JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Faculty of Engineering
and Natural Sciences

Cross-Language Interoperability in a Multi-Language Runtime

DISSERTATION

submitted in partial fulfillment of the requirements
for the academic degree

Doktor der technischen Wissenschaften

in the Doctoral Program in

Engineering Sciences

Submitted by
Dipl.-Ing. Matthias Grimmer, BSc.

At the
Institut für Systemsoftware

Accepted on the recommendation of
o.Univ.-Prof. Dipl.-Ing. Dr.Dr.h.c. Hanspeter Mössenböck (Supervisor)
Univ.-Prof. Dipl.-Ing. Dr. Walter Binder

Linz, October 2015

Abstract

In large-scale software applications, programmers combine different programming languages because it allows them to use the most suitable language for a given problem, to gradually migrate existing projects from one language to another, or to reuse existing source code. However, different programming languages have fundamentally different language implementations, which are hard to combine. The composition of these language implementations often results in complex interfaces between languages, insufficient flexibility, or poor performance.

We propose the *Truffle-Multi-Language Runtime* (TMLR), which can execute different programming languages and is able to compose them in a seamless way. The TMLR supports dynamic languages (e.g. JavaScript and Ruby) as well as statically typed low-level languages (e.g. C). It consists of Truffle language implementations, which translate source code to an intermediate representation that is executed by a shared virtual machine. The TMLR composes these different language implementations via *generic access*. *Generic access* is a language-agnostic mechanism that language implementations use to access foreign data or call foreign functions. It features language-agnostic messages that the TMLR resolves to efficient foreign-language-specific operations at run time. *Generic access* supports multiple languages, enables an efficient multi-language development, and ensures high-performance.

We evaluate *generic access* with three case studies. The first case study explains the transparent composition of JavaScript, Ruby, and C. The second case study shows an implementation of the C extensions API for Ruby. Finally, we show a case study that uses *generic access* to guarantee memory safety in C. We substitute native C allocations with managed data allocations, which demonstrates that the applications of *generic access* are manifold and not limited to cross-language interoperability. We can show that *generic access* guarantees good run-time performance. It avoids conversion or marshalling of foreign objects at the language boundary and allows the dynamic compiler to perform its optimizations across language boundaries.

Kurzfassung

Große Softwareprojekte werden oftmals in verschiedenen Programmiersprachen entwickelt, weil Teilprobleme in einer geeigneten Programmiersprache gelöst werden müssen, bestehende Projekte schrittweise in eine moderne Programmiersprache portiert werden müssen oder ein bestehender Programmcode wiederverwendet werden muss. Die Implementierungen verschiedener Programmiersprachen unterscheiden sich jedoch meist fundamental. Daher haben bestehende Verbindungsmechanismen in der Regel komplizierte Schnittstellen, sind nicht flexibel oder haben eine schlechte Laufzeitperformanz.

Wir stellen die *Truffle-Multi-Language Runtime* (TMLR) vor. Die TMLR ist eine Laufzeitumgebung, welche unterschiedliche Programmiersprachen ausführen und deren Implementierungen verbinden kann. Die Laufzeitumgebung unterstützt sowohl dynamisch typisierte Sprachen (z.B. JavaScript und Ruby) als auch statisch typisierte Sprachen (z.B. C) und besteht aus einzelnen Truffle-Implementierungen. Eine Truffle-Implementierung übersetzt ein Programm in eine Zwischenrepräsentation und führt diese auf einer gemeinsamen virtuellen Maschine aus. Die TMLR verwendet den *Generic Access*-Mechanismus um diese Implementierungen zu verbinden. Dieser Mechanismus ermöglicht einen sprach- und typunabhängigen Zugriff auf Objekte und Funktionen per Nachrichten. Die TMLR löst diese Nachrichten zur Laufzeit auf und ersetzt sie durch effiziente Zugriffsoperationen.

Wir evaluieren den *Generic Access* im Rahmen dreier Fallstudien. Die erste Fallstudie erklärt die transparente Verbindung von JavaScript, Ruby und C. Die zweite Fallstudie beschreibt die Implementierung des C Extensions API für Ruby. Als dritte Fallstudie zeigen wir, wie automatisch verwaltete Datenstrukturen für C verwendet werden können. Die Anwendungsgebiete des *Generic Access* sind vielfältig und nicht auf Interoperabilität zwischen Programmiersprachen beschränkt. Außerdem müssen Objekte an der Grenze zwischen Programmiersprachen nicht kopiert werden und der *Generic Access* ermöglicht dem dynamischen Übersetzer ein Programm unabhängig der verwendeten Sprachen zu optimieren. Das wiederum garantiert gute Laufzeitperformanz.

Contents

Chapter 1

Introduction

A multi-language application requires a system that can compose different language implementations. However, fundamentally different language implementations make this composition hard. We discuss the current state of cross-language interoperability, point out the open challenges, and finally describe the requirements for a novel solution. This section also gives a conceptual overview over the Truffle-Multi-Language Runtime *and summarizes the scientific contributions of this thesis.*

1.1 Problem Setting

In large-scale software development it is common that programmers write applications in multiple languages rather than in a single language [17]. Combining multiple languages allows them to use the most suitable language for a given problem, to gradually migrate existing projects from one language to another, or to reuse existing source code. There exists no programming language that is best for all kinds of problems [6, 17]. High-level languages allow representing a subset of algorithms efficiently but sacrifice low-level features such as pointer arithmetic and raw memory accesses. A typical example is business logic written in a high-level language such as JavaScript that uses a database driver written in a low-level language such as C. Programmers use cross-language interfaces to pick the most suitable language for a given part of a problem. Programmers also combine different languages because it reduces the risks when migrating software from one language to another. For example, programmers can gradually port legacy C code to Ruby, rather than having to rewrite the whole project at once. Finally, cross-language interoperability enables programmers to reuse existing source code. Due to the large body of existing code it is often not feasible to rewrite existing libraries in a different language. A more realistic approach is to interface to the existing code, which allows reusing it.

1.1.1 Problem Statement

Given our scenario of a large-scale software development where programmers mix different languages, it is necessary to compose different language implementations. We distinguish between two different approaches to programming language implementation: An *interpreter* takes a program as its input and executes the instructions on some machine, e.g. a virtual machine (VM) or some computer hardware. A *compiler* also takes a program as its input but translates it into some other language. The output of a compiler can be machine code, which is natively executed by hardware, or code that serves as input to another interpreter or compiler.

Early low-level languages are often implemented by a compiler that produces native code. For example, C code is normally compiled to machine code directly. On the other hand, many modern programming language implementations include elements of both, interpreter and compiler. For example, Java source code is first compiled to Java bytecode, which is then interpreted and dynamically compiled by the Java Virtual Machine (JVM)[1,2]. The JVM starts interpreting this bytecode and can eventually compile it (dynamic compilation) to machine code.

The composition of different languages poses challenges for language implementers as well as application developers:

Language implementation composition: Different implementations of programming languages execute programs differently (e.g. interpretation on a VM and native compilation) and also use, for example, different object model implementations (dynamic objects of JavaScript and byte sequences of C), use different memory models (automatic memory management of the JVM and manual memory management in C), or use different security mechanisms (run-time errors in Java and segmentation faults in C). Cross-language interoperability needs to bridge code that is running on different implementations. For example, a VM needs a mechanism for calling native code and vice versa. Also, this mechanism needs to provide an interface for accessing foreign data and hereby bridge different strategies for memory management and different implementations of object models.

Multi-language development: Application programmers need an interface that allows them to switch execution from one language to another and to access foreign data and functions. This API needs to bridge the different languages on a source code level but also needs to bridge different language paradigms and features.

[1] *Java SE HotSpot at a Glance*, Oracle, 2015: `http://www.oracle.com/technetwork/articles/javase/index-jsp-136373.html`

[2] *IBM Developer Kits: Java Platform Standard Edition (Java SE)*, IBM, 2015: `http://www.ibm.com/developerworks/java/jdk`

Examples are object-oriented and non-object-oriented programming, dynamic and static typing, explicit and automatic memory management, or safe and unsafe memory access.

1.1.2 Problem Analysis

In the following, we analyze three existing approaches for cross-language interoperability between different language implementations: (1) foreign function interfaces (FFIs), (2) inter-process communication, and (3) multi-language runtimes. We identified these approaches as the most relevant techniques and we point out their major limitations, which are *restricted flexibility*, *complex APIs*, and *performance limitations.*

Foreign Function Interfaces

The different approaches for programming language implementation make the composition of languages complicated. It requires a mechanism that can integrate foreign code into a host application and therefore bridge the different implementations. Many modern VMs have an FFI that can integrate native code. An FFI links the native binaries into a VM and also offers an API, which allows the programmer to exchange data between the runtime and the native parts of an application.

The result is a mechanism that caters primarily to composing two *specific* languages rather than *arbitrary* languages. Also, the implementation of these interfaces requires runtime support, which means that the APIs often depend on the original implementation of the languages. For example, interpreted languages such as Perl, Python and Ruby provide support for running extension modules written in the lower-level language C, known as *C extensions* or *native extensions.* C extensions are written in C or a language, which can meet the same application binary interface (ABI) such as C++, and are dynamically loaded and linked into the VM of the high-level language 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, Ruby C extensions[3] are written against the API of the original Ruby implementation MRI[4]. This API contains functions that allow C code to manipulate Ruby objects at a high level.

This model for C extensions worked well for the original implementations of these languages. The API allows programmers to directly access the internal data structures

[3]*Ruby Language*, Yukihiro Matsumoto, 2015: `https://www.ruby-lang.org`
[4]MRI stands for Matz' Ruby Interpreter, after the creator of Ruby, Yukihiro Matsumoto.

of the runtime. Hence, the API is powerful and has low overhead. Also, the implementation of this API is simple; it makes header files public and supports dynamic loading of native modules. However, as popularity of dynamic languages has grown, alternative projects have increasingly attempted to re-implement these languages using modern VM technology such as dynamic or just-in-time (JIT) compilation and advanced garbage collection. Such projects typically use significantly different internal data structures to achieve better performance, which makes an implementation of a C extensions API hard. Modern implementations of these languages need to implement a bridging layer between the internal data structures and the C extensions API. However, this layer introduces costs for transforming the optimized data structures of the more modern implementation to low-level C data and vice versa. As performance is usually the primary goal of using C extensions, this bridging layer is not ideal. For these reasons, modern implementations of dynamic languages often have limited support for C extensions. For example, the JRuby[5] implementation of Ruby on top of the JVM had limited experimental support for C extensions. It used a bridging layer to lower Java objects to low-level C data and vice versa. However, this layer was complicated to maintain and the performance was poor. Eventually, the JRuby team removed the C extensions support completely[6,7].

The Ruby C extensions API, like many other FFIs, is inflexible because application programmers can only use it to integrate C code (or code that meets the same ABI). Furthermore, this interface is language implementation specific, which binds the programmer to MRI. Similar to other FFIs, the C extensions API requires the programmer to use a complex API, which distracts the programmer from the actual task at hand. However, the support for these interfaces is important. For example, lack of support for C extensions is often mentioned as one of the major reasons for the slow adoption of modern implementations of programming languages like Ruby or Python[8]. Finally, linking native code into the VM hinders the compiler to optimize an application across language boundaries, which limits performance [100].

Inter-Process Communication

Rather than composing language implementations on their implementation level, a message-based inter-process communication treats them as black boxes. Examples are the Common Object Request Broker Architecture [42, 104] or Apache's Thrift [80, 94].

[5] *JRuby*, Charles Nutter and Thomas Enebo and Ola Bini and Nick Sieger and others, 2015: `http://jruby.org`

[6] *Ruby Summer of Code Wrap-Up*, Tim Felgentreff, 2015: `http://blog.bithug.org/2010/11/rsoc`

[7] *JRuby C Extensions: CRuby extension support for JRuby*, GitHub repository, 2015: `https://github.com/jruby/jruby-cext`

[8] *Why shouldn't I use PyPy over CPython if PyPy is 6.3 times faster?*, Stackoverflow, 2015: `http://stackoverflow.com/questions/18946662/why-shouldnt-i-use-pypy-over-cpython-if-pypy-is-6-3-times-faster`

Application programmers use a language-agnostic interface description language (IDL) to generate an interface between languages. This interface marshals data to and from a common wire representation.

However, programmers that use these approaches need to learn and apply an IDL, which adds a usability burden. Also, inter-process communication imposes a performance overhead. The marshalling of data introduces a copying overhead whenever language implementations exchange data. Finally, language implementations need to treat each other as black boxes, hence, the compiler cannot optimize a program across language boundaries.

Multi-Language Runtime

Another approach for cross-language interoperability is to compose language implementations that are running on a shared VM. For example, Microsoft's Common Language Runtime (CLR) [16, 56, 70] as well as RPython [15] are runtimes that can host implementations for different languages.

The CLR composes the individual languages by compiling them to a shared intermediate representation (IR). It uses a shared set of IR operations and a shared representation of data for all language implementations, which enables interoperability. A language implementation on top of the CLR needs to use the Common Type System (CTS) of the CLR. We are convinced that a multi-language runtime could be more flexible. First, the CLR does not support integrating low-level, statically compiled languages like C directly. Native code is integrated via an FFI-like interface. Second, the generic data representation cannot be optimized for an individual language. As efficient data representation and data access is critical for the performance of an application, we consider this a limiting factor.

The approach of RPython is different, they compose the language implementations on a very fine granularity (e.g. Python and Prolog [6] or Python and PHP [8]). However, we consider this approach as too inflexible because both projects only compose a pair of languages. Also, RPython has no support for running low-level, statically compiled languages.

1.1.3 Novel Solution

We identified different approaches of language implementation (e.g. native code compilers and interpretation on a VM) as a major issue when implementing a cross-language interoperability mechanism and we are convinced that a multi-language runtime is a promising approach for executing multi-language applications efficiently. We propose

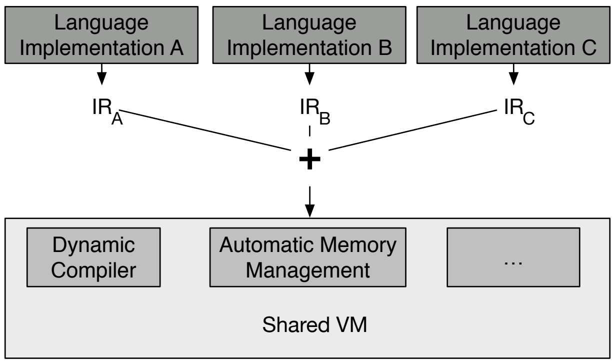

Language Implementation A
Language Implementation B
Language Implementation C
IR_A
IR_B
IR_C
+
Dynamic Compiler
Automatic Memory Management
...
Shared VM

Figure 1.1: A shared VM hosts distinct language implementations and composes them on an IR level.

a novel multi-language runtime that can execute different languages and has support for cross-language interoperability. Our runtime fulfills the following characteristics:

Composition of arbitrary languages: We identified interoperability mechanisms that target a fixed set of languages as too inflexible. Hence, our runtime features a language-agnostic mechanism for cross-language interoperability, i.e., it can compose arbitrary languages rather than only a fixed set of languages.

Efficient multi-language development: Complicated APIs for language composition impose a usability burden to the programmers and distract from the actual task at hand. Our multi-language runtime supports an interface that makes language boundaries mostly invisible to the programmer. However, the support of legacy interfaces between languages is critical. Hence, our runtime can also provide implementations for existing FFIs.

High-performance interoperability: Cross-language interoperability often adds a performance overhead because the cross-language interface marshals data at the language boundaries, because language implementations need to use language-agnostic (and therefore less optimized) data representations, or because compilers cannot optimize an application across language boundaries. We propose a multi-language runtime that does not have these limitations: First, it does not marshal data at the language boundaries. Second, every language implementation can define highly efficient data representations and share it with other language implementations. Finally, it enables the compiler to optimize and inline across any language boundaries. In our multi-language runtime, a language implementation can directly access foreign objects. We expect using heavyweight foreign data to have a negative impact on performance. On the other hand, we expect using efficient foreign data to have a positive effect on performance.

The architecture of our multi-language runtime is simple (see Figure 1.1). A language implementation translates the source code into an IR and interprets it on top of a shared VM. In contrast to other runtimes, we run high-level managed languages as well as low-level unmanaged languages on the same VM. We compose the different language implementations on an IR level. The language implementations support a language-agnostic mechanism for operations on foreign data or code, which we call *generic access*. *Generic access* enables language implementations to efficiently exchange data across any language and to bridge possibly different type systems and their semantics. It is an extension to the IR of the individual language implementations and generates object-specific IR snippets for accessing foreign objects that the runtime inserts into the IR of the host application. A language implementation can therefore directly access foreign objects. Also, *generic access* allows the dynamic compiler of the host VM to inline and optimize across language borders.

We could identify the following requirements for a multi-language runtime and its language implementations with respect to supporting *generic access*:

Compatible and adaptive IR: *Generic access* defines a set of language-agnostic *messages* that a language implementation can use to access a foreign object (we refer to this object as the *receiver*). Upon first execution of a message, we resolve it by adapting the IR of the host application in such a way that it can deal with the foreign object or function directly (*message resolution*). Message resolution embeds IR snippets from a foreign language implementation into the IR of a host language implementation at run time, which requires that the IRs are compatible and inter-mixable. Also, the IR needs a rewriting capability, which can replace IR snippets with different IR snippets at run time.

Foreign data access: Language implementations can define different data structures to represent the data of an application. A foreign data access needs to be able to access the data, allocated by any other foreign language implementation.

Dynamic compilation: The dynamic compiler of the host VM compiles the IR of a program to highly efficient machine code at run time. The shared VM needs support for deoptimization of machine code because *generic access* might change the IR of an application at any time.

We present the *Truffle-Multi-Language Runtime* (TMLR), which is our implementation of the proposed multi-language runtime that can execute and combine multiple programming languages. The TMLR can execute and combine three different languages: JavaScript, Ruby, and C. These languages are implemented as *Truffle Language Implementations (TLIs)*. Truffle [120] is a framework for implementing high-performance language implementations in Java. We execute these languages on a shared JVM, which is in our case the GraalVM. Figure 1.2 depicts the architecture of the TMLR.

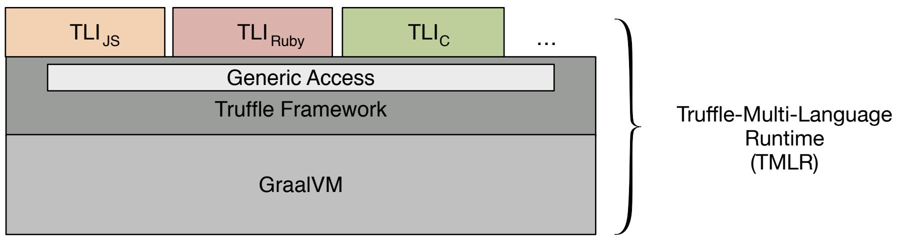


Figure 1.2: The TMLR executes different TLIs on top of the GraalVM, which itself is a modification of the HotSpot™ VM.

1.2 A Novel Multi-Language Runtime Implementation

We present and evaluate the TMLR in five steps: (1) we describe the *GraalVM* including Truffle, (2) we introduce *uniform Truffle language implementations*, (3) we explain the *language implementation composition*, (4) we present two case studies of *multi-language development*, and (5) we present a case study that uses *generic access* to substitute native C allocations with *managed data allocations.*

The GraalVM

The GraalVM, as part of the Graal OpenJDK project[9], is a modification of the Java HotSpot™ VM. The TMLR comes with TLIs for individual languages that we execute on top of the GraalVM.

Uniform Truffle Language Implementations

We base our work on existing TLIs for the modern high-level languages JavaScript and Ruby. We present a TLI for the low-level language C, which is a statically typed language that is usually compiled ahead of time instead of being executed on top of a VM. TruffleC does not create machine code upfront but it is a self-optimizing interpreter that dynamically compiles C code. To the best of our knowledge, TruffleC is the first system that can dynamically compile C code and hereby apply optimistic assumptions based on profile information.

Language Implementation Composition

TLIs implement *generic access*, which we use to combine high-level languages as well as low-level languages. *Generic access* is independent of languages. It allows us to

[9] *OpenJDK: Graal Project*, Oracle, 2015: `http://openjdk.java.net/projects/graal`

```c
struct S {
  int value;
};
struct S * obj = //...
```

Listing 1.1: C source code.

```javascript
// JavaScript can seamlessly access a C struct
var a = obj.value;
```

Listing 1.2: JavaScript source code.

treat languages as modules and we can combine them in a language-agnostic fashion. *Generic access* is independent of data representations or of calling conventions. Each TLI can use data structures that meet the requirements of the individual language best. For example, a C implementation can allocate raw memory on the native heap while a JavaScript implementation can use dynamic objects on a managed heap. We use *generic access* to share these different objects across languages, which avoids lowering them to a common representation. Finally, *generic access* ensures high-performance. Each TLI can access foreign objects or invoke foreign functions directly. Message resolution allows the compiler to optimize a foreign object access like any regular object access. Also, the compiler can widen the compilation scope across language boundaries and thus perform cross-language optimizations. For example, it can inline a JavaScript function into a caller that is written in C.

Multi-Language Application Development

We evaluate *generic access* with two case studies that compose language implementations. The first case study implements a seamless approach for multi-language development and the second one presents an implementation of the C extensions API for Ruby.

In the first case study we describe how TLIs define a mapping from host access operations to language-agnostic messages of *generic access* and vice versa. For example, the JavaScript implementation can use *generic access* to read a C `struct` member (see Listing 1.1 and 1.2). *Generic access* maps a JavaScript property access to a message and then again maps this message to a C `struct`-specific operation. The mapping of access operations to the messages and vice versa implicitly bridges language boundaries and makes them mostly invisible to the programmer. When writing multi-language applications, programmers can access foreign objects and can call foreign functions by simply using the operators of the host language. The JavaScript statement in Listing 1.2 can access the C `struct obj` (Listing 1.1) as if it were a regular JavaScript object. Only if semantics of languages fundamentally differ, programmers should need to revert to an API and therefore an explicit foreign object access.

In the second case study we describe how the TMLR can support C extensions for Ruby. We show an implementation of this API using *generic access*. Every invocation of a C extensions function is substituted by messages of *generic access*. Our solution is source-compatible with the existing Ruby API and our system is able to run existing, almost unmodified C extensions for Ruby written by companies and used today in production. We can demonstrate that the TMLR runs real-world C extensions faster than natively compiled C extensions that interface to conventional implementations of Ruby.

Managed Data Allocations for C

We present a third case study, which demonstrates that the applications of *generic access* are manifold and not limited to cross-language interoperability. We extend TruffleC so that it ensures memory safety, i.e., TruffleC$_M$ can ensure *spatial* and *temporal* safety. We use *generic access* to substitute native allocations with managed allocations in TruffleC$_M$. TruffleC$_M$ allocates all data on the managed heap rather than on the native heap. For example, TruffleC$_M$ allocates an unsafe C `int` array as a safe Java `int` array. The runtime then automatically checks whether all accesses to the array elements fall within the valid array bounds. We retain all characteristics that are typical for unsafe languages (such as pointer arithmetic, pointers that point into objects, or arbitrary casts) by mapping C pointer values to members of a managed object. This case study presents an orthogonal contribution to cross-language interoperability.

1.3 Scientific Contributions

The main contribution of this thesis is a novel approach for the seamless composition of programming languages by means of *generic access* to foreign data and code. We demonstrate this idea by describing the TMLR. This runtime requires a uniform way of language implementation. As part of this work, we present TruffleC, a self-optimizing interpreter for C that can also ensure memory safety. The scientific contributions of this thesis can be grouped into three categories:

Language implementation composition - published in [35, 39, 40]

We present a novel approach for accessing foreign data and functions in a language-agnostic way, which we call *generic access*. *Generic access* is independent of languages, data representations, and calling conventions. It can compose arbitrary languages rather than a fixed set of languages. Also, *generic access* guarantees high-performance of multi-language applications. It can directly access any data representation and does not marshal data at the language boundaries. Also, it enables the compiler to optimize and inline across any language boundaries.

Extensive evaluation by case studies - published in [39, 40]

We evaluate *generic access* using a case study where we compose JavaScript, Ruby, and C. We list the different language paradigms and semantics and explain how we bridge these differences. We evaluate the performance of multi-language applications using non-trivial multi-language benchmarks.

We evaluate *generic access* with a second case study, which is an implementation of the C extensions API for Ruby. We can run real-world C extensions and the evaluation shows that they run faster than natively compiled C extensions that interface to conventional implementations of Ruby.

A self-optimizing C interpreter with memory safe execution - published in [36, 38]

We present a self-optimizing interpreter that can execute and dynamically compile C code, which we call TruffleC. TruffleC aggressively optimizes the C code based on profile information at run time. We extend TruffleC to TruffleC$_{\mathrm{M}}$, which allocates C data as managed objects. Managed objects guarantee a memory-safe execution but still retain the characteristics of C.

We evaluate TruffleC and compare it to an industry standard C compiler in terms of peak performance. Also, we compare the performance of TruffleC (*unsafe*) to the performance of TruffleC$_{\mathrm{M}}$ (*safe*). The evaluation shows that TruffleC$_{\mathrm{M}}$ has no performance overhead compared to TruffleC.

1.4 Project Context

The work of this thesis was done as part of a long-running research cooperation between the Institute of System Software at the Johannes Kepler University and Oracle Labs. Prof. Hanspeter Mössenböck, the head of the Institute for System Software, started this collaboration in 2000 with developing an intermediate representation in static single assignment (SSA) form for the HotSpot™client compiler [71]. The work of collaborators at the Institute for System Software resulted in various research papers and theses:

- Mössenböck, Pfeiffer and Wimmer [72, 107] introduced a linear scan register allocation algorithm.

- Kotzmann et al. [63–65] extended the HotSpot™client compiler with an escape analysis algorithm.
- Wimmer et al. [108–111] presented automatic object and array inlining.
- Würthinger et al. [117, 119] added an array bounds check elimination algorithm to the HotSpot™client compiler.
- Würthinger et al. [118] worked on visualizing program dependency graphs.
- Würthinger et al. [114–116] extended the HotSpot™ VM with support for dynamic code evolution, which allows applying changes to running applications.
- Häubl et al. [46,47] improved the `String` representation within the HotSpot™ VM.
- Schwaighofer [89] added support for tail call optimizations to the HotSpot™ VM.
- Stadler et al. [97, 99] added support for continuations and coroutines to the HotSpot™ VM.
- Schatzl et al. [88] improved various aspects of the garbage collection systems of the HotSpot™ VM.
- Häubl et al. [45, 48–51] worked on generalized trace compilation for Java.

The most recent work in this collaboration is the GraalVM project. The GraalVM is a minor modification of the HotSpot™ VM that adds a third compiler, entirely written in Java. The GraalVM also comes with Truffle, a framework for writing highly efficient language implementations in Java. The following research papers and theses refer to the GraalVM:

- Dubosq et al. [23] introduced an extensible declarative intermediate representation for the compiler.
- Stadler et al. worked on methods for queuing compilation tasks and caching of intermediate results within compilers [95] and did experimental studies on dynamic compiler optimizations [96].
- Würthinger et al. presented Truffle, a framework for building self-optimizing interpreters [121]. This work was extended by a mechanism to generate highly-efficient machine code from the interpreter definition [120].
- Stadler et al. [98] presented a partial escape analysis algorithm with scalar replacement for the compiler.

- Humer et al. [55] introduced a domain-specific language to simplify the building of self-optimizing interpreters.
- Wöss et al. [112] worked on an object storage model for Truffle.
- Dubosq et al. [24, 25] worked on optimizing speculative optimizations within the compiler.

The author's work in this collaboration started in 2012. Together with Manuel Rigger, they implemented a first version of TruffleC, which was the starting point of this thesis and was published in two Master's theses:

Rigger - TruffleC Interpreter [85]: Rigger explains how TruffleC parses C code and how C operations can be implemented in Java (including unsigned operations and `goto` statements).

Grimmer - A Runtime Environment for the TruffleC VM [34]: Grimmer presents a native function interface for the GraalVM [37] and explains how TruffleC uses it to access precompiled native functions efficiently. Also, he describes the memory model of TruffleC that can share data between TruffleC and precompiled native code.

The author's work on this PhD thesis started in 2013. Several papers have been published and significant parts of these papers are integrated into various chapters of this thesis:

- A full research paper describes TruffleC, a self-optimizing interpreter that can execute and dynamically compile C [36]. This work extends the previous work of TruffleC [34, 85] by adding profile based optimizations to the interpreter.
- A position paper about composing JavaScript and C code [41] describes an early version of *generic access* that we use to access C data from JavaScript.
- A doctoral symposium paper [35] generalizes *generic access*. *Generic access* can access arbitrary foreign objects and generates efficient access operations at run time.
- A full research paper [39] presents the TMLR, which composes different TLIs in a seamless way. It describes how TLIs access foreign objects using *generic access*.
- A full research paper [40] shows how we use *generic access* to implement the C extensions API for Ruby. This work was done in collaboration with Chris Seaton (Oracle Labs UK). Chris Seaton leads the TruffleRuby project.

