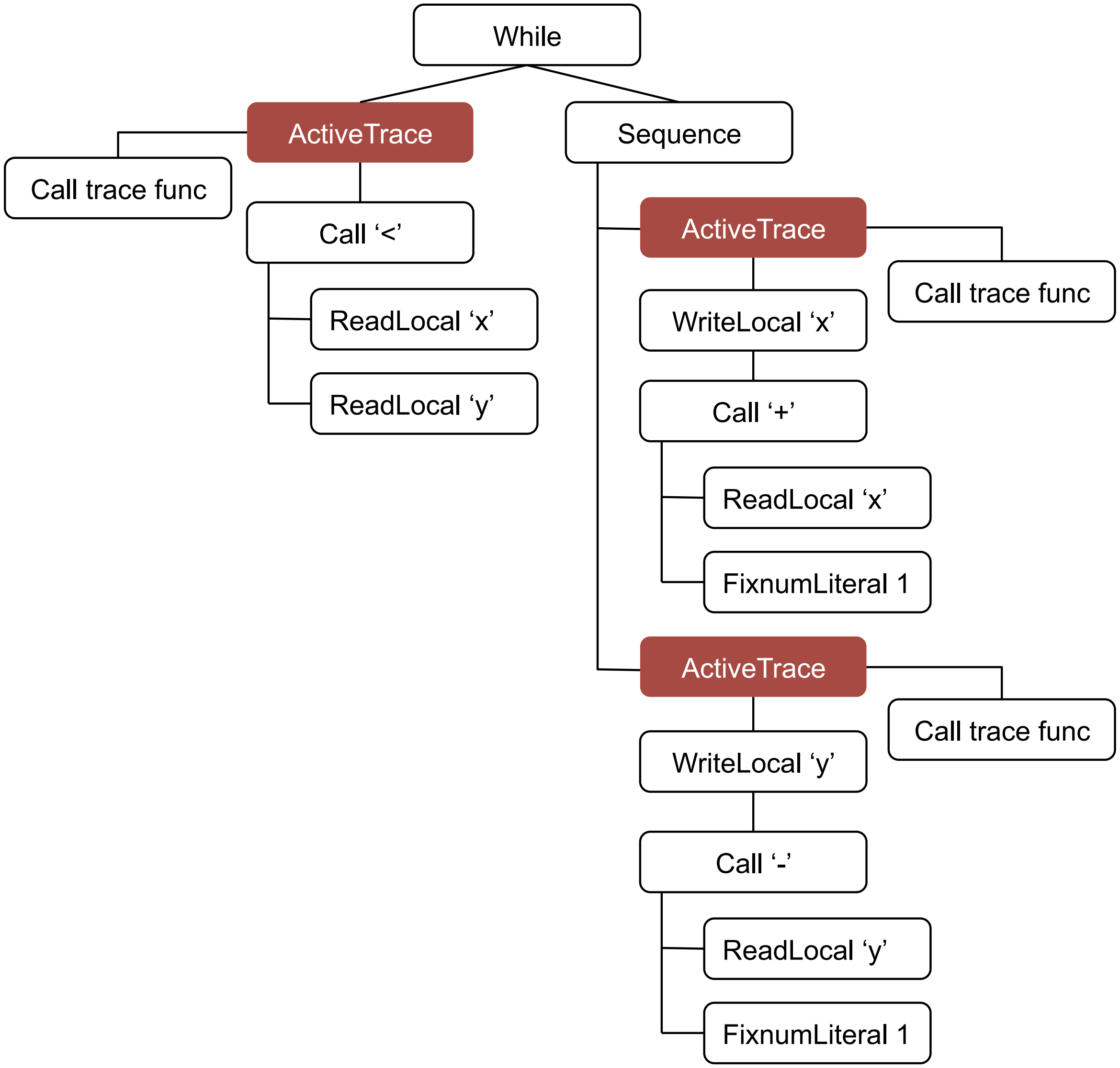
Compile: produces new machine code from the modified AST and the installed debug action.

