Debugging at Full Speed

Chris Seaton
Oracle Labs
University of Manchester
chris.seaton@oracle.com

Michael L. Van De Vanter
Oracle Labs
michael.van.de.vanter
@oracle.com

Michael Haupt
Oracle Labs
michael.haupt@oracle.com

ABSTRACT

Debugging support for highly optimized execution environments is notoriously difficult to implement. The Truffle/-Graal platform for implementing dynamic languages offers an opportunity to resolve the apparent trade-off between debugging and high performance.

Truffle/Graal-implemented languages are expressed as abstract syntax tree (AST) interpreters. They enjoy competitive performance through platform support for type specialization, partial evaluation, and dynamic optimization/deoptimization. A prototype debugger for Ruby, implemented on this platform, demonstrates that basic debugging services can be implemented with modest effort and without significant impact on program performance. Prototyped functionality includes breakpoints, both simple and conditional, at lines and at local variable assignments.

The debugger interacts with running programs by inserting additional nodes at strategic AST locations; these are semantically transparent by default, but when activated can observe and interrupt execution. By becoming in effect part of the executing program, these "wrapper" nodes are subject to full runtime optimization, and they incur zero runtime overhead when debugging actions are not activated. Conditions carry no overhead beyond evaluation of the expression, which is optimized in the same way as user code, greatly improving the prospects for capturing rarely manifested bugs. When a breakpoint interrupts program execution, the platform automatically restores the full execution state of the program (expressed as Java data structures), as if running in the unoptimized AST interpreter. This then allows full introspection of the execution data structures such as the AST and method activation frames when in the interactive debugger console.

Our initial evaluation indicates that such support could be permanently enabled in production environments.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Debugging and Testing—*Ruby*; D.3.4 [**Programming Languages**]: Processors—*run-time environments, interpreters*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
Dyla'14 June 09 - 11 2014, Edinburgh, United Kingdom
Copyright 2014 ACM 978-1-4503-2916-3/14/06 ...$15.00
http://dx.doi.org/10.1145/2617548.2617550.

General Terms

Design, Performance, Languages

Keywords

Truffle, deoptimization, virtual machines

1. INTRODUCTION

Although debugging and code optimization are both essential to software development, their underlying technologies typically conflict. Deploying them together usually demands compromise in one or more of the following areas:

- *Performance*: Static compilers usually support debugging only at low optimization levels, and dynamic compilation may also be limited.
- *Functionality*: Years of research on the topic have most often given priority to optimization, resulting in debugging services that are incomplete and/or unreliable.
- *Complexity*: Debuggers usually require compiler support, in particular the generation of additional information the debugger might need when deciphering execution state. This strategy can strongly couple their respective implementations.
- *Inconvenience*: Putting a system under observation by a debugger requires some form of "debug mode", for example using the `-Xdebug` option when starting the Java[1] Virtual Machine.

As a consequence, debugging is seldom enabled in production environments. Defects that arise must be investigated in a separate development environment that differs enough from production that reproducing the issue may be hard.

The VM Research Group at Oracle Labs proposes to eliminate this conflict in optimized dynamic runtimes by making production-quality code debug-aware without adding overhead. The basis for this proposal is *Truffle* [29], a newly developed platform for constructing high performance implementations of dynamic languages. A Truffle-based language implementation is expressed as an abstract syntax tree

[1]Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

(AST) interpreter, to which the framework applies aggressive dynamic optimizations including type specialization, inlining, and many other techniques.

The group's recently developed implementation of Ruby [5] has already shown promising results [28] and is now being integrated with the existing implementation of Ruby on the JVM, JRuby [18]. As part of a larger inquiry into techniques for adding developer tool support to *Truffle*, the Ruby AST was experimentally extended to include support for a simple yet useful debugger, as well as for Ruby's built-in *tracing* facility. Both extensions exhibit essentially zero runtime overhead until enabled, at which time the debugging/tracing logic accrues overhead proportional to the specific functionality needed.

This strategy was first explored for the Self debugger [9], where debugging support is tightly integrated with execution machinery. A significant consequence of this strategy is that the representation of logic implementing debugging functionality becomes indiscernible from the representation of ordinary application code and thus subject to all the optimization capabilities the VM offers for application code.

The paper is structured as follows. The next section describes the background for this experiment in more detail: the underlying Truffle platform, the Truffle-based implementation of Ruby (Ruby Truffle), the combination of Ruby Truffle and JRuby (JRuby+Truffle), and the prototype Ruby debugger. In section 3, we introduce our use of AST "wrapper" nodes in the debugger. Section 4 gives details about the implementation in the Truffle framework, and about the mechanisms of the underlying VM—including the Graal compiler—that it uses. In section 5, we discuss our implementation in comparison to alternatives, in terms of impact on peak performance. Sections 6 and 7 discuss related work and conclude the paper.

2. BACKGROUND

The context for the experiment presented here is a project at the VM Research Group at Oracle Labs to develop a Truffle-based implementation of the Ruby programming language. An initial implementation produced in less than six months demonstrated extremely high peak performance [28].

During this period the group was also evaluating techniques for extending the Truffle platform with support for developer tools. A goal for these extensions was to exhibit zero peak temporal performance overhead when when debugging is enabled but not in use, and minimal overhead when debugging is in use. We would like to make it possible to debug long running and production processes to catch problems that may only occur infrequently or only in production environments. We would also like to be able to to diagnose problems without restarting long running or production processes in a debug configuration, and we would like to be able to do all of this without reducing the performance of the system until the event we want to debug is actually encountered.

The experimental debugger described here provided a test case for these techniques.

2.1 Truffle and Graal

The *Truffle* framework [29] supports building programming language run-time environments expressed originally as interpreted ASTs. Each Truffle AST node carries the logic required to enact the associated programming language semantics. Based on type information obtained at run-time, AST nodes speculatively replace themselves in the AST with *type-specialized* variants [30]. Should the speculation be wrong, they replace themselves again with other specializations, or with generic nodes that subsume the functionality required for all possible types.

Truffle interpreters perform best when running atop the Graal VM [26]: a modified HotSpot Java VM that hosts the Graal dynamic compiler. The Graal VM supports *partial evaluation* of Truffle ASTs and generates efficient machine code from them. Tree rewriting—e. g., in case of failing speculations—is possible for trees compiled into machine code by means of *dynamic deoptimization* [9], which transitions control from a compiled machine code method back into the original AST interpreter and restores the source-level execution state.

2.2 Ruby Truffle

Ruby [5] is a dynamically typed object-oriented language with features inspired by Smalltalk and Perl. It is best known in combination with the Rails web framework for quick development of database-backed web applications, but it is also applied in fields as diverse as bioinformatics [7] and graphics processing [16]. Although generally considered a relatively slow language with little support from traditional large enterprise, it has powered significant parts of extremely large scale applications, for example Twitter [14] and GitHub [11].

