AST Specialisation and Partial Evaluation for Easy High-Performance Metaprogramming

Chris Seaton
Oracle Labs
chris.seaton@oracle.com

Abstract

The Ruby programming language has extensive metaprogramming functionality. Unlike most similar languages, the use of these features is idiomatic and much of the Ruby ecosystem uses metaprogramming operations in the inner loops of libraries and applications.

The foundational techniques to make most of these metaprogramming operations efficient have been known since the work on Smalltalk and Self, but their implementation in practice is difficult enough that they are not widely applied in existing implementations of Ruby and other similar languages.

The Truffle framework for writing self-specialising AST interpreters, and the Graal dynamic compiler have been designed to make it easy to develop high-performance implementations of languages. We have found that the tools they provide also make it dramatically easier to implement efficient metaprogramming. In this paper we present metaprogramming patterns from Ruby, show that with Truffle and Graal their implementation can be easy, concise, elegant and highly performant, and highlight the key tools that were needed from them.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments

Keywords Virtual Machines, Interpreters, Truffle, Graal, Ruby, Java

1. Introduction

Ruby is an imperative, object oriented, dynamically typed programming language with late-bound dispatch. It has similarities to languages such as Smalltalk and Python. Ruby has extensive metaprogramming functionality that allows parts of program execution to be controlled and observed using runtime data rather than data fixed in the source program. Examples include making method calls using a name that is a dynamic value, proxying method calls, dynamic code evaluation, and access to instance variables and method activation frames.

Many languages include some of this metaprogramming functionality, but a key difference in the Ruby programming language is cultural. While other programming languages tend to discourage use of metaprogramming, Ruby embraces it. Many examples can be found of metaprogramming use in Ruby code that is part of an application's inner loop, rather than just in offline operations such as set up and testing.

JRuby+Truffle (Seaton 2015) is a new implementation of Ruby using the Truffle (Würthinger et al. 2013b) framework for self-optimising AST interpreters, and Graal (Würthinger et al. 2013a), a dynamic compiler that can be used to compile Truffle's ASTs using partial evaluation. In JRuby+Truffle we have developed techniques to make metaprogramming patterns found in Ruby efficient (Seaton 2015; Marr et al. 2015; Seaton et al. 2014; Daloze et al. 2015). The techniques needed for that are in many cases incremental developments on techniques already used for conventional program operations, but we will suggest that in practice they become difficult enough to implement that they are left unoptimised, and that there is a small set of key tools that Truffle and Graal provide that make it much easier to implement efficient metaprogramming.

We are implementing Ruby's metaprogramming functionality in this paper, but many of the methods described are available in other languages such as Python and JavaScript.

1.1 Contributions

Previous publications have described in depth some aspects of implementing metaprogramming functionality in Ruby and other languages implemented using Truffle and Graal. This paper provides a survey of how all the major metaprogramming functionality of Ruby has been implemented, including both concise descriptions of techniques already presented in other publications, and techniques not yet described in the literature.

This paper makes the following research contributions:

- Identification of typical Ruby metaprogramming functionality and the patterns in which they are used.

Meta'16 October 30, 2016, Amsterdam, Netherlands

- A survey of existing published techniques for implementing this functionality.
- Techniques not previously described for the implementation of dynamic code evaluation and meta-access to instance variables.
- Identification of the small set of tools that Truffle and Graal provide that are useful for the efficient implementation of metaprogramming.

2. Ruby Metaprogramming Patterns

2.1 Idiomatic Ruby

We surveyed Ruby code in the standard library and in libraries important to the Ruby ecosystem to find examples of metaprogramming patterns where performance may be important.

We mention three libraries below. Active Support is part of the Rails web framework stack and provides utility methods for data structures, handling dates and times and so on. Chunky PNG is a library for reading and writing PNG image files. PSD.rb is a library for reading and writing Photoshop image files. Sidekiq is a library for enqueueing and processing background tasks.

Code samples that follow have in some cases been simplified for clarity and space.