Ruby Prototype: set_trace_func

```ruby
set_trace_func proc { |event, file, line,
    id, binding, classname|
  puts "We're at line number #{line}"
}
```

- Standard Ruby language feature
- Uses: debuggers, profilers, coverage
- Enabled and disabled programmatically
- Naïve implementation: check for it every single time a bytecode has a different line number
- Rubinius doesn’t support it at all, JRuby has it behind a flag


While
InactiveTrace
Call ‘<’
ReadLocal ‘x’
ReadLocal ‘y’
Sequence
InactiveTrace
WriteLocal ‘x’
Call ‘+’
ReadLocal ‘x’
FixnumLiteral 1
InactiveTrace
WriteLocal ‘y’
Call ‘-’
ReadLocal ‘y’
FixnumLiteral 1


While
ActiveTrace
Call trace func
Call '<'
ReadLocal 'x'
ReadLocal 'y'
Sequence
ActiveTrace
Call trace func
WriteLocal 'x'
Call '+'
ReadLocal 'x'
FixnumLiteral 1
ActiveTrace
Call trace func
WriteLocal 'y'
Call '-'
ReadLocal 'y'
FixnumLiteral 1

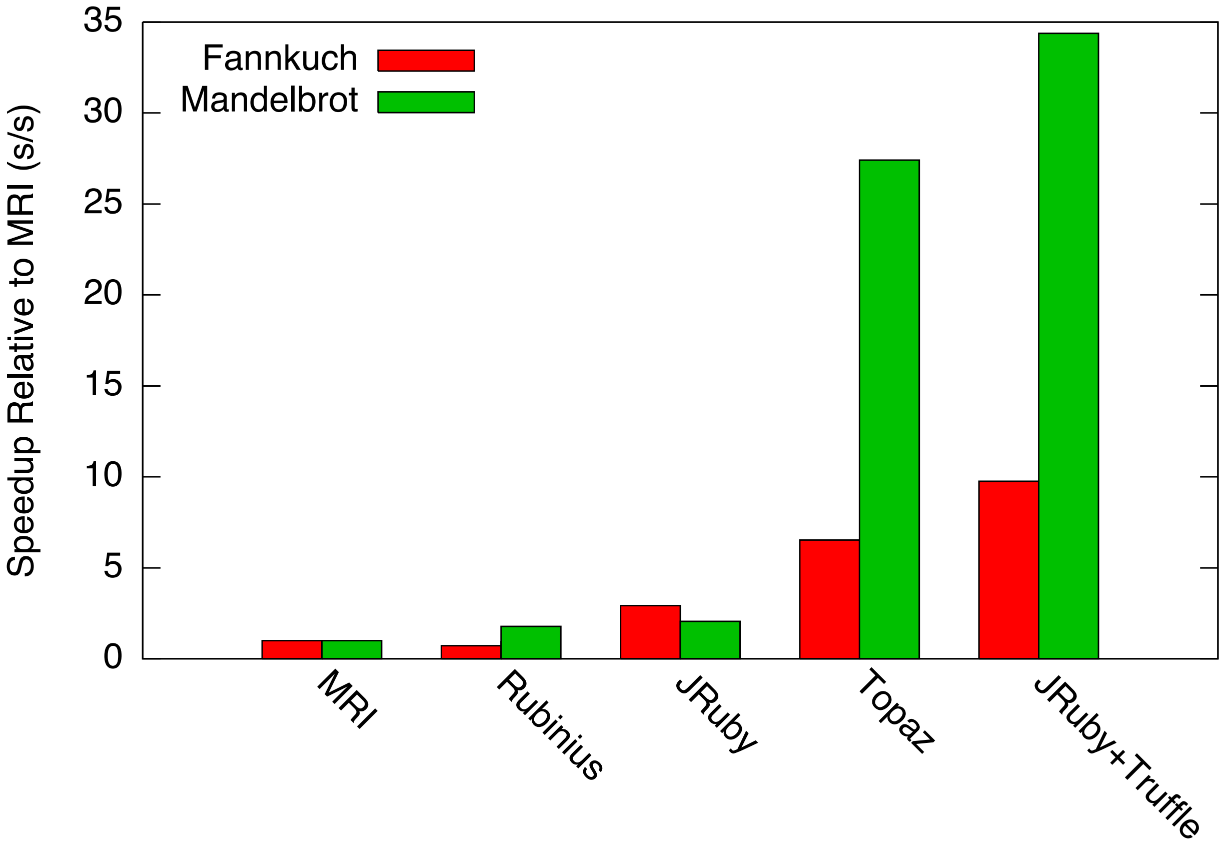
Ruby Prototype: Builtin Debug Operations

```ruby
100.times do
 Debug.break
end
```

```ruby
Debug.break("test.rb", 14) do
 puts "The program has reached line 14"
end
```

