Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages

Matthias Grimmer
Johannes Kepler University, Austria
grimmer@ssw.jku.at

Chris Seaton
Oracle Labs, United Kingdom
chris.seaton@oracle.com

Thomas Würthinger
Oracle Labs, Switzerland
thomas.wuerthinger@oracle.com

Hanspeter Mössenböck
Johannes Kepler University, Austria
moessenboeck@ssw.jku.at

Abstract

Many dynamic languages such as Ruby, Python and Perl offer some kind of 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, which run on the same VM and share the same form of intermediate representation – in this case abstract syntax trees. 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 $3\times$ faster than the conventional implementation of Ruby with native C extensions, and on average over $20\times$ faster than an existing alternate Ruby implementation on the JVM (JRuby) calling compiled C extensions through a bridge interface. We demonstrate that cross-language inlining, which is not possible with native code, is performance-critical by showing how speedup is reduced by around 50% when it is disabled.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments, Code generation, Interpreters, Compilers, Optimization

MODULARITY'15, March 16–19, 2015, Fort Collins, CO, USA.

http://dx.doi.org/10.1145/2724525.2728790

Keywords Cross-language, Language Interoperability, Virtual Machine, Optimization, Ruby, C, Native Extension

1. Introduction

Most programming languages offer some kind of 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 or native extensions. We use the term C extension, because as we will show that native execution is not an essential property of these modules. 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[1]. This API contains functions that allow C code to manipulate Ruby objects at a high level, but also includes routines that let you directly access pointers to internal data such as the character array in a string object. Figure 1 shows the `rb_ary_store` function, which is part of this interface and allows a C extension to store an element into a Ruby array.

```
typedef void* VALUE;
void rb_ary_store(VALUE ary, long idx, VALUE val);
```

Figure 1: Function to store a value into an array as part of the Ruby API.

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 implemen-

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

tations 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, partial evaluation, advanced garbage collection, and specialization. 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.

Some solutions to this problem are not realistic due to the scale at which these languages are used. For example, both Ruby and Python provide a Foreign Function Interface (FFI) with routines for dynamically linking and calling compiled C code and for marshaling values in C's type system. Here there is no need for a wrapper using the language specific API as C function can be called. However these FFIs have come later in the language development, and asking developers to rewrite their C extensions so that the FFI can use them is not realistic due to the large body of existing code.

One more realistic solution is to implement a bridging layer between the API that C extensions expect and the actual internal implementation of the language. However, this introduces costs for lowering the optimized data structures used by the more modern implementation in order to allow them to be accessed using the existing API. Performance is usually one goal of using C extensions, so adding a bridging layer is not ideal.

For these reasons, modern implementations of dynamic languages often have limited support for C extensions. For example, the JRuby [29] implementation of Ruby on top of the Java Virtual Machine (JVM) [6, 7] 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 [3, 9].

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. For example, the top-rated answer on the popular technical forum Stack Overflow for why not to use PyPy is limited C extension support [10]. In modular architectures, and particularly in web services, which often use these kind of dynamic languages, applications are often built on deep stacks of modules where dependencies on C extensions become difficult to remove. We have surveyed 421,817 Ruby modules (known as gems) available in the RubyGems repository. Among these, almost 14 % have a transitive dependency on a gem containing a C extension—although without actually running the gems it is not possible to clarify whether the C extensions are optional or required.

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 by adapting the abstract syntax tree (AST) of an application at runtime in such a way that it can deal with foreign objects and functions directly, i.e., we replace these language-independent messages 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.

Key benefits of our approach include:

Modular composition of languages: Our mechanism is language-independent and allows developers composing arbitrary language interpreters that support this message-based mechanism easily. In this paper we describe how we compose a Ruby interpreter and a C interpreter, but the techniques are immediately applicable to other languages with C extensions such as Python, and for other combinations of languages beside dynamic languages and C.

No object marshaling or conversion: Each language implementation can use its own implementation of internal data structures. However they can be efficiently exchanged across languages, as our mechanism avoids the conversion or marshaling of run-time data between languages completely and uses messages instead. Our C interpreter can access Ruby objects without lowering optimized data structures and vice versa.

Cross-language inlining: Our interoperability mechanism and shared ASTs makes the language boundaries caused by a foreign function call or a foreign object access completely transparent to the dynamic compiler. Removing language boundaries allows the compiler to inline and optimize across different languages.

To evaluate our approach we composed a Ruby interpreter with a C interpreter to support C extensions for Ruby. Our solution is source-compatible with the existing Ruby API. 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 almost unmodified C extensions for Ruby written by companies and used today in production. Our evaluation shows that it outperforms MRI running the same C extensions compiled to native code by a factor of over 3.

We describe our interoperability mechanism in the context of C extensions for Ruby. However, our approach will also work for other cross-language interfaces.

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 [43], 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* [42]. 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 and is used across all Truffle languages.

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 [43] the trees and uses the Graal compiler [30] 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.

JRuby+Truffle: Ruby is a dynamically-typed object-oriented language inspired by Smalltalk and Perl. Ruby is widely used with the Rails web framework for quick development of database-backed web applications, but it is also applied in fields as diverse as bioinformatics [19] and graphics processing [32].

The Truffle implementation of Ruby [33] was developed as a standalone implementation, but after open sourcing it has been merged into the existing JVM-based implementation of Ruby, JRuby. JRuby began as a simple AST interpreter ported directly from the original implementation of Ruby, MRI, but gained bytecode generation and then later was a key motivator of development of the `invokedynamic` instruction [31] for better support of dynamic languages on the JVM.

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: C is a statically typed language and was intended to be easily compilable to machine code. It offers low-level memory access, provides language constructs that map efficiently to machine code instructions and requires minimal run-time support.

TruffleC [22] is the C language implementation on top of Truffle. It dynamically executes C code on top of a JVM and performs well compared to industry standard C compilers such as GCC [14] or LLVM/Clang [12] in terms of peak-performance [22].

Despite C being a static language, TruffleC uses the *self-optimization* capability of Truffle: It uses polymorphic inline caches [24] to efficiently handle function pointer calls, profiles branch probabilities to optimistically remove never executed code, or profiles runtime values and replaces them by constants if they do not change over time.

TruffleC also has the ability to access native C libraries of which no source code is available, using the *Graal native function interface* [21]. This interface can directly access native functions from Java. However this functionality is not used in this evaluation.