- A full research paper [38] describes how we can safely execute C code on top of the TMLR by allocating all data on the managed heap.

1.5 Structure of this Thesis

This thesis is structured as follows: Chapter 2 introduces the context of the TMLR. The TMLR is based on the GraalVM, which is a modified version of the HotSpot™ VM that adds the Graal compiler to it. There are four main chapters:

Chapter 3 presents language implementations for JavaScript, Ruby, and C. This chapter first discusses the implementation of TruffleC in detail. We explain how TruffleC optimizes C code at run time based on profiling information. Afterwards, this chapter introduces the existing JavaScript and Ruby implementations on top of Truffle.

Chapter 4 describes how we can compose different language implementations on top of the TMLR. We describe *generic access*, a language-agnostic object access mechanism, which allows sharing data between different TLIs efficiently. In this chapter we show how we map object accesses to language-agnostic messages and vice versa.

Chapter 5 describes an implementation of *generic access* for JavaScript, Ruby, and C. We present two case studies. First, we discuss how different language paradigms and features can be mapped across different languages for seamless interoperability. Second, we describe a C extensions API implementation for TruffleRuby.

Chapter 6 presents TruffleC$_\mathrm{M}$. TruffleC$_\mathrm{M}$ is a third case study and shows another application of *generic access*. We use *generic access* to substitute native C allocations with managed objects, which guarantees memory safety.

After the technical part we present a performance evaluation in Chapter 7. We first evaluate TruffleC and compare it to an industry-standard C compiler. Afterwards, we show the performance of multi-language applications as well as the performance of Ruby applications using C extensions. Finally, we compare the performance of TruffleC$_\mathrm{M}$ to TruffleC.

This thesis concludes with a discussion of related work in Chapter 8 and a summary including future work in Chapter 9.

Chapter 2

The Graal Virtual Machine

This chapter explains the context of the TMLR and gives an overview over its architecture. It introduces Truffle, the framework that we use to build the individual language implementations (e.g. JavaScript, Ruby, or C) of the TMLR. Also, it describes Graal, the dynamic compiler that Truffle uses to transform applications to machine code.

VMs are mostly monolithic pieces of software, written in one language (mostly C/C++) and executing another language. They offer many services for applications running on top of them, such as dynamic compilation, automatic memory management, threads, as well as synchronization primitives. Implementing all these services for a VM is a non-trivial task and requires a considerable amount of engineering effort. To ease the effort of building new VMs, Truffle language implementations have a layered architecture [120]. Language implementations (e.g. a JavaScript implementation) are running on top of a *Host VM* (e.g., the GraalVM) and hereby reuse the services of the Host VM. Truffle proposes the following three layers (see Figure 2.1):

GraalVM layer (Figure 2.1, bottom): The GraalVM adds the Graal compiler [24, 25, 95, 96, 98] — a Java dynamic compiler written in Java (see Section 2.1) — to the Java HotSpot™ VM. The Graal compiler can substitute the client [65] and server [81] compilers, however, it reuses all other VM components, such as the garbage collector, the interpreter, the class loader and so on, from HotSpot™.

Truffle framework layer (Figure 2.1, middle): Truffle [120] (see Section 2.2) is a framework for building high-performance language implementations in Java. In Truffle, a language is implemented as an abstract syntax tree (AST) interpreter. These interpreters can be executed by any JVM, which allows reusing Java's runtime services such as automatic memory management, threads, synchronization primitives and a well-defined memory management. However, they perform best when running on top of the GraalVM. If Truffle ASTs are executed by the GraalVM, then Truffle uses Graal to dynamically compile ASTs to machine code.

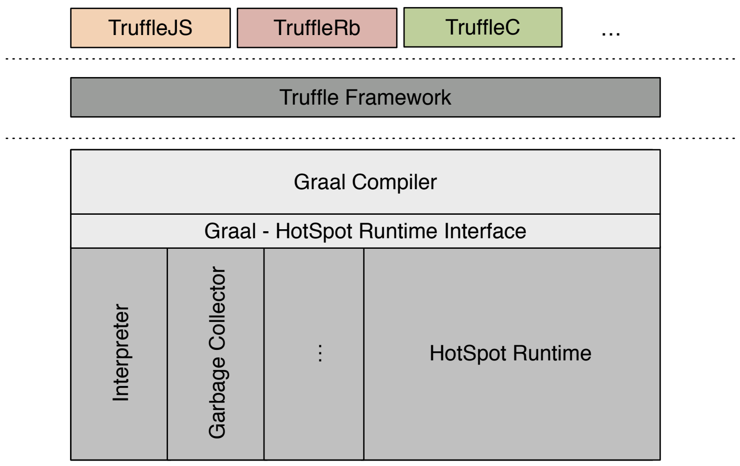


Figure 2.1: The architecture of the TMLR: Three Truffle language implementations (JavaScript, Ruby, and C) are hosted by the GraalVM.

Language implementations (Figure 2.1, top): The TMLR can execute and combine three different languages: JavaScript, Ruby, and C. We describe their implementation in Section 3.

2.1 Graal Compiler

The Truffle framework uses the Graal compiler to dynamically compile ASTs to highly-efficient machine code. However, the Graal compiler is also a regular dynamic compiler for Java bytecode.

The GraalVM starts executing Java programs in the interpreter (part of the HotSpot™ VM). If a method becomes hot, i.e., its execution count exceeds a certain threshold, then the VM compiles it using the Graal compiler. Hereby, the Graal compiler translates Java bytecode to the Graal IR [23], which is a graph-based high-level IR. This IR is in SSA [20] form and captures the control flow and the data flow of instructions. While there is an explicit control flow in this graph, many operations do not have a fixed location but their position is determined by data flow dependencies. These *floating* operations can often be fit in at many places and are brought into a well-defined order in a separate scheduling step. Using this flexible IR, the Graal compiler performs aggressive optimizations, often based on optimistic assumptions about a program, such as branches that are assumed to be never executed or objects that are assumed to have a specific type. The Graal compiler inserts *guards* that check these assumptions and whenever one of these guards fails, the control flow is transferred from machine code back to the interpreter (*deoptimization* [54]). Once Graal compiled a method to machine code, it sends the machine code to the VM, which installs it into its internal data structures. The VM then ensures that subsequent calls to the compiled method immediately jump to the machine code, instead of interpreting the bytecode in the interpreter. Also, the GraalVM offers a native function interface (the Graal Native Function Interface, GNFI) [37], which allows us to efficiently call native functions from within Java. The GNFI is an alternative to other foreign function interfaces of Java (e.g. the *Java Native Interface* [66]) and allows invoking native code from a Java application efficiently.

2.2 Truffle Language Implementation Framework

TLIs are AST interpreters, written in Java. An AST interpreter models constructs of the guest language as nodes. These nodes build a tree (i.e., an AST) that represents the program to be interpreted. Each node extends a common base class `Node` and has an `execute` method, which implements the semantics of the corresponding language construct. By calling these methods recursively, the whole AST is evaluated. The language implementer designs the AST nodes and provides facilities to build the AST from the guest language code. Language developers only write language specific parts when implementing a language.

In the following we describe the two different types of Truffle's optimizations. The language developer can implement *optimizations on AST level* and the Truffle framework itself provides a *dynamic compilation of Truffle ASTs*. Afterwards, we explain the Truffle `Frame`, a data structure to efficiently handle local variables, and also introduce Truffle's object storage model.

2.2.1 Optimizations on AST Level

Truffle AST nodes can speculatively rewrite themselves with *specialized* variants [121] at run time, e.g., based on profile information obtained during execution such as type information. We specialize nodes on a subset of the semantics of an operation, which makes the node implementation simpler. TLIs use self-optimization via tree rewriting as a general mechanism for dynamically optimizing code at run time. If these speculative assumptions turn out to be wrong, the specialized tree can be transformed to a more generic version that provides functionality for all possible cases. Concrete examples of specializations are:

Type Specialization: Operators in dynamic languages often have complex semantics. The behavior of an operation can, for example, depend on the types of the operands. Hence, such an operator needs to check the types of its operands and choose the appropriate version of the operator. However, for each instance of a particular operator in a guest language program, it is likely that the types of its operands do not change at run time. Truffle's self-optimization capability allows us to replace the full implementation of an operation with a specialized version that speculates on the types of the operands being constant. This specialized version then only includes the code for this single case. For example, consider an add operation of a dynamic language (e.g. JavaScript) that has different semantics depending on the types of its operands. Truffle trees can adapt themselves according to type feedback, e.g., a general add operation can replace itself with a faster integer-add operation if its operands have been observed to be integers. If this optimistic specialization of an operator fails at run time, the specialized node changes back to a more generic version.

Polymorphic Inline Caches: Truffle's self-optimization capability allows building up polymorphic inline caches [53] at run time. An inline cache is an optimization that improves the performance of run-time method binding by caching the target method of a previous lookup at the call site. In Truffle, an inline cache is built by chaining nodes. Each node in this chain then checks whether the cached target matches and eventually executes the specialized subtree for this target. If the target does not match, the chain delegates the handling of the operation to the next node. When the chain reaches a predefined length, the whole chain replaces itself with a single node that can handle the fully megamorphic case.

Resolving Operations: Operations of a TLI might include a resolving step that happens at run time. For example, a TLI can resolve and cache the target of a function call lazily at run time. This lazy resolving step is implemented with self-optimization, which in this case replaces the node of an unresolved call operation at run time by its resolved version. The resolved node avoids a subsequent resolving operation.

Optimizations on AST level via node rewriting must fulfill three requirements [120]. First, a specialized node that handles only a subset of the semantics of a guest language operation needs to trigger a subsequent node replacement for all uncovered cases. For example, an add operation of a dynamic language that is specialized on integer operands needs to trigger a replacement if the operands suddenly have a type other than integer. Second, all possible specializations of an operation form a transition DAG with a single final state that can handle the full semantics of this operation, i.e., a series of node replacements eventually ends up with a generic node implementation that handles all possible inputs. This generic node does not trigger subsequent node replacements. For example, the generic node implementation of an add operation of a dynamic language can add operands of any type. Finally, a node rewrite affects only a single node while it is executing. However, such a rewrite might trigger other rewrites and hence change the structure of its complete subtree.

Besides specialization of operations, TLIs inline methods of a guest language program on an AST level. This is done by cloning the AST and replacing it for the calling node. Inlining ASTs allows mitigating the effect of run-time profiling pollution due to different callers using a method in different ways, i.e., inlining can prevent the rewriting of AST nodes to less specialized versions.

2.2.2 Dynamic Compilation of Truffle ASTs

After an AST has become stable (i.e., when no more rewritings occur) and when the execution frequency has exceeded a predefined threshold, Truffle dynamically compiles the AST to machine code. Truffle uses the Graal compiler for dynamic compilation. The compiler hereby assumes that the tree has reached its final state. This allows the compiler to remove all the virtual dispatches between the `execute` methods of the AST nodes and inline them. Inlining produces a combined compilation unit for the whole tree. The Graal compiler can then apply its aggressive optimizations over the whole tree, which results in highly efficient machine code. This special form of interpreter compilation is an application of *partial evaluation* to generate compiled machine code from a specialized interpreter [31].

The compiler inserts deoptimization points [54] in the machine code where the speculative assumptions about the tree are checked. Every control flow path that would cause a node rewrite transfers back from compiled machine code to the interpreted AST, where specialized nodes can be reverted to a more generic version.

2.2.3 Truffle Frame

The Truffle framework provides a `Frame` class. TLIs use `Frame` objects to store the local variables of a function. The `Frame` is essentially an array holding the values of the local variables. As efficient access to local variables is critical, the Graal compiler ensures that this access is fast after dynamic compilation. It forces an *escape analysis* with *scalar replacement* [98] on the `Frame` object, which causes the elements of a `Frame` object to be stored as simple local variables. When all reads of such variables are connected to their most recent writes, the `Frame` object itself becomes *virtual*, i.e., it need not even be allocated in the compiled machine code. The result is a program in SSA form, which allows the compiler to perform standard optimizations such as constant folding or global value numbering without needing a data flow analysis for the `Frame` object.

2.2.4 Truffle Object Storage Model

In a dynamically typed language like JavaScript or Ruby, objects are dynamic data structures and it is possible to add or remove members at run time. The Truffle framework eases the effort of implementing efficient dynamic data structures for a dynamic language implementation and provides an object storage model [112]. The implementation of this object storage model is called `DynamicObject`. The `DynamicObject` supports dynamic resizing and allows adding and removing members (a member can be a field or a method and is stored in a *slot*) at run time and hereby guarantees high-performance. This `DynamicObject` was originally designed for dynamic guest-language implementations on top of Truffle (e.g. TruffleJS and TruffleRuby [112]) to represent their dynamic data structures, however, in Chapter 6 we explain how we can use the `DynamicObject` to efficiently represent C data structures.

Chapter 3

Uniform Language Implementations with Truffle

Truffle language implementations are AST interpreters. In this section we describe the implementations for JavaScript, Ruby, and C. The TMLR composes these TLIs.

The TMLR has a TLI for all different languages. We introduce TruffleC, an implementation for the low-level language C. This implementation allows integrating C code into our multi-language ecosystem and is part of the contributions of this thesis. We describe this implementation in detail.

For the experiments in this thesis, we also use TLIs for the high-level languages JavaScript and Ruby. The TLIs for JavaScript and Ruby were written by others, but we extended them by adding support for *generic access*. Hence, we only describe them briefly and summarize their performance compared to other implementations.

3.1 TruffleC

The following chapter describes the C implementation on top of Truffle in more detail. It recaps the work of Rigger [85] (translating C source code to Truffle ASTs and implementing C operations in Java) and the former work of the author of this thesis [34] (the memory model of TruffleC). However, the main focus is on dynamic compilation of C code, i.e., this chapter extends the previous work and presents C-specific optimizations on an AST level based on run-time feedback.

```c
typedef int (*func)(int,int);

int div(int a, int b) {
  if (b == 0)
    goto error;
  return a / b;
  error:
    // error handling
}

int abs(func f, int a, int b) {
  int value = f(a, b);
  if (value < 0)
    return -value;
  else
    return value;
}
```

Listing 3.1: C source code of the *MyLib* library.

```c
// Profiling information:
// f = div
// a > 0
// b = 2

int result = abs(f, a, b);
```

Listing 3.2: C source code of the Main application.

3.1.1 Example

Listings 3.1 and 3.2 introduce an example that allows us to explain how TruffleC works. We also use this example to discuss advantages of dynamic compilation over static compilation, i.e., we illustrate how TruffleC applies high-level optimizations to the AST by specializing nodes. This example uses a shared user library (*MyLib*, Listing 3.1) containing two functions `div` and `abs`. The function `div` takes two arguments, `a` and `b`, and computes `a / b`. To avoid division by zero, it checks whether `b` is zero and jumps to an error handler in this case. The function `abs` takes three arguments, `f`, `a`, and `b`. The first argument `f` is a function pointer and `a` and `b` are the arguments for the call of this function pointer. The function `abs` uses the function pointer to invoke the target function and then computes and returns the absolute value of the result. The main application of our example (Listing 3.2) uses *MyLib*.

In Section 3.1.4 we describe how TruffleC optimizes the AST of a C function depending on the profile of a program execution. To illustrate these optimizations, we assume the following program profile: `f` will always point to the function `div`, `a` is always a positive value, and `b` is always 2.

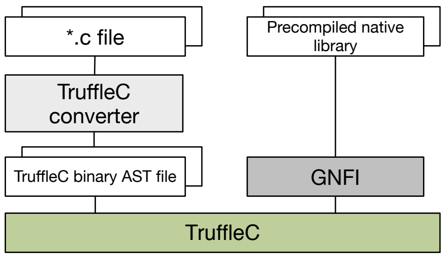


Figure 3.1: Compiling C files to TruffleC binary AST files, which are executed on top of TruffleC.

3.1.2 Compiling C Code to ASTs

Figure 3.1 depicts an architectural overview of TruffleC. The *TruffleC converter* compiles C source code to a binary representation from which we can create a Truffle AST later efficiently. The TruffleC converter performs the preprocessing and syntax checking and produces *TruffleC binary AST files* that contain a serialized form of our ASTs. These files are platform independent so that TruffleC instances running on different platforms can load and execute them.

We translate the source code of our example (Listing 3.1 and 3.2) to two TruffleC binary AST files: The first one contains the library *MyLib* and the second file contains the main application and its entry point. In contrast to that, an industry-standard compiler such as GCC would require us to first compile the shared library to executable machine code before we could compile the main application and link it to the library.

3.1.3 Linking

C applications typically consist of several source files, which we translate to individual TruffleC binary AST files. These files can contain references to symbols declared in other files, which need to be resolved. We combine multiple files by patching symbol references at run time using Truffle's node specialization mechanism. This resolving step happens only once upon the first execution of an *unresolved* node, which replaces itself by a *resolved* version. Afterwards, the application runs at full speed. An example of an unresolved symbol reference is a function call whose target is implemented in a different file. TruffleC resolves the target of a function call using its name upon the first execution. It searches for the target function in all TruffleC binary AST files, creates the AST of the function, and replaces the *unresolved* call node by a specialized and *resolved* version of it. The resolved node ensures that subsequent executions directly call the target function without any prior resolution. Creating the AST of a function is done only once; different call sites to the same target all use the same AST. In our example, TruffleC looks for call targets in the TruffleC files of the main application and *MyLib*.

However, if the target function is located in a *precompiled native library* (e.g. in the standard C library), then TruffleC uses GNFI to efficiently access it. The resolved call node caches a *NativeFunctionHandle*, which is the interface provided by GNFI to allow TruffleC to directly call native functions. GNFI provides *NativeFunctionHandle*s to call native target functions directly from Java.

When using a traditional C compiler to run the example of this thesis, one would create an executable file by linking the main application's object file with the precompiled library. Linking combines the given files by patching symbol references (e.g., by patching call target addresses). In contrast to TruffleC, there is no symbol resolution at run time.

3.1.4 Optimizations on AST Level

After having created the ASTs for the C functions, TruffleC starts interpreting them. During interpretation, the nodes of a tree can replace themselves with specialized versions. If the execution count of a stable tree (i.e., when nodes no longer rewrite themselves because the tree is already fully specialized) exceeds a predefined threshold, then the Graal compiler transforms the AST into machine code. The following sections explain how we implement dynamic optimization for C using the self-optimization capability of Truffle.

Inlining

When traditional compilers (e.g. GCC) compile source files to individual object files, inlining of external functions is not possible by default because the callee's code is usually unknown when the caller is compiled.

Our approach, however, interprets ASTs and only later turns them into machine code, so TruffleC can inline functions as ASTs from other files or from shared libraries at run time. TruffleC inlines functions by copying the AST of the callee and replacing the call node with this copy. TruffleC provides information to the underlying Truffle framework that allows Truffle to take inlining decisions. Truffle's inline heuristic decides for each call site whether to call or to inline the target function. This decision is based on the call count of the caller and the size of the caller and the callee, respectively. For our example, we assume that the call sites of `abs` and `div` (see Listing 3.1) are frequently executed and therefore Truffle decides to inline these functions. Once execution of the main application exceeds a predefined threshold, the Graal compiler transforms the

```
if (f != div)
  // DEOPT
if (b != 2)
  // DEOPT
int result = a >> 1;
if (result < 0)
  // DEOPT
```

Listing 3.3: Result after dynamic compilation (pseudo code).

```
mov     rax,    #f
cmpq    rax,    #div  ;is f != div
jnz     DEOPT
mov     eax,    #b
cmpl    eax,    0x2 ;is b != 2?
jnz     DEOPT
mov     eax,    #a   ;signed shift instead of signed division
mov     esi,    eax
shrl    esi,    0x1F
add     esi,    eax
sarl    esi,    0x1
test    esi,    esi ;is result < 0?
jl      DEOPT
;esi contains result
```

Listing 3.4: Result after dynamic compilation (machine code).

AST (with the functions `abs` and `div` inlined) to machine code. Listings 3.3 and 3.4 show the code (pseudo code and machine code) that the Graal compiler produces for the call to `abs` if all functions are inlined and the profile assumptions are exploited. In the following we describe how we use AST specialization at run time for C code and explain the machine code in Listing 3.4 in more detail.

If our example were compiled with GCC (without any link time optimizations, which are disabled by default) we would get code that calls `abs` directly and `div` via a function pointer. Static compilers cannot inline this external function because they compile the files independently and `abs`' code is not available when the main application is compiled.

Inline Caches for Function Pointer Calls

A regular function call specializes upon its first execution. It replaces itself with a direct call to the AST of the target function and never reverts to a generic version because the target never changes. However, a function pointer call can have different targets, which prevents this specialization. As the target can change, a function pointer call would have to resolve and link the target for each call before executing it. To avoid the overhead of a lookup in TruffleC's function table, we cache the resolved target functions. A subsequent execution of this function pointer call then checks whether the function pointer value matches the cached function. If so, it directly executes the target. By caching multiple targets, we can efficiently handle polymorphic function pointer calls. In case of a new and unseen function pointer value we resolve its target and add it to this cache, i.e., we add an entry to the chain of nodes. An entry in this cache can be seen as constant and therefore Truffle considers these functions for inlining, which creates an inline cache for polymorphic function pointer calls. To prevent excessive growth of this inline cache queue, we replace it with a *generic function pointer call* once it would exceed a predefined length. The generic case resolves and executes the target for every call without caching.

The function `abs` (Listing 3.1, line 12) contains a function pointer call. In our example, this call always executes the function `div`. The call site of this function pointer call resolves the target `div` and caches it. When Graal compiles this call site, it produces machine code that checks whether the function pointer value matches the cached `div` function (Listing 3.4, line 1-3). If so, the inlined function `div` is executed. Otherwise, the function pointer node has to resolve the unseen target and add it to its inline cache. Graal does not compile this resolution to machine code. Instead, it inserts a deoptimization point (Listing 3.4, line 3), which transfers the execution from compiled machine code back to the AST interpreter.

Branch Probability Profiling

Besides caching the target of a function pointer call, TruffleC also records the branch probabilities for conditional statements. It makes the branch probability information available to the Graal compiler by attaching it to the condition. If the profiling information shows that a certain branch is never executed, the compiler excludes the supposedly dead branch from compilation. If the removed code needs to be executed, then the machine code triggers a deoptimization. This technique reduces the size of the machine code produced by the compiler.

If the profile information of `abs` (Listing 3.1) shows that `value` has never been negative, the compiler excludes the then-branch from compilation. If this assumption is violated later on (i.e., `value` is negative), the machine code deoptimizes. If the branch profile does not allow the removal of a dead branch, the Graal compiler can still use this information for other optimizations, e.g., to swap branches.

Value Profiling

While interpreting the AST, TruffleC collects profile information for all variables that cannot be statically replaced with constants. If the profile information shows that the value of a variable does not change, TruffleC speculates on the value being constant using specialization. Every access to this variable checks if the assumption is still valid and uses a constant in this case. When the Graal compiler translates the AST to machine code, this specialization enables further optimizations such as constant propagation or constant folding. If the assumption no longer holds, the AST reverts the assumptions about the variable being constant. In machine code, a no longer valid assumption causes a deoptimization.

In our example, we assume that `b` (Listing 3.2) always has the value `2`. After observing the constant value multiple times, TruffleC specializes the variable's node and assumes that this value is constant. If this assumption no longer holds, the node rewrites itself to the generic case where no assumptions about `b` are made. When Graal compiles our example, it produces machine code that checks if `b` is still `2` (Listing 3.4, line 4-6) and replaces all usages of `b` with the constant value `2`. In our example, this assumption allows the Graal compiler to use a shift operation instead of the division (Listing 3.4, line 11) and to remove the check `b == 0` completely. As this division is a signed operation, the compiler inserts the lines 8-10, which ensure a correct behavior in case of a negative numerator. If the value profile turns out to be invalid, we again deoptimize. We then rewrite the variable's node to the generic case where the above optimizations no longer apply.

Comparison to Static Compilers

The dynamic optimizations of TruffleC are to some extent also possible with a static compiler like GCC. For example, GCC offers *link time optimizations*[1] and *profile guided optimizations*[2]:

Link time optimizations: This optimization (enabled by the compiler flag `-flto`) allows GCC to dump its internal representation of a compilation unit to the object files. When combining units to a single executable, the compiler can optimize it as a single module and can therefore expand the compilation scope across different units. However, the optimizations, including inlining, still happen statically before executing the program. Hence, no profile information about the program execution is available.

TruffleC inlines functions at run time and can use profile information to guide the sophisticated inline heuristics of Truffle. As inlining is one of the most beneficial optimizations [3, 95], good inlining decisions are important for the performance of a program.

[1]*Link Time Optimization*, GCC the GNU Compiler Collection, 2015: `https://gcc.gnu.org/wiki/LinkTimeOptimization`

[2]*Profile Guided Optimizations*, GCC the GNU Compiler Collection, 2015: `https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Optimize-Options.html`

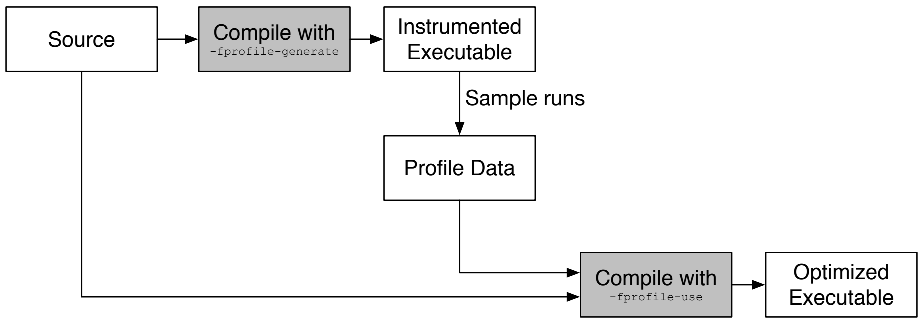


Figure 3.2: Profile guided optimization with the GCC.

Profile-guided optimizations: GCC has support for optimizing C code for a given program profile. This is done by first compiling the source code to an instrumented executable (enabled by the compiler flag `-fprofile-generate`). This executable dumps the profile data of sample runs to a file, which can then be used by a second compilation step. In this second step the C source code is compiled again and the compiler uses the profile data as an additional input (enabled by the compiler flag `-fprofile-use`). The result is an executable that is optimized for the profile of the sample runs. Figure 3.2 summarizes the steps of a profile-guided optimization with GCC.

In contrast, TruffleC collects profile information during AST interpretation and uses it to rewrite AST nodes to a specialized version or provides it to the Graal compiler. The result is machine code, which is optimized for a given program profile. In addition to that, if the behavior of a program changes at run time, TruffleC deoptimizes the machine code and can again collect profile information to specialize the AST on the new behavior. This is not possible with a static compiler such as GCC.

Our technique has two advantages over profile-guided optimizations of GCC. First, there is no explicit sampling step involved. TruffleC starts interpreting a program and collects the profiling information; eventually it dynamically compiles the AST. Profiling and dynamic compilation are transparent to the application programmer, hence, it is not necessary to recompile source code. Second, in case the behavior of a program changes, TruffleC can respecialize on the new behavior. When using GCC, this would require resampling the program and manually recompiling an optimized executable.

3.1.5 Data Allocations

A C program running on top of TruffleC can switch execution from TruffleC to a native function (e.g., one that is part of the standard C library) using GNFI [37]. When switching from the TruffleC interpreter (that is written in Java) to native code, GNFI allows passing parameters from Java to a native function. Besides passing parameters, TruffleC can also exchange data of the running C program via pointers. Hence, TruffleC needs a memory model that allows sharing allocations (e.g., `struct`s, `union`s and arrays) between TruffleC and native code.

Usually, TLIs use `Frame` objects (see Section 2.2.3) for storing data. However, pointers cannot reference variables that are stored in the *virtual* `Frame` because it is not possible to provide a valid memory address of Java objects. The JVM manages Java objects automatically and does not allow a direct access via a raw pointer. Hence, TruffleC cannot use the `Frame` object for those variables of a C application that are referenced via pointers. To fulfill the requirements of C, TruffleC distinguishes two locations of data. Data can be either stored on the *native heap* or in the `Frame` object:

The native heap: Java applications can access the native heap using the Java Unsafe API (available under restricted access in the OpenJDK). The Unsafe API provides functionality to manually allocate memory on the native heap and to load data from it and store data into it.

TruffleC stores all global or static variables of a C application in a memory block on the native heap. This guarantees that static or global objects are not garbage collected and pointers can be used to reference this data. TruffleC manages a native C stack by allocating a second memory block on the native heap, which is used like a stack. This memory block contains all function-local arrays, `struct`s, `union`s, and primitive values that are referenced by a pointer. Pointers to these objects can be shared with native library code because TruffleC and the native code use the same data alignment.

TruffleC represents all pointers as `CAddress` Java objects that wrap a 64-bit integer value. This value is a pointer to the native heap and therefore features any kind of pointer arithmetic. Address values can also be shared with precompiled native code because an address is always represented as a 64-bit value.