The advisability of using these patterns when writing Ruby programs is irrelevant to a discussion of how to efficiently implement them. It could be argued that in some cases they are less safe, but more compact and less repetitive than the alternatives. Another valid criticism of the patterns is their low performance in existing Ruby implementations. The techniques described in this paper reduce that problem.

2.2 Dynamic method sends

In Ruby method calls, also called *message sends* as in the Smalltalk tradition, are usually made using a name written as a literal in the program source code. It is also possible to call a method based on a name that is a dynamic value in the running program, using the `#send` meta-method.

```
# Conventional send
object.method_name(arg1, arg2, ...)
# Metaprogramming send
object.send('method_name', arg1, arg2, ...)
```

Dynamic method sends are used to vary the method being called based on some condition. This can be used to separate the two concerns of deciding which method to call, from providing the arguments and making the call. In this example from Active Support's `Range` class, either the less-than or less-than-or-equal-to are used depending on the `exclude_end?` property of the range.

```
operator = exclude_end? ? :< : :<=
value.send(operator, last)
```

In that case the names appear as literals as in the source code and are selected between based on control flow. In more complex cases the method name can be dynamically generated. In this example from Chunky PNG there are multiple methods for decoding pixel data in an image depending on the bit depth. The string used in the send uses the `#{}` syntax to interpolate the bit depth into the method name.

```
send("decode_png_resample_#{bit_depth}bit_value")
```

2.3 Dynamic method proxies

Dynamic method sends allow the program to control which method is called at runtime. Ruby also allows the program to dynamically control what happens when a method is called. The `#method_missing` method is called when a method is called that does not exist. The program can then call a different method, or perform any other action, in response, forming a kind of dynamic method proxy.

In this example, the Active Support `Duration` class encapsulates an existing time value and provides new methods such as converting it to JSON format. In order to provide all the existing methods of a time object, it uses `#method_missing` to handle calls made to methods not explicitly defined and forwards them, using a dynamic `#send` to the encapsulated value. The `*args` syntax is Ruby's form of variadic arguments.

```
def method_missing(method, *args)
  @encapsulated_value.send(method, *args)
end
```

In this example from PSD.rb a method proxy is used to implement a form of module importing. The importing module defines `#method_missing` to look for methods in the `Color` module if it does not have definitions for them. As well as a dynamic send it also uses `#respond_to?` which indicates ahead of time whether a module has a given method.

```
def method_missing(name, *args)
  if Color.respond_to?(name)
    return Color.send(name, *args)
  end
end
```

2.4 Dynamic code evaluation

Ruby includes an `#eval` method to evaluate a dynamic string value as Ruby program source code at runtime.

Many Ruby applications are web servers and generate HTML pages as output through a template mechanism. The standard template library `erb` compiles templates to Ruby code that concatenates strings with runtime data in variables. To apply the template for each page the generated Ruby code is run with `#eval` and the runtime variables passed as an environment object.

```
eval(generated_template, variables)
```

2.5 Dynamic instance variable access

In the same way as Ruby allows a method to be called with a name that is a dynamic value, it is possible to read and write instance variables (fields) in objects using a dynamic value

as a name. In Ruby, instance variables are normally fully encapsulated, so the metaprogramming approach is the only way to read instance variables from outside of an object.

```
object.instance_variable_get('@variable_name')
object.instance_variable_set('@variable_name', value)
```

The standard library `Set` class uses dynamic access to instance variables to access implementation fields of other objects in basic operations such as equality. In this example the internal `Hash` (hash map) object in the set being compared for equality is accessed using `#instance_variable_get`.

```
def eql?(other)
  @hash.eql?(other.instance_variable_get(:@hash))
end
```

2.6 Dynamic frame access

Ruby allows access to materialised method activation frames, known as `Binding` objects. Local variables captured in a `Binding` can be read and written, and the object can be used as a context in which to execute code, including dynamic code using the `#eval` method.