Figure 2 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 [30, 35], a modification of the

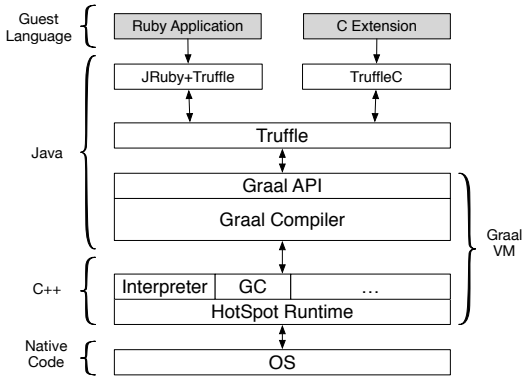


Figure 2: The layered approach of Truffle: The Truffle framework on top of the Graal VM hosts JRuby+Truffle and TruffleC.

Oracle HotSpot™ 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.

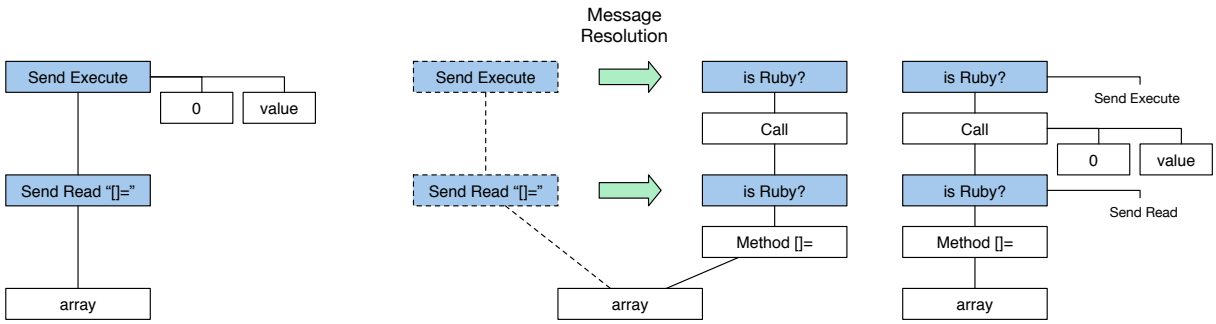
3. Language Interoperability on Top of Truffle

In previous work [20] we drafted the novel idea of a cross-language mechanism that allows us to compose arbitrary interpreters efficiently on top of Truffle. Our goal for this mechanism is that it retains the modular way of implementing languages on top of Truffle and meets the following criteria:

- Languages can be treated as modules and are composable. An implementation of a cross-language interface, such as C extensions for Ruby, requires very little effort because languages that support this mechanism are already interoperable.
- 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. Although some languages, such as Python and Ruby, have superficially similar object models, a shared object model is not applicable to a wider set of languages. For example, JRuby+Truffle uses a specific high-performance object model [41] to represent Ruby runtime data, whereas TruffleC stores C runtime data such as arrays and structures directly on the native heap [22] as is suitable for the semantics of C. 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, rather than lowering all objects to a common representation.
- 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 such as constant propagation and escape analysis across language boundaries without modifications.

In the following sections we describe in detail how we extend the Truffle framework with this mechanism. 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. Therefore our system includes calls both from Ruby to C and from C back to Ruby.

Using ASTs as an internal representation of a user program already abstracts away syntactic differences of object accesses and



(a) Using messages to access a Ruby object.

(b) Performing message resolution.

(c) Accessing a Ruby object after message resolution.

Figure 3: Language independent object access via messages.

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 3a 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 3a, the color blue denotes language-independent nodes, such as *message nodes*.

Given this mechanism, Truffle guest languages can access any foreign object that implements this message-based interface. If an object does not support a certain message we report a runtime error with a high-level diagnostic message.

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 if it is a non-final value, 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 a chain similar to a conventional inline cache [24], except that here the cache handles multiple languages as well as multiple classes of receivers.

Figure 3b illustrates the process of *message resolution* and Figure 3c shows the AST of Figure 3a 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. If not, our mechanism will again send the messages *Read* and *Execute*. Instead of executing the Ruby-specific nodes, the AST then executes the message-nodes again.

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. This mechanism is not limited to C extensions for Ruby but can be used for any other language composition.

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. Our approach creates a single AST that merges different language-specific AST parts. The language-specific parts are completely transparent to the JIT compiler. Removing the language boundaries allows the compiler to inline method calls even if the receiver is a foreign object. Widening the compilation unit across different languages is important [16, 36] as it enables further optimizations such as specialization and constant propagation.

3.3 Shared Objects and Shared Primitive Values

Like a regular object access, a foreign object access produces and returns a result. Our interoperability mechanism distinguishes between two types of values that a foreign object access can return:

Object types: If a foreign object access returns a non-primitive value, this object again has to be *shareable* in the sense that it understands the messages *Read*, *Write*, *Execute*, and *Unbox*. If the returned object is accessed later, it is accessed via these messages.

Primitive types: 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 [2]. The vast majority of languages use some of these types, and as they are provided by the physical architecture their semantics are usually identical. In the course of a foreign object access, a foreign language maps its language-specific primitive values to *shared primitive* values and returns them as language-independent values. When the host language receives a *shared primitive* value it again provides a mapping to host language-specific values.

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. Figure 3 shows how a Ruby array can be accessed via messages.

Ruby objects that represent numbers, such as `Fixnum` and `Float` that can be simply represented as primitives common to many languages, also support the *Unbox* message. This message simply maps the boxed value to the relative shared primitive. For example, a host language other than Ruby might send an *Unbox* message whenever it needs the object's value for an arithmetic operation.

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 [23]. 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 can resolve *Read*/*Write* messages, which access members of the referenced struct.
- A pointer to an array can resolve *Read*/*Write* messages that access a certain array element.
- Finally, `CAddress` objects that reference a C function can be executed using the *Execute* message.

When JRuby+Truffle accesses a function that is implemented within a C extension, it will use an *Execute* message to invoke it. Message resolution will bridge the gap between both languages automatically at runtime. The language boundaries are transparent to the dynamic compiler and it can inline these C extension functions just like normal Ruby functions.

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. TruffleC can then dereference these pointer variables via messages.

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 would be tractable to continue the implementation of API routines so that the set available is reasonably complete.