The `Frame` object: In contrast to elements stored on the native heap, the Graal compiler can apply more aggressive optimizations on values stored in the `Frame` object. To leverage these optimizations, TruffleC keeps all local variables of primitive types that are never referenced by pointers in the `Frame` object.

```
void foo() {
  static int counter;      //The native heap
  int[3] array;            //The native heap
  int value;               //The native heap
  int* pointer = &value;   //Frame object
  // ...
}
```

Listing 3.5: Different locations for local data.

Listing 3.5 summarizes where TruffleC allocates the local variables of a function. The static variable `counter`, the non-primitive variable `array`, and the integer variable `value` (which is referenced by a pointer) are stored on the native heap. The pointer variable `pointer` is never referenced by its address and is therefore stored in the `Frame` object.

The TruffleC memory model as well as the native function calls via GNFI provide efficient interoperability between TruffleC and precompiled native code. However, this forces the implementation of TruffleC to be platform-specific. TruffleC's nodes for address computation ensure that non-primitive data on the native heap uses the alignment of the target platform. This allows exchanging data with precompiled native code. Also, TruffleC needs to follow the *GNFI Java calling convention* [37] for all native calls. As this calling convention differs between platforms, the usage of GNFI is also platform-dependent. The parts of TruffleC that do the address computation and use GNFI need to be customized for each target platform.

3.1.6 Limitations

TruffleC aims to support the C99 standard [29], however, it is not yet fully complete. TruffleC does not yet have support for flexible array members, variable-length automatic arrays, designated initializers, and compound literals. However, adding them would only require additional engineering effort. TruffleC has no conceptual restrictions in this respect. Adding support for these missing features is planned as future work. *Flexible array members*[3] allow defining a C `struct` that has an array member without a given dimension, for example (see Listing 3.6):

[3] *Arrays of length zero*, GCC the GNU Compiler Collection, 2015: `https://gcc.gnu.org/onlinedocs/gcc/Zero-Length.html`

```c
struct vector {
  int length;
  int array[];
}
```

Listing 3.6: A C `struct` using a flexible array member.

Variable-length automatic arrays[4] are array data structures whose length is determined at run time. *Designated initializers*[5] allow assigning values to arrays or `structs` in any order. For example, designated initializers initialize array elements with certain values (see Listing 3.7):

```c
int arr[5] = {[4] = 42, [1] = 12};
```

Listing 3.7: An initialization using a designated initializer.

Compound literals[6] look like casts that contain an initialization. The initialization value is specified as a list of all values to be assigned. For example, a `struct` can be initialized as follows (see Listing 3.8):

```c
struct myComplex {
  double r;
  double i;
} c;
c = ((struct myComplex) {13.0, 4.5});
```

Listing 3.8: Using compound literals for initialization.

TruffleC aims to support the widely used C99 standard, but support for multi-threading is out of scope of this thesis. The latest C standard C11 [30] adds multi-threading support to the C language. At the time of writing this thesis, Truffle has only experimental support for multi-threading but future work on TruffleC will also add multi-threading for C.

[4] *Arrays of variable length*, GCC the GNU Compiler Collection, 2015: `https://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html`

[5] *Designated initializers*, GCC the GNU Compiler Collection, 2015: `https://gcc.gnu.org/onlinedocs/gcc/Designated-Inits.html`

[6] *Compound literals*, GCC the GNU Compiler Collection, 2015: `https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Compound-Literals.html`

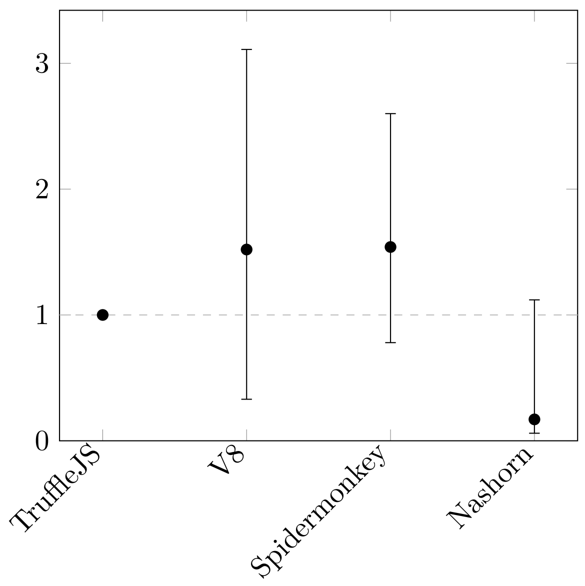


Figure 3.3: Relative speedup of different JavaScript implementations compared to TruffleJS; results taken from [112].

3.2 TruffleJS

TruffleJS is an implementation of JavaScript on top of Truffle and was originally Truffle's proof of concept. JavaScript is a dynamically typed, prototype-based scripting language with first-class functions. It allows an object-oriented, an imperative, as well as a functional programming style, which makes it a good candidate for the evaluation and case study in this thesis. TruffleJS is a state-of-the-art JavaScript engine that is fully compatible with the JavaScript standard. It uses optimizations on the AST level (see Section 2.2.1), e.g., for dynamic type specialization of operations, variables and object properties. Figure 3.3 summarizes the performance evaluation of [112] for JavaScript. The y-axis shows the performance of Google's V8[7], Mozilla's Spidermonkey[8], and Nashorn as included in JDK 8u5[9] relative to TruffleJS where the outer most lines show the minimum and maximum performance and the inner dot shows the average performance. The x-axis shows the different language implementations. This evaluation uses a selected set of benchmarks from the Octane benchmark suite. Google's V8 is between 210% faster and 67% slower (52% faster on average) than TruffleJS; Mozilla's Spidermonkey is between 160% faster and 22% slower (54% faster on average) than TruffleJS; Nashorn as included in JDK 8u5 is between 12% faster and 96% slower (74% slower on average) than TruffleJS.

[7] *V8 JavaScript Engine*, Google, 2015: `http://code.google.com/p/v8`

[8] *SpiderMonkey JavaScript Engine*, Mozilla Foundation, 2015: `http://developer.mozilla.org/en/SpiderMonkey`

[9] *Nashorn JavaScript Engine*, Oracle, 2015: `http://openjdk.java.net/projects/nashorn`

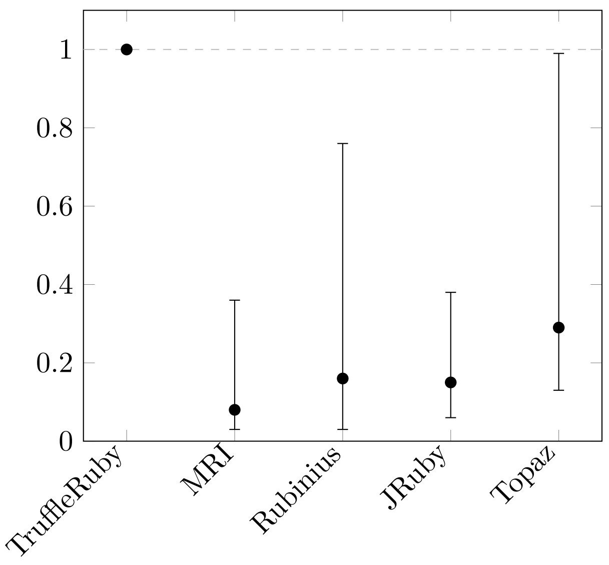


Figure 3.4: Relative speedup of different Ruby implementations compared to TruffleRuby; results taken from [112].

3.3 TruffleRuby

TruffleRuby is an implementation of Ruby on top of Truffle. Ruby is a dynamically typed and object-oriented language that is inspired by Smalltalk and Perl. *TruffleRuby* reuses the parser from JRuby, however, little of the two systems are currently shared and *TruffleRuby* (also called JRuby+Truffle in [40, 90]) should be considered entirely separate from JRuby for this discussion. TruffleRuby performs well compared to existing Ruby implementations. Figure 3.4 summarizes the performance evaluation of [112] for Ruby. The y-axis shows the performance of MRI, Rubinius[10,11], JRuby, and Topaz[12] relative to TruffleRuby where the outer most lines show the minimum and maximum performance and the inner dot shows the average performance. The x-axis shows the different language implementations. This evaluation uses the Richards and DeltaBlue benchmarks from the Octane suite, a neural-net, and an n-body simulation. MRI is between 63% and 97% slower (92% slower on average) than TruffleRuby; Rubinius is between 24% and 97% slower (83% slower on average) than TruffleRuby; JRuby is between 61% and 94% slower (84% slower on average) than TruffleRuby; Topaz is between 1% and 87% slower (71% slower on average) than TruffleRuby.

[10] *An implementation of Ruby*, Rubinius, 2015: `http://rubini.us`
[11] *Rubinius*, GitHub repository, 2015: `https://github.com/rubinius/rubinius`
[12] *Topaz Project*, GitHub repository, 2015: `https://github.com/topazproject/topaz`

Chapter 4

Truffle Language Implementation Composition

This chapter describes the main contribution of this thesis. We introduce generic access, *a novel mechanism that allows accessing objects in a language-agnostic fashion. We use* generic access *to compose different TLIs, which allows writing programs in different languages. To support* generic access, *each TLI defines a mapping from language-specific object accesses to language-agnostic messages and vice versa.*

The TMLR can execute programs that are composed from parts written in multiple languages. Programmers use different files for different programming languages. For example, if parts of a program are written in JavaScript and C (see Listing 4.1 and 4.2), these parts are in different files. Distinct files for each programming language allow us to reuse the existing parsers of each TLI without modification. It is therefore not necessary to combine these languages in a common grammar. Programmers can export data and functions to a shared multi-language scope and can also import data and functions from this scope. Figure 4.1 illustrates that all TLIs access and share a multi-language scope, which allows programmers to explicitly share data among other languages. JavaScript, Ruby, and C provide Truffle-built-ins to export and import data to and from the multi-language scope. For example the C code of Listing 4.1 exports the C `struct obj` to the multi-language scope and the JavaScript code of Listing 4.2 imports it.

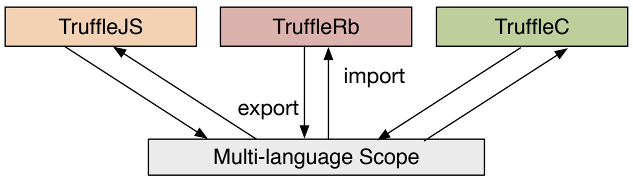


Figure 4.1: TLIs can import and export objects from and to a multi-language scope.

```c
#include<truffle.h>

typedef struct {
  int value;
} S;

S *obj = //...
Export("obj", obj);
```

Listing 4.1: This C snipped exports the variable `obj` which points to a `struct` of type `S`.

```javascript
var obj = Truffle.import("obj");
var a = obj.value;
```

Listing 4.2: This JavaScript snipped imports a variable `obj` and accesses its `value` member.

The TMLR distinguishes between two types of values that can be exchanged among languages:

Primitive types: The TMLR defines a set of *shared primitive types* to exchange primitive values across different languages. We refer to values with such a primitive type as *shared primitives*. This set of types includes all Java built-in types, such as all number classes that extend `java.lang.Number` and also the `java.lang.String` type. Shared primitives do not need to implement the *TruffleObject* interface (see Section 4.1) but can be exchanged directly. A TLI maps its language-specific primitive values to *shared primitive* values and exchanges them as language-agnostic values. Vice versa, a TLI maps *shared primitive* values to language-specific values. Using this set of types works well for TLIs because TLIs are itself written in Java, hence, the TLIs already map the primitive types of the guest language to Java types.

Object types: Every non-primitive type needs to be *shareable* in the sense that it supports *generic access* (see Section 4.1).

In the following we describe *generic access* in detail. *Generic access* allows accessing foreign objects *seamlessly*, i.e., it makes language boundaries mostly invisible to the programmer. In addition, programmers can use an *explicit* form of *generic access* to access foreign objects in cases where this access is not defined by the semantics of the host language (see Section 4.3).

4.1 Generic Access Mechanism

TLIs use different layouts for objects and each TLI uses language-specific AST nodes to access *regular objects*. In the context of this thesis, we use the term *object* for a non-primitive entity of a user program, which we want to share across different TLIs. Examples include data (such as JavaScript objects, Ruby objects, or C pointers), as well as functions, classes or code blocks. If the JavaScript implementation accesses a JavaScript object, the object is considered a *regular object*. If JavaScript (*host language*, L_{Host}) accesses a C `struct` (see Listing 4.2), the C `struct` is considered a *foreign object* (we call C the *foreign language*, $\text{L}_{\text{Foreign}}$). A foreign object has an *unknown type*, hence, the host language accesses it with *generic access*. *Object accesses* are operations that an L_{Host} can perform on objects, e.g., method calls, property accesses, or field reads. The property access `obj.value` (see Listing 4.2) is an example of a foreign object access.

4.1.1 Object Access via Messages

Every object that can be *shared* across different languages needs to support *generic access*. A *sharable* object implements a common interface, i.e., the `TruffleObject` interface. We implement this interface as a set of *messages*. For example, a L_{Host} can access the members of *sharable* objects by *Read* and *Write* messages. The L_{Host} inserts language-agnostic message nodes into the AST of a program to access these foreign objects. In the following we formally describe this transformation, which inserts message nodes into the AST of a host application.

TLIs transform source code to a tree of nodes, i.e., an AST. N^A and N^B define finite sets of nodes of TLIs A and B. Each node has $r : N^A \to \mathbb{N}$ children, where $\mathbb{N}$ denotes the set of natural numbers. If $n \in N^A$ is a node, then $r(n)$ is the number of its children. We call nodes with $r = 0$ *leaf* nodes. For example, nodes that represent constants and labels are *leaf* nodes. An AST $t \in T_{N^A}$ is a tree of nodes $n \in N^A$. By $n(t_1, ..., t_k)$ we denote a tree with root node $n \in N^A$ and k sub-trees $t_1, \ldots, t_k \in T_{N^A}$, where $k = r(n)$.

Generic access defines a set of messages, which are modeled as Truffle nodes N^{Msg}:

$$N^{\text{Msg}} = \{\text{Read}, \text{Write}, \text{Execute}, \text{Unbox}, \text{IsNull}\} \tag{4.1}$$

If TLI A encounters a foreign object at run time and a regular object access operation cannot be used, then TLI A maps the AST with the language-specific object access $t \in T_{N^A}$ to an AST with a language-agnostic, message-based object access $t' \in T_{N^A \cup N^{\text{Msg}}}$ using the function f_A:

$$T_{N^A} \xrightarrow{f_A} T_{N^A \cup N^{\text{Msg}}} \tag{4.2}$$

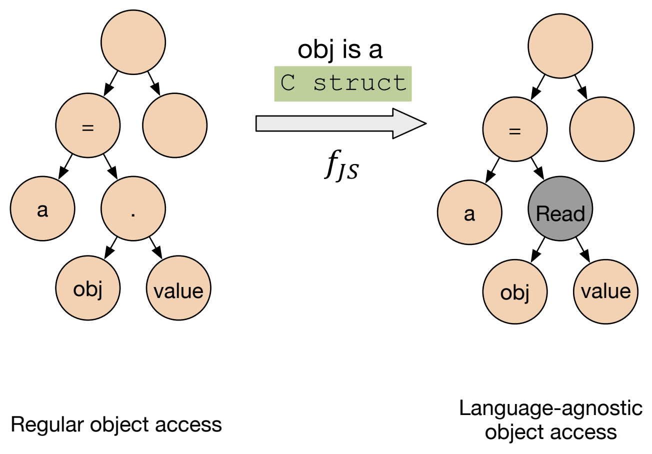


Figure 4.2: Accessing a C `struct` from JavaScript: We replace the property access with a *Read* message.

The function f_A replaces the language-specific access with a message-based access. The tree $t' \in T_{N^A \cup N^{\text{Msg}}}$ consists of language-specific nodes N^A and message nodes N^{Msg}. The other parts of the AST t remain unchanged. An L_{Host} that accesses foreign objects has to define this function f. To compose JavaScript, Ruby, and C we use the messages $n \in N^{\text{Msg}}$ where the sub-trees $t_1, \ldots, t_k \in T_{N^A \cup N^{\text{Msg}}}$ of $n(t_1, ..., t_k)$ evaluate to the arguments of the message.

Read: TLIs use the *Read* message to access a field of an object or an element of an array. It can also be used to access methods of classes or objects, i.e., to lookup executable methods from classes and objects.

$$\text{Read}(t_{\text{rec}}, t_{\text{id}}) \in T_{N^A \cup N^{\text{Msg}}} \tag{4.3}$$

The first subtree t_{rec} denotes the receiver of the *Read* message, the second subtree t_{id} the name or the index.

Consider the example in Figure 4.2 (a property access in JavaScript `obj.value`, see Listing 4.2), f_{JS} replaces the JavaScript-specific object access (node `"."` in the AST, implemented as a node of class *JSReadProperty*) with a *Read* message at run time if JavaScript suddenly encounters a foreign object.

$$\text{JSReadProperty}(t_{\text{obj}}, t_{\text{value}}) \xmapsto{f_{JS}} \text{Read}(t_{\text{obj}}, t_{\text{value}}) \tag{4.4}$$

Rather than using a *JSReadProperty* node to directly access the `value` property of the receiver `obj`, the JavaScript implementation uses a *Read* message to access the foreign object.

Write: A TLI uses the *Write* message to set the field of an object or the element of an array. It can also be used to add or change the methods of classes and objects.

$$\mathrm{Write}(t_{\mathrm{rec}}, t_{\mathrm{id}}, t_{\mathrm{val}}) \in T_{N^A \cup N^{\mathrm{Msg}}} \tag{4.5}$$

The first subtree t_{rec} denotes the receiver of the *Write* message, the second subtree t_{id} the name or the index, and the third subtree t_{val} the written value.

Execute: TLIs execute methods or functions using an *Execute* message.

$$\mathrm{Execute}(t_{\mathrm{f}}, t_1, \ldots, t_i) \in T_{N^A \cup N^{\mathrm{Msg}}} \tag{4.6}$$

The first subtree t_{f} denotes the function/method itself, the other arguments $t_1, \ldots, t_i$ denote the arguments.

Unbox: Programmers often use an object type to wrap a value of a primitive type in order to make it look like a real object. An *Unbox* message unwraps such a wrapper object and produces a primitive value. TLIs use this message to unbox a boxed value whenever a primitive value is required.

$$\mathrm{Unbox}(t_{\mathrm{rec}}) \in T_{N^A \cup N^{\mathrm{Msg}}} \tag{4.7}$$

The subtree t_{rec} denotes the receiver object.

IsNull: Many programming languages use `null`/`nil` for an undefined, uninitialized, empty, or meaningless value. The *IsNull* message allows the TLI to do a language-agnostic `null`-check.

$$\mathrm{IsNull}(t_{\mathrm{rec}}) \in T_{N^A \cup N^{\mathrm{Msg}}} \tag{4.8}$$

The subtree t_{rec} denotes the receiver object.

4.1.2 Message Resolution

The first execution of a message does not directly access the receiver object but triggers *message resolution.* The TMLR resolves the message to a foreign-language-specific AST snippet. $\mathrm{L}_{\mathrm{Foreign}}$ provides an AST snippet that the TMLR inserts into the host AST as a replacement for the message. This foreign-language-specific AST snippet depends on the type of the receiver and contains type-specific nodes for executing the message on the receiver. The TMLR maps the host AST with a language-agnostic access $t' \in T_{N^A \cup N^{\mathrm{Msg}}}$ to an AST with a foreign-language-specific access $t'' \in T_{N^A \cup N^B}$ using the function g_B, which is defined by $\mathrm{L}_{\mathrm{Foreign}}$:

$$T_{N^A \cup N^{\mathrm{Msg}}} \xrightarrow{g_B} T_{N^A \cup N^B} \tag{4.9}$$

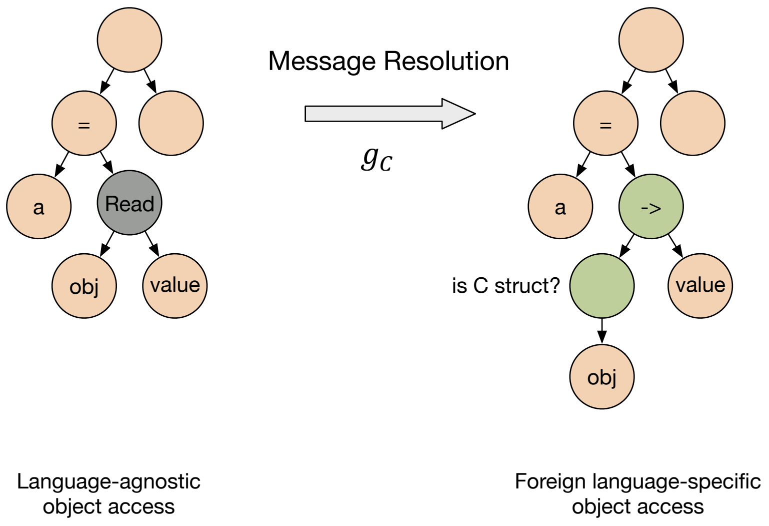

Message Resolution
g_C
=
a
Read
obj
value
=
a
->
is C struct?
value
obj
Language-agnostic object access
Foreign language-specific object access

Figure 4.3: Accessing a C `struct` from JavaScript: Message resolution inserts a C `struct` access into a JavaScript AST.

Message resolution produces an AST that consists of nodes $N^A \cup N^B$. The other parts of t' remain unchanged. With respect to the example in Figure 4.3, the TMLR uses g_C (defined by TruffleC) and replaces the *Read* message with a C-specific `struct` access operation upon its first execution. If the receiver is a pointer to a C `struct`, then TruffleC maps a *Read* message to a *CMemberRead* node (node `"->"` in the AST):

$$\text{Read}(t_{\text{obj}}, t_{\text{value}}) \xmapsto{g_C} \text{CMemberRead}(\text{IsStr}(t_{\text{obj}}), t_{\text{value}}) \tag{4.10}$$

The result is a JavaScript AST that embeds a C access operation $t'' \in T_{N^{\text{JS}} \cup N^C}$. After message resolution the receiver object is accessed directly rather than by a message. In order to notice an access to an object of a previously unseen language or a C object with a different type, message resolution inserts a guard into the AST that checks the receiver's type before it is accessed. This is shown in Figure 4.3 where message resolution inserts a node that checks if `obj` is a C `struct`. If this check fails, the execution falls back to the language-agnostic *Read* message, which will then be resolved to a new AST snippet.

An object access is *language-polymorphic* if it has varying receivers originating from different languages. In the polymorphic case, Truffle embeds the different language-specific AST snippets in a chain like an inline cache [53] to achieve best performance.

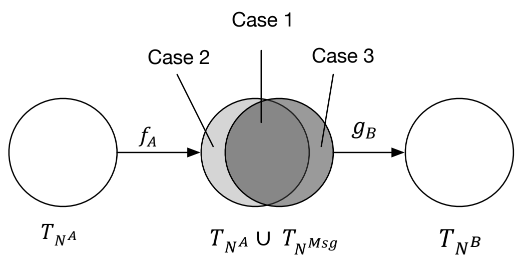


Figure 4.4: Venn diagram of a foreign object access.

4.2 Seamless Foreign Object Access

Language implementers have to define the functions f_A and g_B. The function f_A maps the AST t to an AST $t' \in T_{N^A \cup N^{\text{Msg}}}$ that uses a message to access the foreign object. The foreign language defines the function g_B that maps t' to an AST $t'' \in T_{N^A \cup N^B}$ with a foreign-language-specific object access:

$$\begin{gathered} T_{N^A} \xrightarrow{f_A} T_{N^A \cup N^{\text{Msg}}} \\ T_{N^A \cup N^{\text{Msg}}} \xrightarrow{g_B} T_{N^A \cup N^B} \end{gathered} \tag{4.11}$$

The TMLR composes TLIs automatically by composing f_A and g_B at run time:

$$g_B \circ f_A : T_{N^A} \to T_{N^A \cup N^B} \tag{4.12}$$

It creates an AST $t'' \in T_{N^A \cup N^B}$ where the main part is specific to language A and the foreign object access is specific to language B. When composing f_A and g_B three different cases can occur (see Figure 4.4):

1. If g_B is defined for $t' \in T_{N^A \cup N^{\text{Msg}}}$, a foreign object can be accessed seamlessly. The language B can replace the language-agnostic object access with a foreign-language-specific access.

2. If g_B is *not* defined for $t' \in T_{N^A \cup N^{\text{Msg}}}$, we report a run-time error with a high-level diagnostic message. The foreign object access is not supported. For example, if JavaScript accesses the length property of a C array, we report an error. C cannot provide length information for arrays.

3. A foreign object access might not be expressible in A, i.e., one wants to create $t' \in T_{N^A \cup N^{\text{Msg}}}$ but language A does not provide syntax for this access. For example, a C programmer cannot access the length property of a JavaScript array. In this case one has to fall back to an *explicit* foreign object access.

```
var arr = new Array(5);
Truffle.export("jsArray", arr);
```

Listing 4.3: Array allocation in JavaScript.

```
#include<truffle.h>

int * arr = (int*) Import("jsArray");
int length = Read_int32(arr, "length");
```

Listing 4.4: C code accessing the length property of an array.

4.3 Explicit Foreign Object Access

A host language might not provide syntax for a specific foreign object access. Consider the JavaScript array `arr` of Listing 4.3, which is used in a C program. C does not provide syntax for accessing the length property of an array. We overcome this issue by providing an interface to the programmer that allows explicitly using *generic access*. Using this interface, the programmer can fall back to an *explicit* foreign object access. The programmer directly uses *generic access* in order to access a foreign object. In other words, this interface allows programmers to handcraft the foreign object access of $t' \in T_{N^A \cup N^{\mathrm{Msg}}}$.

Every TLI has an API to use *generic access* explicitly. For example, to access the `length` property of a JavaScript array (see Listing 4.3) from C (see Listing 4.4), the programmer uses the TruffleC-built-in C function `Read_int32`. The C implementation substitutes this `Read_int32` invocation by a *Read* message.

4.4 Discussion

Generic access is a simple and efficient mechanism for cross-language interoperability between multiple languages. It has the following advantages:

Support for multiple languages: *Generic access* is independent of any TLI and can therefore compose multiple languages. Every TLI can access any object that supports *generic access*, which makes the TMLR extensible by new languages. In Section 5.1.3 we explain how a new TLI can be added to the TMLR.

Efficient multi-language development: We can achieve a seamless foreign object access by mapping *host-language access operations* to *messages*, which are then mapped back to *foreign-language-specific operations*. Compared to other approaches that use an explicit API for every interaction with a foreign language,

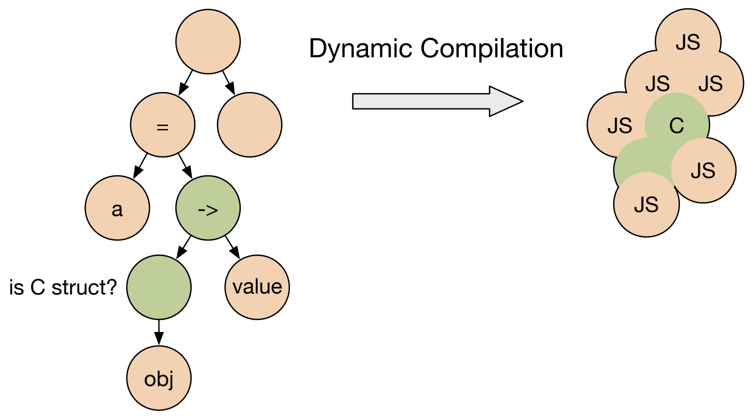


Figure 4.5: Language boundaries are completely transparent to the compiler.

this approach is simpler. It makes the mapping of access operations to messages largely the task of the language implementer rather than the task of the application programmer. Programmers are not forced to write boilerplate code as long as an object access can be mapped from language A to language B ($t \overset{f_A}{\longmapsto} t' \overset{g_B}{\longmapsto} t''$) via *generic access*. Only if not otherwise possible, programmers can explicitly use *generic access* to access foreign objects.

High-performance interoperability: Message resolution only affects the application's performance upon the first execution of an object access. By generating AST snippets for accessing foreign objects we avoid compilation barriers between languages. This allows the compiler to inline method calls even if the receiver is a foreign object (see Figure 4.5). Widening the compilation unit across different languages is important [3,95] as it enables the compiler to apply optimizations to a wider range of code.