The Truffle implementation of Ruby began as a new code base, reusing only the JRuby parser. Truffle allows for rapid implementation: after five months of development Ruby Truffle ran the RubySpec test suite [25], passing about 50 % of the language tests, and ran unmodified micro benchmarks and small kernels from real applications. Ruby Truffle's implementation comprised about 19,000 lines of Java in 155 classes. The prototype debugger was also implemented during this period, allowing it to be used to support Ruby Truffle's development.

2.3 JRuby+Truffle

The Truffle-based implementation of Ruby has been licensed for open source and merged into JRuby: an existing Java-based implementation [18]. The original JRuby began as a straightforward port of the standard Ruby interpreter from C to Java. It has since become arguably the most sophisticated implementation of a dynamic language on the JVM.

JRuby's first tier of execution is an AST interpreter, but it then compiles methods to JVM bytecode. The JRuby project was also an early adopter and key contributor to the design of the `invokedynamic` JVM instruction [23], and the wider JSR 292 framework for supporting dynamic languages [22], so it was already structured around multiple compilation options and backends.

We refer to JRuby running with the Ruby Truffle backend as *JRuby+Truffle*. This combination is the subject of the performance evaluations presented in section 5.

2.4 The Prototype Debugger

Debugging was added to Ruby Truffle by extending Ruby's core library with built-in debugging methods, so that interactive debugging could be conducted via sessions with Ruby's *shell*. Debugging operations include:

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

Figure 1: Example Ruby code

- Set a line breakpoint that halts execution.
- Set a line breakpoint with an associated *action*: a fragment of Ruby code that might be guarded by a *conditional expression* and which might halt execution or anything else.
- Set a data breakpoint on a local variable in some method that halts execution immediately after an assignment.
- Set a data breakpoint with an associated *action*, as with a line breakpoint.
- Continue execution.
- Basic introspection of the program structure and current state such as examining the value of variables and reporting the halted position.

The goal for the prototype was to test whether the goals described earlier could be achieved through techniques that leverage Truffle's AST node abstraction, combined with the partial evaluation, dynamic optimization, and dynamic deoptimization capabilities of Graal. The key strategy is to implement debugging actions as dedicated AST nodes that are *inserted* into the AST, and thus into the flow of program execution. To Truffle's optimization machinery these nodes appear no different than ordinary "language-level" AST nodes.

3. DEBUG NODES

Debugging in the prototype is implemented by strategically modifying the AST under interpretation by Truffle.

3.1 Wrappers

The central construct in the Ruby Truffle debugger is the *wrapper node* or simply *wrapper*. This is a Truffle AST node with one child that:

- is transparent to execution semantics,
- by default just propagates the flow of program execution from parent to child and back, and
- performs debugging actions when needed.

Starting with an AST produced by a conventional parser, we insert a wrapper as the parent of the first node corresponding to each location where we may want to install some debug action.

Figure 1 shows Ruby code that increments a local variable `x` and decrements a local variable `y` while `x < y`. This code has three locations where we might want to set a breakpoint, and two locations where we might want to break on assignment of a local variable.

Figure 2 shows the AST of this code as produced by the parser. Figure 3 shows the same AST with wrappers inserted wherever the Ruby parser tells us that the line number has changed, to implement line breakpoints. Each wraps a single child node.

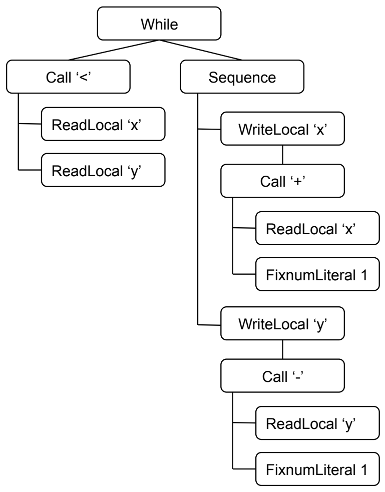


Figure 2: AST of Figure 1 without wrappers

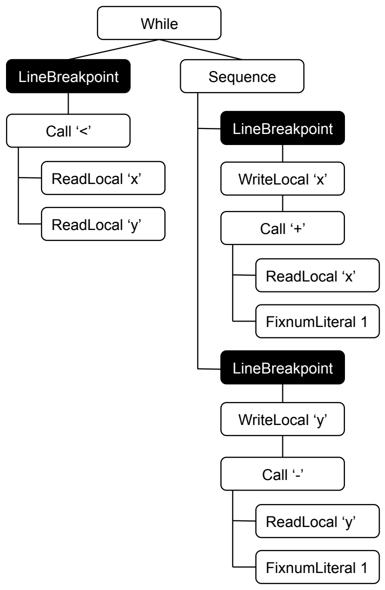


Figure 3: AST of Figure 1 with wrappers to implement line breakpoints

3.2 Debug Operations

Potential debugging operations are implemented in the form of wrappers at every location where the user might want to request a debug action. The wrappers are always added, whether or not a debug action is initially installed, and whether or not a debugging tool is currently attached. When added, the wrappers are initially in an *inactive* state. An inactive wrapper is simple: during each execution it only checks to see if it should be enabled, and if not propagates the flow of program execution to the wrapped node. The debug operation at a particular location can be enabled by replacing just the relevant inactive wrappers with *active* versions. The action that the active wrapper performs depends on the functionality it implements, and we describe several active wrapper nodes in section 3.4. When no longer needed, an active wrapper replaces itself again with an inactive version.

3.3 Assumptions

Many wrappers follow this pattern: a debugging node replaces itself with an alternate version when some presumably rare condition occurs. Truffle aggressively optimizes code when given *hints* about what conditions should be treated as the normal case; instances of the `Assumption` class are one way to do this.

An `Assumption` is implemented as a **`boolean`** that is initially **`true`** until *invalidated*, at which time it becomes permanently **`false`**. For example, debugging code might create an instance to represent the fact that there is no breakpoint at a particular line of source code, and will only invalidate that assumption should a breakpoint be created.

Truffle applies important optimizations speculating that `Assumption.isValid()` always returns **`true`**. When an instance is invalidated (i. e., its value is set to **`false`**), Truffle **`deoptimizes`** any method code that depends on that assumption (i. e., any code that calls `Assumption.isValid()` on the instance). Typically the program then replaces the node associated with the invalid `Assumption` and creates a new (valid) instance of `Assumption`. Section 4 discusses how the `Assumption` class is optimized and how it makes possible very low cost validity checks.

3.4 Wrapper Roles

Our implementation of a Ruby debugger uses wrapper nodes to implement debug and metaprogramming functionality that is similar to that provided by other implementations.

3.4.1 `set_trace_func`

Ruby's core library method `Kernel#set_trace_func` registers a method to be called each time the interpreter encounters certain events, such as moving to a new line of code, entering or leaving a method, or raising an exception (Figure 4 shows an example). This method is used to implement other Ruby debuggers (such as the *debugger* library, detailed in section 5), profilers, coverage tools and so on. The trace method receives a `Binding` object that represents the current environment (local variables in lexical scope) as well as other basic information about where the trace was triggered. One trace method at a time can be installed, and it may be removed by calling **`set_trace_func`** with **`nil`**.

