JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Faculty of Engineering
and Natural Sciences

A Runtime Environment for the Truffle/C VM

Master's Thesis

submitted in partial fulfillment of the requirements
for the academic degree

Diplom-Ingenieur

in the Master's Program

Computer Science

Submitted by
Matthias Grimmer, BSc.

At the
Institut für Systemsoftware

Advisor
o.Univ.-Prof. Dipl.-Ing. Dr.Dr.h.c. Hanspeter Mössenböck

Co-advisor
Dipl.-Ing. Lukas Stadler
Dipl.-Ing. Dr. Thomas Würthinger

Linz, November 2013

Abstract

This thesis presents an efficient runtime environment for a C interpreter implemented in Java. This interpreter uses the *Truffle* framework and is called the *Truffle/C VM*. A runtime environment for this VM has to bridge Java code, running in a Java Virtual Machine (JVM), and native code. This is necessary because C code, which runs in the Truffle/C VM (Java), can call any function present as native code (e.g., C standard library functions). It is not possible to use the Truffle/C VM to execute these native functions because their source code is often not available or written in different programming languages. To accomplish the interoperability between the Truffle/C VM and the native functions, this thesis introduces two concepts.

Firstly, it presents an efficient and dynamic approach for calling native functions from within Java. Traditionally, programmers use the *Java Native Interface* (JNI) to call such functions. This thesis introduces a new mechanism, which is tailored specifically towards calling native functions from Java. It is called the *Graal Native Function Interface* (GNFI). It is faster than JNI in all relevant cases and more flexible because it avoids the JNI boiler-plate code.

The second concept describes a memory model for the Truffle/C VM. This memory model offers sharing of run-time data between the interpreter (written in Java) and native code.

The runtime environment of the Truffle/C VM facilitates GNFI to provide an efficient function handling for the VM. It uses the memory model to establish interoperability between interpreted C code and native C code.

This thesis evaluates GNFI and the Truffle/C VM separately. The measurements demonstrate a significant performance advantage of GNFI compared to JNI and the Java Native Access (JNA) for normal Java applications. Also, the evaluation shows that the Truffle/C VM, using the introduced memory model, has a reasonable performance compared to C code, compiled using GCC.

Kurzfassung

Diese Masterarbeit präsentiert eine effiziente Laufzeitumgebung für einen in Java implementierten C Interpreter. Der Interpreter heißt *Truffle/C VM* und verwendet das Truffle Framework. Eine derartige Laufzeitumgebung muss Java Code (welcher auf einer Java Virtuellen Maschine (JVM) läuft) und nativen Code verbinden. Dies ist notwendig, da C Code, welcher in der Truffle/C VM läuft (Java), auch eine beliebige native Funktion aufrufen kann (z.B. eine C Funktion aus der Standard Bibliothek). Diese nativen Funktionen können nicht mit der Truffle/C VM ausgeführt werden, da der Sourcecode oft nicht vorhanden ist oder in einer anderen Programmiersprache entwickelt wurde. Um eine Interoperabilität zwischen der Truffle/C VM und den nativen Funktionen gewährleisten zu können, stellt diese Masterarbeit zwei Konzepte vor.

Zum Beginn präsentiert diese Arbeit einen effizienten und dynamischen Ansatz um native Funktionen von Java aus aufzurufen. Normalerweise verwenden Programmierer das *Java Native Interface* (JNI) um solche Funktionen aufzurufen. Diese Masterarbeit stellt einen neuen Mechanismus vor, welcher speziell für den Zweck, native Funktionen von Java aus aufrufen, entwickelt wurde. Diese Mechanismus nennt sich *Graal Native Function Interface* (GNFI). Er ist schneller als JNI in allen relevanten Fällen und auch flexibler, da JNI boiler-plate Code vermieden werden kann.

Im Anschluss stellt diese Arbeit das Speichermodell der Truffle/C VM vor. Dieses Speichermodell erlaubt den Austausch von Daten zwischen der VM (implementiert in Java) und nativen Code.

Die Laufzeitumgebung der Truffle/C VM verwendet das GNFI um ein effizientes Funktionsmanagement bereitzustellen. Das Funktionsmanagement erlaubt es, verschiedene interpretierte C Funktionen als auch native C Funktionen aufzurufen. Es verwendet das darunterliegende Speichermodell, welches die Interoperabilität zwischen interpretierten C Code und nativen C Code ermöglicht.

Diese Masterarbeit evaluiert GNFI und die Truffle/C VM getrennt voneinander. Die Messungen zeigen, dass GNFI eine signifikant bessere Performanz gegenüber JNI oder dem Java Native Access (JNA) hat. Weiteres zeigt die Evaluierung, dass die Truffle/C VM unter Verwendung des vorgestellten Speichermodells eine beachtliche Performanz hat, verglichen mit C Programmen die mit dem GCC kompiliert wurden.

