High-Performance Language Composition: Supporting C Extensions for Dynamic Languages

An abbreviated version of [10].

Grimmer Matthias[1], Chris Seaton[2], Thomas Würthinger[2] and Hanspeter Mössenböck[1]

[1] Johannes Kepler University, Linz, Austria
{grimmer,moessenboeck}@ssw.jku.at
[2] Oracle Labs
{chris.seaton,thomas.wuerthinger}@oracle.com

Abstract. Many dynamic languages such as Ruby offer functionality for writing parts of applications in a lower-level language such as C. These C extension modules are usually written against the API of an interpreter, which provides access to the higher-level language's internal data structures. Alternative implementations of the high-level languages often do not support such C extensions because implementing the same API as in the original implementations is complicated and limits performance.
In this paper we describe a novel approach for modular composition of languages that allows dynamic languages to support C extensions through interpretation. We propose a flexible and reusable cross-language mechanism that allows composing multiple language interpreters. This mechanism allows us to efficiently exchange runtime data across different interpreters and also enables the dynamic compiler of the host VM to inline and optimize programs across multiple language boundaries.
We evaluate our approach by composing a Ruby interpreter with a C interpreter. We run existing Ruby C extensions and show how our system executes combined Ruby and C modules on average over 3× faster than the conventional implementation of Ruby with native C extensions.

1 Introduction

Most programming languages offer functionality for calling routines in modules that are written in another language. There are multiple reasons why programmers want to do this, including to run modules already written in another language, to achieve higher performance than is normally possible in the primary language, or generally to allow different parts of the system to be written in the most appropriate language.

Dynamically typed and interpreted languages such as Perl, Python and Ruby often provide support for running extension modules written in the lower-level language C, known as C extensions. C extensions are written in C or a language, which can meet the same ABI such as C++, and are dynamically loaded and linked into the interpreter as a program runs. The APIs that these extensions are written against often simply provide direct access to the internal data structures of the primary implementation of the language. For example, C extensions for Ruby are written against the API of the original implementation of Ruby, known as MRI[3]. This API contains functions that allow C code to manipulate Ruby objects at a high level and to add C implementations of functions.

This model for C extensions worked well for the original implementations of these languages. As the API directly accesses the implementation's internal data structures, the interface is powerful, has low overhead, and was simple for the original implementations to add: all they had to do was make their header files public and support dynamic loading of native modules. However, as popularity of these languages has grown, alternative projects have increasingly attempted to re-implement them using modern virtual machine technology such as dynamic or just-in-time (JIT) compilation or advanced garbage collection. Such projects typically use significantly different internal data structures to achieve better performance, so the question therefore is how to provide the same API that the C extensions expect.

For these reasons, modern implementations of dynamic languages often have limited support for C extensions. For example, the JRuby [17] implementation of Ruby on top of the Java Virtual Machine (JVM) [4] had limited experimental support for C extensions until this was removed after the work proved to be too complicated to maintain and the performance too limited [2,6]. Lack of support for C extensions is often given as one of the major reasons for the slow adoption of modern implementations of such languages.

We would like to enable modern implementations of languages to support C extensions with minimal cost for implementing the existing APIs, and without preventing any advanced optimizations that these implementations use to improve the performance.

Our goal is to run multi-language applications on separate language interpreters, but within the same virtual machine and based on a common framework and using the same kind of intermediate representation. We propose a novel mechanism that allows composing these interpreters, rather than accessing foreign functions and objects via an FFI. Foreign objects and functions are accessed by sending language-independent messages. We resolve these messages at their first execution with language-specific IR snippets that implement efficient accesses to foreign objects and functions. This approach allows composing interpreters at their AST level and makes language boundaries completely transparent to VM performance optimizations.

To evaluate our approach we composed a Ruby interpreter with a C interpreter to support C extensions for Ruby. In our C interpreter, we substitute all invocations to the Ruby API at runtime with language-independent messages that use our cross-language mechanism. Our system is able to run existing unmodified C extensions for Ruby written by companies and used today in pro-

[3] MRI stands for Matz' Ruby Interpreter, after the creator of Ruby, Yukihiro Matsumoto.

duction. Our evaluation shows that it outperforms MRI running the same C extensions compiled to native code by a factor of over 3.