The Ruby Truffle debugger implements **`set_trace_func`** as an (initially inactive) *trace wrapper* at the location of each line. Each time it is executed, the inactive node checks the assumption that there is *no trace method* installed before propagating the flow of program execution. When the check fails, the node replaces itself with an active trace wrapper.

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

Figure 4: Example usage of `set_trace_func`

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

Figure 5: Example command to install a line breakpoint

The active wrapper correspondingly checks the assumption that *there is a trace method* before first invoking the method and then propagating the flow of program execution. When the trace method has been removed, the check fails and an inactive wrapper is swapped back in. Using an `Assumption` object ensures that in the most common case the only overhead is the (inactive) wrappers performing the check.

3.4.2 Line Breakpoints

The line breakpoint and **`set_trace_func`** implementations are similar. However, instead of a single trace method, line breakpoint wrappers check if a method has been installed for their associated line of source code. The debugger maintains a map that relates source locations to `Assumption` objects. A newly constructed line breakpoint wrapper is given access to the `Assumption` that the current method for that line has not changed.

A triggered breakpoint halts program execution and starts an interactive session similar to the standard interactive Ruby shell known as "irb". This debugging session runs in the execution environment of the parent scope at the breakpoint, so that local variables are visible in the debugger. Additional Ruby methods available in the shell include `Debug.where` (displays the source location where the program is halted) and `Debug.continue` (throws an exception that exits the shell and allows program execution to continue). We have not yet implemented debug operations such as *next*, but believe these can be implemented with combinations of these techniques.

The action taken by an active line breakpoint node could be anything that can be expressed in Java (Truffle's host language) or, as with **`set_trace_func`**, a method written in Ruby. Figure 5 shows an example command to install a line breakpoint. This could have been written as part of the program, or typed into an interactive shell. The example prints a message to the log, but it could contain arbitrary Ruby code, including entry into the debugger.

3.4.3 Conditional Line Breakpoints

Conditional line breakpoints are a simple extension to line breakpoints. Since the breakpoint wrapper is a legitimate Truffle AST node, an **`if`** statement can be wrapped around the action that invokes the debugger. To support conditions

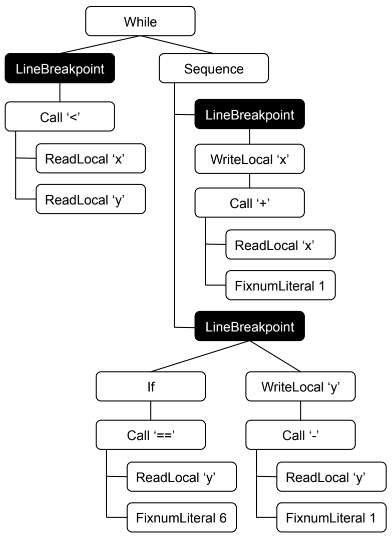


Figure 6: AST of Figure 1 with a line breakpoint with condition `y == 6`

written in Ruby, we can call a user-defined method to test the condition, in exactly the same way as we call a user-defined method in `set_trace_func`. Again, it is also possible to inline this method, so the condition becomes part of the compiled and optimized method.

Figure 6 shows the AST of Figure 1 with a line breakpoint installed on line 3 that contains the condition `y == 6`. The condition forms a first-class part of the AST, alongside the original program, with no distinction between debug code and user code that might inhibit optimization.

3.4.4 Local Variable Watchpoints

Breakpoints on the modification of local variables, as well as the conditional version of the same, are implemented almost exactly as are line breakpoints. A local breakpoint wrapper is inserted at each local assignment node, and the debugging action happens *after* the child has executed, i. e., when the local holds the newly assigned value.

4. IMPLEMENTATION

This section describes the properties of the underlying Truffle/Graal platform that make this approach to debugging effective.

4.1 The Truffle Compilation Model

The Truffle-based implementation of Ruby is expressed as an AST interpreter [30]. Unlike all other modern implementations of Ruby, we do not generate bytecode and do not explicitly generate machine code. Instead, when running on a JVM with the Graal compiler, Truffle will profile AST execution. When it discovers a frequently executed tree, it takes the compiler intermediate representation of all the methods involved in executing the AST—primarily, all the `execute` methods on the AST nodes—and inlines them into a single method. The powerful intra-method optimizations that the JVM normally applies within methods are applied across all the methods, and Truffle produces a single machine code function for the AST. In our case this is a single machine code function for a single Ruby method. This by-default inlining of AST interpreter methods removes the overhead introduced by inactive wrappers.

4.2 Overview

Figure 7 summarizes the transitions of an AST with debug nodes under the Truffle compilation model. It shows an AST with an inactive wrapper node illustrated as a double circle. After enough executions of this AST to trigger compilation, a single machine code method is produced from all nodes in the AST, including the inactive wrapper. When a line breakpoint (illustrated as a filled circle) is installed in place of the inactive wrapper node, we trigger deoptimization (explained in the following subsection) by invalidating the assumption on which the compiled code is produced. The machine code restores interpreter frames and jumps back into interpreted code. In the interpreter we replace the inactive wrapper node with an active line breakpoint node. Execution continues, and after a period to allow the AST to re-stabilize (for example we have to determine the type of operations and fill inline caches in the modified AST) we again reach the threshold of executions for the AST to be compiled. Graal caches parts of the compilation of the AST so compilation with the replaced node does not have to take as long [26].

Figure 8 illustrates how a simplified inactive debug node is compiled to leave zero overhead. The semantic action method of the node, `execute`, checks the assumption that the node should still be inactive. If the assumption is no longer valid the node is replaced. Under compilation, the check is constant and valid. The compiler sees that no exception is thrown and removes the entire `catch` block. The only remaining action is to directly execute the child of the wrapper (the node that is being wrapped). Truffle inlines by default, so there is no method call overhead to execute the child. If the assumption was no longer valid, for example because the user is installing a breakpoint, execution will transition to the interpreter. There the check on the assumption is always performed. Then the exception will be thrown and caught, a new active node will be created and will replace this current inactive node. Execution will then continue with the new active node. At some point Truffle will decide to compile the method again, with the new node in place.

Inactive wrapper nodes play several important roles in ASTs being debugged, even though they compile to nothing most of the time. They make it possible to map significant AST locations, for example the beginning of lines, that could othewise be reached only by tree navigation. They can be relied upon to persist, even when nodes around them are replaced during Truffle AST optimization. Finally, they can be activated and deactivated by *self-replacement*, which is Truffle's fundamental (and safe) mechanism for runtime AST modification.