Contents

Chapter 1

Introduction

This chapter introduces the Truffle/C project, whose goal it is to implement a C interpreter in Java using the Truffle framework, which executes C code on top of a Java Virtual Machine. It also describes the scope of this thesis as part of the Truffle/C VM. It points out the challenges that this project faced and presents the concepts that were developed for this thesis.

1.1 Motivation

On the first impression, the programming languages *Java* and *C* look similar. Both languages are imperative, are statically typed and also share many syntactic elements. A closer look reveals that the apparent similarity is deceiving and that Java and C differ in many aspects.

The first language, Java, is an object-oriented high-level programming language. Its source code gets compiled to *bytecode* that is executed on a Java Virtual Machine (JVM). The intermediate representation of bytecode makes Java platform independent. Furthermore, Java is known to be *type safe* and *memory safe*. Type safety means that the language prohibits discrepancy between different data types for the program's constants, variables and functions. Memory safety is established because Java does not provide raw pointers and performs bounds checks for array accesses. Also, Java does not allow explicit allocation or deallocation of memory. In Java, memory is instantiated by allocating new objects. The JVM uses a *garbage collector*, which frees unreachable objects automatically. The automatic memory management avoids a wide range of memory related problems, like most memory leaks and security problems.

The second language, C, is a low-level programming language and it provides constructs that map efficiently to typical machine code instructions. It allows low-level memory access and provides raw pointers and pointer arithmetic. Memory can be allocated manually and the programmer also has the responsibility to free allocated memory in order to avoid memory leaks.

The goal of the Truffle/C project is to execute C code on top of a JVM. It aims to implement C as a guest language on top of the Truffle framework. Truffle [25] is a framework for implementing managed guest languages in Java. The developer writes an Abstract Syntax Tree (AST) based interpreter for the guest language, which is integrated in the Truffle framework. The overall goal of this project is to execute different C benchmarks on top of Truffle and compare the performance to C code compiled using standard compilers like GCC.

1.2 Goals, Scope and Results

The different aspects of Java and C raise several challenges when implementing an AST interpreter for C in Java:

- **The C language semantic:** The developer has to implement the C language semantic as an AST interpreter. For the Truffle/C project, we use nodes to represent different control structures and operations of the guest language C. The challenge is to model all C language constructs (e.g., loops, if-statements, `GOTO` statements, etc.) in Java by implementing AST nodes.

- **The function handling:** The Truffle/C VM needs facilities to handle any kind of C function call. There are two types of C function calls. A C function executed on top of the Truffle/C VM can call another function available as a Truffle/C function or the caller can call native functions from a native library (e.g., standard library functions). The VM needs an efficient way to perform calls to the different kinds of functions.

- **A fast and flexible native function interface for Java:** The Truffle/C VM needs to be able to call native library functions. For this purpose the Truffle/C project introduces a new approach which is different from other techniques for bridging the Java/native gap like the Java Native Interface (JNI).

- **A memory model:** A memory model for the Truffle/C VM has to provide infrastructure to store and represent run-time data of a C execution. The key challenge of this memory model is to support raw pointers. Another major requirement for this memory model is that it has to be compatible with native C code because pointers can be shared between the Truffle/C VM and native functions.

This thesis focuses on the runtime environment of the Truffle/C VM and therefore on the last three points of the list above. It explicitly omits the process of creating the AST from a C source file. Furthermore, this thesis only describes those AST nodes of the Truffle/C VM which are important to understand the context of the *Truffle/C Runtime Environment*. The interpreter part of the Truffle/C VM is explained in detail in a separate thesis[1].

To evaluate the concepts of the Truffle/C Runtime Environment, this thesis compares the performance of the new native function interface to existing native interfaces for Java, like JNI or the Java Native Access (JNA). Besides the evaluation of this native function interface, this thesis also evaluates the performance of the Truffle/C VM.

1.3 Challenges

The challenge of the Truffle/C Runtime Environment is to bridge Java and native code. To establish such a bridge, we first introduce the *Graal Native Function Interface (GNFI)*, which allows calling native target functions from within Java. Secondly, we introduce a memory model for the Truffle/C VM that allows sharing data between the VM and native code efficiently. Finally, the Truffle/C Runtime Environment facilitates GNFI and the memory model.

1.3.1 Graal Native Function Interface

The Truffle/C VM has to be able to call native libraries because a reimplementation of these libraries in Java would not be feasible or efficient. Even in contexts without the Truffle/C VM, programmers often call native code from Java. This is reasonable, e.g., when highly optimized native libraries exist or when an application is forced to use native legacy code. Calling native code from Java can be seen as a general problem which is not Truffle/C specific. Therefore this thesis describes an implementation

[1] Master's thesis of Manuel Rigger, BSc

that is not only suitable for the Truffle/C VM, but for any Java application. Parts of this native function interface are already published in [13]. The parts of this thesis that describe the native interface are based on this publication.