The `erb` templating library uses the binding of the current method in order to set values for variables in templates. The `#binding` method reifies the current lexical environment as a binding, which is passed into the method to render the template.

```
page_title = 'Home Page'
template.render(binding)
```

The use of the `#binding` method here gives a static indication that the lexical environment is needed as an object, but it is also possible to access an activation frame without any form of 'prior warning', by calling the `#binding` method on a closure which has captured it.

```
a = ...
closure = proc { } # does not refer statically to 'a'

# in a different method
closure.binding.local_variable_get('a')
                              # still able to read 'a'
```

This use case is interesting because it means that it cannot be statically determined that the binding object will be needed ahead of time, and the local variable `a` needs to be made available for all environments.

2.7 Dynamic object graph access

Ruby allows a running program to access all live objects in the object graph by calling `ObjectSpace.each_object` and passing it a callback. Sidekiq uses `each_object` to find all open file objects and to reopen them as part of rotating log files.

```
ObjectSpace.each_object(File) do |file|
  reopen file
end
```

All live objects in the program are returned by this method, whether they are accessible from global variables, current activation frames, frames only accessible via the binding mechanism as described above, or transitively from those roots.

2.8 Tracing

Ruby includes several interfaces for observing and intercepting a running program through tracing execution. `#set_trace_func` allows a method to be installed that is called for meta-events such as method entry, the program reaching new source code lines, and runtime calls.

Tracing is commonly used to implement debugging, as in the basic `debug` standard library module, and historically was used to collect information to report code coverage while running tests, although this functionality is now provided by a native code extension to the standard Ruby implementation to increase performance.

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

With both dynamic access to the object graph and tracing we were not able to find any evidence of people using tracing in production in the inner loop of compute intensive code. However it is always possible to use both features, so we could say that any implementation of Ruby is always constrained by having to be able to turn it on when needed.

3. Implementations of Ruby

The original and most commonly used implementation of Ruby is known as *MRI*. It is a bytecode interpreter written using C.

Two major alternative implementations of Ruby exist. *Rubinius* is a bytecode interpreter with a JIT — both written in C++ — and an additional layer that implements much of the core Ruby library in Ruby itself. *JRuby* runs on the JVM and implements Ruby using Java, emitting JVM bytecode for Ruby methods that are run repeatedly. JRuby was one of the motivating applications for the *invokedynamic* JVM bytecode instruction and associated utilities which have made it easier to implement techniques such as inline caching (described later) on the JVM.

This paper focuses on the implementation of *JRuby+Truffle*, a new implementation of Ruby based on code from both JRuby and Rubinius. JRuby+Truffle runs on the JVM, but instead of generating bytecode at runtime as JRuby does, it uses the Truffle framework to implement a self-optimising AST interpreter (described later) and the Graal dynamic compiler (also described later) to partially evaluate the AST to produce efficient machine code.

The Ruby language supports concurrent threads, but in MRI these are run with a global interpreter lock so that threads are never running in parallel. Rubinius, JRuby, and JRuby+Truffle all support parallel execution of threads with no global lock. Concurrent threads can complicate the im-

plementation of some metaprogramming functionality, and parallel threads even more so.

4. Foundational Techniques

In this section we describe some of the foundational techniques needed to understand our discussion of the efficient implementation of metaprogramming functionality.

4.1 Inline caching

In languages with some degree of late bound dispatch, the method that is actually called for a given name may depend on the class of the receiver. Determining the method to call may also involve a complex lookup operation that walks the class hierarchy.

To make these method calls more efficient it is possible to cache the method that will be called for a given receiver. If this cache is associated with a particular source location or bytecode instruction it is called an *inline cache* (Deutsch and Schiffman 1984). An inline cache that caches multiple receiver and method pairs rather than just one is a *polymorphic inline cache* (Hölzle et al. 1991).

4.2 Truffle - self-specialising AST interpreters

An *abstract syntax tree* is a tree representation of a program. An *AST interpreter* executes a program by recursively walking its AST. AST interpreters are usually thought of as slow. For example, MRI was initially an AST interpreter before a bytecode format was later introduced to improve performance.