4.3 Deoptimization

Truffle provides two implementations of the `Assumption` class. When Graal is unavailable or the method is being interpreted, `Assumption` is implemented with a Boolean flag and explicit checks as described in section 3.3. Since `Assumption` objects are checked often but invalidated rarely, a strategy that treats them as constant and valid during compilation, but ensures that invalidation of the assumption

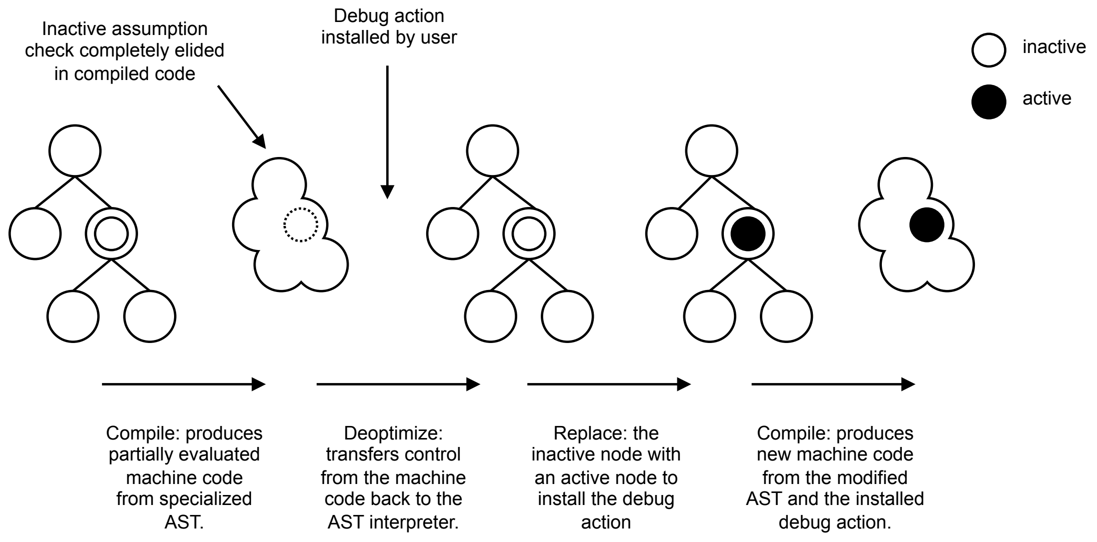
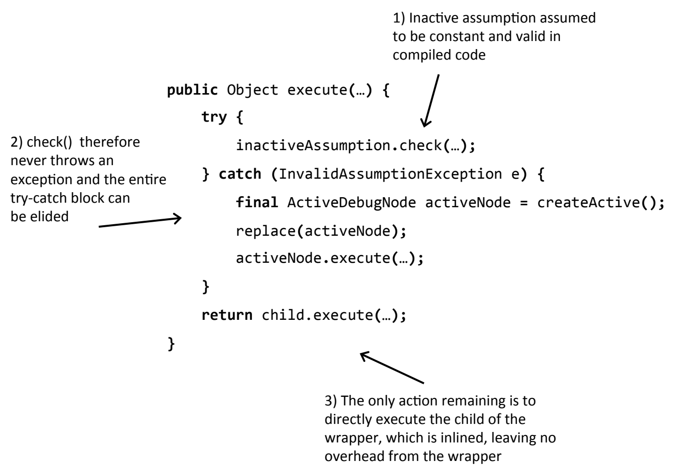

Inactive assumption check completely elided in compiled code
Debug action installed by user
inactive
active
Compile: produces partially evaluated machine code from specialized AST.
Deoptimize: transfers control from the machine code back to the AST interpreter.
Replace: the inactive node with an active node to install the debug action
Compile: produces new machine code from the modified AST and the installed debug action.

Figure 7: Overview of the Truffle compilation model as it applies to debug nodes



```
public Object execute(…) {
    try {
        inactiveAssumption.check(…);
    } catch (InvalidAssumptionException e) {
        final ActiveDebugNode activeNode = createActive();
        replace(activeNode);
        activeNode.execute(…);
    }
    return child.execute(…);
}
```

1) Inactive assumption assumed to be constant and valid in compiled code
2) check() therefore never throws an exception and the entire try-catch block can be elided
3) The only action remaining is to directly execute the child of the wrapper, which is inlined, leaving no overhead from the wrapper

Figure 8: Explanation of how code in an inactive debug node (simplified) is compiled to leave zero-overhead

occurs correctly, can bring large performance benefits. The Graal implementation of `Assumption` on top of the HotSpot JVM provides the mechanism to do this efficiently.

OpenJDK JIT compilers such as *server* [19] and *client* [13] emit machine code at runtime after sufficient invocations of a method, and will then call the machine code version of the method instead of interpreting it. The transition from the initial interpreter into this compiled machine code does not have to be one-way. The transition in the other direction, from machine code to interpreter is called *dynamic deoptimization* [9]. Part of the complexity of deoptimization is that invalidated machine code may be already running and on the stack, potentially with more than one activation, and potentially on more than one thread. Multiple activations in a single thread are deoptimized by examining the entire stack when deoptimizing and transitioning all activations of affected methods. Activations in other threads are deoptimized by cooperatively halting them at a *safepoint* where threads test a page that has its permissions changed to cause a `segfault` and stop the thread. Safepoints are already emitted by the JVM to support systems such as the garbage collector, so they add no overhead in our system to support debugging multi-threaded applications.

The efficient implementation of `Assumption` is made possible by this mechanism, which Graal exploits by maintaining a list for each `Assumption` object of all machine code that depends on the `Assumption` being valid. The invalidate method on an `Assumption` object instructs the underlying JVM to invalidate all dependent machine code, which triggers OpenJDK JVM deoptimization.

The Graal VM is specifically designed to enable aggressive speculative optimizations [3]. It uses dynamic deoptimization internally, and employing the same techniques for debugging introduces no additional runtime overhead.

4.4 Expectation

The inlining of AST interpreter methods and the use of dynamic deoptimization via `Assumption` objects instead of explicit checks to enable debug operations means that our debugger implementation will have no peak temporal performance overhead at all when debugging is enabled but not in use. After optimization, a tree with an inactive line breakpoint wrapper becomes no different in terms of the JVM JIT compiler IR than if the wrapper had not been added, as it had no body after the assumption check was removed, and we remove method boundaries.

It is also possible to inline a trace method or breakpoint condition, rather than making a method call. A copy of the trace method's AST can be inserted as a child node of the active wrapper. In this position it is optimized in exactly the same way as user code, as if inserted at that point in the source code. Part of the reason that `set_trace_func` is expensive in existing implementations (see section 5) is that the trace method is passed the current environment (the *binding* parameter in Figure 4). When our debugger inlines a trace method, Graal observes through escape analysis [12] that the environment argument can be optimized.

5. EVALUATION

We evaluated the temporal performance of JRuby+Truffle against other implementations of Ruby interpreters and debuggers.

We focused on peak temporal performance—that is, performance after all compilation has finished and performance has reached a steady state—because we are interested in debugging long running programs such as servers, where enabling a debugger is usually prohibitively expensive. We do not consider the temporal performance of hitting a breakpoint and entering the debugger to display a user interaction prompt, as we consider this to be an offline operation and not relevant to peak performance.