Chapter 5

Implementation of a Multi-Language Runtime

In this chapter we present two case studies. First, we show an implementation of generic access *for JavaScript, Ruby, and C and explain how we compose these languages. With this case study we back our claim about supporting multiple languages and easing an efficient multi-language development. Second, we describe how we implement the C extensions API for Ruby. We use* generic access *to provide an implementation for this interface, which allows us to run existing production-code. With this case study we back our claim that the TMLR can support legacy interfaces between languages.*

5.1 Interoperability between JavaScript, Ruby, and C

We want to provide a system that allows programmers to write multi-language applications where language boundaries are completely transparent. We first discuss the implementation of *generic access* for JavaScript, Ruby, and C and we present a mapping from host-language-specific operations to messages and from there to foreign-language-specific operations. Second, we discuss how we can bridge differences between JavaScript, Ruby, and C. These differences are object-oriented and non-object-oriented programming, dynamic and static typing, explicit and automatic memory management, or safe and unsafe memory access.

```
// Imports a TruffleObject from the multi-language scope
Truffle.import(id);
// Exports an object to the multi-language scope
Truffle.export(id, val);

// Reads from a TruffleObject
Truffle.read(receiver, id);
// Writes to a TruffleObject
Truffle.write(receiver, id, value);

// ...
```

Listing 5.1: Excerpt of the built-ins (`Truffle`-object methods) for JavaScript.

5.1.1 Implementation of Generic Access

TruffleJS

JavaScript is a prototype-based scripting language with dynamic typing. It is almost completely object-oriented. JavaScript objects are associative arrays that have a prototype, which corresponds to their dynamic type. Object property names are string keys and it is possible to add, change, or delete properties of an object at run time. JavaScript objects are implemented using the `DynamicObject`. In TruffleJS, all data objects are shareable in the sense that they support *generic access*. TruffleJS maps property accesses to *Read* and *Write* messages and vice versa. Functions are first class objects. TruffleJS maps a function invocation to an *Execute* message and vice versa. TruffleJS maps incoming numeric primitive values to objects of type `Number`, i.e., JavaScript's type for representing number values. Also, TruffleJS unboxes (using the *Unbox* message) boxed foreign primitive values (e.g. Ruby's `Float`) and maps them to objects of type `Number`. `Number` objects support the *Unbox* message, which allows sharing them among other languages. *Unbox* maps the value to a numeric *shared primitive*. Table 5.1 summarizes the mapping of JavaScript operations from and to messages.

The implementation of *generic access* for TruffleJS also introduces a built-in object `Truffle` (see Listing 5.1). The `Truffle` object defines functions for importing and exporting objects from and to the multi-language scope. Also, it defines functions to explicitly access foreign objects. TruffleJS substitutes every call to these functions by a *generic access* or a multi-language scope access.

Access	T_{N^A}	$\xrightarrow{f_A}$	$T_{N^A \cup N^{\text{Msg}}}$
Property read	$\text{JSReadProperty}(t_{\text{rec}}, t_{\text{id}})$	$\xmapsto{f_{\text{JS}}}$	$\text{Read}(t_{\text{rec}}, t_{\text{id}})$
Property write	$\text{JSWriteProperty}(t_{\text{rec}}, t_{\text{id}}, t_{val})$	$\xmapsto{f_{\text{JS}}}$	$\text{Write}(t_{\text{rec}}, t_{\text{id}}, t_{val})$
Is null	$\text{JSIsNil}(t_{\text{rec}})$	$\xmapsto{f_{\text{JS}}}$	$\text{IsNull}(t_{\text{rec}})$
Call	$\text{JSCall}(t_{\text{func}}, t_{\text{rec}} \ldots)$	$\xmapsto{f_{\text{JS}}}$	$\text{Execute}(t_{\text{func}}, t_{\text{rec}} \ldots)$

Receiver t_{rec}	$T_{N^A \cup N^{\text{Msg}}}$	$\xrightarrow{g_B}$	$T_{N^A \cup N^B}$
Object	$\text{Read}(t_{\text{rec}}, t_{\text{id}})$	$\xmapsto{g_{\text{JS}}}$	$\text{JSReadProperty}(\text{IsJS}(t_{\text{rec}}), t_{\text{id}})$
Object	$\text{Write}(t_{\text{rec}}, t_{\text{id}}, t_{val})$	$\xmapsto{g_{\text{JS}}}$	$\text{JSWriteProperty}(\text{IsJS}(t_{\text{rec}}), t_{\text{id}}, t_{val})$
Object	$\text{IsNull}(t_{\text{rec}})$	$\xmapsto{g_{\text{JS}}}$	$\text{JSIsNil}(\text{IsJS}(t_{\text{rec}}))$
Object	$\text{Unbox}(t_{\text{rec}})$	$\xmapsto{g_{\text{JS}}}$	$\text{JSNumberToFloat64}(\text{IsJS}(t_{\text{rec}}))$
Object	$\text{Execute}(t_{\text{rec}}, \ldots)$	$\xmapsto{g_{\text{JS}}}$	$\text{JSCall}(\text{IsJS}(t_{\text{rec}}), \ldots)$

Table 5.1: Mapping JavaScript access operations to messages and vice versa.

TruffleRuby

The Ruby language is heavily inspired by Smalltalk, hence, in Ruby there are no primitive types. Every value — including numeric values — is represented as an object. TruffleRuby implements these objects using the `RubyBasicObject` class. The `RubyBasicObject` wraps the `DynamicObject`, which is used to store the data content of a Ruby object. Operations (e.g. arithmetic operations) as well as data access operations (accessing object attributes or array elements) are modeled as function calls on the receiver object. For example, Ruby arrays or hashes provide a setter method `[]=` to set an element of a Ruby array or hash. We map getter and setter invocations (functions `[]` and `[]=`) to *Read* and *Write* messages and vice versa. In TruffleRuby, all data objects as well as all methods support *generic access* and are therefore sharable. Table 5.2 summarizes the mapping of Ruby operations to and from messages. TruffleRuby maps incoming primitive values to objects of numeric type `Fixnum` and `Float` if this is possible without loss of information (e.g. no truncation or rounding). These objects are also sharable with other languages, i.e., they 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 use an *Unbox* message whenever it needs the object's value for an arithmetic operation.

Similar to TruffleJS, TruffleRuby defines a built-in class `Truffle`, which allows exporting and importing variables to and from the multi-language scope and to explicitly access foreign objects.

Access	T_{N^A}	$\xrightarrow{f_A}$	$T_{N^A \cup N^{\text{Msg}}}$
Method call	RbDispatch(RbResolve($t_{\text{rec}}, t_{funcId}$), $t_{\text{rec}} \ldots$)	$\xmapsto{f_{\text{Ruby}}}$	Execute(Read($t_{\text{rec}}, t_{funcId}$), $t_{\text{rec}} \ldots$)
Is null	RbIsNil(t_{rec})	$\xmapsto{f_{\text{Ruby}}}$	IsNull(t_{rec})
Getter call	RbDispatch(RbResolve(t_{rec}, "[]"), $t_{\text{rec}}, t_{\text{id}}$)	$\xmapsto{f_{\text{Ruby}}}$	Read($t_{\text{rec}}, t_{\text{id}}$)
Setter call	RbDispatch(RbResolve(t_{rec}, "[]="), $t_{\text{rec}}, t_{\text{id}}, t_{\text{val}}$)	$\xmapsto{f_{\text{Ruby}}}$	Write($t_{\text{rec}}, t_{\text{id}}, t_{val}$)

Receiver t_{rec}	$T_{N^A \cup N^{\text{Msg}}}$	$\xrightarrow{g_B}$	$T_{N^A \cup N^B}$
Object	Execute(Read(t_{rec}, t_{id}), $t_{\text{rec}} \ldots$)	$\xmapsto{g_{\text{Ruby}}}$	RbDispatch(RbResolve(IsRbObj(t_{rec}), t_{id}), $t_{\text{rec}} \ldots$)
Object	IsNull(t_{rec})	$\xmapsto{g_{\text{Ruby}}}$	RbIsNil(IsRbObj(t_{rec}))
Array	Read($t_{\text{rec}}, t_{\text{id}}$)	$\xmapsto{g_{\text{Ruby}}}$	RbDispatch(RbResolve(IsRbAry(t_{rec}), "[]"), $t_{\text{rec}}, t_{\text{id}}$)
Array	Write($t_{\text{rec}}, t_{\text{id}}, t_{val}$)	$\xmapsto{g_{\text{Ruby}}}$	RbDispatch(RbResolve(IsRbAry(t_{rec}), "[]="), $t_{\text{rec}}, t_{\text{id}}, t_{val}$)
Hash	Read($t_{\text{rec}}, t_{\text{id}}$)	$\xmapsto{g_{\text{Ruby}}}$	RbDispatch(RbResolve(IsRbHsh(t_{rec}), "[]"), $t_{\text{rec}}, t_{\text{id}}$)
Hash	Write($t_{\text{rec}}, t_{\text{id}}, t_{val}$)	$\xmapsto{g_{\text{Ruby}}}$	RbDispatch(RbResolve(IsRbHsh(t_{rec}), "[]="), $t_{\text{rec}}, t_{\text{id}}, t_{val}$)
FixNum	Unbox(t_{rec})	$\xmapsto{g_{\text{Ruby}}}$	RbFixNumToInt64(IsRbFN(t_{rec}))
Float	Unbox(t_{rec})	$\xmapsto{g_{\text{Ruby}}}$	RbFloatToFloat64(IsRbFt(t_{rec}))
Function	Execute($t_{\text{rec}}, \ldots$)	$\xmapsto{g_{\text{Ruby}}}$	RbDispatch(IsRbFunc(t_{rec}), ...)

Table 5.2: Mapping Ruby access operations to messages and vice versa.

TruffleC

TruffleC can share primitive C values, mapped to *shared primitive values*, as well as pointers to C data with other languages. In our implementation, pointers are objects that support *generic access*, which allows them to be shared across all TLIs. TruffleC represents all pointers (i.e., pointers to values, arrays, `struct`s or functions) as `CAddress` objects that wrap a 64-bit value. This value is a pointer to the native heap. Besides the address value, a `CAddress` object also stores type information about the pointee. Depending on the type of the pointee, TruffleC resolves 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. Any pointer can resolve the *IsNull* message, which checks whether the pointer is a `null`-pointer. Finally, `CAddress` objects that reference a C function can be executed using the *Execute* message.

TruffleC can bind `CAddress` objects as well as shared foreign objects to pointer variables. TruffleC uses *generic access* to access these foreign objects. Table 5.3 summarizes how we map C operations to messages and vice versa.

```
// Imports a TruffleObject from the multi-language scope
void* Import(const char* id);
// Exports a pointer to the multi-language scope
void  Export(const char* id, void* val);

// Reads a 32-bit integer from a TruffleObject
int   Read_int32(void* receiver, const char* id);
// Writes a 32-bit integer to a TruffleObject
void  Write_int32(void* receiver, const char* id, int value);

// ...
```

Listing 5.3: Excerpt of the `truffle.h` header file.

C is a statically typed language and every expression has a static type. Hence, if TruffleC accesses a foreign object and the access expression has a primitive type, then it tries to convert the result to a value that has the same type as the expression. Consider the following example (Listing 5.2):

```
int *arr = Import("arr");
int value = arr[0];
```

Listing 5.2: C `int` expression that reads from a foreign object.

TruffleC converts the result of `arr[0]` to a C `int` value. It directly converts shared primitive values to an `int` value if this is possible without loss of information. If the result is a `TruffleObject`, TruffleC uses *Unbox* and then does the conversion. If a conversion is not possible, TruffleC raises a run-time error and reports the type-incompatibility.

The implementation of *generic access* for TruffleC also introduces the header file `truffle.h` (see Listing 5.3). This header file defines functions for importing and exporting objects from and to the multi-language scope. Also, it defines functions to explicitly access foreign objects. There are different versions of these functions for all primitive C types. For example, there is an explicit *Read* function `Read_int32` that reads from a foreign object and tries to convert the result to an `int` value. None of the functions that are defined in `truffle.h` have an implementation in C. Instead, TruffleC substitutes every invocation by a *generic access* or a multi-language scope access.

Access	T_{N^A}	$\xrightarrow{f_A}$	$T_{N^A \cup N^{\text{Msg}}}$
`struct` read	$\text{CMemberRead}(t_{\text{rec}}, t_{\text{id}})$	$\xmapsto{f_C}$	$\text{Read}(t_{\text{rec}}, t_{\text{id}})$
`struct` write	$\text{CMemberWrite}(t_{\text{rec}}, t_{\text{id}}, t_{val})$	$\xmapsto{f_C}$	$\text{Write}(t_{\text{rec}}, t_{\text{id}}, t_{val})$
Array read	$\text{CElementRead}(t_{\text{arr}}, t_{\text{idx}})$	$\xmapsto{f_C}$	$\text{Read}(t_{\text{arr}}, t_{\text{idx}})$
Array write	$\text{CElementWrite}(t_{\text{arr}}, t_{\text{idx}}, t_{val})$	$\xmapsto{f_C}$	$\text{Write}(t_{\text{arr}}, t_{\text{idx}}, t_{val})$
Is null	$\text{CIsNullPointer}(t_{\text{rec}})$	$\xmapsto{f_C}$	$\text{IsNull}(t_{\text{rec}})$
Function call	$\text{CCall}(t_{\text{func}}, \ldots)$	$\xmapsto{f_C}$	$\text{Execute}(t_{\text{func}}, \ldots)$

Receiver t_{rec}	$T_{N^A \cup N^{\text{Msg}}}$	$\xrightarrow{g_B}$	$T_{N^A \cup N^B}$
Ptr to a `struct`	$\text{Read}(t_{\text{rec}}, t_{\text{id}})$	$\xmapsto{g_C}$	$\text{CMemberRead}(\text{IsStr}(t_{\text{rec}}), t_{\text{id}})$
Ptr to a `struct`	$\text{Write}(t_{\text{rec}}, t_{\text{id}}, t_{val})$	$\xmapsto{g_C}$	$\text{CMemberWrite}(\text{IsStr}(t_{\text{rec}}), t_{\text{id}}, t_{val})$
Ptr to an array	$\text{Read}(t_{\text{rec}}, t_{idx})$	$\xmapsto{g_C}$	$\text{CElementRead}(\text{IsAry}(t_{\text{rec}}), t_{idx})$
Ptr to an array	$\text{Write}(t_{\text{rec}}, t_{idx}, t_{val})$	$\xmapsto{g_C}$	$\text{CElementWrite}(\text{IsAry}(t_{\text{rec}}), t_{idx}, t_{val})$
Any ptr	$\text{IsNull}(t_{\text{rec}})$	$\xmapsto{g_C}$	$\text{CIsNullPointer}(\text{IsC}(t_{\text{rec}}))$
Function ptr	$\text{Execute}(t_{\text{rec}}, \ldots)$	$\xmapsto{g_C}$	$\text{CCall}(\text{IsFunc}(t_{\text{rec}}), \ldots)$

Table 5.3: Mapping C access operations to messages and vice versa.

5.1.2 Different Language Paradigms and Features

In this section we describe an intuitive approach for bridging the different paradigms and features of JavaScript, Ruby, and C. We focus on these languages and explain how we deal with dynamic and static typing, object-oriented and non-object oriented programming, explicit and automatic memory management, as well as safe and unsafe memory accesses. For this discussion, consider a multi-language application, which consists of two files. The first file (Listing 5.4) contains JavaScript code and the second file (Listing 5.5) contains C code. The example allocates a JavaScript object (the `counter` object, see Listing 5.4), which is then used by the C code (see Listing 5.5). The JavaScript code exports the object `counter` using a built-in function. This built-in stores the reference to the JavaScript object into the multi-language scope of the application. In turn, the C code imports the object using a C built-in, defined in `truffle.h`.

Ruby and JavaScript are more similar (both languages are dynamically typed, object-oriented, and use the automatic memory management of the TMLR). To keep the example simple, we write this application in JavaScript and C. However, the following ideas directly apply also for Ruby.

```
var counter = {
    counterValue: 0,
    add: function(val) {
        counterValue += val;
    },
    myPrint: function() {
        print(counterValue);
    }
}
Truffle.export("counter", counter);
```

Listing 5.4: Allocation of a JavaScript object.

```
#include<truffle.h>

typedef struct {
    void (*add)(void *this, int val);
    void (*myPrint)(void *this);
} Counter;

int main() {
    Counter *c = (Counter*) Import("counter");
    c->add(c, 42);
    c->myPrint(c);
}
```

Listing 5.5: C type definition for the foreign object and an object-oriented access operation.

Dynamic and Static Typing

In [113], Wrigstad et al. describe a concept called *like types*, which allows integrating dynamically typed objects in statically typed languages. Dynamically typed objects can be used in a statically typed language by binding them to *like-type* variables. Operations on like-type variables are syntactically and semantically checked against the static type of these variables, but the actual validity of these operations is only checked at run time. Our approach is similar, except that in our case any pointer variable in C can be bound to a foreign object. We bind foreign dynamically typed objects to pointer variables that are associated with static type information. If a pointer is bound to a dynamically typed value, we check the usage dynamically, i.e., upon each access we check whether the operation on the foreign object is possible. We report a run-time error otherwise.

Listing 5.5 shows a C program, which uses a JavaScript object `counter`. The C code associates the variable `c` with the static type `Counter*`, which is defined by the programmer. When the C code accesses the JavaScript object, we check whether `add` or `myPrint` exist and report an error otherwise.

Object-Oriented and Non-Object-Oriented Programming

The object-oriented programming paradigm allows programmers to create objects that contain both data and code, known as *fields* and *methods*. Also, objects can extend each other (e.g. class-based inheritance or prototype-based inheritance); when accessing fields or methods, the object does a lookup and provides a field value or a method. *Generic access* allows us to retain the object-oriented semantics of an object access even if the host language is not object-oriented. Consider the `add` method invocation (from C to JavaScript) in Listing 5.5. TruffleC maps this access to the following messages:

$$\begin{gathered}\text{CCall}(\text{CMemberRead}(t_\text{c}, t_\text{add}), t_\text{c}, t_{42}) \\ \xmapsto{f_C} \text{Execute}(\text{Read}(t_\text{c}, t_\text{add}), t_\text{c}, t_{42})\end{gathered} \tag{5.1}$$

TruffleJS resolves this access to an AST snippet that does the lookup of method `add` and executes it:

$$\begin{gathered}\text{Execute}(\text{Read}(t_\text{c}, t_\text{add}), t_\text{c}, t_{42}) \\ \xmapsto{g_{JS}} \text{JSCall}(\text{IsJS}(\text{JSReadProperty}(\text{IsJS}(t_\text{c}), t_\text{add})), t_\text{c}, t_{42})\end{gathered} \tag{5.2}$$

A method call in an object-oriented language passes the `this` object (i.e., the receiver) as an implicit argument. Non-object oriented languages that invoke methods therefore need to explicitly pass the `this` object. For example, the JavaScript function `add` (see Listing 5.4) expects the `this` object as an implicit first argument. Hence, the first argument of the `add` method call in C is the `this` object `c`.

Vice versa, the signature of a non-object-oriented function needs to contain the `this` object argument if the caller is an object-oriented language. For example, if JavaScript calls the C function, JavaScript passes the `this` object as the first argument. The signature of the C function needs to add the `this` object argument to its signature explicitly. This approach allows us to access object-oriented data from a non-object-oriented language and vice versa.

Limitations: Our current approach does not feature cross-language inheritance, i.e., class-based inheritance or prototype-based inheritance is only possible with objects that originate from the same language.

Explicit and Automatic Memory Management

TLIs are running within the TMLR and can exchange data, independent of whether the data is managed or unmanaged:

Unmanaged allocations: TLIs keep unmanaged allocations on the native heap, which is not garbage collected. For example, TruffleC allocates data such as arrays or `structs` on the native heap. When accessing a `CAddress` object via *generic access*, the access will resolve to a raw memory accesses. *Generic access* allows accessing unmanaged data from a language that otherwise only uses managed data.

Managed allocations: TLIs keep managed allocations on the Java heap, which is garbage collected. For example, the JavaScript and Ruby implementations use Truffle's `DynamicObject` to represent their data structures. If an application binds a managed object to a C variable (e.g. `Counter *c = Import("counter");`, see Listing 5.5), then TruffleC keeps the pointer to this variable in the Truffle `Frame`. The `Frame`'s `Object` array holds the values of the function's local variables (see Section 3.1.5). Thus, the Java garbage collector can trace managed objects even if they are referenced from unmanaged languages.

Limitations: If a C pointer variable references an object of a managed language, operations are restricted. First, pointer arithmetic on foreign objects is only allowed as an alternative to array indexing. For example, C programmers can access a JavaScript array either with indexing (e.g. `jsArray[1]`) or by pointer arithmetic (`*(jsArray + 1)`). However, it is not allowed to manipulate a pointer variable that is bound to a managed object in any other way (e.g. `jsArray = jsArray + 1`). Second, C pointer variables that are bound to managed objects cannot be casted to primitive values (such as `long` or `int`). References to the Java heap cannot be represented as primitive values like it is possible for raw memory addresses. Finally, unmanaged data structures cannot store references to managed objects. For example, it is not possible to assign a reference to a managed JavaScript object to a C array of pointers. `Counter *array[] = {jsReference};` is forbidden. Non-primitive C data is stored as a sequence of bytes on the native heap and it is not possible to keep managed references on the native heap. We report a run-time error-message in these cases.

Safe and Unsafe Memory Accesses

C is an unsafe language and does not check memory accesses at run time, i.e., there are no run-time checks that ensure that pointers are only dereferenced if they point to a valid memory region and that pointers are not used after the referenced object has been deallocated. TruffleC allocates data on the native heap and uses raw memory operations to access it, which is unsafe. This has the following implications on multi-language applications:

Unsafe accesses: If C shares data with a safe language, all access operations are *unsafe*. For example, accessing a C array in JavaScript is unsafe. If the index is out of bounds, the access has an undefined behavior (as defined by the C specification). However, accessing a C array is more efficient than accessing a dynamic JavaScript array because less run-time checks are required.

Safe accesses: Accessing data structures of a safe language (such as JavaScript) from C is *safe*. For example, accessing a JavaScript array in C is safe. TruffleC implements the access by a *Read* or *Write* message, which TruffleJS resolves with operations that check if the index is within the array bounds and grow the array in case the access was out of bounds.

5.1.3 Discussion

In contrast to many other cross-language mechanisms (e.g., FFIs), our *generic access* mechanism works for any TLI. This applies also to new TLIs if they support the following:

Transforming foreign object accesses to language-agnostic messages: If a TLI wants to act as a host language and access another foreign language, it needs to map object accesses to messages, i.e., a TLI (L_{New}) has to define $T_{N^{\text{New}}} \xrightarrow{f_{\text{New}}} T_{N^{\text{New}} \cup N^{Msg}}$. Optionally, a TLI can provide an API that allows programmers to explicitly use *generic access* (see Section 4.3).

Transforming language-agnostic messages to regular object accesses: If a TLI (L_{New}) wants to be used as a foreign language and share objects with other languages, shared objects need to support *generic access*. The TLI needs to define a mapping from language-agnostic messages to access operations that are specific to L_{New}: $T_{N^{\text{A}} \cup N^{Msg}} \xrightarrow{g_{\text{New}}} T_{N^{A} \cup N^{\text{New}}}$.

Multi-language scope: The TLI has to provide infrastructure for the application programmer to export and import objects to and from the multi-language scope.

The TMLR, including the TLIs for JavaScript, Ruby, and C, eases an efficient multi-language development, which we evaluate by writing and executing multi-language benchmarks (see also Section 7). We modified single-language benchmarks, which are available in C, Ruby, and JavaScript, such that parts of them were written in a different language. We extracted all array and object allocations into factory functions. We then replaced these factory functions with implementations in different languages, making the benchmarks multi-language applications. The other parts did not have to be changed, because accesses to foreign objects can simply be written in the language of the host. The only extra code that we needed was for importing and exporting objects from and to the multi-language scope.

5.2 C Extensions Support for TruffleRuby

In this second case study we implement the C extensions API for Ruby. A C extension is a C program that can access the data and metadata of a Ruby program by using a set of API functions (*C extension functions*), which are part of the Ruby VM MRI. Developers of a C extension for Ruby access this API by including the `ruby.h` header file. The C extension code is then dynamically loaded and linked into the Ruby VM as a program runs. Figure 5.1 gives an architectural overview of a Ruby application using a C extension.

We provide the same C extensions API as Ruby does, i.e., we 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 the C extension functions. Listing 5.6 shows an excerpt of this header file including a description of the function's semantic. In the following we discuss how we can provide an implementation for these functions. We distinguish between *local* and *global* functions in the C extensions API. Local functions access and manipulate Ruby objects from within C. Global functions manipulate the global object of a Ruby application from C or directly access the Ruby engine.

5.2.1 Local Functions

TruffleC substitutes every call to a local C extension function with a message in the AST that accesses the foreign Ruby data directly. The result is an AST $t' \in T_{N^C \cup N^{\mathrm{Msg}}}$, which uses messages to access the foreign object rather than calling a function of the C extensions API.

The TMLR can resolve these messages because TruffleRuby provides a mapping g_{Rb}. Message resolution uses g_{Rb} to map the messages in $t' \in T_{N^C \cup N^{\mathrm{Msg}}}$ to an AST with a Ruby-specific access $t'' \in T_{N^C \cup N^{\mathrm{Rb}}}$:

$$T_{N^C \cup N^{\mathrm{Msg}}} \xrightarrow{g_{\mathrm{Rb}}} T_{N^C \cup N^{\mathrm{Rb}}} \tag{5.3}$$

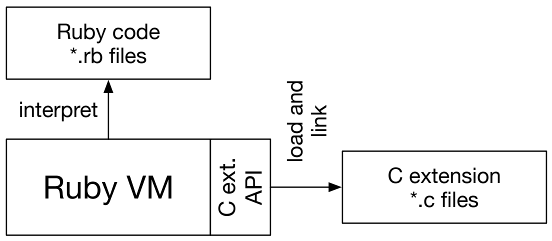


Figure 5.1: Architecture of a Ruby application using C extensions.

```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 a C long
long FIX2INT(VALUE value);
```