Truffle is a framework for writing AST interpreters that increase their performance by replacing nodes with more specialized versions based on profiling information such as the actual types seen in a program. A simple example is a Truffle AST node for a type-generic add operation that optimises to one that performs simple integer arithmetic if integers are the only observed types. A *guard*, which is just a conditional statement, will usually be added to check that the condition on which we based our specialization is still valid each time the node is executed.

4.3 Dynamic optimization and deoptimization

The performance of interpreters, even those that use sophisticated bytecode dispatch loops or self-specialization and inline caching, are often very limited. *Dynamic optimization* (Deutsch and Schiffman 1984), or JIT compilation, can translate a program to machine code at run time. Hopefully this will reduce the overhead of the interpreter loop, and it provides opportunities to optimise and simplify code.

Dynamic deoptimization (Hölzle et al. 1992) is the counterpoint to optimization. A language implementation with dynamic deotpimisation can jump from optimised compiled code back into the unoptimised interpreter. Consequently, the compiler can perform more aggressive speculative optimizations (Duboscq et al. 2013) and avoid compiling any code for not-yet-seen or slow-path cases because it can always fall back to the interpreter.

4.4 Graal - partial evaluation

Truffle's dynamic optimization is implemented by the *Graal* compiler. Graal uses *partial evaluation* to compile a Truffle AST which has reached a stable state to machine code. Partial evaluation has a specific technical meaning, but it can be more simply described as aggressive inlining and constant folding.

5. Implementation

5.1 Dynamic method sends

An efficient implementation of dynamic method sends in Ruby has been previously described by Marr et al. (2015) as *dispatch chains*. A conventional polymorphic inline cache is designed to handle a single method name, but multiple classes of receiver object. Each time the cache is used the actual receiver is searched for in the cache and the method which was cached against that class is run. Dispatch chains generalise this single dimension (in the receiver class) data structure, to two dimensions by adding the method name. Now the cache maps tuples of method names and receiver classes to methods, allowing the method name to vary in the same way that the receiver does.

In JRuby+Truffle we implement `#send` using such a dispatch chain. In fact, we implement all method calls as dispatch chains, and rely on Graal's partial evaluation to automatically remove the extra degree of freedom that in most cases is not needed. In the case of `#send`, which often varies in method name but not receiver class, we can apply the same technique to eliminate the receiver class from the cache key.

5.2 Dynamic method proxies

The efficient implementation of method proxies in Ruby was also previously described by Marr et al. (2015), using a modification of their dispatch chain technique. In dispatch chains, a cache maps a tuple of method name and receiver class to the method handle that will be called. To support `#method_missing`, we augment the range of the cache to indicate whether `#method_missing` will be called. Such a modification is necessary to support both normal methods and `#method_missing` calls as `#method_missing` calls require special argument handling (the first argument is the target method name followed by the arguments for that method).

5.3 Dynamic code evaluation

In other implementations of Ruby the `#eval` method parses the supplied Ruby code from scratch on each invocation before executing it. We applied the original inline caching hypothesis to this problem, and reasoned that strings passed to `#eval` are potentially stable. We built a specialized polymorphic inline cache that maps from source code strings to

versions of this code that have already been parsed and compiled to Truffle AST methods. The source code is treated in the same way as the method name.

5.4 Dynamic instance variable access

JRuby+Truffle implements object instance variables using a sophisticated object storage model (Wöß et al. 2014). Ruby is a class-based object oriented language but because metaprogramming functionality like `#instance_variable_set` can create new instance variables, the class of an object is not a reliable way to determine what instance variables it has. Instead, we actually discard all static class hierarchy information and rely on recreating the actual structure of objects through a graph of hidden classes and transitions between them as new instance variables are added to existing objects.