All of the code used in this evaluation is open source. Our experimental artifact contains scripts to download and build all the relevant software including our patches, to run the experiments and to produce the results.[2]

5.1 Compared Implementations

The reference implementation of Ruby is often referred to as **MRI** (Matz's Ruby Interpreter) or CRuby [17]. Prior to version 1.9 it was an AST interpreter written in C, but it now also has a simple bytecode interpreter [24]. We used version 2.1.0.

Rubinius [20] is Ruby 'built using Ruby' with stated goals of high temporal performance and concurrency, as well as being more accessible for Ruby developers. It has a substantial VM core implemented in C++, but most of the Ruby specific behavior is implemented in Ruby with an interface to the VM. Rubinius uses LLVM [15] to implement a JIT compiler and also implements its own debugger. We used version 2.2.4.

We evaluated **JRuby** [18] separately, without the Ruby Truffle backend enabled. We ran with `invokedynamic` enabled when possible, and used a development build at revision 59185437ae86.

Based on the PyPy project [21], **Topaz** [6] is Ruby implemented in RPython, a subset of Python that can be statically translated to C for native compilation. At runtime Topaz can also apply the RPython tracing JIT to the interpreter. Topaz is comparable in completeness to JRuby+Truffle, but includes no support for debugging yet. We used a development build at revision 4cdaa84fb99c, built with PyPy at revision 8d9c30585d33.

[2]http://lafo.ssw.uni-linz.ac.at/truffle/debugging/dyla14-debugging-artifact-0557a4f756d4.tar.gz

The Ruby standard library includes a simple debugger that is implemented using `set_trace_func`. The name of the library is *debug*, but for clarity we will refer to it as **stdlib-debug**. As part of the standard library, it is not separately versioned.

Providing a superset of the functionality of stdlib-debug, **ruby-debug** (also referred to as *rdebug*) is implemented as a C extension to MRI to reduce overheads. This library is the foundation for most debugging tools commonly used with Ruby, e. g., in RubyMine. We used version 1.6.5.

As JRuby has limited support for C extensions, ruby-debug has been ported to Java as **jruby-debug**. We used version 0.10.4. jruby-debug is not compatible with the development version of JRuby, so we ran experiments using jruby-debug with the latest compatible version, JRuby 1.7.10.

5.2 Experimental Setup

All experiments were run on a system with 2 Intel Xeon E5345 processors with 4 cores each at 2.33 GHz and 64 GB of RAM. We used 64bit Ubuntu Linux 13.04, using system default compilers. Where an unmodified Java VM was required, we used the system default 64bit OpenJDK 1.7.0_51. For implementations using Graal we used version 0.1.

Each configuration evaluated was sampled 30 times. We found that 10 runs was at least sufficient to reach steady state with subsequent runs being statistically independently and identically distributed. We informally verified this using lag plots [10]. We disregarded these first 10 runs to obtain sample measures of peak temporal performance. An arithmetic mean of the samples gives us the reported time. Reported errors are the standard error. When summarizing across multiple benchmarks we report a geometric mean.

We used common simple benchmarks from the Computer Language Benchmarks Game (née Shootout)—`fannkuch` and `mandelbrot` [2]. These are small synthetic benchmarks with well-understood limitations when used for comparing performance. For our experiments, they are only used as a base on which to install breakpoints (with the exception of section 5.3) so we do not believe their simplicity poses any threat to the validity of our results.

5.3 Relative Default Performance

Keeping in mind the limitations of these simple benchmarks, we first present an overview of the relative default performance of the implementations under evaluation. It is important to understand the orders of magnitude involved here, as providing low overhead debugging is only important in a fast system. Figure 9 shows the performance of the different implementations in their default configuration, relative to the performance of MRI. As the different implementations offer different default functionality (for example, JRuby does not support `set_trace_func` by default, where we support both `set_trace_func` and full debugging) the comparison is not fair, but it is indicative of the temporal performance perceived by users during normal operation.

5.4 Overhead of `set_trace_func`

We looked at the temporal overhead of using `set_trace_func` to understand the ability of each implementation to

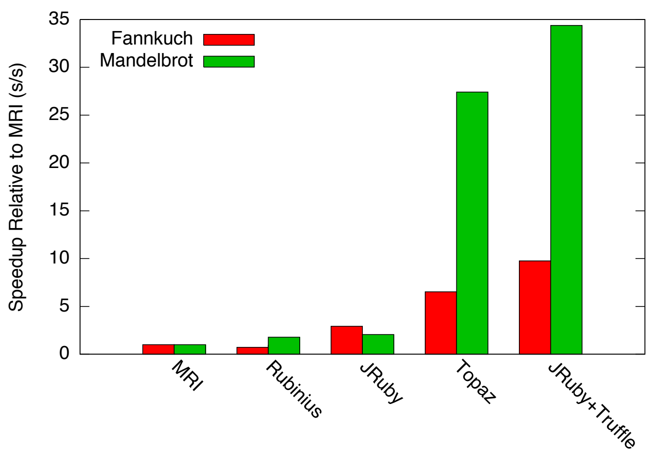


Figure 9: Speed of Ruby implementations in default configuration, relative to MRI (taller is linearly better)

efficiently attach user code to a running program. First we looked at the overhead of being able to use `set_trace_func` but not actually doing so, by comparing with it disabled and with it enabled. Then we looked at the overhead of running with a trace method actually installed, and finally after it has been removed. The trace method we used simply counted the number of times it was called.

As MRI and Topaz have no option to turn off `set_trace_func`, we patched them to remove support. JRuby requires the `--debug` flag to turn on support for `set_trace_func` and requires compilation to JVM bytecode to be disabled. Rubinius does not support `set_trace_func`, so it is not mentioned in this subsection. For JRuby+Truffle, we added an option to not create trace nodes so we could evaluate performance with tracing not enabled.

Table 1 shows benchmark run times when `set_trace_func` is disabled with standard deviation and standard error (*disabled*). We then show the percentage overhead relative to the disabled time when `set_trace_func` is enabled but no method is installed (*before*), when the simple method is installed (*during*) and after it has been removed (*after*).

MRI shows low overhead because one extra memory read per line is a tiny proportion of the work done in executing the rest of line under their execution model. The overhead when a trace method is installed is high but not unreasonable given it has no mechanism to elide the allocation of a `Binding` object and so must actually allocate it on the heap for each trace event.

JRuby's initial overhead results from having to disable compilation to JVM bytecode, which is required in order to use the feature. The overhead of calling the trace method is limited by having to allocate the `Binding` object.

Topaz has low but statistically significant overhead for enabling tracing. However the implementation does not appear to be optimized for having a trace method actually installed, showing a pathological overhead as large as three orders of magnitude.

JRuby+Truffle shows very low overhead for enabling tracing, and a reasonable overhead of 4–5x to install a trace method. Our implementation inlines the trace method, allowing the binding (the object representing the current environment) to be elided if not actually used. If the trace method is used, and if escape analysis determines that the binding cannot be referenced outside the method, then the frame can be allocated on the stack for better performance than via default heap allocation.