Listing 5.6: Excerpt of the `ruby.h` implementation.

```c
#include<ruby.h>

VALUE array = ... ; // Ruby array of Fixnums
VALUE value = ... ; // Ruby Fixnum

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

Listing 5.7: Accessing a Ruby array from C.

The function to resolve the messages to Ruby-specific access operations (g_{Rb}) remained unchanged and we were able to reuse the infrastructure that was already implemented in TruffleRuby.

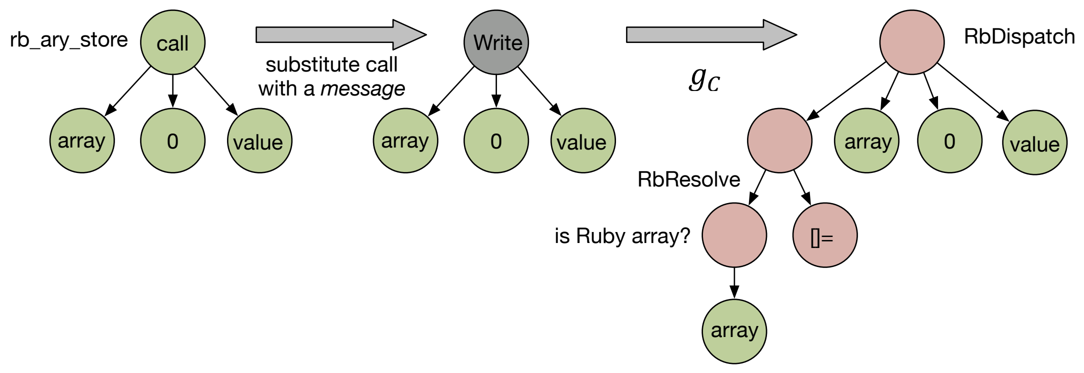


Figure 5.2: TruffleC substitutes invocations of C extension functions with messages; the TMLR resolves them to Ruby-specific operations.

The following examples explain how we substitute the C extension functions `rb_ary_store`, `rb_funcall`, and `FIX2INT`.

`rb_ary_store` allows writing an element of a Ruby array. Figure 5.2 shows how TruffleC substitutes the call of `rb_ary_store` (see source code in Listing 5.7) with a *Write* message for setting the Ruby array element. Upon first execution, this message is resolved by the TMLR, which results in a TruffleC AST that does a Ruby array access via a setter function (`[]=`). The resolved AST replaces the AST for the *Write* message and is executed from now on.

`rb_funcall` allows invoking a method on a Ruby object from within a C extension. TruffleC substitutes this call by two messages, namely a *Read* message to get the method from the Ruby receiver and an *Execute* message, which invokes the method.

`FIX2INT` transforms a Ruby Fixnum object to a C integer value. TruffleC substitutes this call by an *Unbox* message to the Ruby object.

5.2.2 Global Functions

The C extensions API also offers functions that manipulate the global object class of a Ruby application from C. For example, these functions can define global variables, modules, or functions. Global functions can also directly access the Ruby engine (e.g., to convert a C string to an immutable Ruby object). TruffleC forwards invocations of these global C extension functions to an API of the TruffleRuby engine.

In the following we discuss how TruffleC implements calls to `rb_define_method` and `rb_intern`.

`rb_define_method` allows defining a new method in a Ruby class. To substitute an invocation to this function, TruffleC directly accesses the Ruby engine and adds a C function pointer to a Ruby class object. The function pointer `VALUE(*func)()` is a `CAddress` object. When TruffleRuby invokes this method later, it uses *generic access*, i.e., it uses an *Execute* message to invoke the C function.

`rb_intern` provides a shared immutable Ruby object representation for a C string. TruffleRuby exposes a utility function that allows resolving these immutable Ruby objects, which TruffleC uses to substitute invocations of this methods.

5.2.3 Pointers to Ruby Objects

In TruffleC, a pointer to a Ruby object is a Java reference to a *sharable* `TruffleObject`. If the C extension introduces additional indirections (i.e., taking a pointer to a variable that holds a Ruby object) we create an `MAddress` object (see Chapter 6). `MAddress` objects wrap the Ruby pointee and also hold a numeric offset value. They allow introducing arbitrary levels of indirection. Also, the offset value allows supporting pointer arithmetic. Whenever the Ruby object needs to be accessed, we use *generic access* again.

5.2.4 Discussion

This thesis claims that the TMLR eases an efficient multi-language development that also supports legacy interfaces between languages. In this case study we show an implementation of the C extensions API using *generic access*. We can provide an implementation that allows us to run Ruby code with C extensions that have been developed to meet a real business need (see also Section 7). We were able to successfully execute the existing modules `chunky_png`[1] and `psd.rb`[2], 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. Running the C extensions of these gems on top of the TMLR required the following modifications for compatibility: TruffleC does not support variable-length arrays (see Section 3.1.6), hence, we replaced two instances of variable size stack allocations with

[1] *Chunky PNG*, Willem van Bergen and others, 2015: `https://github.com/wvanbergen/chunky_png`

[2] *PSD.rb from Layer Vault*, Ryan LeFevre, Kelly Sutton and others, 2015: `https://cosmos.layervault.com/psdrb.html`

a heap allocation via `malloc` and `free`. Running the C extensions on TruffleC also allowed us to find two bugs. A value of type `VALUE` (64-bit pointer value) was stored in a variable of type `int` (32-bit integer value), which caused different results between the Ruby module and the C extensions on all Ruby implementations. We have reported this implementation bug to the module's authors[3]. Apart from these minor modifications we are running all native routines from the two non-trivial gems unmodified.

The implementation of the C extensions API for Ruby serves as empirical evidence that the TMLR can support legacy interfaces between languages.

[3] *PSDNative*, Bug report, 2015: `https://github.com/layervault/psd_native/pull/4`

Chapter 6

Managed Data Allocations for C

In this chapter we present a third case study, which demonstrates that the applications of generic access *are manifold and not limited to cross-language interoperability. We use* generic access *to substitute native allocations with managed allocations in TruffleC, hence, we can ensure* spatial *and* temporal *memory safety of a C program execution.*

This thesis claims that *generic access* is a flexible approach for language composition, which is independent of languages and data structures. We backed this claim with case studies of cross-language interoperability (see Section 5), however, in this section we provide another case study where we apply *generic access* differently. We extend TruffleC and use *generic access* to substitute native allocations (i.e., objects on the native heap) with managed allocations (i.e., objects on the Java heap). Managed allocations ensure memory safety of a C program execution. Note that this is an additional contribution of our thesis, which is orthogonal to the other contributions in the area of cross-language interoperability. Furthermore, the case study in this chapter allows us to back the flexibility claims of *generic access* and provides a further performance evaluation that measures *generic access* (see Section 7).

We call the memory safe version of TruffleC *TruffleC$_M$*, where M stands for *managed*. TruffleC$_{\mathrm{M}}$ uses only managed Java objects to represent the data of a C program execution and does not allocate native data at all. TruffleC$_{\mathrm{M}}$ ensures *spatial* and *temporal* memory safety of C programs. A program execution is considered *memory safe* [75,76,101] if it ensures *spatial* and *temporal* memory safety. *Spatial* memory safety ensures that pointers are only dereferenced if they point to a valid memory region [93], i.e., if the access is within the bounds of the accessed object. This prevents errors such as *null pointer dereferences* or *buffer overflows*. *Temporal* memory safety ensures that pointers are not used after the referenced object has been deallocated [93]. This prevents errors such as *dangling pointers* or *illegal deallocations* (e.g. calling `free` on a pointer twice).

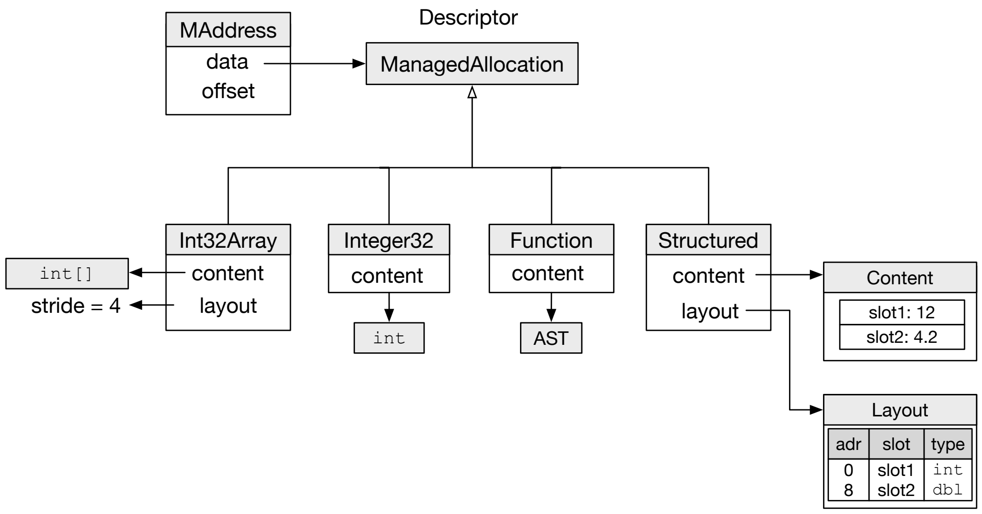


Figure 6.1: `MAddress` objects are pointers to managed allocations.

In the following, we first describe the new data structures that we define (Section 6.1) and how they are allocated (see Section 6.2). Afterwards, we explain how we can use *generic access* to safely access these data structures (Section 6.3).

6.1 Managed Addresses and Managed Objects

Managed objects cannot be referenced by `CAddress` objects, which wrap raw pointer values. Hence, TruffleC$_M$ introduces `MAddress` objects that substitute the `CAddress` objects of TruffleC. `MAddress` objects store a `data` field and an `offset` (see Figure 6.1). The `data` field references a descriptor representing the managed allocation, while the `offset` field holds a byte offset relative to the beginning of the managed allocation. It is updated by address computations and pointer arithmetic and allows us to represent pointers that point into an object (e.g. to a `struct` member). When dereferencing an `MAddress` object, the offset is passed to the descriptor, which can map (e.g. using a layout table) the offset to a data entry of the content (`content` field of a `ManagedAllocation`). Thus we can check whether a pointer references a valid member.

Managed Objects

In TruffleC$_M$, objects of a C program can be represented as *array objects*, *primitive objects*, *function objects*, and *structured objects* (see Figure 6.1). All of them are `ManagedAllocations` and can be referenced by an `MAddress` object.

Array objects: We represent C arrays by objects that box a Java array. For example, if C code allocates an `int` array, TruffleC$_M$ allocates an `Int32Array` object that wraps a Java `int` array. An array object holds the size of an element in bytes (`layout` field). When an element is accessed, the offset of the `MAddress` is divided by this size to get the index of the array element.

Primitive objects: We represent primitive C values (e.g., `int`, `double`, ...) as Java objects that box a Java primitive value.

Function objects: C functions are represented as ASTs, wrapped in `Function` objects. `MAddress` objects can point to a `Function` object (function pointers), which can be executed by TruffleC$_\text{M}$.

Structured objects: C `structs` and `unions` are represented by `Structured` objects. The content of an `Structured` object is an object of type `DynamicObject` [112] (see Section 2.2.4) that contains the values of all assigned members. The slots of a `DynamicObject` can contain primitive values, address values, as well as other managed allocations. Consider the example in Listing 6.1, which shows the declaration of a `struct S`. We assume the object representation of the `struct` on an x86-64 platform. The `int` member `a` (4 bytes) is stored at offset 0 and the `double` member `b` (8 bytes) at offset 8. The 4 bytes from offset 4 to 8 are not in use and their content is therefore *undefined*. The content of the `Structured` object of `struct S` contains two values:

$$\{\text{slot1} \mapsto 12, \text{slot2} \mapsto 4.2\} \tag{6.1}$$

The layout table (`layout` field) is a map storing the offsets, types and slot names of the members and is used to map an offset to the corresponding slot and its type. The layout table of the `Structured` object of `struct S` (see Listing 6.1) contains two entries:

$$\{0 \mapsto (\text{slot1}, \text{int}), 8 \mapsto (\text{slot2}, \text{double})\} \tag{6.2}$$

If a C program accesses a `Structured` object with a certain offset, the offset is mapped to a slot of the content. For example, when accessing the member `b` of `struct S`, the layout table maps the offset 8 to *slot2*, which contains an 8-byte `double` value.

For non-primitive members (e.g., arrays and `structs`) of a `struct` or `union` we allocate a separate array or `Structured` object and store it into the slot of the content. When accessing sub-objects (e.g. when a `struct` has an array member), we compute the offset within the sub-object and access it in the same way. `Union` member slots can be accessed with the types of the `union` members.

```c
struct S {
  int a;
  double b;
};

void foo() {
  struct S *s = malloc(sizeof(struct S));
  s->a = 12;
  s->b = 4.2;
}
```

Listing 6.1: Writing a `struct` member.

Address Computation

Accessing array elements or `struct` members in C involves an address computation. The same is true for pointer arithmetic. Such computations can be conveniently done with `MAddress` objects, i.e., `MAddress` object have an `offset` field, which we use to implement pointer arithmetic in C. For processing the assignment `s->b = 4.2;` (see Listing 6.1) a C compiler would add the offset of `b` (8) to the address `s` and assign the value `4.2` to this address. TruffleC$_M$ represents the pointer `s` using an `MAddress` object that references a managed allocation of type `struct S`; its offset is 0. When accessing `s->b`, we copy the `MAddress` object of `s` and add the value 8 to its `offset` field thus referencing the field `b`. TruffleC$_M$ uses this new `MAddress` object to access the member `b`.

To compute the address of an array element, a C compiler would multiply the array index with the element size and add this as an offset to the address of the array. TruffleC$_M$ uses `MAddress` objects instead of raw addresses; it also multiplies the index with the element size and stores this value in the `offset` field of an `MAddress` object.

For pointer arithmetic, Section 6.5.6 of the C99 standard [29] defines the following semantics: For an addition/subtraction, one operand shall be an integer type. When two pointers are subtracted, both shall point to elements of the same array object and the result is the difference of the subscripts of the two array elements. If an integer value is added/subtracted to/from a pointer, we add/subtract this integer value times the size of the pointed-to type to/from the offset value of an `MAddress` object. If two address values are subtracted, we calculate the difference of the subscripts using the offset values of the `MAddress` objects. If two addresses do not reference the same array object, the result is undefined, which conforms to the standard (see Section 6.5.6. §9 of the C99 standard).

Data Access

When accessing managed allocations, we distinguish between a *strict mode* and a *relaxed mode*. When running TruffleC$_M$ in *strict mode*, only *well-defined memory accesses* are allowed. A *well-defined access* cannot read from uninitialized memory and the pointer used in the access must reference a valid destination, i.e., the value at the destination must have a type that is expected by the access operation. For example, it is not allowed to dereference a *type-punned* pointer, i.e., a pointer of type B that was casted from a pointer of type A. An access via such a pointer would not resolve to a valid destination. Sections 6.5 §5 and 6.3.2.3 §7 of the C99 standard state that the program shall not access type-punned pointers [29]. In *strict mode*, it is only possible to access a primitive object as well as an array object if the access operation is well-defined, i.e., the type of the access operation has to match the element type and the offset of the `MAddress` object needs to be aligned (the offset must be 0 for primitive objects or a whole multiple of the array's element size). It is only possible to access a `Structured` object if the layout table can map an offset to a slot and if the type of the access operation matches the type of the slot.

When running TruffleC$_M$ in *relaxed mode*, programmers can read from uninitialized memory and can access any memory destination (*undefined memory access*). For example, a pointer can be casted to a pointer with a different type (*type-punning*) and it is possible to dereference it. In these cases, TruffleC$_M$ mimics the behavior of plain C compilers but still ensures spatial and temporal safety. In the following, we describe the relaxed mode in detail.

Undefined Object Accesses

TruffleC$_M$ maps the offset of an `MAddress` object to a member of a managed allocation. If this is not possible or if the mapped member does not have the type that is expected by the access operation, we call this an *undefined access*.

Primitive object: An undefined read operation to a primitive object returns the same value as a raw memory read would have produced, i.e., we mimic the object representation of native C primitives. Consider the example in Listing 6.2:

```c
double v = 42.5;
double *d = &v;
int i = ((int*)d)[0];
```

Listing 6.2: Undefined read operation to a primitive object.

The read operation returns an `int` value that contains the lower 4 bytes of the `double` value `42.5`. Upon an undefined write access to a primitive object, we first allocate a new `Structured` object and copy the data to its content. We also initialize the layout table with one entry for the primitive value. The program finally accesses this `Structured` object and uses it for the rest of the program execution.

Array object: In case of an undefined read operation from an array object, TruffleC$_{\mathrm{M}}$ reads from all elements that overlap with the accessed element and returns the same value as a raw memory read would have produced. Let us assume that we cast an `int` array to a `double*`, see Listing 6.3:

```c
int a[5];
double v = ((double *)a)[0];
```

Listing 6.3: Undefined read operation to an array object.

The read operation reads the `int` elements (32 bits) at indexes 0 and 1 and composes their values to a 64 bit `double` value. Again, undefined write operations transform the array object to a `Structured` object. We initialize the layout table with one entry for each array element.

Structured object: A `Structured` object can handle any undefined access operation. We use the layout table to map the offset of an `MAddress` object to a slot and hence mimic the object representation of a native allocation on the x86-64 platform. An undefined read operation reads all slots that overlap with the accessed element, composes their values, and returns the bit pattern at the given offset. This produces the same value as a raw memory read from native data would have produced. Let us assume that we cast a pointer `s` (of type `Struct S*`) to a pointer of type `int*` and perform an *undefined* read operation (see Listing 6.4):

```c
int *i = (int*)s;
int v = i[2];
i[2] = 13;
```

Listing 6.4: Undefined read and write operation to a structured object.

The read access of `i[2]` should return the first 4 bytes of the `double` value `s->b`. Thus, TruffleC$_{\mathrm{M}}$ reads *slot2* (the `double` value `s->b`) and returns the first 4 bytes of the `double` value encoded as an `int`.

An undefined write operation can partially overwrite one or more members of a `Structured` object. For example, the statement `i[2] = 13` attempts to write a 4 byte `int` value to offset 8, which is mapped to an 8 byte `double` slot. This means that slots in a `Structured` object need to be partially overwritten. We remove all slots that are partially overwritten. Then we add a new slot to the content for the new value and also slots for the remaining bytes of the partially overwritten slots. For the write access `i[2] = 13`, we first remove the entry $8 \mapsto (\text{slot2}, \text{double})$ from our layout table and also remove `slot2` from the content. Afterwards we add two new entries to our layout table ($\{8 \mapsto (\text{slot2}, \text{int}), 12 \mapsto (\text{slot3}, \text{undefined})\}$). *Slot2* contains the value `13` and *slot3* contains the remaining 4 bytes of the `double` value `4.2`, which was previously stored at offset 8. The result is a layout table with three entries:

$$\begin{gathered}\{0 \mapsto (\text{slot1}, \text{int}), 8 \mapsto (\text{slot2}, \text{int}),\\ 12 \mapsto (\text{slot3}, \text{undefined})\}\end{gathered} \tag{6.3}$$

The content stores three values:

$$\{\text{slot1} \mapsto 12, \text{slot2} \mapsto 13, \text{slot3} \mapsto ...\} \tag{6.4}$$

Allocations Without Type Information

In C, programmers can allocate memory without providing information about its type. Consider the example in Listing 6.5:

```c
void *p = malloc(16);
```

Listing 6.5: Allocation without type information.

In this case, we create a `Structured` object with a layout table starting in an *uninitialized* state, i.e., if no data was written to this allocation yet, the layout table does not contain any entries. A write operation then adds a new slot to the content and creates an entry in the layout table. In other words, when a pointer `p` is casted to `struct S` and dereferenced to write to a field (e.g. `p->a = 3;`), a new slot (e.g., `slot0`) is added to the content, and a corresponding entry is made in the layout table (e.g., $0 \mapsto (\text{slot0}, \text{int})$).

A read operation that cannot be mapped to a slot by the layout table because the memory is uninitialized produces a default value.

6.2 Allocation and Deallocation

We distinguish between stack allocations (memory that is automatically allocated when a function is called and automatically deallocated when the function returns) and heap allocations (memory that is manually allocated using `malloc`, `calloc`, or `realloc`, as well as memory that lives throughout the entire program execution such as static local variables):

Stack allocations: When a C function is called, a new stack frame is created for its local variables. TruffleC$_{\text{M}}$ allocates all local variables as managed allocations on the Java heap. When a function returns, these allocations are marked as *deallocated*. The GC of the JVM can then reclaim these objects.

Heap allocations: TruffleC$_{\text{M}}$ allocates a managed allocation whenever a C program manually allocates memory (e.g. using `malloc`). It marks the allocation as *deallocated* if it is deallocated manually (e.g. using `free`). If the C program tries to access this allocation later or if it calls `free` twice, we report a high-level run-time error.

We use the memory management of the JVM to guarantee temporal safety. The GC automatically deallocates an object if and only if it is not referenced anymore. Marking managed allocations as deallocated allows us to mimic the behavior of C by simulating deallocations. We mark managed objects as deallocated by setting the `content` field of the descriptor (see Figure 6.1) to `null`. Thus, we can detect if a pointer is used after the referent has been deallocated, i.e., when a dangling pointer is accessed or an object is deallocated twice. This ensures temporal safety.

TruffleC$_{\text{M}}$ can even do away with memory leaks that result from forgetting to deallocate objects. Such errors cannot occur in TruffleC$_{\text{M}}$, because the GC automatically frees a managed allocation as soon as it is not referenced any longer. In fact, manual deallocation becomes superfluous, because managed allocations are garbage collected.

6.3 Implementation with Generic Access

Extending TruffleC by managed objects is simple. We use *generic access* wherever TruffleC would normally access the native heap. All managed allocations implement the `SafeAllocation` interface, which itself extends the `TruffleObject` interface. Hence, all managed allocations can be accessed via *messages*. TruffleC$_{\text{M}}$ removes all TruffleC nodes that access the native heap and replaces them with message nodes $n \in N^{\text{SafeCMsg}}$.

There are read and write messages for all primitive C types as well as messages for reading and writing an `MAddress`:

$$\begin{aligned} N^{\text{SafeCMsg}} = \{&\text{SafeReadDouble}, \text{SafeWriteDouble}, \\ &\text{SafeReadInt32}, \text{SafeWriteInt32}, ..., \\ &\text{SafeReadMAddress}, \text{SafeWriteMAddress}, \\ &\text{SafeExecute}\} \end{aligned} \tag{6.5}$$

TruffleC$_\text{M}$ defines a function f_{SafeC} that maps every unsafe memory access in $t \in T_{N^{\text{TruffleC}}}$ to one of these messages. We apply this function to the TruffleC ASTs statically before execution. For example, the TruffleC node to write a `struct` member of type `double` (*CMemberWrite*) is mapped to a *SafeWriteDouble* message (see Figure 6.3 on page 72). This node has three children. The first child provides the receiver of this access operation, e.g., a pointer to a `struct`. The second child provides the member to be accessed, i.e., this node returns the byte offset of the member. Finally, the third child provides the value to be written.

TruffleC$_\text{M}$ also defines a resolution function g_{SafeC} that maps every message to a memory safe access operation. This function is applied at run time and is part of message resolution. It provides AST snippets that contain access operations, which are specific to the managed allocations. For example, a `Structured` object requires an AST snippet that maps an offset to a slot and eventually accesses this slot whereas an `DoubleArray` requires an AST snippet that divides an offset by 8 (the size of an element) and accesses a Java `double` array. During later execution, the value of an `MAddress` can change so that it points to a different type of managed allocation. In order to detect that, *generic access* inserts a guard into the AST that checks the object's type before accessing the object.

Figure 6.3 on page 72 shows the AST for the statement `s->b = 4.2`, where a *SafeWriteDouble* message node is used to write the `double` value `4.2` to the `struct` member `b`. Message resolution replaces the *SafeWriteDouble* node with a *SafeWriteSlot* (a `Structured`-specific AST node). This node takes the offset of the `MAddress` object (`s.offset`) and adds the member offset (`member.offset`). It uses this offset to lookup a slot within the `Structured` object (`strct.layout(off)`). Finally, it stores the value into this slot (`strct.content.set(slot, val)`). Before the AST accesses `s`, it checks if its data really references a `Structured` object (*is `Structured`?* node). If during execution `s` would change to point to a `DoubleArray` say, the execution would fall back to using the *SafeWriteDouble* message again, which would be resolved to a `DoubleArray`-specific AST snippet.

The `Structured`-specific access operations are later specialized according to their execution profile. We use tree rewriting and replace the *SafeWriteSlot* with a specialized version (see Section 2.2.1). This gives good performance because of two observations. First, it is likely that a C program accesses an allocation in a well-defined fashion, which means that the layout table of a `Structured` object does not change during run time. Second, statements that access an allocation are likely to do so with a constant offset. For example, a `struct` access `s->b` always accesses the `Structured` object with the same offset. Thus, we speculate that the offset of the `MAddress` and the layout table of the `Structured` object are constant and specialize the *SafeWriteSlot* on this assumption. The specialized node caches the slot and directly accesses the content without any lookup in the layout table.

6.4 Compliance with the C99 Standard

`MAddress` objects comply with the C99 standard. Section 6.2.6.1 §4 and §5 of the C99 standard define requirements on the representation of pointer types and Section 6.3.2.3 defines the conversions that are valid on pointer values. In the following list we explain how TruffleC$_M$ complies with these requirements:

6.2.6.1 §4: The standard states that *two values (other than NaNs) with the same object representation compare equal, but values that compare equal may have different object representations*[1] [29].

An `MAddress` object uses a unique ID of the descriptor object plus the offset as its object representation. Even though a pointer is an `MAddress` object, the system provides the illusion that it can be read as a word-sized sequence of bytes. Based on this object representation, we can compare `MAddress` objects for equality and also convert them to integers. Two pointers that reference the same object thus have the same object representation.

6.2.6.1 §5: The standard states that *certain object representations need not represent a value of the object type. If the stored value of an object has such a representation and is read by an lvalue expression that does not have character type, the behavior is undefined. If such a representation is produced by a side effect that modifies all or any part of the object by an lvalue expression that does not have character type, the behavior is undefined* [29].

In TruffleC$_M$ it is not possible for programmers to handcraft a valid object representation of a pointer. The C standard explicitly states that modifying an

[1]The object representation of data is the set of $n \times$ `CHAR_BIT` bits, where n is the size of an object of that type in bytes.

object representation by an lvalue expression that does not have character type is not allowed, but it does not say anything about modification with an lvalue expression that has character type. We decided that directly modifying an object representation always creates an invalid pointer. Otherwise it would be possible to access objects that should be inaccessible.

6.3.2.3 §1, 2, 7, and 8: The standard (6.3.2.3 §1) states that *a pointer to void may be converted to or from a pointer to any incomplete or object type. A pointer to any incomplete or object type may be converted to a pointer to void and back again; the result shall compare equal to the original pointer* [29]. In 6.3.2.3 §2 it states that *for any qualifier q, a pointer to a non-q-qualified type may be converted to a pointer to the q-qualified version of the type; the values stored in the original and converted pointers shall compare equal* [29]. In 6.3.2.3 §7 it states that *a pointer to an object or incomplete type may be converted to a pointer to a different object or incomplete type. If the resulting pointer is not correctly aligned for the pointed-to type, the behavior is undefined. Otherwise, when converted back again, the result shall compare equal to the original pointer* [29]. Finally, in 6.3.2.3 §8 it states that *a pointer to a function of one type may be converted to a pointer to a function of another type and back again; the result shall compare equal to the original pointer* [29].

In TruffleC$_M$ a pointer of type A can be converted to a pointer of type B without modifying the corresponding `MAddress` objects (i.e., the `data` and `offset` fields remain unchanged). Hence, TruffleC$_M$ complies with the requirements above.

6.3.2.3 §3 and 4: The C standard (6.3.2.3 §3) states that *an integer constant expression with the value 0, or such an expression cast to type void *, is called a null pointer constant. If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer, is guaranteed to compare unequal to a pointer to any object or function* [29]. In 6.3.2.3 §4 it states that *a conversion of a null pointer to another pointer type yields a null pointer of that type. Any two null pointers shall compare equal.*

The integer constant `0` can be assigned to an `MAddress` object. The `data` field of such an `MAddress` object is then `null` and the `offset` field is 0, which we consider the *null pointer constant.* This constant compares unequal to any other pointer to an object or function.

6.3.2.3 §5: The C standard (6.3.2.3 §5) states that *an integer may be converted to any pointer type. Except as previously specified, the result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation* [29].

An integer may be converted to any pointer type by storing the integer value in the `offset` field of the `MAddress` object and setting the `data` field to `null`. According to the C99 standard, converting an integer to a pointer results in an undefined behavior. In our case, the pointer value cannot be dereferenced (the `data` field is `null`), which is in accordance with the standard.

6.3.2.3 §6: The C standard (6.3.2.3 §6) states that *any pointer type may be converted to an integer type. Except as previously specified, the result is implementation-defined. If the result cannot be represented in the integer type, the behavior is undefined. The result need not be in the range of values of any integer type* [29].

Any `MAddress` object may be converted to an integer type. We take the descriptor's ID (`data` field of an `MAddress` object) and concatenate it with the offset value (`offset` field of an `MAddress` object).

In TruffleC$_{\text{M}}$, each pointer variable starts as a *null pointer*, i.e., the `data` field is `null` and hence the pointer cannot be dereferenced. A valid pointer value is produced by applying the address-of operator (`&`) on any C object or by an allocation of memory (e.g., by a manual `malloc` or an automatic stack allocation). In these cases, we create an `MAddress` object where the `data` field references the managed allocation. These valid pointer values can then be modified by pointer arithmetic and address computations (i.e., by updating the offset of an `MAddress` object; see Section 6.1). If the `data` field of an `MAddress` is `null`, any access to it causes a high-level run-time error.

6.5 Discussion

We discuss TruffleC$_{\text{M}}$, which can safely execute C code in a *strict* and in a *relaxed* mode. The *strict* mode can be used during development to detect undefined operations of a program, such as accessing type-punned pointers. To run existing source code without modification we offer the *relaxed* mode, which mimics the behavior of industry-standard C compilers. In this mode, TruffleC$_{\text{M}}$ can run existing code that depends on the behavior of industry-standard C compilers without sacrificing spatial or temporal memory safety. TruffleC$_{\text{M}}$ detects memory errors as well as undefined access operations at run time, immediately before memory is accessed in some illegal way. TruffleC$_{\text{M}}$ throws a Java exception that describes the error and includes also a stack trace at the error position. Currently, such Java exceptions are caught, the error and the stack trace are printed, and the program exits. Since the C standard does not specify the effects of memory-unsafe accesses and undefined memory accesses, this behavior is fully compliant. This kind of error reporting could either be used during testing to find bugs, or at run time to further enforce correctness. Listing 6.6 shows an example where the index of an array access is out of bounds (spatial memory access violation). The

```c
#include<stdlib.h>
void doWork(int *p, int N) {
  int i;
  for (i = 0; i <= N; i++) {
    p[i] = 0;
  }
}

int main() {
  int N = 5;
  int *p = malloc (N * sizeof(int));
  doWork(p, N);
  free(p);
  return 0;
}
```

Listing 6.6: The function `doWork` accesses the array `p` out of bounds.