In summary, this paper contributes the following:

- We present a novel language interoperability mechanism that allows programmers to compose interpreters in a modular way. It allows exchanging data between different interpreters without marshaling or conversion.
- We describe how our interoperability mechanism avoids compilation barriers between languages that would normally prevent optimizations across different languages.
- We describe how we use this mechanism to seamlessly compose our Ruby and C interpreters, producing a system that can run existing Ruby C extensions
- We provide an evaluation, which shows that our approach works for real C extensions and runs faster than all existing Ruby engines.

2 System Overview

We base our work on Truffle [26], a framework for building high-performance language implementations in Java. Truffle language implementations are AST interpreters. This means that the input program is represented as an AST, which can be evaluated by performing an execution action on nodes recursively. All nodes of this AST, whatever language they are implementing, extend a common `Node` class.

An important characteristic of a Truffle AST is that it is *self-optimizing* [27]. Nodes or subtrees of a Truffle AST can replace themselves with specialized versions at runtime. For example, Truffle trees self-optimize as a reaction to type feedback, replacing an add operation node that receives two integers with a node that only performs integer addition and so is simpler. The Truffle framework encourages the optimistic specialization of trees where nodes can be replaced with a more specialized node that applies given some assumption about the running program. If an assumption turns out to be wrong as the program continues to run, a specialized tree can undo the optimization and transition to a more generic version that provides the functionality for all required cases. This self-optimization via tree rewriting is a general mechanism of Truffle for dynamically optimizing code at runtime.

When a Truffle AST has arrived at a stable state with no more node replacements occurring, and when execution count of a tree exceeds a predefined threshold, the Truffle framework partially evaluates [26] the trees and uses the Graal compiler [18] to dynamically-compile the AST to highly optimized machine code. Graal is an implementation of a dynamic compiler for the JVM that is written in Java. This allows it to be used as a library by a running Java program, including the Truffle framework.

In this research we have composed two existing languages implemented in Truffle, Ruby and C.

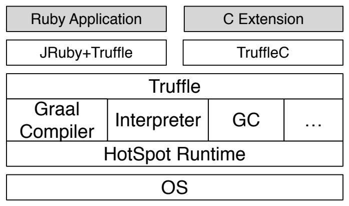


Fig. 1: The layered approach of Truffle: The Truffle framework on top of the Graal VM hosts JRuby+Truffle and TruffleC.

JRuby+Truffle: The Truffle implementation of Ruby [20]. JRuby is the foundation, on which our implementation is built, but beyond the parser and some utilities, little of the two systems are currently shared and JRuby+Truffle should be considered entirely separate from JRuby for this discussion.

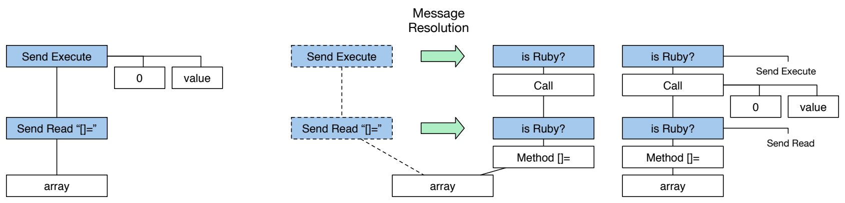
TruffleC: TruffleC [9] is the C language implementation on top of Truffle and can dynamically execute C code on top of a JVM.

Figure 1 summarizes the layered approach of hosting language implementations with Truffle. The Truffle framework provides reusable services for language implementations, such as dynamic compilation, automatic memory management, threads, synchronization primitives and a well-defined memory management. Truffle runs on top of the Graal VM [18,21], a modification of the Oracle HotSpot^{TM} VM. The Graal VM adds the Graal compiler but reuses all other parts, including the garbage collector, the interpreter, the class loader and so on, from HotSpot.