Figure 4 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:

4.1 Local Functions

The Ruby API also offers a wide variety of functions that are used to access and manipulate Ruby objects from within C. In the following we explain how we substitute the Ruby API functions `rb_ary_store`, `rb_iv_get`, `rb_funcall`, and `FIX2INT`:

- `rb_ary_store`:
  Normally TruffleC would insert *call nodes* for regular function calls, however, TruffleC handles invocations of these API functions differently. Consider the invocation of the `rb_ary_store`

```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);

// Get the Ruby internal representation of an
// identifier
ID rb_intern(const char* name);

// Get instance variables of a Ruby object
VALUE rb_iv_get(VALUE object,
    const char* iv_name)

// 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);
```

Figure 4: Excerpt of the `ruby.h` implementation.

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

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

Figure 5: Calling `rb_ary_store` from C.

function (Figure 5): Instead of a call node, TruffleC inserts message nodes that are sent to the Ruby array (`array`). The AST of the C program (Figure 5) 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 3a). Upon first execution these messages are resolved (Figure 3b), which results in a TruffleC AST that embeds a Ruby array access (Figure 3c).

- `rb_iv_get`:
  In contrast to Ruby object attributes, which are accessed via getter and setter methods, Ruby instance variables can be accessed directly. Therefore the substitution of `rb_iv_get` sends a *Read* message and provides the name of the accessed instance variable.
- `rb_funcall`:
  The function `rb_funcall` allows calling a Ruby method from within a C function. We substitute this call again by two messages. The first one is a *Read* message that resolves the method from the Ruby receiver object. The second message is an *Execute* message that finally executes the method.
- `FIX2INT`:
  We replace functions that convert numbers from Ruby objects to C primitives (such as `FIX2INT`, Fixnum to integer) by *Unbox* messages, sent to the Ruby object (`VALUE value`). As the naming convention suggests, `FIX2INT` is usually implemented as a C preprocessor macro. For the gems we studied this difference did not matter, and if it did we could implement it as a macro that simply called another function.

4.2 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. Also, these functions allow developers to retrieve the Ruby internal representation of an identifier (e.g. `rb_intern`). 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.

For instance, we substitute calls to `rb_define_method` and `rb_intern` as follows:

- `rb_define_method`:
  To define a new method in a Ruby class, developers use the `rb_define_method` function. TruffleC substitutes this function invocation and sends a *Write* message to the Ruby class object (first argument, `VALUE class`). The Ruby class object resolves this message and adds the C function pointer (`VALUE(*func)()`) as a new method. The function pointer (`VALUE(*func)()`) is represented as a `CAddress` object. When invoking the added function from within Ruby, JRuby+Truffle uses an *Execute* message and can therefore directly invoke this C function.
- `rb_intern`:
  `rb_intern` converts a C string to a reference to a shared immutable Ruby object which can be compared by reference to increase performance. TruffleC substitutes the invocation of this method and directly accesses the JRuby+Truffle engine. JRuby+Truffle exposes a utility function that allows resolving these immutable Ruby objects.

Given this implementation of the API we can run C extensions without modification and are therefore compatible with the Ruby MRI API. Given the interoperability mechanism presented in this paper, implementing this API via message-based substitutions was a trivial task: The implementation of TruffleC simply uses the interoperability mechanism and replaces invocations of Ruby API methods with messages. Besides these changes, no modifications of JRuby+Truffle or TruffleC were necessary to support C extensions for Ruby. This demonstrates that our cross-language mechanism is applicable in practice and makes language compositions easy.

4.3 Pointers to Ruby Objects

A normal pointer to a Ruby object is modeled in C as a simple Java reference to a *sharable* `TruffleObject`. If additional indirection is introduced and a pointer is taken to a Ruby object, TruffleC creates a `TruffleObjectReference` object (which itself is *sharable*) that wraps the pointee. Additional wrappers can achieve arbitrary levels of indirection. As with any object that does not escape, as long as the pointer objects are created and used within a single compilation unit (which may of course include multiple Ruby and C methods) the compiler can optimize and remove these indirections (see Section 5) and thus do not introduce a time or space overhead.

However, C also allows arithmetic on pointers. This is frequently used in Ruby C extensions, as the API allows a pointer to be obtained to internal data structures such as the C array that backs a Ruby string or array. It is then common to iterate directly

over these array pointers rather than using Ruby API functions, in order to achieve higher performance. Supporting this in JRuby is a major source of overhead as the JVM uses handles to specifically disallow pointers to object data via JNI. JRuby's solution was to copy string and array data into the native heap and back at every transition from Ruby to C [3] after a pointer has been requested.

To support this in TruffleC a second wrapper, a *sharable* `TruffleObjectReferenceOffset` object, holds both the object reference (`TruffleObjectReference`) and the offset from that address. Further pointer arithmetic just produces a new wrapper with a new offset. Any dereference of this object-offset wrapper will use the same messages to read or write from the object as a normal array access would.

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. Also, we wanted to use code that is computationally bound rather than I/O intensive, as our system does nothing to improve I/O performance.

To the best of our knowledge, there is no benchmark suite that evaluates the performance of native C extensions of Ruby extensively. Therefore we use the existing modules `chunky_png` [38] and `psd.rb` [32], which are both open source and freely available on the RubyGems website. `chunky_png` is a module for reading and writing image files using the Portable Network Graphics (PNG) format. It includes routines for resampling, PNG encoding and decoding, color channel manipulation, and image composition. `psd.rb` is a module for reading and writing image files using the Adobe Photoshop format. It includes routines for 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` [39] and `psd-native` [26], which allows us to compare the C extension against the pure Ruby code. These modules, written in C, heavily access Ruby objects, which makes them good candidates for our evaluation.

It is important to note that the C extensions are designed to produce the same output as the Ruby code, but they are not always identical in how they achieve this. For example, where the Ruby code might use array-indexing functions, the C code might use pointer arithmetic.

There are 43 routines in the two gems for which a C extension equivalent is provided. All 43 routines were set up in a benchmark harness for evaluation. Routine output was compared for each iteration to check for incorrect results. To focus on computational performance, I/O operations such as reading from files were mocked to read from memory.

5.2 Required Modifications

Running these gems on our system required just one minor modification for compatibility: we had to replace two instances of stack allocation of a variable-sized array with heap allocation via `malloc` and `free`. Variable-sized array allocation on the stack is a feature from C99 which TruffleC does not support yet. We will address completeness issues of our system in future work.

We also had to fix two bugs where an incorrect type was used, `int` instead of `VALUE`, causing a tagged integer to be used as if it was untagged. This was an implementation bug in the gem rather than a TruffleC incompatibility, as it was causing a different result between the Ruby module and the C extension on all Ruby language implementations. However, TruffleC was able to differentiate between the two types where the error was missed by the original authors when running with GCC[2].