Again, this is a problem of mapping dynamic names into another value, which here is the location of the instance variables storage in an object. We solve the problem using yet another form of dispatch chains. In this case, the flexibility of the hidden class system we are using means that for a given instance variable name the location at which it is stored in objects can vary even if they have the same class, so the domain for this cache is the hidden class, or shape, of the object, not its logical Ruby class.

5.5 Dynamic frame access

In MRI method activation frames are heap-allocated objects that form a stack parallel to the actual machine stack which runs the interpreter. Alternative implementations of Ruby such as JRuby have attempted to put Ruby local variables onto the machine stack (or the host virtual machine stack, which normally re-uses the machine stack). However this process, already requiring additional work to implement, is deeply complicated by the multiple ways to obtain activation frames as objects (described in Section 2), and by the limitation of the JVM that it is not possible to read virtual machine local variables from outside their scope. Such factors mean that in many cases it cannot be determined that local variables will never be accessed via metaprogramming and subsequently it cannot be determined that it is safe to store them on the virtual machine stack.

Using the Truffle framework, JRuby+Truffle exclusively uses heap objects to store method activation frames, and does not attempt to manually store local variables on the stack. This means that activation frames are always available to be accessed as a Ruby `Binding` object, because they are already heap allocated objects. Graal then performs the job of storing the contents of the activation frame objects on the machine stack where possible, through sophisticated partial escape analysis (Stadler 2014). This process can be speculative and optimistic, being applied even when it is not certain that the activation frame will never be used as a binding, because through deoptimization Graal can re-materialise the activation frame as a heap object when required.

5.6 Dynamic object graph access

To provide a list of all live objects we must be able to access all the locations where objects could be stored. The object storage model as described earlier allows us to examine an object to find all other objects it references. Likewise, global variables and other such simple program state is easy to traverse. The final place where objects can be referenced from is the program stack and method activation frames. As previously described, in JRuby+Truffle these are conceptually heap allocated objects. Truffle provides an API which allows the current stack of activation frame objects to be iterated, deoptimising and re-materialising them if they were previously optimised away.

The task is complicated because in implementations of Ruby with concurrent threads there may be multiple call stacks which need to be visited to find roots, and multiple mutators modifying the graph as it is visited. If the threads are parallel then some kind of pre-emption is needed to interrupt them from the thread wanting to visit their stacks.

Daloze et al. (2015) describe *guest-language safepoints* as a tool to solve these two problems. The technique conceptually allows one thread to synchronously pause all other threads and to execute some arbitrary semantic action, provided by the initiating thread as a Java lambda.

This solves the problem of needing to visit another thread's call stack because the lambda sent to all other threads can be the same code to iterate stack frames as was used by the initiating thread in the single-threaded case. The lambda is executed by the other threads in their own call stack as if the action were a normal part of their program, so when the same lambda is executed on each thread it sees the stack of that thread.

The technique also solves the second problem of needing to pause the program to prevent the object graph from continuing to be modified as live objects are collected, because threads can be made to wait at a barrier until they have all finished before they continue to execute user code.

5.7 Tracing

An efficient implementation of tracing in Ruby has been previously described by Seaton et al. (2014), re-using the self-specialization and deoptimization capabilities in Truffle and Graal.

To support tracing, the JRuby+Truffle AST includes nodes in each location where a tracing event could be raised, such as method entry, moving from one source line to another, runtime calls and so on. When an AST for a method is first generated, these nodes are in an inactive state and do nothing. To enable tracing, the trace nodes are replaced with active nodes, that call the installed trace method.

The principle in Truffle is that trees are self-specialising, rather than being specialized from afar, as this is a simpler model to understand. Therefore trace nodes are activated lazily, replacing themselves with active versions the first

time they are used. Trace nodes can monitor for the need to replace themselves using the `Assumption` class provided by Truffle. Assumption objects represent a flag that can be set to indicate some action should be performed, but they integrate with the deoptimization mechanism in Graal so that in optimised code no check of the flag needs to be made. Instead, when the flag is set optimised code that would have checked the flag is deoptimised, storing the explicit flag check.