3 Language Interoperability on Top of Truffle

The goal of our work is to retain the modular way of implementing languages on top of Truffle but make them composable by implementing a cross-language interface. Given this interface, composing two languages, such as C and Ruby, requires very little effort. We do *not* want to introduce a new object model that all Truffle guest languages have to share, which is based on memory layouts and calling conventions. We introduce a common *interface* for objects that is based on code generation via ASTs. Our approach allows sharing language specific objects (with different memory representations and calling conventions) across languages. Finally, we want to make the language boundaries completely transparent to Truffle's dynamic compiler, in that a cross-language call should have exactly the same representation as an intra-language call. This transparency allows the JIT compiler to inline and apply advanced optimizations across language boundaries without modifications.

We use the mechanism to access Ruby objects from C and to forward Ruby API calls from the TruffleC interpreter back to the JRuby+Truffle interpreter.



(a) Using messages to access a Ruby object. (b) Performing message resolution. (c) Accessing a Ruby object after message resolution.

Fig. 2: Language independent object access via messages.

Using ASTs as an internal representation of a user program already abstracts away syntactic differences of object accesses and function calls in different languages. However, each language uses its own representation of runtime data such as objects, and therefore the access operations differ. Our research therefore focused on how we can share such objects with different representations across different interpreters.

In this paper we call every non-primitive entity of a program an *object*. This includes Ruby objects, classes, modules and methods, and C immediate values and pointers. An object that is being accessed by a different language than the language of its origin is called a *foreign object*. A Ruby object used by a C extension is therefore considered foreign in that context. If an object is accessed in the language of its origin, we call it a *regular object*. A Ruby object, used by a Ruby program is therefore considered regular. Object accesses are operations that can be performed on objects, e.g. method calls or property accesses.

3.1 Language-independent Object Accesses

In order to make objects (objects that implement `TruffleObject`) *shareable* across languages, we require them to support a common interface. We implement this as a set of *messages*:

Read: We use the *Read* message to read a member of an object denoted by the member's identity. For example, we use the *Read* message to get properties of an object such as a field or a method, and to read elements of an array.

Write: We use the *Write* message to write a member of an object denoted by its identity. Analogous to the *Read* message, we use it to write object properties.

Execute: The *Execute* message, which can have arguments, is used to evaluate an object. For example, it can evaluate a Ruby method or invoke the target of a C function pointer.

Unbox: If the object represents a boxed numeric value and receives an *Unbox* message, this message unwraps the boxed value and returns it. For example, if an *Unbox* message is sent to a Ruby Fixnum, the object returns its value as a 4 byte integer value.

We call an object *shareable* if we can access it via these language-independent messages. Truffle guest-language implementations can insert language-independent message nodes into the AST of a program and send these messages in order to access a foreign object. Figure 2a shows an AST that accesses a Ruby array via messages in order to store `value` at index 0. This interpreter first sends a *Read* message to get the array setter function `[]=` from the array object (in Ruby writing to an element in an array is performed via a method call). Afterwards it sends an *Execute* message to evaluate this setter function. In Figure 2a, the color blue denotes language-independent nodes, such as *message nodes*.

3.2 Message Resolution

The receiver of a cross-language message does *not* return a value that can be further processed. Instead, the receiver returns an AST snippet — a small tree of nodes designed for insertion into a larger tree. This AST snippet contains language-specific nodes for executing the message on the receiver. *Message resolution* replaces the AST node that sent a language-independent message with a language-specific AST snippet that directly accesses the receiver. After message resolution an object is accessed directly by a receiver-specific AST snippet rather than by a message.

During the execution of a program the receiver of an access can change, and so the target language of an object access can change as well. Therefore we need to check the receiver's language before we directly access it. If the foreign receiver object originates from a different language than the one seen so far we access it again via messages and do the message resolution again. If an object access site has varying receivers, originating from different languages, we call the access *language polymorphic*. To avoid a loss in performance, caused by a language polymorphic object access, we embed AST snippets for different receiver languages in an inline cache [12].

Figure 2b illustrates the process of *message resolution* and Figure 2c shows the AST of Figure 2a after message resolution. Message resolution replaced the *Read* message by a Ruby-specific node that accesses the getter function `[]=`. The *Execute* method is replaced by a Ruby-specific node that evaluates this getter method. Message resolution also places other nodes into this AST, which check whether the receiver is really a Ruby object.