Apart from these minor modifications we are running all of the native routines from two non-trivial gems unmodified.

5.3 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. Rubinius uses internal data structures and implementation techniques different from those in MRI. Most importantly, it uses C++ instead of C; so 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. It uses dynamic classfile generation and the `invokedynamic` instruction to JIT compile Ruby to JVM bytecode, and thus to machine code. 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. Our TruffleC implementation and the shared messaging API are not yet open source, but in the Ruby implementation there are only minor differences with the open source version of JRuby+Truffle at commit `df2aaff`.

In Section 6 we describe how each of the compared implementations support C extensions in more depth.

5.4 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.

Due to the extreme differences in performance, the benchmark input parameters were configured so that each iteration runs in the conventional Ruby interpreter without the C extensions for at least several seconds. 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 [25]. We then sampled the final 20 iterations and took a mean of their runtime as the reported time. Where we report an error we use a 95% confidence interval calculated using the Kalibera-Jones bootstrap method [25],

[2] We have reported this issue to the module's authors as `https://github.com/layervault/psd_native/pull/4`.

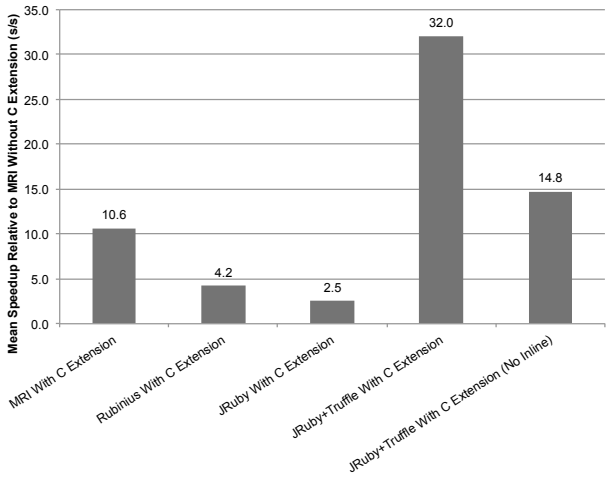


Figure 6: Summary of speedup across all benchmarks.

as implemented by Barrett et al [18]. Where we summarize across different benchmarks we report a geometric mean [1].

5.5 Results

Figure 6 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 6) over pure Ruby code is around $11\times$. Rubinius (*Rubinius With C Extensions*, Figure 6) only achieves around one third of this speedup. Although Rubinius generally achieves better performance than MRI for Ruby code [33], 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 in a reasonable time frame so they were considered to have timed out. The performance of JRuby (*JRuby With C Extensions*, Figure 6) is $2.5\times$ faster than MRI running the pure Ruby version of the benchmarks without the C extensions. JRuby uses JNI [27] 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 6) the performance is over $32\times$ better than MRI without C extensions and over $3\times$ better than *MRI With C Extension*. When compared to the other alternative implementations of C extensions, we are over $8\times$ faster than Rubinius, and over $20\times$ 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 [37] can trace objects, allocated in one language but consumed in another, and eventually apply scalar replacement [37] 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 6) the speedup over MRI is roughly halved, although it is still around $15\times$ 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.

Figure 7 shows detailed graphs for all benchmarks and Figure 8 shows tabular data. The first column of the tabular shortly describes the application of each benchmark that we evaluate. All other columns show the results of our performance evaluation of the different approaches. We show the absolute average time needed for a single run, the error, and the relative speedup to MRI without C extensions.

The speedup achieved for *MRI With C Extensions* compared to MRI running Ruby code (the topmost bar of each cluster, in red) varies between slightly slower at 0.69x (`psd-blender-compose`) and very significantly faster at 84x (`chunky-operations-compose`).

The speedup of *JRuby+Truffle With C Extensions* compared to MRI varies between 1.37x faster (`chunky-encode-png-image`) up to 101x faster (`psd-compose-exclusion`). When comparing our system to *MRI With C Extensions* we perform best on benchmarks that heavily access Ruby data from C but otherwise do little computation on the C side. These scenarios are well suited for our system because our compiler can easily optimize foreign object accesses and cross-language calls. In some benchmark such as `chunky-color-r` the entire benchmark, including Ruby and C, will be compiled into a single compact machine code routine. However, if benchmarks do a lot of computations on the C side and exchange little data the performance of our system is similar to *MRI With C extensions*.

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 $20\times$ faster. However we also evaluate against MRI directly running native C and find our system to be on average over $3\times$ faster, indicating that our system might be preferable even when it is possible to run the original native code.

6. Related Work

Although we are not aware of any other implementation of support for C extensions using an interpreter, 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 Barrett et al. [17], 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. As described earlier,