5.5 Overhead of a Breakpoint on a Line Never Taken

We examined the overhead of setting a breakpoint on a line that is never taken during execution, compared to running with debugging disabled, and to running with debugging enabled but without any breakpoints. This represents the cost of setting a line breakpoint on some rarely taken erroneous path. The question we are asking is this: If such an erroneous state is only observed intermittently, such as once a week, what is the cost of having the breakpoint set during the whole run of the program to catch the one time when it is? The breakpoint was set on a line in the inner loop of the benchmarks. The condition we used to guard the line was not statically determinable to be always `false` by any of the implementations. We also looked at the overhead after the breakpoint has been removed.

As with `set_trace_func`, JRuby requires the `--debug` flag which disables compilation. We found that the Rubinius debugger silently failed to work in combination with JIT compilation[3] so like JRuby we were forced to disable compilation. However, Rubinius runs methods with breakpoints in a special interpreter anyway, so disabling the JIT should not affect the reported overhead. Topaz does not support any debuggers, so it is not mentioned in experiments from this point on. We added an option to JRuby+Truffle to not create debug nodes to test performance when debugging is not enabled.

For this and the following experiments, we considered multiple combinations of implementation and debugger where possible. For example, we show JRuby with both stdlib-debug and jruby-debug. In later summaries, we show only the best performing combination. Normally, JRuby+Truffle would detect that the branch is never taken during interpretation and speculatively elide it for compilation, but we disabled conditional branch profiling for all these experiments.

Table 2 shows time taken for the benchmark when the debugger is disabled with standard deviation and standard error (*disabled*). We then show the percentage overhead relative to that disabled time of running with the debugger enabled and attached, but no breakpoint set (*before*), then with a breakpoint set on a line never taken (*during*), and finally after the breakpoint has been removed (*after*).

The overhead of using stdlib in either MRI or JRuby is extremely high as it is based on the already inefficient implementations of `set_trace_func`. The native extension variants ruby-debug and jruby-debug show two orders of magnitude less overhead, bringing it down to around a reasonable 5x. Rubinius also has a reasonable overhead of 1.2–6.8x. JRuby+Truffle shows very low overhead for all states. Overhead is negative in some cases due to normal variation in produced machine code (the Graal compiler is non-deterministic). The overhead appears to be negative for the

[3]We reported this issue along with test cases to demonstrate the problem (https://github.com/rubinius/rubinius/issues/2942) but have not received a response at the time of writing.

Fannkuch				
	Disabled (s (sd) se)	Before	During	After
MRI	0.995 (0.006) ±0.142%	0.1x	24.6x	0.1x
JRuby	0.358 (0.008) ±0.514%	3.9x	199.4x	3.7x
Topaz	0.154 (0.001) ±0.204%	0.0x	661.1x	0.0x
JRuby+Truffle	0.091 (0.003) ±0.692%	0.0x	4.0x	0.0x
Mandelbrot				
	Disabled (s (sd) se)	Before	During	After
MRI	2.015 (0.001) ±0.014%	0.0x	30.7x	0.0x
JRuby	0.992 (0.013) ±0.304%	2.8x	153.5x	2.8x
Topaz	0.073 (0.000) ±0.054%	0.2x	5680.4x	0.0x
JRuby+Truffle	0.060 (0.000) ±0.179%	0.0x	5.0x	0.0x

Table 1: Overhead of `set_trace_func` (lower is better)

mandelbrot benchmark due to normal non-determinism in the Graal compiler caused by profiling.

5.6 Overhead of a Breakpoint With a Constant Condition

We looked at the overhead of setting a line breakpoint with a constant condition that is statically determinable to always evaluate to `false`. This tests the overhead of a conditional breakpoint where the condition itself should have no overhead. Again the breakpoint was set on a line in the inner loop of the benchmarks.

We could not find any support in stdlib-debug for conditional breakpoints, so it is not mentioned further.

Table 3 shows the overheads in the same format as before. Results for MRI, Rubinius and JRuby are broadly the same as before, except with a significant additional overhead caused by evaluating the condition. JRuby+Truffle now shows a significant overhead when the conditional breakpoint is installed. Although the condition is constant, we are not yet able to inline the condition in the line where the breakpoint is installed, so this overhead represents a call to the condition method.

5.7 Overhead of a Breakpoint With a Simple Condition

Finally, we looked at the overhead of setting a line breakpoint with a simple condition, comparing a local variable against a value it never holds. This tests the normal use of conditional breakpoints, such as breaking when some invariant fails. Again the breakpoint was set on a line in the inner loop of the benchmarks.

Table 4 shows the overheads in the same format as before. The overhead when there is a simple condition to test compared to a constant condition is not great in MRI. The overhead in JRuby+Truffle when the breakpoint is installed is increased but is still reasonable at up to 10x.

5.8 Summary

Table 5 summarizes the results across both benchmarks, using the highest performing debugger implementation for each implementation of Ruby. We show the overhead for different debug tasks, in each case compared to when `set_trace_func` or debugging is disabled. Figure 10 shows self-relative performance on a logarithmic scale, with relative performance of 1 being no-overhead and one vertical grid line being an extra order of magnitude of overhead.

JRuby+Truffle has on average 0.0x overhead for enabling tracing, debugging and setting a breakpoint on a line never reached. For constant and simple conditional breakpoints on lines in the inner loop of the benchmarks which we cannot optimize away entirely JRuby+Truffle has a very reasonable overhead in the range 5–9x: about an order of magnitude less than other implementations.

When interpreting these results we should also keep in mind the extreme performance variation among language implementations (see Figure 9). These overheads are *on top* of those differences. In terms of absolute wall-clock performance, JRuby+Truffle is over two orders of magnitude faster than the next fastest debugger, MRI with ruby-debug, when running with a breakpoint on a line never taken.

5.9 Other Implementations of Ruby

Serious implementations of Ruby are suprisingly numerous, relative to similar languages such as Python, PHP or Perl. We did not consider implementations that have been unsupported for years, such as Ruby Enterprise Edition (an older version of MRI with performance patches, most of which are now in MRI) and IronRuby (Ruby implemented on the Dynamic Language Runtime). We also did not consider some variants of Ruby that are platform specific or differ significantly from standard Ruby such as MacRuby (using OS X system libraries), RubyMotion (on iOS) or MRuby (for embedded environments). We did review MagLev [4], the implementation of Ruby on a commercial Smalltalk VM: performance was not competitive with JRuby, it only supports legacy versions of the Ruby language, it does not support `set_trace_func` at all, and it does not compile methods with breakpoints.

6. RELATED WORK

6.1 Self

Debugging support was one of the primary motivations behind the development of *dynamic deoptimization* in Self, which was claimed to be “the first practical system providing full expected [debugging] behavior with globally optimized code” [9]. Along with other seminal innovations in Self, this derived from its creators’ firm commitment that the experience of using a language is fully as important as performance [27].