Programmers typically call native functions using JNI [12]. JNI is a standardized interface, which allows applications to switch execution from Java to a function available as native code and to manipulate Java objects from native code. Programmers can pass arbitrary Java objects or primitives to such a function, which is declared *native* (but is not implemented) on the Java side and which is implemented as a C function or C++ method. This native JNI function has a specific name and signature, which is implied by JNI. Therefore it is only possible to call this special native JNI function from Java.

Whenever the Java application needs to call a native target function, this is done through the indirection of these C or C++ native JNI functions/methods. Such a method whose sole purpose is to call another native function is called a *wrapper*. Within the context of this thesis we define native code or native functions as code that the machine can execute directly. For example, the thesis considers both, the compiled C or C++ JNI wrapper as well as the target function to be native code or native functions. We define the native target function to be the actual function of interest. The native target function can follow any calling convention, such as the C calling convention. Hence the C or C++ implementation, which performs the call to the function of interest, is not a native target function.

While it is common to write JNI wrapper functions to call native target functions, this has several disadvantages:

1. In order to call a native target function at least two calls have to be performed. The first call is from the Java application to the JNI wrapper and the second from the JNI wrapper to the native target function. Theoretically, only one call could suffice, namely from the Java application to the native target function.

2. JNI has additional overhead when setting up parameters. When JNI wrapper functions pass array references to other native functions, JNI does not guarantee to avoid copying the array [16, 18].

3. The call to the native method is opaque to the JIT compiler, meaning that it cannot inline the call. We refer to this as a compilation barrier. Because of the compilation barrier, the JIT compiler cannot optimize the native code of the JNI wrapper function that sets up the parameters.

4. The programmer has to implement, compile and link C/C++ code instead of being able to call the native target function from Java directly.

5. JNI does not allow to dynamically load libraries at runtime or call different native functions that are unknown at compile time.

The first three disadvantages are related to performance issues, while the fourth one is a usability problem. The last problem is an architectural problem of JNI, which implies that JNI is not applicable for the Truffle/C VM. The VM requires to call arbitrary functions at runtime that are not known at compile time as described in Section 4.3.

This thesis describes a native function interface for the Graal VM [21], which avoids these disadvantages. The Graal Native Function Interface (GNFI) exhibits better run-time performance, as well as an improved usability. Furthermore, GNFI allows an application to dynamically load libraries and call functions at runtime.

GNFI avoids the first disadvantage by applying a special inlining mechanism. Usually it is not possible to call native code directly from Java. To circumvent this issue, GNFI produces a Graal intermediate representation (IR) and compiles it to a piece of code that assembles the parameters in the calling convention of the target. This Graal IR can then be inlined by the Graal compiler, just like IR derived from Java bytecode. After inlining, it is possible to remove the additional layer in most cases and directly call the native target function.

The second disadvantage of JNI is the fact that JNI cannot guarantee to get references to Java array data without copying. This is also solved by GNFI. The fact that we use Graal IR to do native calls enables us to pass pointers into the Java heap, without having the risk of possible garbage collections. Java applications, in addition to the Truffle/C VM, will benefit from this GNFI feature.

GNFI avoids the third disadvantage by shifting the compilation barrier down to the call of the native target function. Setting up the parameters is opaque to the compiler in JNI, because it performs this parameter marshaling on the native side, while the GNFI approach performs this step on the Java side. Thus, the JIT compiler can still construct an intermediate representation of this transition and optimize it up to the native target function call. Depending on the use case, GNFI is up to twice as fast as JNI and is an order of magnitude faster than the Java Native Access (JNA).

GNFI avoids the fourth disadvantage by providing a Java interface that can be used to call native functions. The Java interface does not only avoid the need for writing native wrapper functions, it also solves the fifth disadvantage. Following this approach, the programmer can acquire handles to native libraries during runtime and call a native target function by providing its name and signature.

GNFI offers functionality similar to JNI, namely to directly invoke native target functions from within Java, and is specially designed for this purpose. Therefore the goal of this approach is not to replace JNI, but GNFI should rather provide an alternative and faster way to dynamically invoke native target functions.

In terms of security, GNFI raises the same issues as JNI. Both techniques allow the programmer to escape Java's guarantees of safety and security [24]. Since calling native function is an inherently unsafe operation, this thesis does not focus on the issue of security.