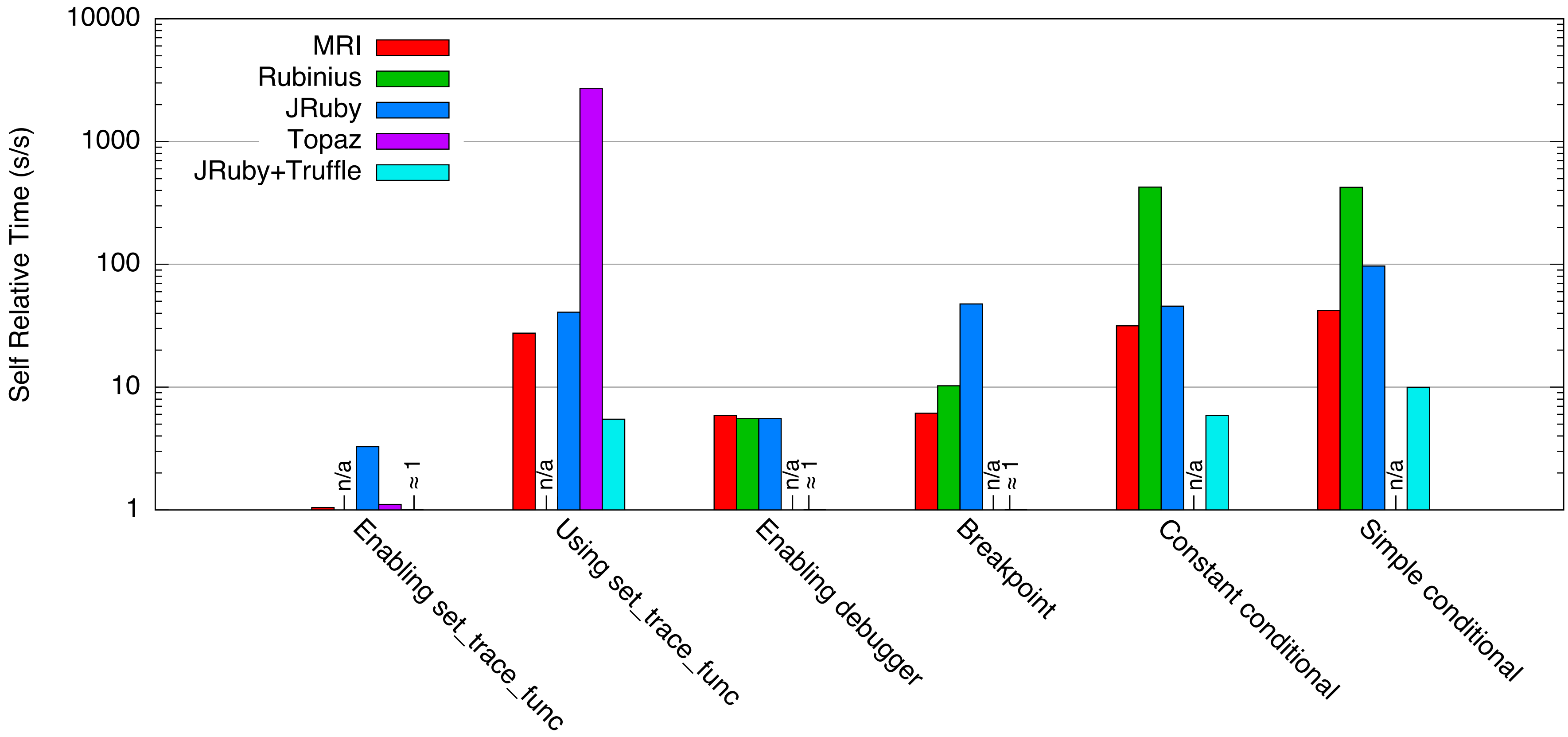
```ruby
Debug.break("test.rb", 14) do |binding|
  if binding.local_variable_get(:foo) == 14
    Debug.break
  end
end
```

Evaluation – general performance



Chris Seaton, Michael Van De Vanter, Michael Haupt. Debugging at Full Speed. In Proceedings of DYLA 2014.

Evaluation – summary



Chris Seaton, Michael Van De Vanter, Michael Haupt. Debugging at Full Speed. In Proceedings of DYLA 2014.