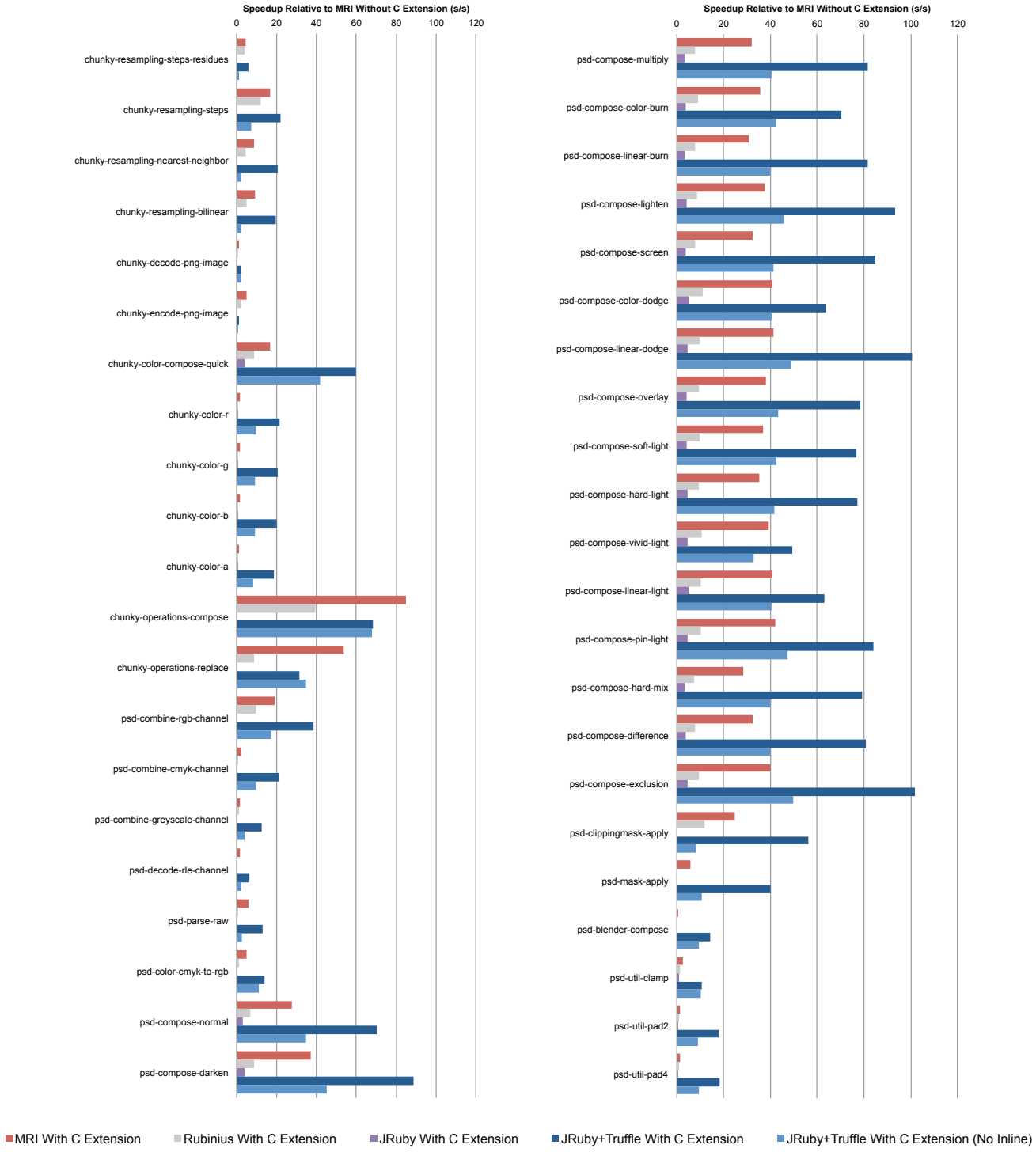


Figure 7: Speedup for individual benchmarks.