If the installed trace method is later removed, trace nodes can again replace themselves with inactive versions and restore the original behaviour and performance.

5.8 Storing state in Truffle

Five of the seven techniques that we have described need to store state in some sense. Inline caches need to be located somewhere alongside the program code that they support. In custom interpreters such as MRI this requires specialized bytecode instruction formats, but in JRuby which runs on the JVM they cannot define their own bytecode.

Truffle solves this problem by giving you consistent and universal access to a place to store state. A Truffle program is made up from a tree of nodes. Each node is a Java object that can contain mutable fields, and Truffle interpreters like JRuby+Truffle can store arbirary state in these fields.

5.9 Implementing caches in Truffle

Four of the seven techniques that we have described use sophisticated versions of multi-dimension inline caching mechanisms to efficiently implement metaprogramming functionality. While it would be possible to build these caches in many language implementation systems, we know from experience that it is usually prohibitively difficult.

This example shows an approximation[1] of the code for caching metaprogramming access to an instance variable via `#instance_variable_get`, using the DSL that Truffle provides (Humer et al. 2014). Our subclass of node has a method which performs the semantic action of the node, taking as parameters the receiver object and the dynamic name value. We use an object model with hidden classes so we need to map the name to a location within the current shape (layout) of the object (Wöß et al. 2014). We add three synthetic parameters that are annotated with `@Cached` and an expression that fills the cache on first use. These are caches for the name, the shape of the object, and the location that results from combining these. The `@Specialization` annotation on the execute method then includes code for guards that must hold before the cached parameters can be used. In this case we check that the shape of the object is as expected, and that the name is as expected. If these are true then we can use the cached location of the storage in the object.

```
class InstanceVariableGetNode extends Node {
  @Specialization(guards={
    "obj.getShape() == shape",
    "name.equals(cachedName)"
  })
  public Object executeGet(
    DynamicObject obj,
    String name,
    @Cached("name") String cachedName,
    @Cached("obj.getShape()") Shape cachedShape,
    @Cached("cachedShape.getLoc(name)")
        Location cachedLoc) {
      return obj.getValueAt(cachedLoc);
  }
}
```

Caches for other metaprogramming functionality are conceptually similar. For example the cache for `#eval` caches and then guards on the source code text, and caches the compiled source code.

Truffle will automatically chain a sensible number of these caching execute methods together to create a polymorphic inline cache, and will also automatically fallback to a megamorphic case if the case becomes too large. The fallback implementation performs the same actions as a cache miss but does not add a new entry to the chain. In this case our cache varies in two dimensions–the name and the receiver object's shape–so this is already a more sophisticated cache than found in existing implementations of Ruby, even in these few lines.

Cache entries can be removed when they are no longer applicable, for example if a cached method is overriden, by linking to caches from the methods that they reference.

5.10 Avoiding megamorphisation

Most of the techniques we have described are vulnerable to a problem which we will call *megamorphisation*. We have described how we can implement `#send` using a dispatch chain, but if we use a single dispatch chain for dynamic sends for an entire program the size of the chain will likely become very long. A long dispatch chain is a not a useful one, as looking up methods in the chain becomes time consuming and the code involved in implementing it becomes large. Also, we hope that in ideal cases where a dynamic send is in practice static with just one or two method names that we can produce code similar or identical to a conventional method call.

At some point creating longer and longer chains does not make sense, and no more entries are added. Accesses to the chain which miss will perform a full slow-path lookup. We say that the dispatch chain has become *megamorphic*. How do we prevent the dispatch chains for `#send`, `#eval` and `#instance_variable_get` from becoming megamorphic?

Truffle will split, or create multiple copies of, a method that contains megamorphic code. In the case of a method like `#send`, Truffle will in-practice create a separate copy of the

[1] The real implementation separates the two tasks of caching the instance variable name, and caching the location of the instance variable of that name in the object, into two separate nodes. This is because only the latter functionality is needed when the name is written statically in the source code.

method for every location that calls it, so that the caches can work independently. This is an automatic feature of Truffle, which knows about the size of caches and the degree of megamorphism in trees.