	Fannkuch			
	Disabled (s (sd) se)	Before	During	After
MRI/stdlib-debug	1.043 (0.006) ±0.124%	154.2x	182.9x	196.3x
MRI/ruby-debug	1.043 (0.006) ±0.124%	4.3x	4.7x	4.3x
Rubinius	1.459 (0.011) ±0.174%	4.5x	6.8x	3.6x
JRuby/stdlib-debug	0.562 (0.010) ±0.402%	1375.2x	1609.2x	1573.3x
JRuby/jruby-debug	0.562 (0.010) ±0.402%	4.5x	43.3x	41.9x
JRuby+Truffle	0.091 (0.003) ±0.692%	0.0x	0.0x	0.0x
	Mandelbrot			
	Disabled (s (sd) se)	Before	During	After
MRI/stdlib-debug	2.046 (0.001) ±0.009%	139.6x	179.0x	166.8x
MRI/ruby-debug	2.046 (0.001) ±0.009%	5.5x	5.6x	5.5x
Rubinius	1.151 (0.002) ±0.031%	4.6x	11.7x	3.8x
JRuby/stdlib-debug	1.096 (0.008) ±0.170%	1698.3x	1971.4x	1884.1x
JRuby/jruby-debug	1.096 (0.008) ±0.170%	4.6x	49.8x	48.3x
JRuby+Truffle	0.060 (0.000) ±0.179%	0.0x	0.0x	0.0x

Table 2: Overhead of setting a breakpoint on a line never taken (lower is better)

	Fannkuch			
	Disabled (s (sd) se)	Before	During	After
MRI/ruby-debug	1.043 (0.006) ±0.124%	4.3x	25.8x	4.2x
Rubinius	1.459 (0.011) ±0.174%	3.7x	187.4x	3.7x
JRuby/jruby-debug	0.562 (0.010) ±0.402%	4.6x	41.2x	41.4x
JRuby+Truffle	0.107 (0.003) ±0.528%	0.0x	1.7x	0.0x
	Mandelbrot			
	Disabled (s (sd) se)	Before	During	After
MRI/ruby-debug	2.046 (0.001) ±0.009%	5.5x	35.4x	5.5x
Rubinius	1.151 (0.002) ±0.031%	4.6x	662.8x	4.0x
JRuby/jruby-debug	1.096 (0.008) ±0.170%	4.3x	48.0x	47.4x
JRuby+Truffle	0.059 (0.001) ±0.188%	0.0x	8.1x	0.0x

Table 3: Overhead of setting breakpoint with a constant condition (lower is better)

	Fannkuch			
	Disabled (s (sd) se)	Before	During	After
MRI/ruby-debug	1.043 (0.006) ±0.124%	4.3x	31.6x	4.2x
Rubinius	1.459 (0.011) ±0.174%	3.7x	187.7x	4.5x
JRuby/jruby-debug	0.562 (0.010) ±0.402%	4.4x	86.2x	41.9x
JRuby+Truffle	0.107 (0.003) ±0.528%	0.1x	10.1x	0.1x
	Mandelbrot			
	Disabled (s (sd) se)	Before	During	After
MRI/ruby-debug	2.046 (0.001) ±0.009%	5.6x	50.7x	6.7x
Rubinius	1.151 (0.002) ±0.031%	4.7x	659.8x	4.5x
JRuby/jruby-debug	1.096 (0.008) ±0.170%	4.5x	105.3x	46.8x
JRuby+Truffle	0.059 (0.001) ±0.188%	0.0x	7.9x	0.0x

Table 4: Overhead of setting breakpoint with a simple condition (lower is better)

	MRI	Rubinius	JRuby	Topaz	JRuby+Truffle
Enabling set_trace_func	0.0x	n/a	2.3x	0.1x	0.0x
Using set_trace_func	26.6x	n/a	39.9x	2714.2x	4.5x
Enabling debugging	4.9x	4.6x	4.6x	n/a	0.0x
Breakpoint on a line never taken	5.1x	9.3x	46.5x	n/a	0.0x
Breakpoint with constant condition	30.6x	425.1x	44.6x	n/a	4.9x
Breakpoint with simple condition	41.2x	423.7x	95.8x	n/a	9.0x

Table 5: Summary of overheads (lower is better)

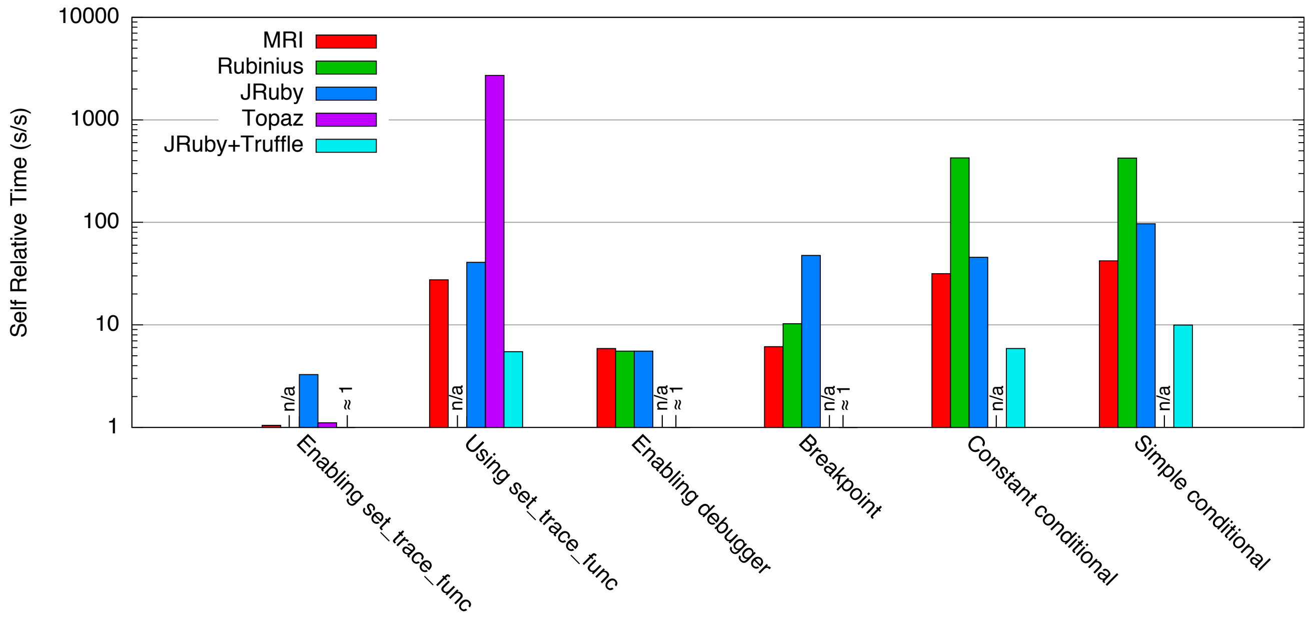


Figure 10: Summary of relative performance when using debug functionality (taller is exponentially worse)

Debugging in the original Self was primitive; setting a breakpoint required manually inserting "a send of `halt` into the source method". It was also deeply entwined with the language implementation. For example, the `finish` operation was implemented by "changing the return address of the selected activation's stack frame to a special routine ...".