Message resolution and building object accesses at runtime has the following benefits:

Language independence: Messages can be sent to any shareable object. The receiver's language of origin does not matter and messages resolve themselves to language-specific operations at runtime.

No performance overhead: Message resolution only affects the application's performance upon the first execution of an object access for a given language. Once a message is resolved and as long as the languages used remain stable, the application runs at full speed.

Cross-language inlining: Message resolution allows the dynamic compiler to inline methods even across language boundaries. By generating AST snippets for accessing foreign objects we avoid the barriers from one language to another that would normally prevent inlining.

3.3 Shared Primitive Values

In order to exchange primitive values across different languages we define a set of *shared primitive types*. We refer to values with such a primitive type as *shared primitives*. The primitive types include signed and unsigned integer types (8, 16, 32 and 64 bit versions) as well as floating point types (32 and 64 bit versions) that follow the IEEE floating point 754 standard.

3.4 JRuby+Truffle: Foreign Object Accesses and Shareable Ruby Objects

In Ruby's semantics there are no non-reference primitive types and every value is logically represented as an object, as in the tradition of languages such as Smalltalk. Also, in contrast to other languages such as Java, Ruby array elements, hash elements, or object attributes cannot be accessed directly but only via getter and setter calls on the receiver object. For example, a write access to a Ruby array element is performed by calling the `[]=` method of the array and providing the index and the value as arguments.

In our Ruby implementation all runtime data objects as well as all Ruby methods are shareable in the sense that they implement our message-based interface.

Ruby objects that represent numbers, such as `Fixnum` and `Float` that can be simply represented as primitives common to many languages, and also support the *Unbox* message. This message maps the boxed value to the relative shared primitive.

3.5 TruffleC: Foreign Object Accesses and shareable C Pointers

TruffleC can share primitive C values, mapped to *shared primitive values*, as well as pointers to C runtime data with other languages. In our implementation, pointers are objects that implement the message interface, which allows them to be shared across all Truffle guest language implementations. TruffleC represents all pointers (so including pointers to values, arrays, structs or functions) as `CAddress` Java objects that wrap a 64-bit value [11]. This value represents the actual referenced address on the native heap. Besides the address value, a `CAddress` object also stores type information about the referenced object. Depending on the type of the referenced object, `CAddress` objects can resolve the following messages: A pointer to a C struct/array can resolve *Read*/*Write* messages, which access members of the referenced struct/a certain array element. Finally, `CAddress` objects that reference a C function can be executed using the *Execute* message.

```c
typedef VALUE void*;
typedef ID void*;

// Define a C function as a Ruby method
void rb_define_method
(VALUE class, const char* name,
VALUE(*func)(), int argc);

// Store an array element into a Ruby array
void rb_ary_store
     (VALUE ary, long idx, VALUE val);

// Invoke a Ruby method from C
VALUE rb_funcall(VALUE receiver ID method_id,
    int argc, ...);

// Convert a Ruby Fixnum to C long
long FIX2INT(VALUE value);
```

Fig. 3: Excerpt of the `ruby.h` implementation.

```c
VALUE array = … ; // Ruby array of Fixnums
VALUE value = … ; // Ruby Fixnum

rb_ary_store(array, 0, value);
```

Fig. 4: Calling `rb_ary_store` from C.

TruffleC allows binding foreign objects to pointer variables declared in C. Hence, pointer variables can be bound to `CAdress` objects as well as shared foreign objects.

4 C Extensions for Ruby

Developers of a C extension for Ruby access the API by including the `ruby.h` header file. We want to provide the same API as Ruby does for C extensions, i.e., we want to provide all functions that are available when including `ruby.h`. To do so we created our own source-compatible implementation of `ruby.h`. This file contains the function signatures of all of the Ruby API functions that were required for the modules we evaluated, as described in the next section. We believe it is tractable to continue the implementation of API routines so that the set available is reasonably complete.

Figure 3 shows an excerpt of this header file.