Instrumenting Truffle Programs

1. Truffle: Execution Overview
2. Tools: Opportunity, Challenge, Strategy
3. Proof of Concept: Ruby, Ruby debugger
4. Truffle: new Instrumentation API
5. Applications & Status

Generalize, Scale, Simplify

- Ruby prototype:
  - Extensions (builtins) to guest language
  - Function-specific wrappers
  - Multiple, chained wrappers
  - Wrappers only added statically
  - Activate tool functions via node replacement

- Instrumentation API:
  - Language-agnostic API for tools
  - General purpose wrappers
  - Single wrapper per location
  - Add multiple tool functions dynamically
  - Activate tool functions by state changes

Language Implementation API

- Define language-specific wrapper node type(s)
- Wrap appropriate nodes when ASTs constructed
- Ensure complete *source attribution*
  - By default, Probes are tracked by source location
- Additional tool-specific support (e.g. debugging)
  - E.g. eval

Tool Client API: Overview

Disclaimer: work in progress

- SourceSection
  - Text range corresponding to a *user-sensible* AST node
- SyntaxTag
  - Annotation to guide tool behavior at an AST node (e.g. STMT)
- ExecutionEvent
  - Flow of control into or out of an AST node
- Instrument
  - Tool-specific receptor of ExecutionEvents at an AST node
- Probe
  - Manager for Instruments & SyntaxTags at an AST node

Tool Client API: A Truffle Instrument

Interposes in Truffle interpretation

```
public abstract class Instrument implements ExecutionEvents {

  public void enter(Node node, Frame frame) {
  }

  public void leave(Node node, Frame frame) {
  }

  public void leave(Node node, Frame frame, Object result) {
  }

  public void leave(Node node, Frame frame, boolean result) {
      leave(node, frame, (Object) result);
  }

  . . .

  public void leaveExceptional(Node node, Frame frame, Exception e) {
  }
}
```

Tool Client API: A Truffle Probe

Manages Instruments & Tags at an AST location

```
public interface Probe {

  void addInstrument(Instrument newInstrument);

  void removeInstrument(Instrument oldInstrument);

  void tagAs(SyntaxTag tag);

}

public interface ProbeListener {

  void newProbeInserted(SourceSection location, Probe probe);

  void probeTaggedAs(Probe probe, SyntaxTag tag);

}
```

Tool Client API: Finding Probes Dynamically

```
public abstract class ExecutionContext {

  Probe getProbe(SourceSection sourceSection);

  Collection<Probe> findProbesTaggedAs(SyntaxTag tag);

  Collection<Probe> findProbesByLine(LineLocation lineLocation);

. . .

}
```

Instrumenting Truffle Programs

1. Truffle: Execution Overview
2. Tools: Opportunity, Challenge, Strategy
3. Proof of Concept: Ruby, Ruby debugger
4. Truffle: new Instrumentation API
5. **Applications & Status**

Application: Language Implementation

Ruby implementation by Chris Seaton

- Ruby `set_trace_function`
  - Invokes arbitrary *proc* at each statement
- Ruby heap iteration, possibly filtered
  - `ObjectSpace.each_object()`
  - `ObjectSpace.count_objects()`
  - `ObjectSpace._id2ref(object_id)`
- Ruby core library thread quiescence
  - Implemented with Instruments

Application: Simple Profiling

Collaboration with UC Irvine

- ZipPy performance optimizations
  - Exploratory work to locate hot spots
  - Gathering interpreter-level execution data
  - Building maps when needed

Application: Debugging (1/3)

Very limited, but useful functionality

- Debugging engine prototype
  - Breakpoints; step in/over/out; show frame vars; backtrace
  - Evaluate string in halted context
  - Language-agnostic (mostly)
- Language-specific adaptation
  - Some language-specific code needed
  - *Ruby*, *JavaScript* working at same functional level
  - *Simple* (demonstration language) underway
  - *R* just begun

Application: Debugging (2/3)

```
public final class DebugBreakInstrument extends DebugInstrument {

    @CompilationFinal private boolean isEnabled = true;

    public DebugBreakInstrument(DebugInstrumentCallback callback) {
        super(callback);
    }

    @Override
    public void enter(Node astNode, VirtualFrame frame) {
        if (isEnabled) {
            debuggerCallback.haltedAt(astNode, frame.materialize());
        }
    }

    . . .
}
```

Application: Debugging (3/3)

- Command-Line debugger for testing & demonstration
  - Client/server architecture emulated
  - Client is language-agnostic
  - Server built around a language-specific Debugging Engine
- IDE integration
  - Looking at simple experiments with NetBeans

Status

- Instrumentation is part of the public Truffle API
- Debugging is under internal development

Hardware and Software

Engineered to Work Together