		MRI Without C Extesnion			MRI With C Extension			Rubinius With C Extension			JRuby With C Extension			JRuby+Truffle With C Extension			JRuby+Truffle With C Extension (No Inline)		
Benchmark	Application	Mean Time (s)	Error (± s)	Speedup (s/s)	Mean Time (s)	Error (± s)	Speedup (s/s)	Mean Time (s)	Error (± s)	Speedup (s/s)	Mean Time (s)	Error (± s)	Speedup (s/s)	Mean Time (s)	Error (± s)	Speedup (s/s)	Mean Time (s)	Error (± s)	Speedup (s/s)
chunky-resampling-steps-residues	Calculate offsets for scaling an image	4.1615	0.0006	1.0000	0.8884	0.0003	4.6844	1.0955	0.0013	3.7986		failed		0.6809	0.0075	6.1113	2.9943	0.0158	1.3898
chunky-resampling-steps	Calculate just the integer offsets for scaling	3.1536	0.0032	1.0000	0.1896	0.0003	16.6289	0.2635	0.0043	11.9675		failed		0.1445	0.0002	21.8320	0.4222	0.0002	7.4695
chunky-resampling-nearest-neighbor	Scale an image with no interpolation	4.8069	0.0644	1.0000	0.5449	0.0001	8.8212	1.0989	0.0030	4.3744		failed		0.2354	0.0022	20.4203	2.3568	0.0303	2.0396
chunky-resampling-bilinear	Scale an image with bilinear interpolation	4.9273	0.0262	1.0000	0.5445	0.0003	9.0496	1.0205	0.0103	4.8282		failed		0.2513	0.0038	19.6074	2.3411	0.0295	2.1047
chunky-decode-png-image	Decode a PNG image stream to pixel values	31.9720	0.3122	1.0000	32.0908	0.3098	0.9963	100.3601	2.3741	0.3186		failed		13.7589	0.0101	2.3237	15.7590	0.0120	2.0288
chunky-encode-png-image	Encode pixel values to a PNG image stream	48.0001	0.5826	1.0000	10.0598	0.0503	4.7715	21.8336	0.0560	2.1985		failed		34.8494	0.4705	1.3774	57.5937	0.4539	0.8334
chunky-color-compose-quick	Compose one colour value on top of another	20.0624	0.0049	1.0000	1.2092	0.0017	16.5917	2.2468	0.0022	8.9293	5.2565	0.0749	3.8166	0.3366	0.0003	59.6119	0.4796	0.0028	41.8315
chunky-color-r	Extract colour channel from packed 32bit pixel value	16.0452	0.0205	1.0000	11.2803	0.0221	1.4224	18.1816	0.1635	0.8825	25.5462	0.1627	0.6281	0.7479	0.0003	21.4537	1.6514	0.0133	9.7164
chunky-color-g	Extract colour channel from packed 32bit pixel value	15.1524	0.0397	1.0000	10.6781	0.0188	1.4190	18.4940	0.0289	0.8193	25.2880	0.2154	0.5992	0.7403	0.0006	20.4679	1.6792	0.0222	9.0238
chunky-color-b	Extract colour channel from packed 32bit pixel value	15.0363	0.0445	1.0000	10.1860	0.0042	1.4762	17.7032	0.0409	0.8494	24.7702	0.1849	0.6070	0.7503	0.0051	20.0391	1.6476	0.0234	9.1265
chunky-color-a	Extract colour channel from packed 32bit pixel value	13.8268	0.0514	1.0000	10.0445	0.0244	1.3766	17.9353	0.0412	0.7709	24.6747	0.1415	0.5604	0.7377	0.0006	18.7418	1.6336	0.0144	8.4640
chunky-operations-compose	Compose one image on top of another	18.2619	0.0181	1.0000	0.2156	0.0000	84.7060	0.4620	0.0003	39.5269		failed		0.2682	0.0018	68.1033	0.2689	0.0014	67.9133
chunky-operations-replace	Replace the contents of one image with another	7.5622	0.0063	1.0000	0.1411	0.0000	53.5976	0.8628	0.0004	8.7651		failed		0.2409	0.0067	31.3914	0.2182	0.0045	34.6492
psd-combine-rgb-channel	Decode RGB channel data into pixel values	9.1537	0.0268	1.0000	0.4760	0.0002	19.2303	0.9657	0.0099	9.4787		failed		0.2366	0.0014	38.6804	0.5347	0.0005	17.1209
psd-combine-cmyk-channel	Decode CMYK channel data into pixel values	8.7897	0.0080	1.0000	4.2160	0.0035	2.0848	11.3257	0.1693	0.7761		failed		0.4161	0.0030	21.1240	0.9220	0.0153	9.5333
psd-combine-greyscale-channel	Decode greyscale channel data into pixel values	3.8683	0.0014	1.0000	2.1293	0.0003	1.8167	2.8351	0.0004	1.3645		failed		0.3079	0.0035	12.5636	0.9681	0.0004	3.9960
psd-decode-rle-channel	Decode run-length-encoded data	3.6114	0.0474	1.0000	2.4145	0.0137	1.4957		failed			failed		0.5744	0.0029	6.2866	1.7794	0.0257	2.0296
psd-parse-raw	Decode raw encoded data	3.1876	0.0590	1.0000	0.5426	0.0054	5.8748	5.5173	0.0385	0.5777		failed		0.2460	0.0018	12.9551	1.3409	0.0172	2.3773
psd-color-cmyk-to-rgb	Convert from CMYK to RGB	25.4834	0.0056	1.0000	5.2850	0.0040	4.8218	24.4951	0.0086	1.0403		failed		1.8211	0.0107	13.9934	2.2905	0.0221	11.1257
psd-compose-normal	Compose two pixel values	9.5798	0.0017	1.0000	0.3490	0.0024	27.4506	1.4110	0.0055	6.7895	3.1666	0.0602	3.0253	0.1370	0.0004	69.9514	0.2765	0.0008	34.6531
psd-compose-darken	Compose two pixel values using a filter	12.4882	0.0021	1.0000	0.3358	0.0003	37.1902	1.4342	0.0015	8.7074	3.2332	0.0598	3.8625	0.1408	0.0003	88.6946	0.2770	0.0019	45.0837
psd-compose-multiply	Compose two pixel values using a filter	11.3139	0.0015	1.0000	0.3530	0.0021	32.0465	1.4093	0.0013	8.0278	3.2066	0.0541	3.5284	0.1390	0.0015	81.4240	0.2796	0.0006	40.4717
psd-compose-color-burn	Compose two pixel values using a filter	13.6794	0.0028	1.0000	0.3846	0.0007	35.5709	1.4947	0.0008	9.1519	3.5064	0.0574	3.9013	0.1951	0.0012	70.1328	0.3208	0.0003	42.6349
psd-compose-linear-burn	Compose two pixel values using a filter	11.2797	0.0020	1.0000	0.3664	0.0011	30.7812	1.4525	0.0009	7.7659	3.2223	0.0476	3.5005	0.1385	0.0003	81.4716	0.2822	0.0037	39.9707
psd-compose-lighten	Compose two pixel values using a filter	12.7023	0.0023	1.0000	0.3389	0.0004	37.4836	1.4511	0.0032	8.7538	3.1088	0.0444	4.0860	0.1360	0.0017	93.3989	0.2780	0.0027	45.6998
psd-compose-screen	Compose two pixel values using a filter	11.3777	0.0011	1.0000	0.3491	0.0012	32.5957	1.4418	0.0028	7.8913	2.9852	0.0430	3.8113	0.1344	0.0015	84.6555	0.2761	0.0008	41.2086
psd-compose-color-dodge	Compose two pixel values using a filter	15.6079	0.0031	1.0000	0.3809	0.0007	40.9720	1.4253	0.0018	10.9508	3.0824	0.0449	5.0635	0.2448	0.0014	63.7576	0.3856	0.0031	40.4821
psd-compose-linear-dodge	Compose two pixel values using a filter	13.9772	0.0024	1.0000	0.3381	0.0012	41.3362	1.3973	0.0011	10.0031	3.0626	0.0606	4.5639	0.1389	0.0010	100.6281	0.2850	0.0061	49.0516
psd-compose-overlay	Compose two pixel values using a filter	13.7380	0.0025	1.0000	0.3597	0.0007	38.1892	1.4684	0.0004	9.3555	3.2616	0.0351	4.2120	0.1755	0.0013	78.2569	0.3188	0.0040	43.0996
psd-compose-soft-light	Compose two pixel values using a filter	14.0483	0.0018	1.0000	0.3806	0.0003	36.9104	1.4558	0.0014	9.6496	3.2347	0.0554	4.3431	0.1829	0.0004	76.8086	0.3296	0.0037	42.6223
psd-compose-hard-light	Compose two pixel values using a filter	13.2856	0.0020	1.0000	0.3765	0.0002	35.2864	1.4366	0.0034	9.2477	2.8690	0.0455	4.6307	0.1725	0.0003	77.0179	0.3191	0.0011	41.6411
psd-compose-vivid-light	Compose two pixel values using a filter	15.7572	0.0020	1.0000	0.4015	0.0014	39.2481	1.4759	0.0008	10.6764	3.3599	0.0491	4.6898	0.3199	0.0004	49.2566	0.4782	0.0057	32.9510
psd-compose-linear-light	Compose two pixel values using a filter	15.2815	0.0031	1.0000	0.3725	0.0002	41.0285	1.5195	0.0020	10.0569	3.1666	0.0423	4.8258	0.2426	0.0003	62.9775	0.3796	0.0010	40.2621
psd-compose-pin-light	Compose two pixel values using a filter	15.1247	0.0016	1.0000	0.3601	0.0002	42.0034	1.4739	0.0010	10.2616	3.1750	0.0439	4.7636	0.1800	0.0026	84.0026	0.3203	0.0003	47.2277
psd-compose-hard-mix	Compose two pixel values using a filter	11.2739	0.0022	1.0000	0.3966	0.0012	28.4271	1.5099	0.0017	7.4667	3.1402	0.0598	3.5902	0.1426	0.0003	79.0594	0.2816	0.0033	40.0422
psd-compose-difference	Compose two pixel values using a filter	11.0951	0.0025	1.0000	0.3424	0.0001	32.4050	1.4161	0.0019	7.8352	2.9586	0.0324	3.7501	0.1372	0.0003	80.8976	0.2759	0.0019	40.2215
psd-compose-exclusion	Compose two pixel values using a filter	14.0561	0.0016	1.0000	0.3503	0.0007	40.1213	1.4592	0.0029	9.6328	3.0012	0.0400	4.6836	0.1383	0.0003	101.6350	0.2833	0.0038	49.6157
psd-clippingmask-apply	Apply a clipping mask to an image	6.5653	0.0031	1.0000	0.2642	0.0004	24.8485	0.5486	0.0006	11.9680		failed		0.1169	0.0002	56.1856	0.7980	0.0003	8.2272
psd-mask-apply	Apply an image mask to an image	10.3243	0.0192	1.0000	1.7499	0.0016	5.9001		failed			failed		0.2582	0.0033	39.9857	0.9720	0.0214	10.6212
psd-blender-compose	Blends two images using one of the compose modes	12.7997	0.0762	1.0000	18.6183	0.0019	0.6875		failed			failed		0.9081	0.0040	14.0959	1.3786	0.0199	9.2849
psd-util-clamp	Return a value or minimum or maximum bounds	27.3576	0.0123	1.0000	10.6073	0.0248	2.5791	20.0002	0.0430	1.3679	25.8236	0.2004	1.0594	2.5784	0.0043	10.6105	2.6979	0.0228	10.1401
psd-util-pad2	Round an integer to a multiple of 2	14.4757	0.0443	1.0000	9.7852	0.0098	1.4793	17.8647	0.0140	0.8103	22.6366	0.1362	0.6395	0.8032	0.0004	18.0237	1.6267	0.0141	8.8991
psd-util-pad4	Round an integer to a multiple of 4	15.1029	0.0098	1.0000	10.2633	0.0067	1.4715	17.8847	0.0408	0.8445	23.9633	0.1667	0.6303	0.8314	0.0005	18.1656	1.6326	0.0085	9.2508