We do not provide an implementation for these functions in C code. Instead, we implement the API by substituting every invocation of one of the functions at runtime with a language-independent message send or directly access the Ruby runtime.

We can distinguish between *local* and *global* functions in the Ruby API:

Local Functions: The Ruby API offers a wide variety of functions that are used to access and manipulate Ruby objects from within C. Consider the function `rb_ary_store` (Figure 4): Instead of a call, TruffleC inserts message nodes into the AST that are sent to the Ruby array (`array`). The AST of the C program (Figure 4) now contains two message nodes (namely a *Read* message to get the array setter method `[]=` and an *Execute* message to eventually execute the setter method, see Figure 2a). Upon first execution these messages are resolved (Figure 2b), which results in a TruffleC AST that embeds a Ruby array access (Figure 2c).

Global Functions: The Ruby API offers various different functions that allow developers to manipulate the global object class of a Ruby application from C or to access the Ruby engine.

The API includes functions to define global variables, modules, or global functions (e.g., `rb_define_method`) etc. In order to substitute invocations of these API functions, TruffleC accesses the global object of the Ruby application using messages or directly accesses the Ruby engine.

Given this implementation of the API we can run C extensions without modification and are therefore compatible with the Ruby MRI API.

5 Evaluation

We evaluated the performance in terms of running time for our implementation of Ruby and C extensions against other existing implementations of Ruby and its C extension API. Ruby is primarily used as a server-side language, so we are interested in peak performance of long running applications after an initial warm-up.

5.1 Benchmarks

We wanted to evaluate our approach on real-world Ruby code and C extensions that have been developed to meet a real business need. Therefore we use the existing modules `chunky_png` [23] and `psd.rb` [19], which are both open source and freely available on the RubyGems website. `chunky_png` is a module that includes routines for resampling, PNG encoding and decoding, color channel manipulation, and image composition. `psd.rb` is a module that includes color space conversion, clipping, layer masking, implementations of Photoshop's color blend modes, and some other utilities.

Both modules have separately available C extension modules to replace key routines with C code, known as `oily_png` [24] and `psd-native` [14], which allows us to compare the C extension against the pure Ruby code. There are 43 routines in the two gems for which a C extension equivalent is provided.

5.2 Compared Implementations

The standard implementation of Ruby is known as **MRI**, or CRuby. It is a bytecode interpreter, with some simple optimizations such as inline caches for method dispatch. MRI has excellent support for C extensions, as the API directly interfaces with the internal data structures of MRI. We evaluated version 2.1.2.

Rubinius is an alternative implementation of Ruby using a significant VM core written in C++ and using LLVM to implement a simple JIT compiler, but much of the Ruby specific functionality in Rubinius is implemented in Ruby. To implement the C extension API, Rubinius has a bridging layer. We evaluated version 2.2.10.

JRuby is an implementation of Ruby on the Java Virtual Machine. JRuby used to have experimental support for running C extensions, but after initial development it became unmaintained and has since been removed. We evaluated the last major version where we found that the code still worked, version 1.6.0.

JRuby+Truffle is our system, using Truffle and Graal. It interfaces to TruffleC to provide support for C extensions. To explore the performance impact of cross-language dynamic inlining, which is only possible in our system, we also evaluated JRuby+Truffle with this optimization disabled.

5.3 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, running 64bit Ubuntu Linux 14.04. Where an unmodified Java VM was required, we used the 64bit JDK 1.8.0u5 with default settings. For JRuby+Truffle we used the Graal VM version 0.3. Native versions of Ruby and C extensions were compiled with the system standard GCC 4.8.2.

We ran 100 iterations of each benchmark to allow the different VMs to warm up and reach a steady state so that subsequent iterations are identically and independently distributed. This was verified informally using lag plots [13]. We then sampled the final 20 iterations and took a mean of their runtime as the reported time. We summarize across different benchmarks and report a geometric mean [1].