Two decades of progress in the underlying technologies have led to the Truffle platform, which supports multiple languages, and into which nearly transparent debugging code can be inserted without the language specificity and fragility of its original incarnation.

6.2 Debugging Optimized Code

Debugging statically compiled, optimized code has been a problem worthy of many PhD dissertations. As the Self creators pointed out, that work generally gave priority to optimization, and results generally were complex and supported only limited debugging functionality [9]. The Ruby debugger demonstrates that the compromise can be avoided.

6.3 Wrapper Nodes

The idea of wrapping nodes to transparently introduce extra functionality was applied before in a machine model for aspect-oriented programming languages [8]. The abstractions used there are generic enough to be used for debugging as well, as this work shows.

6.4 Low Overhead for `set_trace_func` in Ruby

Topaz provides high peak temporal performance even with `set_trace_func` enabled by default, using techniques comparable to ours. However, performance with a trace method installed causes an extremely high performance overhead, showing that only the inactive path is optimized. Topaz implements `set_trace_func` by declaring the current trace method to be a *green variable* [1]. A green variable is one that should be the same every time a compiled method (actually a *trace* in the case of Topaz) is entered. Without a trace method installed, the variable is `nil` upon entering the trace and can be assumed to be `nil` throughout, meaning that the check at each line if there is a trace method installed is a constant. If a trace method is installed, the compiled trace will be found to be invalid when the green variables are checked, and will be recompiled. This is similar to how checking our `Assumption` class becomes a constant operation. However, to implement `set_trace_func` the developers of Topaz needed to define the trace method as a green variable in their main JIT object and at each *merge point* where the interpreter could enter a compiled trace. We believe that our system where an `Assumption` is only of concern to the subsystem that is using it, rather than a 'global' object, is more elegant.

7. CONCLUSIONS

Early experience with an experimental Ruby debugger suggests that it is possible to build debuggers on the Truffle platform without the compromises listed in the introduction.

- *Performance*: Runtime overhead is extremely low, and is arguably minimal relative to optimization supported by Truffle. Inactive AST node wrappers incur zero overhead when dynamically optimized along with program code. Activated debugging actions, whether expressed in Java or the implemented language, are subject to full optimization.
- *Functionality*: We have yet to see any limitations imposed by Truffle on the kind of debugging functionality represented in the prototype.
- *Complexity*: There is almost no interaction between the inserted debugging code and Truffle's mechanisms for compilation, dynamic optimization, and dynamic deoptimization. Debugging code need only follow standard Truffle techniques for AST construction and use the `Assumption` class correctly. The "wrapper" AST nodes that implement debugging actions are almost completely transparent to the flow of program execution around them.
- *Inconvenience*: We see no reason that such a debugging infrastructure should not be present in any environment, developmental or production.

This approach is applicable to the range of languages that can be implemented on Truffle. Nothing reported here other than the `set_trace_func` functionality is specific to Ruby.

Moreover, this approach places only modest demands on other parts of a language implementation. We anticipate language implementers adding debugging support incrementally during development, with evident advantage to both to themselves and early users. We also anticipate applying this general approach to supporting development tools other than debugging.

8. ACKNOWLEDGEMENTS

We gratefully acknowledge the contributions of the wider Virtual Machine Research Group at Oracle Labs, the Institute for System Software at JKU Linz and everyone else who has contributed to Graal and Truffle. Special thanks go to Mario Wolczko for his technical leadership, instructive comments on an early draft, and deep insight concerning SELF. We likewise thank Tom Rodriguez for his help writing the explanation of the JVM's deoptimization mechanism. Finally, we thank everyone who reviewed drafts, including Tobias Pape and the DYLA program committee.

9. REFERENCES

[1] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM, 2009.

[2] Brent Fulgham et al. The Computer Language Benchmarks Game. http://benchmarksgame.alioth.debian.org, 2013.

[3] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 1–10, New York, NY, USA, 2013. ACM.

[4] T. Felgentreff, M. Springer, et al. MagLev. http://maglev.github.io/, 2014.

[5] D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O'Reilly Media, Inc., 2008.

[6] A. Gaynor, T. Felgentreff, et al. Topaz. http://docs.topazruby.com/, 2014.

[7] N. Goto, P. Prins, M. Nakao, R. Bonnal, J. Aerts, and T. Katayama. BioRuby: bioinformatics software for the Ruby programming language. *Bioinformatics*, 26(20):2617–2619, 2010.

[8] M. Haupt and H. Schippers. A machine model for aspect-oriented programming. In E. Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 501–524. Springer Berlin Heidelberg, 2007.

[9] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, 1992.

[10] T. Kalibera and R. Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 ACM SIGPLAN International Symposium on Memory Management (ISMM)*, 2013.

[11] B. Keepers. Ruby at GitHub. In *Proceedings of MountainWest RubyConf*, 2013.

[12] T. Kotzmann and H. Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of 1st Conference on Virtual Execution Environments (VEE)*, 2005.

[13] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1), 2008.

[14] R. Krikorian. Twitter: From Ruby on Rails to the JVM. In *O'Reilly Open Source Convention (OSCON)*, 2011.

[15] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.

[16] R. LeFevre et al. PSD.rb from Layer Vault. https://cosmos.layervault.com/psdrb.html, 2013.

[17] Y. Matsumoto et al. Ruby programming language. http://ruby-lange.org/en/, 2014.

[18] C. Nutter, T. Enebo, O. Bini, N. Sieger, et al. JRuby. http://jruby.org/, 2014.

[19] M. Paleczny, C. Vick, and C. Click. The Java HotSpot server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposum*, 2001.

[20] E. Phoenix, B. Shirai, R. Davis, D. Bussink, et al. Rubinius. http://rubini.us/, 2014.

[21] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 944–953, New York, NY, USA, 2006. ACM.

[22] J. Rose, D. Coward, O. Bini, W. Cook, S. Pedroni, and J. Theodorou. JSR 292: Supporting dynamically typed languages on the Java platform. https://www.jcp.org/en/jsr/detail?id=292, 2008.

[23] J. R. Rose. Bytecodes meet combinators: invokedynamic on the JVM. In *Proceedings of the The 3rd workshop on Virtual Machines and Intermediate Languages*, 2009.

[24] P. Shaughnessy. *Ruby Under a Microscope*. No Starch Press, 2013.

[25] B. Shirai et al. RubySpec. http://rubyspec.org/, 2014.

[26] L. Stadler, G. Duboscq, T. Würthinger, and H. Mössenböck. Compilation queuing and graph caching for dynamic compilers. In *Proceedings of The 6th Workshop on Virtual Machines and Intermediate Languages (VMIL)*, 2012.

[27] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM.

[28] C. Wimmer and C. Seaton. One VM to rule them all. In *In Proceedings of the JVM Language Summit*, 2013.

[29] 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.

[30] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST interpreters. In *Proceedings of the 8th Dynamic Languages Sympsium*, 2012.