```
BufferOverflowError:
at doWork (Example.c)
at main (Example.c)
```

Figure 6.2: The example of Listing 6.6 raises a `BufferOverflowError`.

implementation of `doWork` erroneously loops with the condition `i <= N`, rather than `i < N`, which causes it to access one element beyond the end of the array within the loop. TruffleC$_M$ reports this as `BufferOverflowError` and tells the user that the error occurred in `doWork`, which was called by `main` (see Figure 6.2).

TruffleC$_M$ cannot share managed allocations with precompiled native code. Therefore, TruffleC$_M$ requires that the source code of the entire C program is available and is executed under TruffleC$_M$. We provide a Java implementation for functions that are not available in source code (e.g. functions of standard libraries). Rather than doing a native call, TruffleC$_M$ then uses these Java implementations that substitute the native implementations. TruffleC$_M$ currently has substitutions for various functions defined in `assert.h`, `limits.h`, `math.h`, `stdarg.h`, `stdbool.h`, `stdio.h`, `stdlib.h`, `string.h`, `time.h`, and `wchar.h`. The list of standard library substitution is not yet complete, however, our future work will focus on completing this and hence make TruffleC$_M$ more complete. Under these premises, we can run C programs entirely on top of the JVM without ever accessing precompiled native code or unsafe native data.

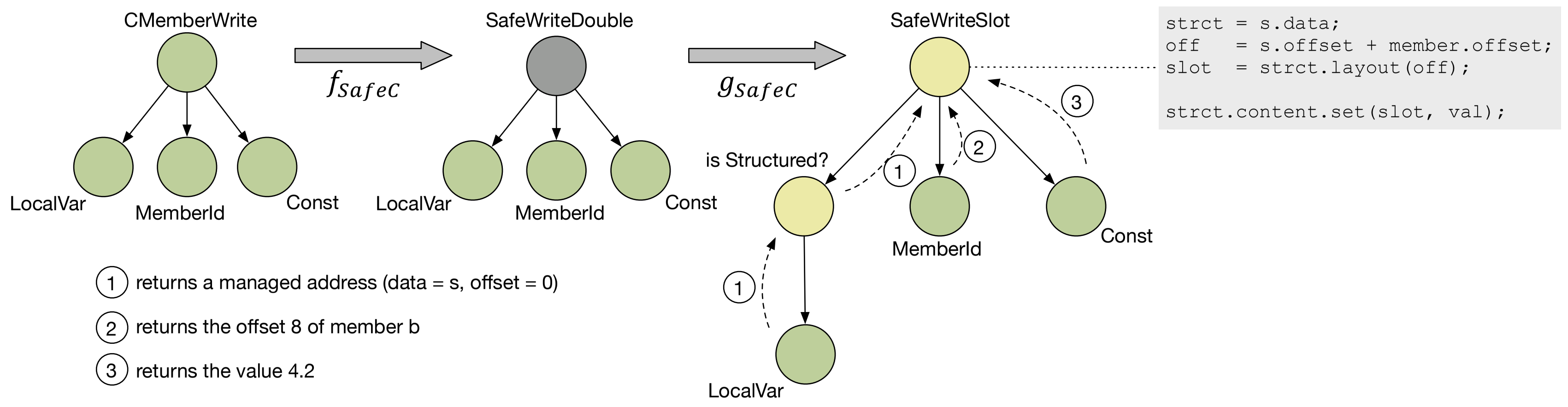
This case study showed an application of *generic access*, which is orthogonal to cross-language interoperability. We used *generic access* to create a memory-safe implementation of TruffleC by substituting all native allocations of a C program with managed allocations and treating them like foreign objects. This required only the following modifications:

Managed allocations: We introduced managed Java objects that were used as a substitution for native data and raw pointer values. We extend the TruffleC nodes for address computation and pointer arithmetic so that they can operate on pointers to managed objects.

Allocations: We removed any kind of native memory allocation and replaced it with a managed allocation.

Generic Access: We used *generic access* for all operations that would normally access native data.

This case study also allows us to present a performance evaluation of *generic access* apart from cross-language interoperability.



```
strct = s.data;
off   = s.offset + member.offset;
slot  = strct.layout(off);