5.4 Results

Figure 5 shows a summary of our results. We show the geometric mean speedup of each evaluated implementation over all benchmarks, relative to the speed at which MRI ran the Ruby code without the C extension. When using MRI the average speedup of using the C extension (*MRI With C Extension*, Figure 5) over pure Ruby code is around 11×. Rubinius (*Rubinius With C Extensions*, Figure 5) only achieves around one third of this speedup. Although Rubinius generally achieves better performance than MRI for Ruby code [20], its performance for C extensions is limited by having to meet MRI's API, which requires a bridging layer. Rubinius failed to make any progress with 3 of the benchmarks

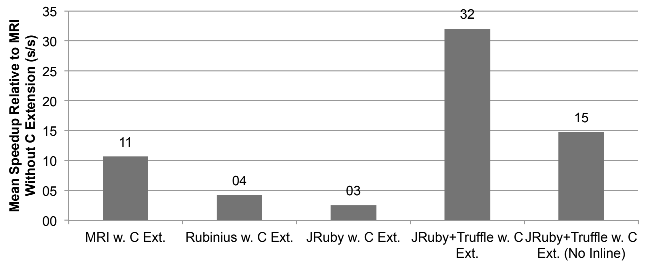


Fig. 5: Summary of speedup across all benchmarks.

in a reasonable time frame so they were considered to have timed out. The performance of JRuby (*JRuby With C Extensions*, Figure 5) is 2.5× faster than MRI running the pure Ruby version of the benchmarks without the C extensions. JRuby uses JNI [15] to access the C extensions from Java, which causes a significant overhead. Hence it can only achieve 25% of the *MRI With C Extension* performance. JRuby failed to run one benchmark with an error about an incomplete feature. As with Rubinius, 17 of the benchmarks did not make progress in reasonable time. Despite a 8GB maximum heap, which is extremely generous for the problems sizes, some benchmarks in JRuby were spending the majority of their time in GC or were running out of heap.

When running the C extension version of the benchmarks on top of our system (*JRuby+Truffle With C Extension*, Figure 5) the performance is over 32× better than MRI without C extensions and over 3× better than *MRI With C Extension*. When compared to the other alternative implementations of C extensions, we are over 8× faster than Rubinius, and over 20× faster than JRuby, the previous attempt to support C extensions for Ruby on the JVM. We also run all the extensions methods correctly, unlike both JRuby and Rubinius.
We can explain this speedup as follows:
In a conventional implementation of C extensions, where the Ruby code runs in a dedicated Ruby VM and the C code is compiled and run natively, the call from one language to another is a barrier that prevents the implementation from performing almost any optimizations. In our system the barrier between C and Ruby is no different to the barrier between one Ruby method and another. We found that allowing inlining between languages is a key optimization, as it permits many other advanced optimizations in the Graal compiler. For example, partial escape analysis [22] can trace objects, allocated in one language but consumed in another, and eventually apply scalar replacement [22] to remove the allocation. Other optimizations that benefit from cross language inlining include constant propagation and folding, global value numbering and strength reduction. When disabling cross-language inlining (*JRuby+Truffle With C Extension (No Inline)*, Figure 5) the speedup over MRI is roughly halved, although it is still around 15× faster, which is around 39% faster than *MRI With C Extension*.

In this configuration the compiler cannot widen its compilation units across the Ruby and C boundaries, which results in performance that is similar to MRI.

If we just consider the contribution of a high performance reimplementation of Ruby and its support for C extensions, then we should compare ourselves against JRuby. In that case our implementation is highly successful at on average over 20× faster. However we also evaluate against MRI directly running native C and find our system to be on average over 3× faster, indicating that our system might be preferable even when it is possible to run the original native code.

6 Related Work

We can compare our work against other projects that seek to compose two languages, and against existing support for C extensions or alternatives in Ruby implementations.

6.1 Unipycation

The work that is closest to our interoperability mechanism is that of Barret et al. [8], in which the authors describe a novel combination of Python and Prolog called Unipycation. We share the same goals, namely to retain the performance of different language parts when composing them and to find an approach that is applicable for any language composition.

However, our approach is quite different both in application and technique. We are concerned in this research in running existing C extensions, so there is immediate utility.