The splitting process is also nestable. If a method calls another method, which then calls `#send`, it is possible to split the call to `#send`, and then if that intermediate method is still used by multiple locations with different method names, to further split it to its multiple call sites.

In many cases, this will actually give you not just one instance of the `#send` dispatch chain for the whole application, and not just one for every location which calls `#send`, but actually for every location which leads to a call to it with a stable method name (if the name is ever stable).

6. Evaluation

Prior publications (Seaton 2015; Marr et al. 2015; Seaton et al. 2014; Daloze et al. 2015) have already evaluated the performance of the metaprogramming implementation techniques that they introduced. In this section we give a representative evaluation of one of the new techniques we have presented here: dynamic instance variable access.

6.1 Methodology

We evaluated the version of JRuby+Truffle that is distributed as part of GraalVM 0.15, JRuby 9.1.2.0, Rubinius 2.11 (later versions have had the JIT removed for maintenance reasons), and MRI 2.3.1. We also tried disabling the metaprogramming instance variable cache in the same version of JRuby+Truffle, so that we could compare against exactly the same implementation, just without the cache.

We ran benchmarks using the standard Ruby benchmark-ips tool, warming up for 2 seconds, and then measuring for 5, running 3 iterations in total. Error bars show ± one standard deviation.

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. JRuby was run using Oracle JDK 1.8.0_102.

6.2 Dynamic instance variable access

To evaluate dynamic instance variable access we used the already present example of `Set#eql?`, which uses metaprogramming to read an implementation field of a `Set` object being compared against for equality. First of all we evaluated the performance of the metaprogramming access to the instance variable compared to a conventional access to the same instance variable. Figure 1 shows how far the performance of the metaprogramming operation matches the conventional operation. In all cases except JRuby+Truffle, performance of the metaprogramming operation is around 80% of the conventional operation.

This showed that in JRuby+Truffle the metaprogramming operation is the same performance (within the margin of error) as the conventional operation. However, this would not be useful if JRuby+Truffle were generally slower than other implementations. We therefore looked at the performance of the code using metaprogramming between implementations. Figure 2 shows the slowdown of other implementations relative to JRuby+Truffle for a simple `#instance_variable_get`, and then for the whole `Set#eql?` operation which uses it.

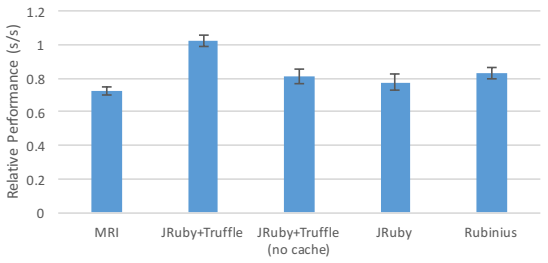

Figure 1. Relative performance of metaprogramming access to instance variables relative to conventional access

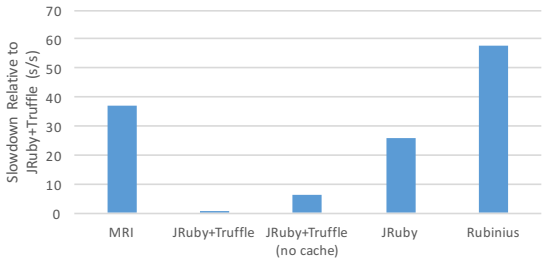

Figure 2. Slowdown of metaprogramming access to instance variables relative to JRuby+Truffle

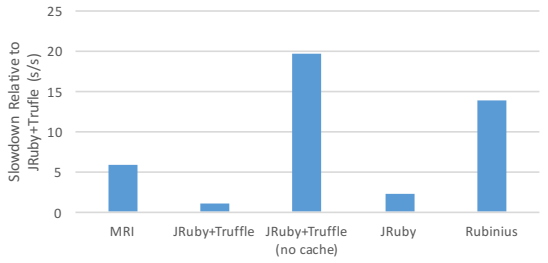

Figure 3. Slowdown of `Set#eql?` relative to JRuby+Truffle

For the simple operation, other implementations are an order of magnitude slower than JRuby+Truffle. For the set operation, other implementations are between 2.3× and 13.8× slower than JRuby+Truffle. When JRuby+Truffle's caching for metaprogramming is disabled performance is reduced by 6.1× and 19.6× respectively.