Figure 8: Description of benchmarks and evaluation data.

current poor support for C extensions is often described as a limiting factor in high performance re-implementations of languages. 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. We allow foreign objects to be used from any language. When accessed for the first time, these objects dynamically generate foreign-language-specific IR (compiler intermediate representation) and embed it into the IR of the host application. Since the IR nodes themselves implement interpretation we can combine IR nodes of different origin without needing glue code. As in our approach, Unipycation maps foreign primitive values to primitive host values. However, Unipycation wraps non-primitive objects using *adapters* when passing them to a foreign language, while we allow foreign objects to be accessed directly by creating language-specific access IR at run time.

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) [28]. The Common Language Specification (CLS), as part of the CLI (specified in ECMA-355 [4]), describes a cross-language interoperability mechanism for these language implementations. The 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. Object accesses are dynamically generated using the object model of the foreign language. Therefore we believe that our approach is more flexible.

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 [13, 15, 34]. 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. Also, we target languages running on the same VM and thus use a more lightweight approach.

6.4 Language-neutral Object Model

Another approach towards cross-language interoperability are language-neutral object models. Wegiel and Krintz [40] 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. All VMs map their shared memory segments to the same virtual address and use shared objects directly via pointers. 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. A shared object model also does not bridge the language gap that prevents a JIT compiler from widening its compilation span across different languages using inlining. Interpreters of different languages can exchange runtime data via a shared object model, however, invocations across languages are compilation barriers that prevent inlining. Our approach of using message resolution completely obliterates the language boundaries and allows inlining across different languages.

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 [27], Java Native Access [8], or the Compiled Native Interface [5]. 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.

It is also possible to implement an FFI in the dynamic language so that it can interface directly with C code. However, as we described in the introduction this approach is simply not tractable due to the large volume of existing code using the current API. It also does nothing to allow the compiler to optimize through to the native code.

6.6 Ruby C Extensions

In Section 5.3 we briefly described how other Ruby implementations support 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. It might be more correct to say that the C extension is MRI as it was when it became widely used. MRI is now not free to change the API, and this means that it also not free to alter their 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. At this point many other useful optimizations no longer can be applied. Despite having a significantly more advanced implementation than MRI, it runs C extensions about half as fast as MRI (see Section 5).

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. For example, the Ruby C API makes it possible to take the pointer to the character array representing a string. MRI and Rubinius are able to directly return the actual pointer, but in JRuby using JNI it is not possible to obtain the address of the character array in a string. In order to implement this routine, JRuby must copy the string data from the JVM onto the native heap. When the native string 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. We believe that this is the cause for the benchmarks timing out in JRuby.

7. Future Work

There are several issues that remain to be tackled, and opportunities for further research:

Completeness We are currently working on completeness of JRuby+Truffle and TruffleC. Both interpreters do not yet fully support all language features. For example, JRuby+Truffle has limited support for the core library and does not support classes such as the `File` class, the `Time` class, or the `Socket` class. These classes are not critical for performance. TruffleC currently does not support multi-threaded programs and also has only limited support for GNU C extensions [11]. In case programmers use these unsupported features, we report a runtime error. In future work we will address these completeness issues and will therefore be able to support more Ruby gems that use C extensions.

Threading Ruby is a multi-threaded language with both native threads scheduled by the operating system and lightweight threads called *fibers*, scheduled by the runtime. MRI has a global interpreter lock which prevents parallel execution of multiple threads, which simplifies C extensions as they do not have to deal with parallel updates to runtime data structures. However, on a modern multi-core system this approach severely limits the proportion of available processing power that it utilized. Alternative implementations of MRI's API either have to also adopt a global interpreter lock, or provide a more fine-grained mechanism. Research into parallelizing Truffle languages and providing support for languages such as Ruby and C is ongoing.

Multi-tenant runtimes JRuby supports multiple isolated Ruby programs running in the same instance of a JVM. This is used for various purposes, such as improving startup performance by running subsequent programs in an already running JVM, or to amortize the JVM overhead of many small applications. Our current implementation has a global state in the native heap, which would need to be isolated for each runtime.

8. Conclusions

We have presented a new approach to composing implementations of different language interpreters as modules and hosted in a shared underlying VM. 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. TruffleC substitutes invocations of Ruby API functions and uses our cross-language mechanism for accessing Ruby objects instead. Our system is therefore compatible with MRI's Ruby API and can execute real-world applications.

Our evaluation demonstrates that this novel approach of composing languages and their interpreters exhibits excellent performance. Our cross-language mechanism makes language borders between Ruby an C completely transparent. The JIT compiler can inline C functions into Ruby and vice versa and therefore enables optimizations across language boundaries. The peak performance of our system is over $3\times$ faster compared to Ruby MRI when running benchmarks which stress interoperability between Ruby code and C extensions.