In contrast, Unipycation is a novel combination with no immediate industrial application. Unipycation composes Python and Prolog by combining their interpreters using glue code (which is specific to Python and Prolog) and compiles code using a meta-tracing JIT compiler. In contrast, we do not write glue code for a specific pair of interpreters but rather create this glue code at runtime for any pair of interpreters. Since the IR nodes themselves implement interpretation we can combine IR nodes of different origin without needing glue code.

6.2 Common Language Infrastructure

The Microsoft Common Language Infrastructure (CLI) supports writing language implementations that compile different languages to a common IR and execute it on top of the Common Language Runtime (CLR) [16]. The Common Language Specification (CLS) describes how language implementations can exchange objects across different languages. This standard defines a fixed set of data types and operations that all language implementations have to use. CLS-compliant language implementations generate metadata to describe user-defined types. This metadata contains enough information to enable cross-language operations and foreign object accesses. Also, the CLS specifies a set of basic language features that every implementation has to provide and therefore developers can rely on their availability in a wide variety of languages. This approach is different from ours because it forces CLS-compliant languages to use the same object model. Our approach, on the other hand, allows every language to have its own object model.

6.3 Interface Description Language

Interface Description Languages (IDLs) are also widely used to implement cross-language interoperability. To compose software components, written in different languages, programmers use an IDL to describe the interface of each component. Such IDL interfaces are then compiled to stubs in the host language and in the foreign language. Cross-language communication is done via these stubs [7]. However, an IDL is much more heavyweight. It is mainly targeted to remote procedure calls and often not only aims at bridging different languages but also at calling code on remote computers. Our approach is different because we neither require new interfaces nor a mapping between languages. Foreign objects can be accessed via messages without needing any boilerplate code that converts or marshals an object.

6.4 Language-neutral Object Model

Another approach towards cross-language interoperability are language-neutral object models. Wegiel and Krintz [25] propose a language-neutral object model, which allows different programming languages to exchange runtime data. In their system, the language-neutral objects are stored on an independent shared heap. Each language implementation then transparently translates a shared object to a private object. We argue that sharing objects between different languages and VMs does not require a special object model. Instead, objects should be shared between languages directly. Also, a shared object model would not solve the performance problems that Ruby engines, other than MRI, have when running C extensions.

6.5 Foreign Function Interfaces

Low-level APIs allow developers to integrate C code into another high-level language. Java developers can use a wide variety of different FFIs to integrate C code into Java, for example the Java Native Interface [15], Java Native Access [5], or the Compiled Native Interface [3]. VM engineers that implement new interpreters for dynamic languages in Java, e.g. the original JRuby without Truffle, could use these FFIs to support C extensions. However, the experience of JRuby, described below, shows that this approach is cumbersome and also has limited performance.

Rather than accessing precompiled C extensions via FFIs we follow a completely different approach. We use TruffleC to run these C extensions within a Truffle interpreter and use an efficient cross-language mechanism to compose the JRuby+Truffle and TruffleC interpreter. Our approach hoists optimizations such as cross-language inlining and performs extremely well compared to existing solutions.

6.6 Ruby C Extensions

MRI should have very straightforward support for C extensions as its implementation defines the API. However this does not mean that it poses no problems for MRI. As the interface is well established, MRI is now bound by it as much as any other implementation.

Rubinius supports C extensions through a compatibility layer. This means that in addition to problems that MRI has with meeting a fixed API, Rubinius must also add another layer that converts routines from the MRI API to calls on Rubinius' C++ implementation objects. The mechanism Rubinius uses to optimize Ruby code, an LLVM-based JIT compiler, cannot optimize through the initial native call to the conversion layer.

JRuby uses the JVM's FFI mechanism, JNI, to call C extensions. This technique is almost the same as used in Rubinius, also using a conversion layer, except that now the interface between the VM and the conversion layer is even more complex. In order to exchange data between the JVM and native code, JRuby must copy the data from the JVM onto the native heap. When the native data is then modified, JRuby must copy it back into the JVM. To keep both sides of the divide synchronized, JRuby must keep performing this copy each time the interface is passed.