strct.content.set(slot, val);
```

Figure 6.3: Resolving managed allocation-specific access operations.

Chapter 7

Performance Evaluation

This chapter presents a performance evaluation of individual parts of the TMLR. We present benchmarks that combine different languages and show that combining a slower language with a faster one yields an overall performance that is somewhere in the middle. We show that the C extensions API implementation with generic access *runs benchmarks faster than all other Ruby implementations using C extensions. We also demonstrate that message resolution and cross-language inlining are essential for the performance by measuring the effect of temporarily disabling them. Finally, we demonstrate that* generic access *does not lead to a performance degradation (on average) when substituting native allocations with managed allocations in TruffleC.*

We structure our performance evaluation in three parts. First, we summarize the performance of *uniform language implementations with Truffle* by comparing TruffleC to an industry-standard C compiler. Secondly, we evaluate *generic access* for *composing Truffle language implementations.* Finally, we compare TruffleC$_\mathrm{M}$ to TruffleC and thus measure the costs of a memory-safe implementation of C.

Uniform Language Implementations with Truffle

The TMLR comes with three language implementations: TruffleJS, TruffleRuby, and TruffleC. TruffleJS and TruffleRuby are existing implementations and we extend them by *generic access* but did not contribute to these implementations in any other way. Hence, we only provide an extensive performance evaluation for TruffleC. A summary of the JavaScript and Ruby performance can be found in Section 3.

In Chapter 1 we stated that TruffleC can dynamically compile C code, which features profile-based optimizations such as inline caches for function pointer calls. We described the implementation of these optimizations in Chapter 3.1 and claimed that TruffleC benefits from them in terms of performance. In this chapter we evaluate the effect of these optimizations. We execute a micro benchmark (the example of Chapter 3.1) that allows TruffleC to exploit all these dynamic optimizations. TruffleC can outperform GCC on this benchmark. However, we also evaluate TruffleC on number-crunching benchmarks where dynamic optimizations can hardly be applied. Programs compiled with TruffleC are on average 28% slower than programs compiled with GCC.

Truffle Language Implementation Composition

We evaluate the composition of languages on top of the TMLR with two different performance measurements:

First, we want to evaluate the performance of multi-language applications. Every language implementation can define efficient data representations, which can be shared across different languages. *Generic access* ensures that a TLI can directly access foreign objects. We expect using heavyweight foreign data to have a negative impact on performance. On the other hand, we expect using efficient foreign data to have a positive effect on performance. For example, accessing a Ruby array in C is less efficient than accessing a C array because Ruby arrays require more run-time checks. A C array in Ruby, however, is more efficient because a C array access performs a raw memory access without additional run-time checks. Message resolution inserts a foreign-language-specific access into the AST of a host application, however, it also inserts a language and type check before the foreign object is accessed. The dynamic compiler can optimize this additional check. We discuss how the compiler minimizes the performance overhead of this check in detail. Furthermore, we claim that inlining across language boundaries and cross-language optimizations are critical for performance. We evaluate the performance impact when disabling message resolution. When disabling message resolution, the TMLR does not replace a message with a foreign-language-specific AST snippet that implements the access operation, but we invoke this AST snippet like a function. The function target (i.e., the foreign-language-specific AST snippet) depends on the language and type of the receiver object. We do not introduce any additional complexity to a foreign object access. However, L_{Host} has to treat L_{Foreign} as a black box, which introduces a language boundary.

Second, we compare the performance of the TMLR to the C extensions API implementations of MRI, Rubinius, and JRuby. We claim that the TMLR can run C extension functions on average over 200% faster than natively compiled C code using MRI's C extensions API. In this evaluation we back this claim and run image processing libraries that have a C extensions implementation. Also, we state that cross-language inlining and cross-language optimizations are the most beneficial optimization compared to other implementations. Hence, we disable message resolution to verify this claim.

Managed Data Structures for C

TruffleC$_M$ is a case study that allows us to back the flexibility claims of *generic access*. We substitute native allocations with managed allocations. *Generic access* ensures that TruffleC$_M$ can directly access managed allocations. However, these access operations to managed data perform additional checks compared to access operations to unmanaged data with TruffleC. *Generic access* introduces language and type checks, and the access operations introduce checks that ensure memory safety. The dynamic compiler can optimize these additional checks, which we discuss in detail. We compare the performance of the managed execution of C code (TruffleC$_M$ accesses managed allocations with *generic access*) to the unmanaged execution (TruffleC accesses native allocations directly) and can show that TruffleC$_M$ and TruffleC have the same performance on average. For this evaluation, we use the same benchmarks as for the TruffleC evaluation.

7.1 Evaluation Methodology

To account for the adaptive compilation techniques of Truffle and Graal, we set up a harness that executes each benchmark 50 times. After these warm-up iterations, every benchmark reaches a steady state such that subsequent iterations are identically and independently distributed. This was verified informally using lag plots [60]. We then sampled the final 10 iterations and calculated the averages for each configuration using the arithmetic mean. Where we report an error we show the standard deviation. Where we summarize across different benchmarks we report a geometric mean [28]. Our harness reports scores for each benchmark and its configurations, which is the proportion of the execution count of the benchmark and the time needed (executions per second). We ran the TruffleC benchmarks, the multi-language benchmarks, and the TruffleC$_M$ benchmarks on an Intel Core i7-4770 quad-core 3.4GHz CPU running 64 Bit Debian 7 (Linux3.2.0-4-amd64) with 16 GB of memory. The C extension benchmarks are long-running applications and the measurement takes multiple days. Therefore, we ran these benchmarks on a different hardware. We used a server machine 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.

We focus this evaluation on peak performance of long-running applications where the startup performance plays a minor role. Hence, we neglect the startup time and present performance numbers after an initial warm-up. A detailed evaluation of the start-up performance of Truffle is out of scope and can be found in [68].

7.2 Uniform Language Implementations with Truffle

In this section we compare TruffleC to the industry standard C compiler GCC (version 4.7.2-5) in terms of peak performance. We based TruffleC on Graal revision `5b24a15988fe` from the official OpenJDK Graal repository[1].

First, we execute a micro benchmark (Listings 3.1 and 3.2 from Chapter 3.1 on page 22) that allows TruffleC to exploit its profile-based optimizations. We set up the function `abs` in a benchmark harness and pass it a function pointer as well as two integer values. Using run-time profiling, TruffleC finds out that the value of the function pointer is always `div` and the value of `b` is always 2. On this benchmark we expect TruffleC to outperform GCC because of the dynamic optimizations.

Second, we evaluate the performance of TruffleC with benchmarks from the SciMark benchmark suite[2] and from the Computer Language Benchmarks Game[3]. The benchmarks consist of a Fast Fourier Transformation (FFT), a Jacobi successive overrelaxation (SOR), a Monte Carlo integration (MC), a sparse matrix multiplication (SM), a dense LU matrix factorization (LU), a simulation of the N-body problem (NB), a tree sort algorithm (TS), a generation of random DNA sequences (FA), an algorithm to solve the Towers of Hanoi problem (TW), a computation of the spectral norm of a matrix (SN), as well as the Fannkuch (FK) and Mandelbrot (MB) benchmarks, which both do a lot of integer and array accesses. These number-crunching benchmarks demonstrate the performance of TruffleC where dynamic optimizations can hardly be applied.

Compared Implementations

The charts in Figures 7.1 and 7.2 are arranged on a linear, higher-is-better scale. We show the TruffleC performance, the *GCC Best* performance (the best performance out of the three optimization levels *O1, O2* and *O3*), and the performance of GCC with optimization level *O0.* We normalize our results to GCC *O0*, i.e., 1 is the performance of GCC with optimization level *O0.*

Results

Figure 7.1 shows the evaluation of the micro benchmark; the y-axis shows the normalized score of the benchmark. TruffleC is more than 600% faster than the best GCC performance. We explain the big difference as follows:

[1] *OpenJDK Graal repository*, Oracle, 2015: `http://hg.openjdk.java.net/graal/graal`
[2] *SciMark 2.0*, Roldan Pozo and Bruce R Miller, 2015: `http://math.nist.gov/scimark2/index.html`
[3] *The Computer Language Benchmarks Game*, Brent Fulgham and Isaac Gouy, 2015: `http://benchmarksgame.alioth.debian.org/`

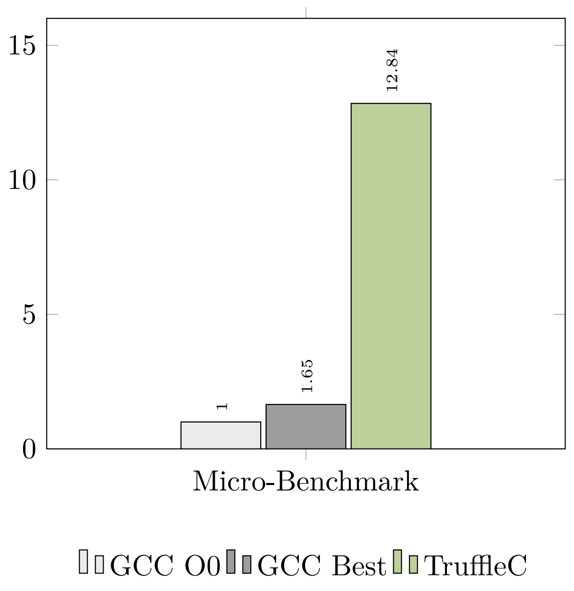


Figure 7.1: TruffleC performance of a micro-benchmark. TruffleC inlines library functions, caches a function pointer call, and profiles values (normalized to GCC *O0* performance; higher is better).

Inlining of library functions: In contrast to GCC, TruffleC can inline the library function `abs`, whereas GCC cannot inline independently compiled library functions. TruffleC can inline functions as ASTs from shared libraries at run time by copying the AST of the callee and replacing the call node with this copy.

Inline caches for function pointer calls: TruffleC speculates on the function pointer `f` in `abs` being constant (`div`) and caches the target function, i.e., it builds an inline cache with one entry. Whenever this call is done via a pointer to `div`, TruffleC calls the target directly. As entries in the inline cache are always constant, TruffleC can even inline the function pointer call. GCC, on the other hand, has to do a function pointer call because the library does not know that `f` is always `div`.

Value profiling: TruffleC profiles all run-time values. In our example it speculates on the variable `b` being 2, because this is the value that was seen during the interpretation so far. Truffle produces machine code that checks if this assumption is still valid and then uses the constant 2 for `b`. GCC cannot exploit run-time feedback nor can it deoptimize machine code in case of an invalid assumption. Hence, it cannot perform these specializations.

TruffleC can exploit profile information and use it for dynamic optimizations. Thus, it outperforms the static compilation of GCC.

In Figure 7.2 we apply TruffleC to benchmarks from SciMark and the Computer Language Benchmarks Game; the x-axis shows the benchmarks that we evaluated. The y-axis shows the normalized score of the benchmarks (higher is better). The evaluation

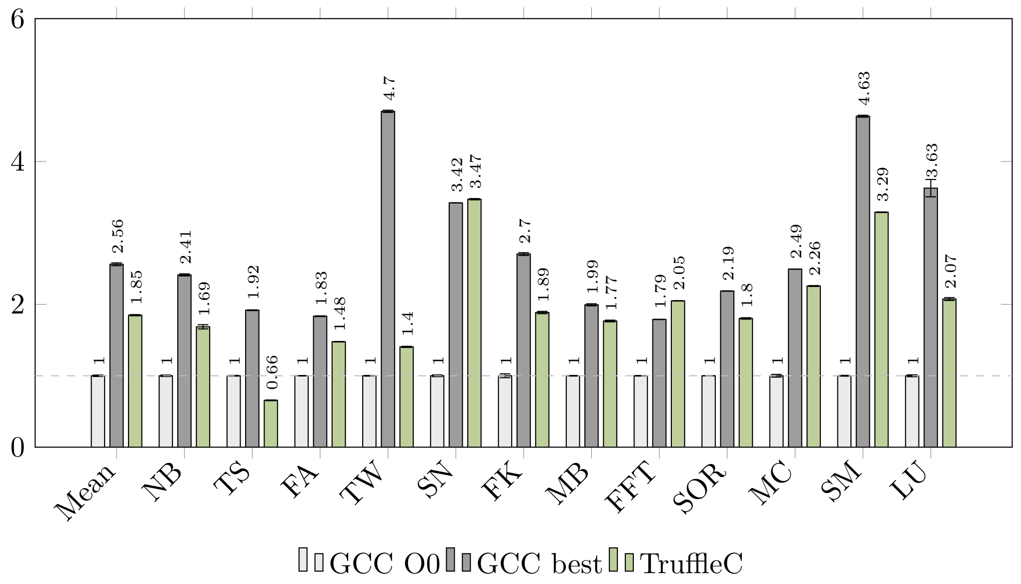


Figure 7.2: Performance numbers of TruffleC (normalized to GCC *O0* performance; higher is better).

shows that TruffleC is mostly faster than GCC with optimization level O0 and on average 28% slower compared to GCC best. We explain the differences as follows:

Function call overhead: A function call in Truffle is much more heavyweight than in normal C. Truffle needs to pack arguments into an `Object` array, which is passed to the callee and afterwards unpacked. Hence, for call-intensive benchmarks (e.g., the recursive benchmarks TS and TW) TruffleC has the highest overhead (TruffleC is 70% slower than GCC best on TW). However, future work on Truffle and the Graal compiler will further optimize calls in Truffle. We expect this gap to be reduced as part of future work.

Compiler optimizations: For the rest of the benchmarks, TruffleC very much depends on the optimizations of the Graal compiler. Performance varies between 43% slower (LU) and 16% faster (FFT) compared to GCC best. The Graal compiler does not exploit all optimizations possible for arithmetic operations or memory addressing, which explains the performance difference on these number-crunching benchmarks. However, future work on the compiler will close this gap.

7.3 Truffle Language Implementation Composition

In this chapter we evaluate the peak performance of multi-language applications running on top of the TMLR. First, we measure the peak performance of our multi-language version of the SciMark and Computer Language Benchmarks Game benchmarks. Second, we measure the peak performance of C extensions benchmarks for Ruby.

7.3.1 Interoperability between JavaScript, Ruby, and C

First, we compare the performance of the individual TLIs on our benchmarks. The results in Figure 7.3 are normalized to the TruffleC performance. Second, we run the multi-language version of our benchmarks. We modified the benchmarks, which are available in C, Ruby, and JavaScript, such that parts of them were written in a different language. We extracted all array and object allocations into factory functions. We then replaced these factory functions with implementations in different languages, making the benchmarks multi-language applications. Compared to the TruffleC evaluation, not all of our benchmarks were suitable for writing them in multiple languages. We do not include the benchmarks NB, SN, FK and MB, because some of them only use primitive values or their implementation is simply too short such that it would not make sense writing them in multiple languages. We selected the following benchmarks: FFT, SOR, MC, SM, LU, TS, FA, and TW. The SciMark benchmarks (FFT, SOR, MC, SM, LU) already had factory functions for all allocations and it was easy to replace these functions with versions that are written in different languages. For TS we implemented the allocation of the tree data structure in a different language. The FA benchmark was modified in a sense that the DNA data structures were allocated using a different language. Finally, the TW benchmark also allocates the tower data structures using a different language. We grouped our evaluations such that their main part was either written in C, in JavaScript, or in Ruby. For each group we used the single-language implementation as the baseline and show how multi-language applications perform compared to single-language applications. The x-axis of each chart in Figures 7.3, 7.4, 7.5, 7.6, and 7.7 shows the different benchmarks. The y-axis of each chart shows the average scores (higher is better) of the benchmarks. We base the TMLR on Graal revision `bf586af6fa0c`.

Results of Single-Language Benchmarks

Figure 7.3 shows that JavaScript code is on average 37% slower and Ruby code on average 67% slower than C code. C is efficient because C data accesses do not require run-time checks (such as array bounds checks), but the memory is accessed directly.

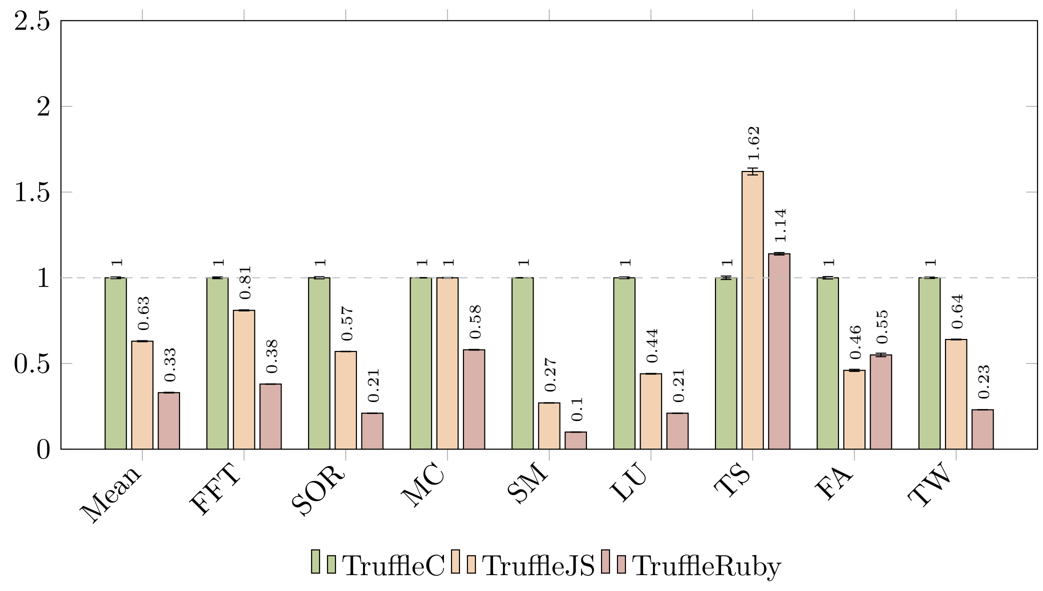


Figure 7.3: Performance of individual languages on our benchmarks (normalized to C performance; higher is better).

This efficient data access makes C the fastest language for most benchmarks. However, if a program allocates data in a frequently executed part of the program, the managed languages (JavaScript and Ruby) can outperform C. Allocations in TruffleC (using `calloc`) are more expensive than the instantiation of a new object on the Java heap. TruffleC does a native call to execute the `calloc` function of the underlying OS. TruffleJS or TruffleRuby allocate a new object on the Java heap using sequential allocation in thread-local allocation buffers, which explains why JavaScript and Ruby perform better than C on TS. This benchmark allocates data in a hot loop. The Ruby semantics require that Ruby objects are accessed via getter or setter methods. TruffleRuby uses a dispatch mechanism to access these methods. This dispatch mechanism introduces additional run-time checks and indirections, which explains why Ruby is in general slower than JavaScript or C.

Results of Multi-Language Benchmarks

The multi-language versions of our benchmarks heavily access foreign objects:

C Objects: C data structures are unsafe; access operations are not checked at run time, which makes them efficient in terms of performance. Hence, using C data structures in JavaScript or Ruby applications improves the run-time performance. However, an allocation with `calloc` is more expensive than an allocation on the Java heap. Factory functions in JavaScript or Ruby perform better than factory functions written in C.

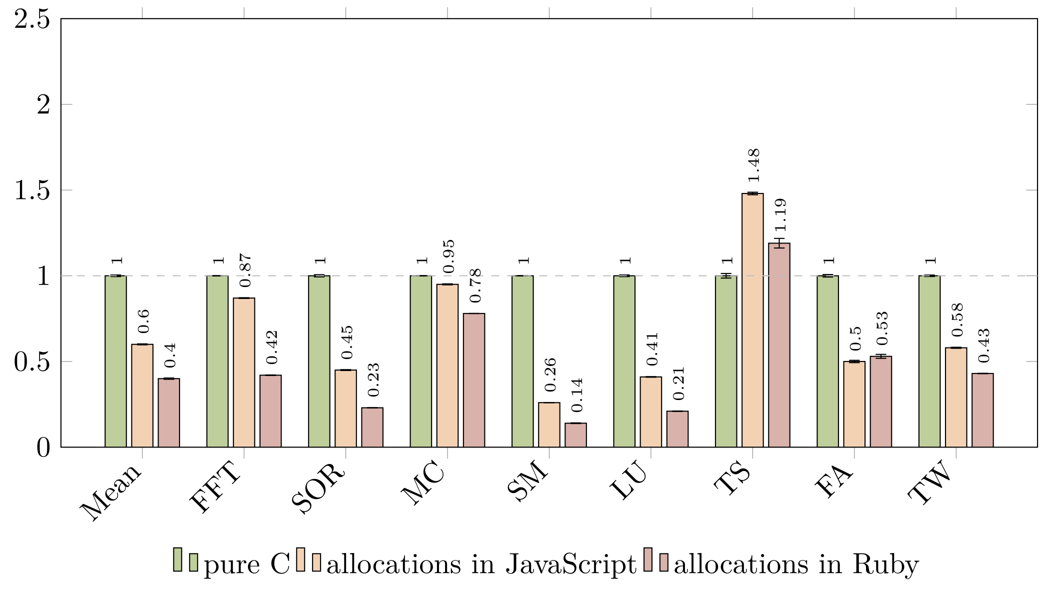


Figure 7.4: Main part in C, allocations in different languages (normalized to pure TruffleC performance; higher is better).

JS Objects: TruffleJS uses a dynamic object implementation where each access involves run-time checks. Examples of such checks are array bounds checks to dynamically grow JavaScript arrays or property access checks to dynamically add properties to an object. These checks are the reason why accesses to JavaScript objects perform worse than accesses to C objects.

Ruby Objects: TruffleRuby's dispatch mechanism for accessing objects introduces a performance overhead compared to JavaScript and even more so to C. TruffleRuby implements Ruby objects as `RubyBasicObject`s, which wrap a `DynamicObject` that actually contains the data. According to the Ruby semantics, TruffleRuby invokes getter and setter methods to access the `DynamicObject`. This additional indirection is the reason why accesses to Ruby objects are in general slower than accesses to JavaScript objects or C objects.

The TMLR does not marshal objects at the language boundary but directly passes them from one language to another and the TLIs use *generic access* to access them. Message resolution only affects the performance at the first execution of an object access. After that, the application runs at full speed. The dynamic compiler can minimize the effect of *generic access*'s language and type check on the receiver. Using conditional elimination [96], the Graal compiler can even remove the additional type/language check by merging it with the type check on the receiver, which is necessary in dynamically typed languages anyway. Conditional elimination reduces the number of conditional expressions by performing a control-flow analysis over the IR graph and pruning conditions that can be proven to be true. Also, it can move *generic access*'s check out of loops if

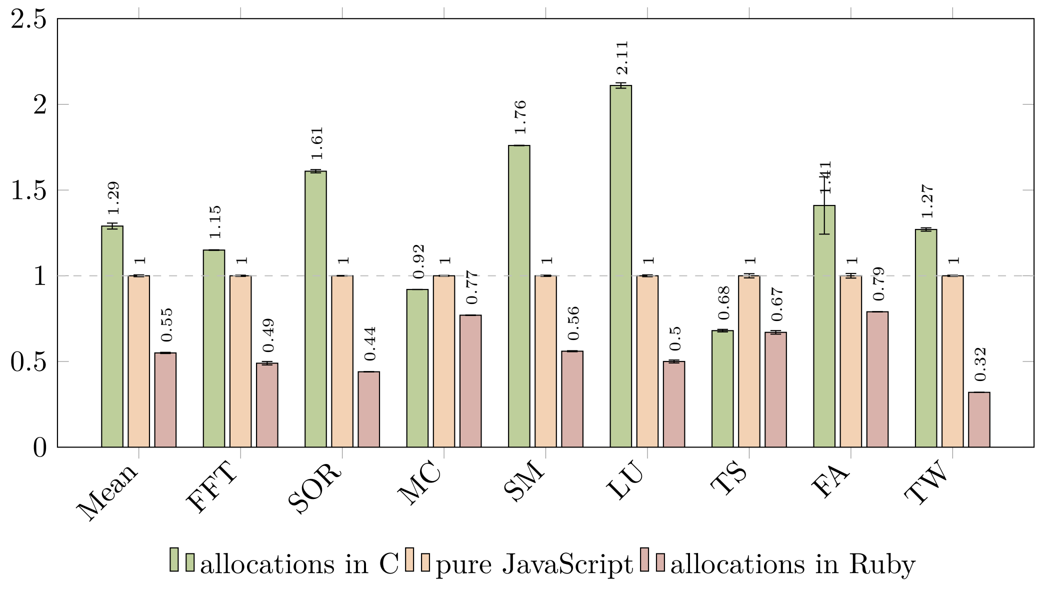


Figure 7.5: Main part in JavaScript, allocations in different languages (normalized to pure TruffleJS performance; higher is better).

the compiler can prove that a condition is loop-invariant [24, 25]. Hence, we can show that the performance of a multi-language program mainly depends on the performance of the individual language parts. Using JavaScript or Ruby data in C programs has a negative impact on the overall performance. Figures 7.4 and 7.5 show that using Ruby objects in C or JavaScript programs causes an overhead of up to 86%. On average, using Ruby data in C causes a performance overhead of 60% and using Ruby data in JavaScript causes a performance overhead of 45%. On the other hand, using efficient foreign data has a positive effect on performance. For example, Figures 7.5 and 7.6 show that using efficient C data in JavaScript or Ruby programs can improve performance by up to 322%. On average, using C data improves the JavaScript performance by 29% and the Ruby performance by 85%.

Message resolution allows the Graal compiler to apply its optimizations across language boundaries (*cross-language inlining*). Widening the compilation span across different languages enables the compiler to apply optimizations to a wider range of code. Message resolution also allows the Graal compiler to apply *escape analysis* and *scalar replacement* [98] to foreign objects. Consider a JavaScript program that allocates an object, which is used by a C part of the application. Message resolution ensures that Graal's escape analysis can analyze the object access, independent of the host language. If the JavaScript object does not escape the compilation scope, scalar replacement can remove the allocation and replace all usages of the object with scalar values. To demonstrate the performance improvement due to message resolution we disable it. In Figure 7.7 we show the performance of our JavaScript benchmarks using C data structures with and without message resolution. When disabling message reso-

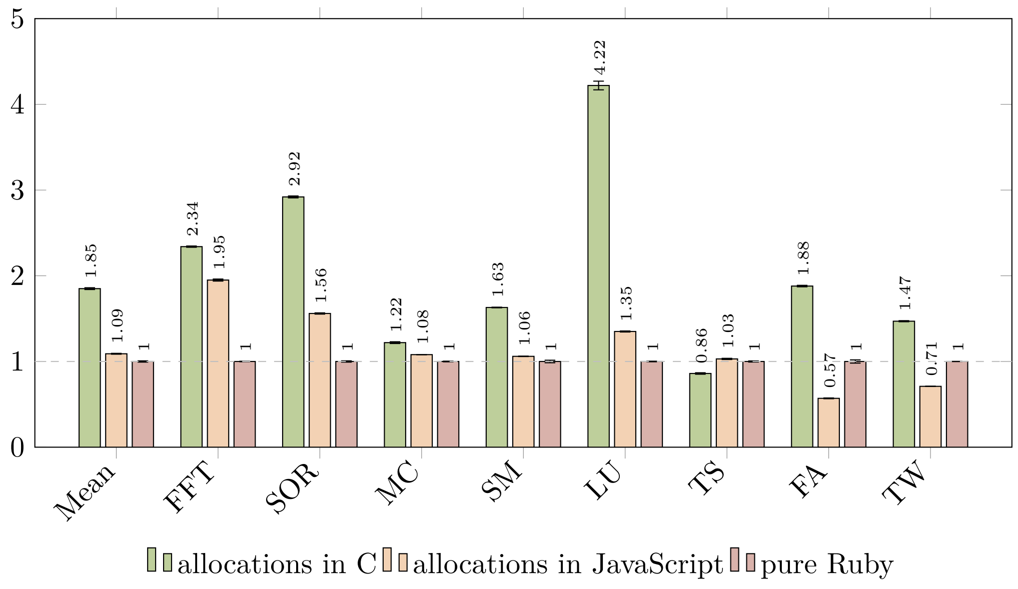


Figure 7.6: Main part in Ruby, allocations in different languages (normalized to pure TruffleRuby performance; higher is better).

lution, every data access as well as every function call crosses the language boundary, which results in a performance overhead of more than 500% for JavaScript with allocations in C compared to the same configuration but with message resolution. The Graal compiler cannot perform optimizations across language boundaries, which explains the loss in performance. We expect similar results for the other configurations, however, we have not measured them because disabling message resolution for an TLI requires a significant engineering effort.

7.3.2 C Extensions Support for Ruby

In this section we compare the TMLR to other multi-language systems that can compose Ruby code and native C code, namely MRI, Rubinius, and JRuby. We show that the TMLR performs better than these related approaches.

We benchmark examples of real-world C extensions that have been developed to meet a real business need (see also Section 5.2.4). Also, we used 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 extensively. Therefore, we use the existing modules `chunky_png` and `psd.rb`. Both modules have separately available C extension modules. `Oily_png`[4] includes C extensions for resampling, PNG encoding and decoding, color

[4] *OilyPNG*, Willem van Bergen and others, 2015: `https://github.com/wvanbergen/oily_png`

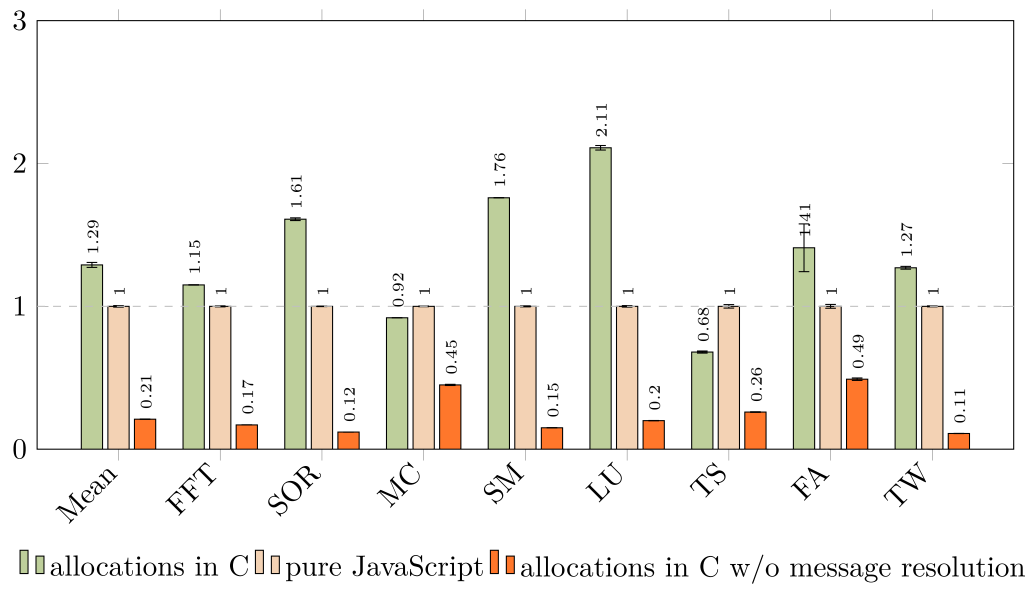


Figure 7.7: Main part in JavaScript and allocations in C with and without message resolution (normalized to pure TruffleJS performance; higher is better).

channel manipulation, and image composition. `Psd-native`[5] contains C extensions for color space conversion, clipping, layer masking, implementations of Photoshop's color blend modes, and some other utilities. In total, we evaluate 43 C extensions. The 43 routines were set up in a benchmark harness for evaluation. The Ruby harness allocates Ruby data, which is then processed in a C extension, i.e., the computations of these routines are implemented in C but the data is provided by Ruby.

We base the TMLR on Graal revision `9535eccd2a11`. Where an unmodified Java VM was required, we used the 64bit JDK 1.8.0u5 with default settings. Native versions of Ruby and C extensions were compiled with the system standard GCC 4.8.2. Figure 7.8 summarizes our peak performance results of all benchmarks by showing the geometric mean speedup over all benchmarks (y-axis, higher is better). We compare all implementations (x-axis) relative to the speed at which the C extensions run when using MRI's implementation of the C extensions API.

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 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 struc-

[5] *PSDNative*, Ryan LeFevre, 2015: `https://github.com/layervault/psd_native`

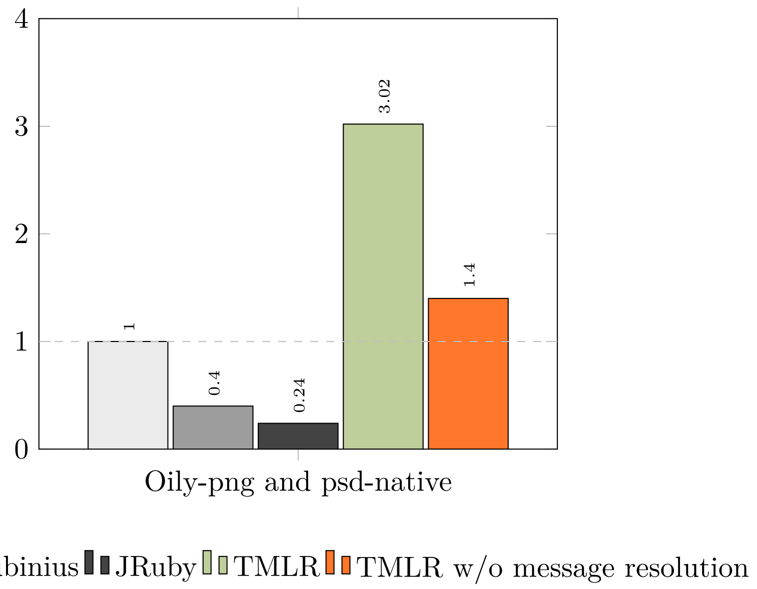


Figure 7.8: C extensions benchmarks (normalized to natively compiled C extensions that interface to MRI; higher is better).

tures and implementation techniques different from those in MRI. Most importantly, it uses C++ instead of C; so to implement the C extensions API, Rubinius has a bridging layer. This layer converts C extensions API calls to calls on Rubinius' C++ implementation objects. 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 uses Java's JNI [66] to implement a bridging layer to support MRI's C extensions API. This technique is almost the same as in Rubinius, except that now the interface between the VM and the conversion layer is even more complex. To share Ruby data with C extensions, JRuby must copy the data from the managed Java heap onto the unmanaged native heap. Whenever the native data is modified, JRuby copies the changes back to the managed Ruby object. To keep both sides of the divide synchronized, JRuby must keep performing this copy each time the interface is passed. 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.

TMLR is our system. We implement the C extensions API with *generic access*. As before (see Section 7.3.1), we add performance numbers for a configuration that disables message resolution.

Results

The baseline of our evaluation are natively compiled C extensions that interface to MRI. C extensions that interface to Rubinius are on average 60% slower than C extensions that interface to MRI. Rubinius needs a bridging layer to meet MRI's API, which introduces a significant run-time overhead. The C extensions also failed to make any progress on three of the benchmarks; we considered these benchmarks to have timed out and did not include these numbers in the reported mean value. C extensions that interface to JRuby are on average 76% slower than C extensions that interface to MRI. JRuby also has a bridging layer to meet MRI's API that uses JNI, which causes a significant overhead. The C extensions failed one benchmark with an error about a missing feature and did not make progress on 17 benchmarks in reasonable time. C extensions that run on top of the TMLR are on average 202% faster than native C extensions that interface to MRI. The TMLR performs better because MRI, Rubinius, and JRuby run the Ruby code in a dedicated VM and the C extensions are statically compiled and run natively. Every call or data access form native code to the VM (and vice versa) is a compilation barrier that prevents the compiler from performing any optimizations across the language boundaries. However, in our system, the C extensions are executed on top of TruffleC and are therefore running in the same VM. We use *generic access* for any foreign object access (C extensions accessing Ruby data), which removes the language boundaries completely and allows optimizations across languages. Our system performs best on C extensions that heavily access Ruby data but otherwise do little computation. *Generic access* removes all language boundaries, which in the best case allows compiling the entire benchmark into a single machine code routine. Performance is similar to native C extensions that interface to MRI if the benchmarks are computationally intense. In these cases, the performance numbers are dominated by the computationally intense parts rather than by the foreign data access. Without message resolution the C extensions run 54% slower. However, the performance is still 40% faster compared to native C extensions that interface to MRI.

7.4 Managed Data Allocations for C

In this section we compare TruffleC and TruffleC$_{\mathrm{M}}$. TruffleC can directly access the native data without any run-time checks. In TruffleC$_{\mathrm{M}}$ there is a type check for every data access (introduced by *generic access*) and also checks that ensure memory safety (introduced by the managed allocations). We discuss Graal's efficiency in optimizing these checks. Also, we compare the performance of TruffleC$_{\mathrm{M}}$ to related work that ensures spatial and temporal safety. TruffleC as well as TruffleC$_{\mathrm{M}}$ are based on Graal revision `5b24a15988fe`. For this evaluation we reuse the benchmarks that we have used to evaluate TruffleC.

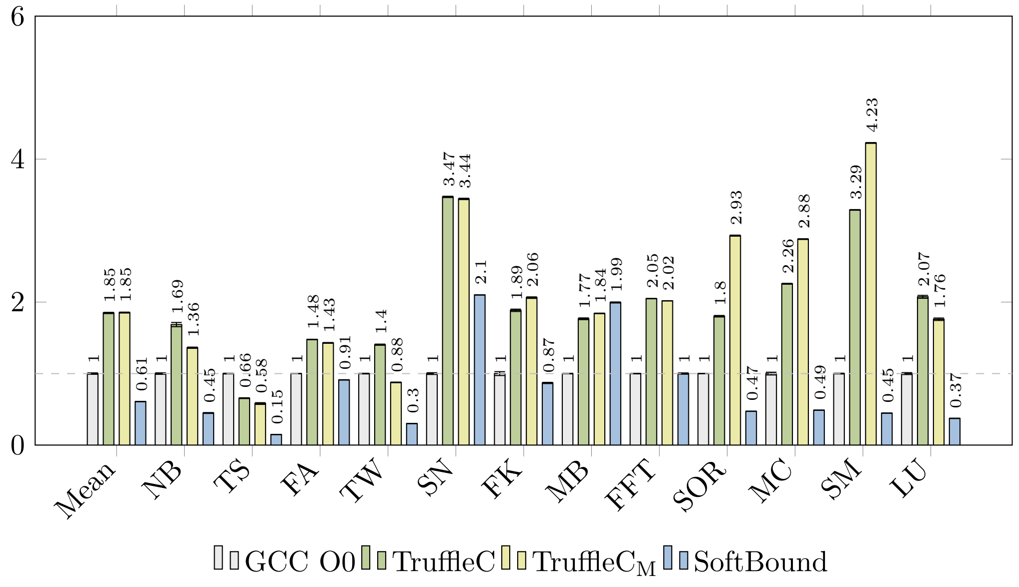


Figure 7.9: Performance numbers of TruffleC and TruffleC$_M$ (normalized to GCC *O0* performance; higher is better).

Figure 7.9 shows the results of the benchmarks. The x-axis of this chart shows the different benchmarks. The y-axis of this chart shows the average scores (higher is better) of the benchmarks. The baseline (1) is the performance of **GCC O0** (the GCC performance without optimization). **TruffleC** uses the same memory management as plain C compilers, i.e., it allocates data on the native heap and accesses it via *unsafe* access operations. **TruffleC$_M$** replaces these *unsafe* allocations with *safe* managed allocations. TruffleC$_M$ executes these benchmarks in *strict* mode (see Section 6.1). **SoftBound** [75,76] is an alternative, LLVM-based approach for memory-safe execution of C code, which we include in our performance evaluation. SoftBound is a recent approach for memory-safe execution of C code with good performance and uses compile-time transformations to insert run-time checks for detecting spatial and temporal safety violations.

Results

TruffleC$_M$ performs various checks when accessing a managed allocation. For example, when accessing an *Array object*, it checks if the managed allocation is not `null` and if the allocation is an *Array object*, if the index is within the bounds of the array, and finally if the expected type of the access expression matches the type of the array element.

The Graal compiler can eliminate many of these checks and can therefore reduce the run-time overhead [24, 25, 96, 117]. Using conditional elimination [96], the Graal compiler can remove redundant run-time checks by merging them. Conditional elimination reduces the number of conditional expressions by performing a data-flow analysis over the IR graph and by pruning conditions that can be proven to be true. Loop invariant code motion [24,25] moves run-time checks out of loops if the compiler can prove that a condition is loop-invariant. In this case, the static number of checks remains the same, but a check outside a loop is likely to be executed less often than inside a loop. Array bounds check elimination [117] fully removes bounds checks if the compiler can prove that they never fail. Also, whenever possible, the compiler moves bounds checks out of loops. In addition to that, the Graal compiler speculatively moves guards out of loops even if they have a control dependence in the loop. In this case, the (possibly stricter) condition before the loop implies the guard in the loop, which allows removing the check inside the loop. The performance evaluation shows that the safe execution of a C program with TruffleC$_M$ is on average as fast as the unsafe execution with TruffleC. TruffleC$_M$ performs equally well as TruffleC when the Graal compiler is able to remove all run-time checks (FA, SN, MB, FFT). In other words, there is no performance difference between an unsafe memory access and a safe managed object access when the compiler can remove all run-time checks. Only if the compiler cannot remove all run-time checks (NB, TW, LU), then TruffleC$_M$ is 20% (NB), 37% (TW), and 15% (LU) slower than TruffleC. Managed allocations allow the compiler to apply further optimizations. It can apply a sophisticated partial escape analysis with scalar replacement [98] on the objects of a C program. Also, it enables further copy propagation and common subexpression eliminations. If the compiler can remove all run-time checks and can exploit further optimizations, then TruffleC$_M$ can even outperform TruffleC. In this case, TruffleC$_M$ is between 9% (FK) and 63% (SOR) faster than TruffleC.

If we compare the performance of C code compiled with *SoftBound* to TruffleC$_M$, SoftBound is between 8% faster (MB) and 89% slower (SM) (67% slower on average) than TruffleC$_M$. SoftBound is based on LLVM and the compiler cannot optimize/remove the run-time checks as well as the Graal compiler can. We can conclude that TruffleC$_M$ is significantly faster than SoftBound in terms of peak performance.

7.5 Discussion

In this performance evaluation we show that if a TLI uses *generic access* to access data (e.g. foreign data), then the performance of this data access mainly depends on the implementation of the data structure and its access operations. We demonstrate that the dynamic Graal compiler efficiently removes the language and type checks in *generic access*. The performance of a multi-language program depends on the performance of the individual language parts. Using heavyweight foreign data has a negative impact on performance (e.g. heavyweight Ruby objects used in C). On the other hand, using efficient foreign data has a positive effect on performance (e.g. efficient C data used in Ruby).

In addition, *generic access* ensures excellent performance of multi-language applications because of two reasons: First, message resolution replaces language-agnostic messages with efficient foreign-language-specific operations. Accessing foreign objects becomes as efficient as accessing objects of the host language. Second, the dynamic compiler can perform optimizations across language borders because these borders were removed by message resolution. For example, the compiler can inline C functions into Ruby code and vice versa, which enables optimizations across language boundaries.

Chapter 8

Related Work

To put the TMLR in context, we discuss related approaches of executing C code on top of a JVM including approaches that dynamically compile C code. We also compare the TMLR to related approaches to cross-language interoperability, including foreign function interfaces, inter-process communication, and multi-language runtimes. Finally, we compare TruffleC$_M$ to other approaches that ensure memory safety.

8.1 C Language Implementations

Previous efforts to execute C programs on the JVM mostly rely on the translation of C to bytecode or Java source code. Most approaches do this by first translating the C code to an IR, such as MIPS machine code or LLVM IR. NestedVM [2] targets the translation of unsafe native code to safe Java bytecode. It uses a MIPS compiler to obtain machine code, which is then further translated using two different modes. The first mode translates this machine code to Java source code, which is then compiled by `javac`. The second mode is a direct conversion to Java bytecode. Cibyl [59] is similar to NestedVM but targets Java J2ME devices. It first translates C to MIPS code. Afterwards, it generates Java bytecode as well as Java wrappers for system calls. Cibyl supports the full C language. LLJVM[1] compiles a C program to LLVM IR using a frontend such as Clang[2]. It then translates the LLVM IR to Java bytecode. We execute C code on top of a JVM without creating bytecode. TruffleC produces a self-optimizing AST from the C source code, which is interpreted and eventually compiled to machine code. Truffle ASTs are flexible, e.g., we can rewrite them to apply optimizations on the AST level. When the Graal compiler finally compiles the ASTs to machine code, it uses the profile information collected during interpretation to guide its optimizations.

[1]*LLJVM 2.7*, GitHub repository, 2015: `https://github.com/davidar/lljvm`
[2]*clang: a C language family frontend for LLVM*, LLVM, 2015: `http://clang.llvm.org`

Dynamic compilation allows optimizations to exploit profiling information, e.g., by replacing variables with constants if their values do not change. To use this information when compiling C code, Auslander et al. [4] partially evaluate the code by producing pre-optimized machine code templates in a static compiler. These machine code templates contain placeholders for variables that will be constant at run time. They use a fast dynamic compiler that fills the templates with the missing data at run time. The approach of Auslander et al. [4] requires the programmer to annotate program fragments in order to guide the template creation. TruffleC profiles variable values at run time and automatically specializes them to constants if possible. TruffleC makes optimistic assumptions about values being constants, function pointer calls being constant and code parts being never executed. These assumptions can later lead to deoptimization of the machine code if they are violated. Execution then continues in the AST interpreter and TruffleC can re-compile the code with a different specialization at a later point.

'C and the tcc compiler[3] [82] allow the programmer to explicitly designate C statements and expressions in the source code for dynamic code generation at run time rather than statically compiling them. 'C is an extension of ANSI C. However, the changes are small, which makes it easy to learn and use. TruffleC, on the other hand, does not require the programmer to use a different programming language or to explicitly compile statements dynamically. Truffle schedules compilation of frequently executed code parts automatically and uses the collected profiling information to guide its optimizations.

Industry standard C compilers, such as GCC, can compile and optimize C code based on profile information. Profile-guided optimization contains the following three steps (see also Section 3.1.4):

1. Programmers can configure the compiler to produce binaries that profile an application and dump the *profile* of a program run (profile code generation).

2. This binary is executed and produces the profile of a program run (profiling run).

3. After the profiling run, the program is compiled a second time (profile-guided compilation). This time the compiler uses the profile information, which allows optimizations such as loop unrolling, or the specialization of certain program paths.

This approach clearly differs from TruffleC because it requires the programmer to manually collect profile information and to recompile a program. In contrast to that, TruffleC starts interpreting the C code and automatically collects profile information, which is

[3] *'C*, Max Poletto, 2015: `http://pdos.csail.mit.edu/tickc`

used when the IR is dynamically compiled. All this happens without any user interaction. The drawback of dynamic compilation is that it optimizes a program every time it runs, whereas static compilers optimize an application only once, which saves energy. Also, a dynamically compiled program reaches its peak performance only after a warm-up phase (in which frequently executed program parts are dynamically compiled and optimized), whereas as a statically compiled program runs at full speed from the beginning. However, a multi-language environment has to deal with foreign objects of possibly varying languages and thus has to specialize to these languages at run time. Hence, in our scenario of multi-language development dynamic compilation is preferable.

8.2 Cross-Language Interoperability

8.2.1 Foreign Function Interfaces

Most modern VMs expose an FFI such as Java's JNI [67], Java's Native Access[4], or Java's Compiled Native Interface[5]. An FFI defines a specific interface between two languages. Programmers can compose a pair of languages by using an API that allows accessing foreign objects. The result is rather inflexible, i.e., in order to interact with a foreign language, the programmer has to write glue code and this code only works for a specific pair of languages. Also, FFIs primarily allow integrating C/C++ code, e.g., Ruby and C (Ruby's C extensions mechanism), R and C (native R extensions), or Java and C [67]. They hardly allow integrating code written in a different language than C.

Wrapper generation tools (e.g. the tool Swig [9] or the tool described by Reppy and Song [84]) use annotations to generate FFI code from C/C++ interfaces, rather than requiring users to write FFI glue code by hand. A similar approach is described in [62], where existing interfaces are transcribed into a new notion instead of using annotations.

Compilation barriers at language boundaries have a negative impact on performance. To widen the compilation span across multiple languages, Stepanian et al. [100] describe an approach that allows inlining native functions into a Java application using a JIT compiler. They can show how inlining substantially reduces the overhead of JNI calls.

Kell et al. [61] describe *invisible VMs*, which allow a simple and low-overhead foreign function interfacing. They implement the Python language and minimize the FFI overhead to natively compiled code.

[4] *Java Native Access (JNA)*, GitHub repository, 2015: `https://github.com/twall/jna`
[5] *Compiled Native Interface (CNI)*, GCC the GNU Compiler Collection, 2015: `http://gcc.gnu.org/onlinedocs/gcj/About-CNI.html`

Jeannie [52] allows toggling between C and Java, hence, the two languages can be combined without writing boilerplate code. In Jeannie, programmers can mix both Java and C code in the same file and Jeannie compiles this code down to JNI.

There are many other approaches that target a fixed pair of languages [10,32,58,86, 113]. These approaches are all tailored towards interoperability between two specific languages and cannot be generalized for arbitrary languages and VMs. In contrast to them, our solution provides true cross-language interoperability between any Truffle-based languages rather than just pairwise interoperability. We can compose languages and reduce boilerplate code to a minimum, we do not target a fixed set of languages, and *generic access* does not introduce a compilation barrier when crossing language boundaries.

8.2.2 Inter-Process Communication