7. Conclusions

Looking at all the Ruby metaprogramming functionality which we have identified, we can pick out the key tools that

the AST specialization of Truffle and the partial evaluation of Graal provide which have made it possible to implement the metaprogramming functionality with reasonable effort. We can then further generalise away from Truffle and Graal to suggest a list of capabilities that any new framework for implementing languages should consider providing in order to support efficient metaprogramming for languages such as Ruby.

We believe that currently Truffle and Graal is the only system to both provide all these capabilities and to make them easy to use.

Somewhere to store state is a basic tool for efficient implementation of dynamic programming languages and in particular their metaprogramming functionality. Inline caches are one example of state, but others include profiling to observe the actual types, ranges and values of variables. In Truffle's AST nodes are Java objects and so provide an easy place to store arbitrary state.

Low-effort caching makes it tractable to add caching for all metaprogramming operations where they make sense, even if the caches need to complex, multi-dimensional dispatch chains. We know from the experience of JRuby and Rubinius that if adding caches is very complex then they may be omitted.

Dynamic optimization is needed for a baseline of performance, but for metaprogramming a powerful form of dynamic optimization such as *partial evaluation* is important to remove degrees of freedom that metaprogramming allows but are not used in practice, such as calls to `#send` where the method name is actually stable.

Dynamic deoptimization is needed so that code paths and functionality can be removed during optimization and then restored by returning to the interpreter if it is needed. For example this allows the interpreter to always support tracing, but for the optimised code to elide this functionality.

Automatic inlining and splitting is used to remove the overhead of intermediate method calls in functionality such as `#method_missing`, and to push state such as inline caches further down the call stack to reduce the degree of their polymorphism.

Programmatic access to frames allows metaprogramming functionality to read, write and reference method activation frames. In the case of Ruby we always need to be able to reference a method activation frame due to the `#binding` method.

Acknowledgments

Oracle, Java, and HotSpot are trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

JRuby+Truffle includes work from many people including Benoit Daloze, Kevin Menard, Petr Chalupa and Brandon Fish.

References

B. Daloze, C. Seaton, D. Bonetta, and H. Mössenböck. Techniques and applications for guest-language safepoints. In *Proceedings of the 10th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages and Tools*, 2015.

L. P. Deutsch and A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System, 1984.

G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *VMIL '13: Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*, 2013.

U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*. 1991.

U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, 1992.

C. Humer, C. Wimmer, C. Wirth, A. Wöß, and T. Würthinger. A domain-specific language for building self-optimizing AST interpreters. In *Proceedings of the International Conference on Generative Programming: Concepts and Experiences*, 2014.

S. Marr, C. Seaton, and S. Ducasse. Zero-overhead metaprogramming: Reflection and metaobject protocols fast and without compromises. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

C. Seaton. *Specialising Dynamic Techniques for Implementing The Ruby Programming Language*. PhD thesis, The University of Manchester, 2015.

C. Seaton, M. L. Van De Vanter, and M. Haupt. Debugging at Full Speed. In *Proceedings of the 8th Workshop on Dynamic Languages and Applications (DYLA)*, 2014.

L. Stadler. *Partial Escape Analysis and Scalar Replacement for Java*. PhD thesis, 2014.

A. Wöß, C. Wirth, D. Bonetta, C. Seaton, C. Humer, and H. Mössenböck. An object storage model for the Truffle language implementation framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, 2014.

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 *Onward! '13: Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, 2013a.

T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST interpreters. In *Proceedings of the 8th Symposium on Dynamic languages*, 2013b.