7 Conclusion

We have presented a new approach to composing implementations of different language interpreters. The cross-language mechanism composes interpreters without additional infrastructure or glue code. We introduce an interface for shareable objects, which allows different language implementations to exchange objects. Language implementations access shared objects via object- and language-independent messages. Our resolving approach transforms these messages to an object- and language-specific access at runtime. The mechanism therefore refrains from converting objects, instead we adapt the IR of a program to deal with the foreign objects. The resolved IR of a program completely obliterates the language boundaries, which enables a JIT compiler to perform its optimizations across any language boundaries.

We use our mechanism to compose the JRuby+Truffle interpreter and the TruffleC interpreter to support C extensions for Ruby. Our evaluation demonstrates that this novel approach exhibits excellent performance. The peak performance of our system is over 3× faster compared to Ruby MRI when running benchmarks which stress interoperability between Ruby code and C extensions.

References

1. How Not To Lie With Statistics: The Correct Way To Summarize Benchmark Results. *Communications of the ACM*, 29, March 1986.
2. Ruby Summer of Code Wrap-Up. `http://blog.bithug.org/2010/11/rsoc`, 2011.
3. CNI (Compiled Native Interface). `http://gcc.gnu.org/onlinedocs/gcj/About-CNI.html`, 2013.
4. HotSpot JVM. Java version history (J2SE 1.3). `http://en.wikipedia.org/wiki/Java_version_history`, 2013.
5. Java Native Access (JNA). `https://github.com/twall/jna#readme`, 2013.
6. jruby-cext: CRuby extension support for JRuby. `https://github.com/jruby/jruby-cext`, 2013.
7. Common Object Request Brooker Architecture (CORBA) Specification. `http://www.omg.org/spec/CORBA/3.3/`, 2014.
8. E. Barrett, C. F. Bolz, and L. Tratt. Unipycation: A case study in cross-language tracing. In *Proceedings of VMIL '13*, New York, NY, USA.
9. M. Grimmer, M. Rigger, R. Schatz, L. Stadler, and H. Mössenböck. TruffleC: Dynamic Execution of C on a Java Virtual Machine. In *Proceedings of PPPJ '14*.
10. M. Grimmer, C. Seaton, T. Wuerthinger, and H. Moessenboeck. Dynamically composing languages in a modular way: Supporting c extensions for dynamic languages. In *Proceedings of MODULARITY '15*.
11. M. Grimmer, T. Würthinger, A. Wöß, and H. Mössenböck. An Efficient Approach for Accessing C Data Structures from JavaScript. In *Proceedings of ICOOOLPS '14*.
12. U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91*, pages 21–38.
13. T. Kalibera and R. Jones. Rigorous benchmarking in reasonable time. In *Proceedings of ISMM '13'*.
14. R. LeFevre. PSDNative, 2013.
15. S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.
16. E. Meijer and J. Gough. Technical overview of the common language runtime. *language*, 29:7, 2001.
17. C. Nutter, T. Enebo, O. Bini, N. Sieger, et al. JRuby. `http://jruby.org/`, 2014.
18. Oracle. OpenJDK: Graal project. `http://openjdk.java.net/projects/graal/`, 2013.
19. K. S. Ryan LeFevre et al. PSD.rb from Layer Vault, 2013.
20. C. Seaton, M. L. Van De Vanter, and M. Haupt. Debugging at full speed. In *Proceedings of DYLA'14*. ACM.
21. L. Stadler, G. Duboscq, H. Mössenböck, and T. Würthinger. Compilation queuing and graph caching for dynamic compilers. In *Proceedings of VMIL '12*.
22. L. Stadler, T. Würthinger, and H. Mössenböck. Partial escape analysis and scalar replacement for java. In *Proceedings of CGO '14*.
23. W. van Bergen et al. Chunky PNG, 2013.
24. W. van Bergen et al. OilyPNG, 2013.
25. M. Wegiel and C. Krintz. Cross-language, type-safe, and transparent object sharing for co-located managed runtimes. Technical Report 2010-11, UC Santa Barbara, 2010.
26. 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*. ACM.
27. T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST interpreters. In *Proceedings of DLS'12*.