Acknowledgments

We thank all members of the Virtual Machine Research Group at Oracle Labs and the Institute of System Software at the Johannes Kepler University Linz for their valuable feedback on this work and on this paper. We especially thank Danilo Ansaloni, Michael Haupt, Christian Wirth and Tim Felgentreff for feedback on this paper.

Edd Barrett, Carl Friedrich Bolz and Laurence Tratt kindly gave us permission to use their unpublished implementation of the Kalibera-Jones method.

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

References

[1] How Not To Lie With Statistics: The Correct Way To Summarize Benchmark Results. *Communications of the ACM*, 29, March 1986.

[2] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008. .

[3] Ruby Summer of Code Wrap-Up. `http://blog.bithug.org/2010/11/rsoc`, 2011.

[4] Standard ECMA-335. Common Language Infrastructure (CLI). `http://www.ecma-international.org/publications/standards/Ecma-335.htm`, 2012.

[5] CNI (Compiled Native Interface). `http://gcc.gnu.org/onlinedocs/gcj/About-CNI.html`, 2013.

[6] HotSpot JVM. Java version history (J2SE 1.3). `http://en.wikipedia.org/wiki/Java_version_history`, 2013.

[7] IBM. Java 2 Platform. Standard Edition. `http://www.ibm.com/developerworks/java/jdk/`, 2013.

[8] Java Native Access (JNA). `https://github.com/twall/jna#readme`, 2013.

[9] jruby-cext: CRuby extension support for JRuby. `https://github.com/jruby/jruby-cext`, 2013.

[10] Why shouldn't I use PyPy over CPython if PyPy is 6.3 times faster? `http://stackoverflow.com/questions/18946662/why-shouldnt-i-use-pypy-over-cpython-if-pypy-is-6-3-times-faster`, 2013.

[11] Extensions to the C Language. `https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html`, 2014.

[12] Clang/LLVM. `http://clang.llvm.org/`, 2014.

[13] Common Object Request Brooker Architecture (CORBA) Specification. `http://www.omg.org/spec/CORBA/3.3/`, 2014.

[14] GCC (GNU C Compiler). `http://gcc.gnu.org/`, 2014.

[15] XPCOM Specification. `https://developer.mozilla.org/en-US/docs/Mozilla/XPCOM`, 2014.

[16] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005. ISSN 0018-9219. .

[17] E. Barrett, C. F. Bolz, and L. Tratt. Unipycation: A case study in cross-language tracing. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 31–40, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2601-8. . URL http://doi.acm.org/10.1145/2542142.2542146.

[18] E. Barrett, C. F. Bolz, and L. Tratt. Approaches to interpreter composition. To appear, 2014.

[19] 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. . URL http://bioinformatics.oxfordjournals.org/content/26/20/2617.abstract.

[20] M. Grimmer. High-performance language interoperability in multi-language runtimes. In *Proceedings of the 2014 Companion Publication for Conference on Systems, Programming, Applications: Software for Humanity*, SPLASH '14, New York, NY, USA, 2014. ACM.

[21] M. Grimmer, M. Rigger, L. Stadler, R. Schatz, and H. Mössenböck. An Efficient Native Function Interface for Java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, pages 35–44, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2111-2. . URL http://doi.acm.org/10.1145/2500828.2500832.

[22] M. Grimmer, M. Rigger, R. Schatz, L. Stadler, and H. Mössenböck. TruffleC: Dynamic Execution of C on a Java Virtual Machine. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, New York, NY, USA, 2014. ACM. . URL http://dx.doi.org/10.1145/2647508.2647528.

[23] M. Grimmer, T. Würthinger, A. Wöß, and H. Mössenböck. An Efficient Approach for Accessing C Data Structures from JavaScript. In *Proceedings of 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE - Workshop on Programming Language Evolution, 2014*, ICOOOLPS '14, New York, NY, USA, 2014. ACM. . URL http://dx.doi.org/10.1145/2633301.2633302.

[24] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In P. America, editor, *ECOOP'91 European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 21–38. Springer Berlin Heidelberg, 1991. ISBN 978-3-540-54262-9. . URL http://dx.doi.org/10.1007/BFb0057013.

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

[26] R. LeFevre. PSDNative, 2013. URL https://github.com/layervault/psd_native.

[27] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999. ISBN 0201325772.

[28] E. Meijer and J. Gough. Technical overview of the common language runtime. *language*, 29:7, 2001.

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

[30] Oracle. OpenJDK: Graal project. http://openjdk.java.net/projects/graal/, 2013.

[31] J. R. Rose. Bytecodes meet combinators: Invokedynamic on the jvm. In *VMIL '09: Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, pages 1–11, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-874-2. . URL http://portal.acm.org/citation.cfm?id=1711506.1711508.

[32] K. S. Ryan LeFevre et al. PSD.rb from Layer Vault, 2013. URL https://cosmos.layervault.com/psdrb.html.

[33] C. Seaton, M. L. Van De Vanter, and M. Haupt. Debugging at full speed. In *Proceedings of the Workshop on Dynamic Languages and Applications*, pages 1–13. ACM, 2014.

[34] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5, 2007.

[35] L. Stadler, G. Duboscq, H. Mössenböck, and T. Würthinger. Compilation queuing and graph caching for dynamic compilers. In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '12, pages 49–58, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1633-0. . URL http://doi.acm.org/10.1145/2414740.2414750.

[36] L. Stadler, G. Duboscq, H. Mössenböck, and T. Würthinger. Compilation queuing and graph caching for dynamic compilers. In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '12, pages 49–58, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1633-0. . URL http://doi.acm.org/10.1145/2414740.2414750.

[37] L. Stadler, T. Würthinger, and H. Mössenböck. Partial escape analysis and scalar replacement for java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 165:165–165:174, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2670-4. . URL http://doi.acm.org/10.1145/2544137.2544157.

[38] W. van Bergen et al. Chunky PNG, 2013. URL https://github.com/wvanbergen/chunky_png.

[39] W. van Bergen et al. OilyPNG, 2013. URL https://github.com/wvanbergen/oily_png.

[40] 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.

[41] 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*, PPPJ '14. ACM, 2014. . URL http://dx.doi.org/10.1145/2647508.2647517.

[42] 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*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1564-7. . URL http://doi.acm.org/10.1145/2384577.2384587.

[43] 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 the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 187–204. ACM, 2013.