IDLs (interface description languages) implement cross-language interoperability via message-based inter-process communication between separate runtimes. This approach is mainly targeted to remote procedure calls and often not only aims at bridging different languages but also at calling code on remote computers. Programmers can define an interface in an IDL that can then be mapped to multiple languages. An IDL interface is translated to stubs in the host language and in the foreign language, which can then be used for cross-language communication [42,80,94,104]. These per-language stubs marshal data to and from a common wire representation. However, this approach introduces a marshalling and copying overhead as well as an additional maintenance burden (learning and using an IDL, together with its toolchain).

Using IDLs in the context of single-process applications has only been explored in limited ways [27, 105]. Also, these approaches retain the marshalling overhead and cannot share objects directly. *Generic access* is also based on messages, but they are resolved at run time and are replaced with direct access operations to foreign objects. These messages are transparent to the programmer and are automatically generated. The TMLR makes the mapping of foreign language operations to messages the task of the language implementer rather than the task of the application programmer. Our approach accesses foreign objects directly instead of copying them at language borders. In fact, language borders are completely eliminated so that the dynamic compiler can optimize across languages and can thus improve the performance of multi-language applications significantly.

8.2.3 Multi-Language Runtimes

The TMLR composes language implementations that are running on a shared VM, which is closely related to Microsoft's Common Language Runtime [16, 56, 70] as well as to the RPython [15] runtime. In the following we compare these runtimes to the TMLR.

The Microsoft Common Language Infrastructure (CLI) [56] describes language implementations that compile different languages to a common IR that is executed by the CLR [16]. The CLR provides a common type system, an automatic memory management, a JIT compiler (a function is compiled just before execution), a garbage collector, a security manager, and a class loader. The CLR can execute conventional object-oriented imperative languages, dynamically typed language, and the functional languages (e.g. F#).

The Dynamic Language Runtime (DLR) [43] is a framework for implementing dynamic languages on top of the CLR, which is similar to Truffle as a framework on top of the JVM. Language developers parse source code to an expression tree, which is the DLR's representation of source code. The DLR defines a fixed set of language-agnostic expressions that language implementers use to build up an expression tree. DLR expression trees can be interpreted by the DLR's interpreter or converted to the CLR's IR, which is directly compiled to machine code. A DLR language implementation transforms operations of dynamically typed operands to *call sites* [43]. For example, an implementation does not emit IR code that adds two numbers for a JavaScript `+` operation, but it emits a call site. A call site is a placeholder for an operation that is resolved at run time. The DLR uses a *delegate* to implement a call site. A delegate then calls the different implementations of an operation. In the case of a JavaScript `+` operation, the delegate can call the `double` instance of a `+` operation (implemented as a type check followed by a `double` addition). This approach is different to Truffle because Truffle ASTs are self-optimizing and speculatively rewrite themselves with *specialized* variants at run time, e.g., based on profile information. This technique allows specializing on a subset of the semantics of a particular operation. Truffle compiles frequently executed ASTs to machine code and deoptimizes them if a tree needs to be *re-specialized*.

A language implementation on top of the DLR needs to use the object model of the CLR to implement the objects of a guest language, i.e., it has to represent them using CLR's static classes [43]. DLR languages can make dynamic calls on objects defined in other languages. Similar to *generic access*, the DLR is inspired by Smalltalk. The DLR defines a meta-object protocol with defines a set of language-agnostic operations on objects. These operations on objects are again implemented with call sites and delegates.

Microsoft's approach is different from ours because of the following reasons:

Object representation: Language implementations on top of the CLR (including the DLR languages) need to use the statically typed and managed object model of the CLR. Tight interoperability on the IR level is only possible between languages whose type system corresponds to the CTS. The object access is implemented using the CLR's IR.

Generic access, on the other hand, allows every language to have its own representation of objects and to define individual access operations. TLIs are not bound to a common object representation, e.g., TruffleJS allocates objects as `DynamicObject` instances (on the managed Java heap) whereas TruffleC allocates them as plain byte sequences (on the unmanaged native heap). *Generic access* resolves and embeds language-specific AST snippets for each access at run time (e.g., access operations to the managed Java heap or a raw memory access to the native heap).

Languages: The CLR accesses unmanaged code (e.g. C code) via the annotation-based PInvoke and the FFI-like IJW interface, which uses explicit marshalling and a pinning API. The TMLR treats unmanaged languages (e.g. C) as first-class citizens and provides a TLI for them. TruffleC supports *generic access* and can access managed and dynamically typed objects. Also, other high-level languages (e.g. JavaScript) can access unmanaged C data efficiently.

Object access: The DLR uses cached delegates to access foreign objects, which causes a call site for every foreign object access. Jeff Hardy states in [43] that a foreign object access is *as fast as other dynamic calls, and almost as fast as static calls*. *Generic access* avoids this indirection, i.e., there are no call sites between languages because *generic access* directly embeds the foreign object access into the AST of the host application. We specialize a foreign object access on the language and the type of the foreign object and embed it into the host language's AST thus eliminating any boundaries between languages. If the foreign object suddenly has a different type or comes from a different language, the execution falls back to the AST interpreter and *generic access* resolves a new foreign object access.

Cross-language interoperability on top of RPython [6–8] allows the programmer to toggle between syntax and semantics of languages on the statement level. Barrett et al. describe a combination of Python and Prolog called Unipycation [6] or a combination of Python and PHP called PyHyp [8]. Unipycation and PyHyp compose languages by combining their interpreters. Both approaches glue the interpreters together on the language implementation level. The new interpreter is then compiled using a meta-tracing JIT. To share data across languages, Unipycation and PyHyp wrap objects using *adapters.* Like the TMLR, the approach of Barrett et al. avoids compilation boundaries between languages and multi-language applications show good performance. In contrast to Barret et al.'s approach, however, the TMLR is not restricted to a fixed set of languages. Unipycation and PyHyp both compose a specific pair of language implementations whereas *generic access* is a general mechanism to compose arbitrary Truffle language implementations. Unipycation, and PyHyp propose a more fine-grained language composition compared to our approach. However, when languages are mixed at source code level, editors, compilers, and debuggers have to be adapted.

8.2.4 Multi-Language Semantics

The semantics of language composition is a well-researched area [1,32,33,69,102,113], however, most of these approaches do not have an efficient implementation. Our work partially bases on ideas from existing approaches (i.e., *like types* from Wrigstad et al. [113], Section 5.1.2) and therefore stands to complement such efforts.

8.3 Spatial and Temporal Memory Safety

TruffleC_M uses managed data allocations for C and can therefore ensure memory safety. We discuss the Boehm-Demers-Weiser GC [11–14] that automatically manages C objects. Also, we discuss *software-based* approaches that ensure memory safety of a C program execution. TruffleC_M is a software-based approach, thus we focus on research in this area rather than on hardware-based research such as [18,21,73,74,83,91,92,103]. Like existing literature surveys [76], we distinguish between *pointer-based approaches* and *object-based approaches.*

8.3.1 Boehm-Demers-Weiser Garbage Collector

The Boehm-Demers-Weiss GC is a conservative GC for C and C++ that can provide temporal safety. It uses a mark-sweep algorithm and provides incremental and generational collection. It works with unmodified C programs by replacing native memory allocations with *GC allocations* and removing `free` calls completely. Also, it can run in a *leak detection* mode that allows ensuring temporal safety.

Like TruffleC_M, this approach automatically deallocates memory that is not used anymore. However, the architecture of the two approaches is different. TruffleC_M executes C code on a VM via AST interpretation. It introduces `MAddress` objects to represent pointers and allocates managed objects, which allows reusing the GC of an existing JVM directly.

8.3.2 Pointer-Based Approaches

Our technique is inspired by *pointer-based metadata* approaches. Pointer-based approaches [5, 73, 75–78] store additional information for each pointer of a C program so that pointers become multi-word values (*fat pointers*) [5, 77] that hold the actual pointer value along with metadata (e.g., the upper and the lower bounds of the referenced object). Pointer arithmetic then modifies the actual pointer value, but the metadata remains unchanged. When a pointer is dereferenced, the actual pointer value is checked against the bounds of the object, which ensures spatial safety. To ensure temporal safety, every allocated object gets a unique identifier (capability) [5]. These capabilities stay in existence even after the deallocation of an object, which allows checking the pointer's validity when it is dereferenced. SafeC [5] detects spatial and temporal memory errors by using fat pointers that encode the pointer value, base and size information, a storage class (heap, local, or global allocation), and a capability to the referent. CCured [77, 78] classifies pointers in three categories: *SAFE* pointers, which cannot be used for pointer arithmetic, array indexing or type casts and cause almost no run-time overhead; *SEQ* pointers, which are fat pointers that support pointer arithmetic, array indexing and primitive casts; and *WILD* pointers, which support arbitrary casts but have additional metadata and require run-time checks. Nagarakatte et al. describe SoftBound [75] and CETS [76], which use compile-time transformations and insert runtime checks for detecting spatial and temporal safety violations. These approaches keep the metadata in a separate metadata space (in contrast to fat pointers), which retains memory layout compatibility.

An `MAddress` object can be seen as a fat pointer (to ensure spatial safety) and the memory management of the JVM ensures temporal safety. The novelty of TruffleC$_{\mathrm{M}}$ is that we transferred the idea of *fat pointers* to a C interpreter (i.e., TruffleC), which is implemented in Java. We can reuse the sophisticated automated memory management (the JVM garbage collector) of the host VM. This allows us to ensure spatial and temporal safety with little effort. We use a layout table to map the offset of an `MAddress` to the corresponding member of a managed allocation and eventually do a Java member access, which ensures spatial safety. We mark freed objects as deallocated, which allows us to ensure temporal safety. The GC of the JVM eventually deallocates the object. Also, TruffleC$_{\mathrm{M}}$ is source-compatible with regular C programs and does not require any changes in them because we mimic the behavior of industry-standard C compilers. We dynamically compile ASTs with a state-of-the-art dynamic compiler, which is very good at removing access checks or moving them out of hot loops. After an initial warm-up and dynamic compilation of the AST we can report that TruffleC$_{\mathrm{M}}$ is on average 28% slower than the best performance of GCC. The dynamic compilation of TruffleC$_{\mathrm{M}}$ introduces a warm-up overhead, which related approaches do not need. However, we can specialize the AST on the profile of a C program execution, which results in excellent peak performance. Hence, for long running C applications that require memory safety, TruffleC$_\mathrm{M}$ is preferable.

8.3.3 Object-Based Approaches

Object-based approaches track information about each object such as its status (allocated/deallocated) or its bounds and store it in an auxiliary data structure [19, 22, 26, 44, 57, 79, 87]. Spatial and temporal safety is ensured by mapping pointer values to the tracked information (e.g., using a splay tree [57] or a trie [79]) and by checking that pointer arithmetic and pointer dereferencing fall within the bounds of the object. Purify [44] traps every memory access by instrumenting the object code of a program and by allocating *red zones* before and after each allocation. Eigler's mudflap system [26] inserts an additional pass into GCC's normal compilation to instrument the C code and to assert a validity predicate at every use of a pointer. Mudflap caches the lookup in the auxiliary data structure. Dhurjati and Adve [22] use a fine-grained partitioning of memory to provide run-time bounds checking for arrays and strings. Ruwase and Lam [87] prevent buffer overflows by introducing out-of-bound objects for all out-of-bound pointers. Any pointer derived from an out-of-bound object is bound-checked before it can be dereferenced.

Our system represents pointer values by `MAddress` objects that use a Java reference to refer to the allocation. We detect spatial memory errors when the `offset` of the `MAddress` object cannot be mapped to a member of the referent and temporal errors when the referent is not allocated (`data` is `null`) or the referent is marked as deallocated. This is different to object-based approaches, hence, we consider TruffleC$_\mathrm{M}$ as distinct from object-based approaches.

Chapter 9

Summary

This section outlines future work and finally concludes this thesis.

9.1 Future Work

9.1.1 Cross-Language Interoperability

The TMLR is a good basis for future research in cross-language interoperability. Examples include research on multi-language inheritance or multi-language concurrency. Also, *generic access* can be used to implement further FFIs, e.g., a C extensions API for other languages.

Additional languages: The TMLR allows adding new TLIs easily. An TLI has to support *generic access* and needs access to the multi-language scope. Having these extensions, the TLI can be added to the TMLR. As future work we want to add further languages to the runtime. There is a TLI for the functional language Clojure, for the dynamic language Python[1] [106,122], and also for the mathematical language R[2]. Like we did for JavaScript, Ruby, and C (see Section 5.1.2), this work requires bridging different language paradigms and features. We did not include these languages to the case study of this thesis (see Chapter 5) because at the time of writing this thesis Clojure was in an early state and under heavy development. Python or R were developed by external collaborators. Hence, adding these TLIs to the TMLR would have required a major engineering effort and was therefore intentionally left for future work.

[1] *Zippy - a Python implementation on top of Truffle*, Bitbucket repository, 2015: `https://bitbucket.org/ssllab/zippy`

[2] *FastR - an R implementation on top of Truffle*, Bitbucket repository, 2015: `https://bitbucket.org/allr/fastr`

C extensions support for other languages: Besides Ruby, other dynamic languages also have a C extensions API, e.g. Python[3] or R[4]. Similar to our implementation for TruffleRuby, an implementation of the C extensions API could simply substitute invocations of these extension functions with a *generic access*.

Multi-language inheritance: Currently, there is no support for cross-language inheritance, i.e., class-based inheritance or prototype-based inheritance is only possible with classes or objects that originate from the same language. However, we are convinced that the TMLR is extensible in this respect and therefore future research could focus on inheritance across language boundaries.

Cross-language debuggers: The Truffle framework allows the implementation of debuggers with zero-overhead [90]. Future research will focus on generalizing the existing debuggers for TLIs so that they can be used for multi-language applications. The goal of this work is a zero-overhead debugger for multi-language applications that allows developers to step into functions or inspect data, which were implemented or allocated in different languages.

Multi-language concurrency and parallelization: Modern programming languages use a wide variety of different models for concurrency. Applications, written in multiple languages need to unify these models and bridge the differences across languages. Our future research will investigate concurrency and parallelization across language borders.

As a first step, the author of this thesis started exploring this area of research in collaboration with Daniele Bonetta (Oracle Labs) at the time of writing this thesis. The aim of this work is to explore an implicit parallel-data programming model combining JavaScript data structures with C functions.

9.1.2 TruffleC and TruffleC$_M$

Our future work will extend TruffleC and complete its implementation.

Completeness: TruffleC is not yet complete and does not yet support all features of the C99 standard (see Section 3.1.6). Features such as flexible array members, designated initializers, or compound literals are left out for future work. These features only require an implementation effort and there are no restrictions that would limit supporting the full C standard.

[3] *Python Language*, Python Software Foundation, 2015: `https://www.python.org/`
[4] *The R Project for Statistical Computing*, The R Foundation, 2015: `http://www.r-project.org/`

Multi-threading: At the time of writing this thesis, Truffle had only experimental support for multi-threading. Hence, there is no multi-threading support for TruffleC yet. We plan to add multi-threading for TruffleC as soon as the Truffle framework itself supports concurrent executions of ASTs.

TruffleLLVM: As part of our future work on TruffleC we are planning to replace TruffleC's parser (the parser that transforms C code to a Truffle AST) with a parser for LLVM bitcode. Rather than interpreting C code we would then interpret LLVM bitcode. LLVM bitcode is the internal representation of source code when using LLVM. There are various front-ends that parse source code (e.g. C/C++ or Fortran) to LLVM bitcode, which is then optimized and transformed to machine code by LLVM. A TLI that can execute LLVM bitcode would allow us to execute all languages that have a LLVM front-end, such as C, C++ or Fortran. This TLI can reuse large parts of TruffleC. Most of the TruffleC nodes can be directly reused because the semantics of LLVM bitcode is very similar to that of C. Also, all dynamic optimizations and the memory-safe execution of a program that we have implemented for C can be directly applied to LLVM bitcode. Other languages such as C++ or Fortran would then also benefit from the TMLR approach.

Our future work on TruffleC$_\mathrm{M}$ will improve completeness. We will complete the list of substitutions of standard library functions. This work will make TruffleC$_\mathrm{M}$ applicable to full-sized applications.

9.2 Conclusion

In this thesis we proposed the TMLR, a runtime with a set of language implementations that can efficiently execute multi-language applications. The language implementations of the TMLR translate source code into a self-optimizing AST. The runtime hosts managed high-level languages (JavaScript and Ruby) as well as unmanaged low-level languages (C). We presented TruffleC, which is an AST interpreter for C that dynamically compiles C code.

Our work allows us to conclude that a uniform approach of language implementation on the same VM is the most essential factor for efficient cross-language interoperability. We compose different language implementations on an AST level via a language-agnostic mechanism, which we call *generic access*. Language implementations use language-independent messages to access foreign objects that are resolved at their first execution and transformed to efficient foreign-language-specific operations. *Generic access* is independent of languages, which allows adding new languages to the

TMLR without affecting existing languages. This approach leads to excellent performance of multi-language applications because of two reasons. First, message resolution replaces language-agnostic messages with efficient foreign-language-specific operations. Accessing foreign objects becomes as efficient as accessing objects of the host language. Second, the dynamic compiler can perform optimizations across language borders because these borders were removed by message resolution. We show that using heavyweight foreign data has a negative impact on performance whereas using lightweight foreign data has a positive effect on performance. Our evaluation shows that the dynamic compiler of the TMLR can minimize the effect of *generic access*'s language and type check on the receiver.

We presented two different case studies that evaluate *generic access*. First, we discussed seamless cross-language interoperability between JavaScript, Ruby, and C. The TMLR allows programmers to directly access foreign objects using the operators of the host language. The *generic access* makes the mapping of access operations to messages largely the task of the language implementer rather than the task of the end programmer. Second, we used the *generic access* to implement the C extensions API for TruffleRuby. TruffleC substitutes invocations of C extensions API functions and uses *generic access* for accessing Ruby objects instead. Our system is therefore compatible with MRI's C extensions API and can execute real-world applications.

The TMLR can also ensure memory safety of a C program execution. We introduce TruffleC$_{\mathrm{M}}$, which uses *generic access* to substitute native allocations with managed objects. Access operations to managed data perform additional checks compared to access operations to unmanaged data. *Generic access* introduces a type check and the access operations implement checks that ensure memory safety. We show that the dynamic compiler can efficiently optimize these additional checks. TruffleC$_{\mathrm{M}}$ and TruffleC have the same performance on average.

The TMLR can be the basis for a wide variety of different areas of future research. Topics are, for example, multi-language concurrency and parallelism, cross-language inheritance, or cross-language debuggers.

List of Figures

List of Listings

List of Tables

Bibliography

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic Typing in a Statically-typed Language. In *Proceedings of the 16th Symposium on Principles of Programming Languages*, POPL '89, pages 213–227, New York, NY, USA, 1989. ACM.

[2] Brian Alliet and Adam Megacz. Complete Translation of Unsafe Native Code to Safe Bytecode. In *Proceedings of the 2004 Workshop on Interpreters, Virtual Machines and Emulators*, IVME '04, pages 32–41, New York, NY, USA, 2004. ACM.

[3] M. Arnold, S.J. Fink, D. Grove, M. Hind, and P.F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of the IEEE*, 93(2):449–466, 2005.

[4] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, Effective Dynamic Compilation. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 149–159, New York, NY, USA, 1996. ACM.

[5] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 290–301, New York, NY, USA, 1994. ACM.

[6] Edd Barrett, Carl Friedrich Bolz, and Laurence 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.

[7] Edd Barrett, Carl Friedrich Bolz, and Laurence Tratt. Approaches to Interpreter Composition. *CoRR*, abs/1409.0757, 2014.

[8] Edd Barrett, Lukas Diekmann, and Laurence Tratt. Fine-grained language composition. *CoRR*, abs/1503.08623, 2015.

[9] David M Beazley et al. SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk Workshop*, pages 129–139, 1996.

[10] Matthias Blume. No-longer-foreign: Teaching an ML Compiler to Speak C Natively. *Electronic Notes in Theoretical Computer Science*, 59(1):36–52, 2001.

[11] Hans-J. Boehm. Reducing Garbage Collector Cache Misses. In *Proceedings of the 2nd International Symposium on Memory Management*, ISMM '00, pages 59–64, New York, NY, USA, 2000. ACM.

[12] Hans-J. Boehm. Bounding Space Usage of Conservative Garbage Collectors. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 93–100, New York, NY, USA, 2002. ACM.

[13] Hans J. Boehm. Space Efficient Conservative Garbage Collection. *SIGPLAN Not.*, 39(4):490–501, April 2004.

[14] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly Parallel Garbage Collection. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 157–164, New York, NY, USA, 1991. ACM.

[15] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[16] D Box and C Sells. Essential .NET. The Common Language Runtime, vol. I, 2002.

[17] David Chisnall. The Challenge of Cross-language Interoperability. *Commun. ACM*, 56(12):50–56, 2013.

[18] Weihaw Chuang, Satish Narayanasamy, and Brad Calder. Accelerating Meta Data Checks for Software Correctness and Security. *Journal of Instruction-Level Parallelism*, 9:1–26, 2007.

[19] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 351–366. ACM, 2007.

[20] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[21] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 103–114, New York, NY, USA, 2008. ACM.

[22] Dinakar Dhurjati and Vikram Adve. Backwards-compatible Array Bounds Checking for C with Very Low Overhead. In *Proceedings of the 28th international conference on Software engineering*, pages 162–171. ACM, 2006.

[23] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, and Christian Wimmer. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[24] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. Speculation Without Regret: Reducing Deoptimization Meta-data in the Graal Compiler. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 187–193, New York, NY, USA, 2014. ACM.

[25] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 1–10, New York, NY, USA, 2013. ACM.

[26] Frank Ch Eigler. Mudflap: Pointer Use Checking for C/C+. In *GCC Developers Summit*, page 57. Citeseer, 2003.

[27] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling Hell from Heaven and Heaven from Hell. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, ICFP '99, pages 114–125, New York, NY, USA, 1999. ACM.

[28] Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, March 1986.

[29] International Organization for Standardization. C99 Standard: ISO/IEX 9899:TC3. `www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`, 2007.

[30] International Organization for Standardization. C11 Standard: ISO/IEX 9899:201x. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`, 2011.

[31] Yoshihiko Futamura. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[32] Kathryn Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained Interoperability Through Mirrors and Contracts. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 231–245, New York, NY, USA, 2005. ACM.

[33] Kathryn E. Gray. Safe Cross-Language Inheritance. In *ECOOP 2008 – Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 52–75. Springer Berlin Heidelberg, 2008.

[34] Matthias Grimmer. A Runtime Environment for the Truffle/C VM. Master's thesis, Johannes Kepler University, Linz, 2013.

[35] Matthias Grimmer. High-performance Language Interoperability in Multi-language Runtimes. In *Proceedings of the Companion Publication of the 2014 ACM SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '14, pages 17–19, New York, NY, USA, 2014. ACM.

[36] Matthias Grimmer, Manuel Rigger, Roland Schatz, Lukas Stadler, and Hanspeter 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.

[37] Matthias Grimmer, Manuel Rigger, Lukas Stadler, Roland Schatz, and Hanspeter 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.

[38] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. Memory-safe Execution of C on a Java VM. In *Proceedings of the Tenth Workshop on Programming Languages and Analysis for Security*, PLAS'15, New York, NY, USA, 2015. ACM.

[39] Matthias Grimmer, Chris Seaton, Roland Schatz, Würthinger, and Hanspeter Mössenböck. High-Performance Cross-Language Interoperability in a Multi-Language Runtime. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS '15, New York, NY, USA, 2015. ACM.

[40] Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages. In *Proceedings of the 14th International Conference on Modularity*, MODULARITY 2015, pages 1–13, New York, NY, USA, 2015. ACM.

[41] Matthias Grimmer, Thomas Würthinger, Andreas Wöß, and Hanspeter 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.

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

[43] Jeff Hardy. The Dynamic Language Runtime and the Iron Languages. In Amy Brown and Greg Wilson, editors, *The Architecture Of Open Source Applications, Volume II.* `http://aosabook.org`, 2008.

[44] Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proc. of the Winter 1992 USENIX Conference.* Citeseer, 1991.

[45] Christian Häubl and Hanspeter Mössenböck. Trace-based Compilation for the Java HotSpot Virtual Machine. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 129–138. ACM, 2011.

[46] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Optimized Strings for the Java HotSpot&Trade; Virtual Machine. In *Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, PPPJ '08, pages 105–114, New York, NY, USA, 2008. ACM.

[47] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Compact and efficient strings for Java. *Science of Computer Programming*, 75(11):1077 – 1094, 2010. 23rd ACM Symposium on Applied Computing 08.

[48] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Evaluation of Trace Inlining Heuristics for Java. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1871–1876. ACM, 2012.

[49] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Context-sensitive Trace Inlining for Java. *Computer Languages, Systems & Structures*, 39(4):123–141, 2013.

[50] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Deriving Code Coverage Information from Profiling Data Recorded for a Trace-based Just-in-time Compiler. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 1–12. ACM, 2013.

[51] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Trace Transitioning and Exception Handling in a Trace-based JIT Compiler for Java. *ACM Trans. Archit. Code Optim.*, 11(1):6:1–6:26, February 2014.

[52] Martin Hirzel and Robert Grimm. Jeannie: Granting Java Native Interface Developers Their Wishes. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 19–38, New York, NY, USA, 2007. ACM.

[53] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-typed Object-oriented Languages with Polymorphic Inline Caches. In Pierre 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.

[54] Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM.

[55] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A Domain-specific Language for Building Self-optimizing AST Interpreters. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, pages 123–132. ACM, 2014.

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

[57] Richard WM Jones and Paul HJ Kelly. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *AADEBUG*, pages 13–26. Citeseer, 1997.

[58] Simon Peyton Jones, Thomas Nordin, and Alastair Reid. GreenCard: A Foreign-language Interface for Haskell. In *Proc. Haskell Workshop*, 1997.

[59] Simon Kågström, Håkan Grahn, and Lars Lundberg. Cibyl: An Environment for Language Diversity on Mobile Devices. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 75–82. ACM, 2007.

[60] Tomas Kalibera and Richard Jones. Rigorous Benchmarking in Reasonable Time. In *Proceedings of the 2013 ACM SIGPLAN International Symposium on Memory Management (ISMM)*, 2013.

[61] Stephen Kell and Conrad Irwin. Virtual Machines Should Be Invisible. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 289–296, New York, NY, USA, 2011. ACM.

[62] F Klock II. The layers of Larceny's Foreign Function Interface. In *Scheme and Functional Programming Workshop*. Citeseer, 2007.

[63] Thomas Kotzmann and Hanspeter Mössenböck. Escape Analysis in the Context of Dynamic Compilation and Deoptimization. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 111–120. ACM, 2005.

[64] Thomas Kotzmann and Hanspeter Mossenbock. Run-time Support for Optimizations based on Escape Analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 49–60. IEEE Computer Society, 2007.

[65] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot Client Compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):7, 2008.

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

[67] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification.* Addison-Wesley Professional, 1999.

[68] Stefan Marr and Stephane Ducasse. Tracing vs. Partial Evaluation: Comparing Meta-Compilation Approaches for Self-Optimizing Interpreters. In *Proceedings of*

the 2015 ACM International Conference on Object Oriented Programming Systems Languages; Applications, OOPSLA '15, New York, NY, USA. ACM.

[69] Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 3–10, New York, NY, USA, 2007. ACM.

[70] Erik Meijer and John Gough. Technical Overview of the Common Language Runtime. *language*, 29:7, 2001.

[71] Hanspeter Mössenböck. Adding Static Single Assignment Form and a Graph Coloring Register Allocator to the Java HotSpot Client Compiler. In *Technical Report 15, Institute for Practical Computer Science, Johannes Kepler University Linz*, 2000.

[72] Hanspeter Mössenböck and Michael Pfeiffer. Linear Scan Register Allocation in the Context of SSA Form and Register Constraints. In *Compiler Construction*, pages 229–246. Springer, 2002.

[73] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 189–200, Washington, DC, USA, 2012. IEEE Computer Society.

[74] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 175:175–175:184, New York, NY, USA, 2014. ACM.

[75] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[76] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 31–40, New York, NY, USA, 2010. ACM.

[77] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[78] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 128–139, New York, NY, USA, 2002. ACM.

[79] Nicholas Nethercote and Julian Seward. How to Shadow Every Byte of Memory Used by a Program. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 65–74. ACM, 2007.

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

[81] Michael Paleczny, Christopher Vick, and Cliff Click. The Java Hotspot Server Compiler. In *Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[82] Massimiliano Poletto. *Language and Compiler Support for Dynamic Code Generation.* PhD thesis, Messachusetts Institute of Technology, 1999.

[83] Feng Qin, Shan Lu, and Yuanyuan Zhou. SafeMem: Exploiting ECC-memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 291–302. IEEE, 2005.

[84] John Reppy and Chunyan Song. Application-specific Foreign-interface Generation. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, pages 49–58, New York, NY, USA, 2006. ACM.

[85] Manuel Rigger. Truffle/C Interpreter. Master's thesis, Johannes Kepler University, Linz, 2014.

[86] John R. Rose and Hans Muller. Integrating the Scheme and C Languages. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, pages 247–259, New York, NY, USA, 1992. ACM.

[87] Olatunji Ruwase and Monica S Lam. A Practical Dynamic Buffer Overflow Detector. In *NDSS*, 2004.

[88] Thomas Schatzl, Laurent Daynes, and Hanspeter Mössenböck. Optimized Memory Management for Class Metadata in a JVM. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 151–160. ACM, 2011.

[89] Arnold Schwaighofer. Tail Call Optimizations for the Java HotSpot VM. Master's thesis, Johannes Kepler University, Linz, 2009.

[90] Chris Seaton, Michael L Van De Vanter, and Michael Haupt. Debugging at Full Speed. In *Proceedings of the Workshop on Dynamic Languages and Applications*, pages 1–13. ACM, 2014.

[91] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[92] Julian Seward and Nicholas Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

[93] Matthew S. Simpson and Rajeev K. Barua. MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime. *Software: Practice and Experience*, 43(1):93–128, 2013.

[94] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable Cross-language Services Implementation. *Facebook White Paper*, 5, 2007.

[95] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, and Thomas 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.

[96] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. An Experimental Study of the Influence of Dynamic Compiler Optimizations on Scala Performance. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 9:1–9:8, New York, NY, USA, 2013. ACM.

[97] Lukas Stadler, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck, and John Rose. Lazy Continuations for Java Virtual Machines. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 143–152. ACM, 2009.

[98] Lukas Stadler, Thomas Würthinger, and Hanspeter 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.

[99] Lukas Stadler, Thomas Würthinger, and Christian Wimmer. Efficient Coroutines for the Java Platform. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, pages 20–28. ACM, 2010.

[100] Levon Stepanian, Angela Demke Brown, Allan Kielstra, Gita Koblents, and Kevin Stoodley. Inlining Java Native Calls at Runtime. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 121–131, New York, NY, USA, 2005. ACM.

[101] L. Szekeres, M. Payer, Tao Wei, and D. Song. SoK: Eternal War in Memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62, 2013.

[102] Valery Trifonov and Zhong Shao. *Safe and principled language interoperation.* Springer, 1999.

[103] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. Memtracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 273–284. IEEE, 2007.

[104] Nanbor Wang, Douglas C Schmidt, and Carlos O'Ryan. Overview of the CORBA Component Model. In *Component-Based Software Engineering*, pages 557–571. Addison-Wesley Longman Publishing Co., Inc., 2001.

[105] Michal Wegiel and Chandra Krintz. Cross-language, Type-safe, and Transparent Object Sharing for Co-located Managed Runtimes. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 223–240, New York, NY, USA, 2010. ACM.

[106] Christian Wimmer and Stefan Brunthaler. ZipPy on Truffle: A Fast and Simple Implementation of Python. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, Applications: Software for Humanity*, SPLASH '13, pages 17–18, New York, NY, USA, 2013. ACM.

[107] Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 132–141. ACM, 2005.

[108] Christian Wimmer and Hanspeter Mössenböck. Automatic Object Colocation Based on Read Barriers. In *Modular Programming Languages*, pages 326–345. Springer, 2006.

[109] Christian Wimmer and Hanspeter Mössenböck. Automatic Feedback-directed Object Inlining in the Java Hotspot Virtual Machine. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 12–21. ACM, 2007.

[110] Christian Wimmer and Hanspeter Mössenböck. Automatic Array Inlining in Java Virtual Machines. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 14–23. ACM, 2008.

[111] Christian Wimmer and Hanspeter Mössenbösck. Automatic Feedback-directed Object Fusing. *ACM Trans. Archit. Code Optim.*, 7(2):7:1–7:35, October 2010.

[112] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter 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, pages 133–144, New York, NY, USA, 2014. ACM.

[113] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating Typed and Untyped Code in a Scripting Language. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 377–388, New York, NY, USA, 2010. ACM.

[114] Thomas Würthinger, Danilo Ansaloni, Walter Binder, Christian Wimmer, and Hanspeter Mössenböck. Safe and atomic run-time code evolution for java and its application to dynamic aop. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 825–844, New York, NY, USA, 2011. ACM.

[115] Thomas Würthinger, Walter Binder, Danilo Ansaloni, Philippe Moret, and Hanspeter Mössenböck. Applications of Enhanced Dynamic Code Evolution for Java in GUI Development and Dynamic Aspect-oriented Programming. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 123–126, New York, NY, USA, 2010. ACM.

[116] Thomas Würthinger, Walter Binder, Danilo Ansaloni, Philippe Moret, and Hanspeter Mössenböck. Improving Aspect-oriented Programming with Dynamic Code Evolution in an Enhanced Java Virtual Machine. In *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution*, page 5. ACM, 2010.

[117] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array Bounds Check Elimination for the Java HotSpot Client Compiler. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, PPPJ '07, pages 125–133, New York, NY, USA, 2007. ACM.

[118] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Visualization of Program Dependence Graphs. In *Compiler Construction*, pages 193–196. Springer, 2008.

[119] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array Bounds Check Elimination in the Context of Deoptimization. *Science of Computer Programming*, 74(5):279–295, 2009.

[120] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario 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*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. ACM.

[121] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian 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.

[122] Wei Zhang, Per Larsen, Stefan Brunthaler, and Michael Franz. Accelerating Iterators in Optimizing AST Interpreters. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages; Applications*, OOPSLA '14, pages 727–743, New York, NY, USA, 2014. ACM.