1.3.2 Truffle/C Memory Model

Given GNFI, a C program running on top of the Truffle/C VM can switch between a Truffle/C function (on top of the Truffle/C VM) and a native function (e.g., library code). When switching from the Truffle/C VM, a running Java program, to the native code, parameters can be passed from Java to native code. The interface of GNFI allows the VM to easily pass parameters to native functions. Besides passing parameters, the VM can exchange memory allocations of the running C program via pointers. The Truffle/C VM has to share memory allocations (e.g., structures, unions and arrays) between the VM and a native function. The *Truffle/C Memory Model* is responsible for aligning memory (including structures, unions and arrays) so that the VM can exchange it. Native code expects a specific platform dependent alignment and padding of the data. This alignment and padding is the same that a normal C function (compiled using a standard C compiler, e.g., GCC) would apply to the data. Because we cannot change or influence this alignment and padding, all shared data for an execution switch between the Truffle/C VM and the native function has to be aligned as expected by the native code.

A naïve solution for this task would be to copy all shared data into a native memory block before the Truffle/C VM calls a native function. The VM would then have to align the data according to the alignment and padding of the target platform. Java offers facilities for allocating native memory and for storing data into it, using the Java Unsafe API (available in the OpenJDK). This approach, however, would not be efficient because the VM might have to copy a large amount of data.

For the Truffle/C project, this thesis introduces a different approach. The Truffle/C VM itself uses the same alignment and padding for C data structures (structures, unions and arrays) as the native code. This means that the Truffle/C Memory Model does not introduce a new alignment or padding, but instead uses the exact same representation as the native code does. To replicate the data representation of native code for the Truffle/C VM, the VM uses the Java Unsafe API to allocate and deallocate memory for the run-time data on the Native Heap. The Truffle/C Memory Model offers an interface for the Truffle/C VM to manage run-time data, stored in the platform specific representation. Using the

same representation of data structures as the native C code has two advantages: Firstly, the Truffle/C Memory Model can provide valid addresses to the memory locations of the Native Heap, which then allows the VM to pass real pointers to native functions. Secondly, using the same alignment and padding as the native C code avoids copying and preparing data before an execution switches from the Truffle/C VM (Java) to a native function.

1.4 Structure of the Thesis

This thesis aims to separate GNFI from the Truffle/C VM because GNFI i not only suitable for the Truffle/C VM, but is also applicable to any Java function that calls native target functions. Therefore, this thesis presents GNFI as a general concept that any Java application can use. Nevertheless, the main focus of this thesis is on the Truffle/C Runtime Environment and how the VM facilitates GNFI.

Chapter 2 provides an overview of the context of the Truffle/C VM and GNFI. This chapter also introduces the Truffle framework and how the Truffle/C VM is implemented in this context.

Afterwards, Chapter 3 explains GNFI in detail. It illustrates how GNFI works and describes the interface that regular Java applications can use to call native target functions.

This thesis continues with describing the Truffle/C Runtime Environment in Chapter 4. This chapter starts with an overview of all parts of the VM and briefly describes them. After the overview, Section 4.2 introduces the Truffle/C Memory Model. The goal of this memory model is to provide interoperability between the Truffle/C VM and the native C code. Given GNFI and the Truffle/C Memory Model, Section 4.3 combines these concepts and introduces the Truffle/C Function Handling. This chapter describes how the VM does linking and how it is able to call the two different kinds of functions: *Truffle/C functions* and *native C functions*.

After the explanation of all parts of the Truffle/C Runtime Environment, Chapter 5 presents a case study where all concepts are illustrated on a non-trivial C program.

Chapter 6 provides an evaluation of GNFI and the Truffle/C VM using the runtime environment presented in this thesis. Again, the thesis aims to also evaluate GNFI separate from the Truffle/C VM.

This thesis concludes with related work in Chapter 7, future work in Chapter 8 and finally a conclusion in Chapter 9.

Chapter 2

System Overview

This chapter explains the context of GNFI and the Truffle/C VM. For this purpose, Section 2.1 introduces the Graal VM. GNFI is an extension of the Graal VM. Given the underlying VM and its extension, Section 2.2 describes the Truffle framework and explains how the Truffle/C VM is implemented on top of Truffle.

2.1 The Graal VM

The Graal VM, as part of the Graal OpenJDK project [21], is a modification of the Java HotSpot™ VM. The Java HotSpot™ VM has two just-in-time (JIT) compilers, the client compiler and the server compiler. The client compiler aims at fast compilation performance, while the server compiler aims at better optimizations at the expense of compilation time. Both the client compiler and the server compiler are implemented in C++.

The Graal VM extends the HotSpot™ VM by a third compiler, the Graal compiler. It reuses all other parts, including the interpreter, from the HotSpot™ VM. The new compiler, Graal, is written in Java and aims to produce highly optimized code. Figure 2.1 gives a schematic overview of the components of the Graal VM.

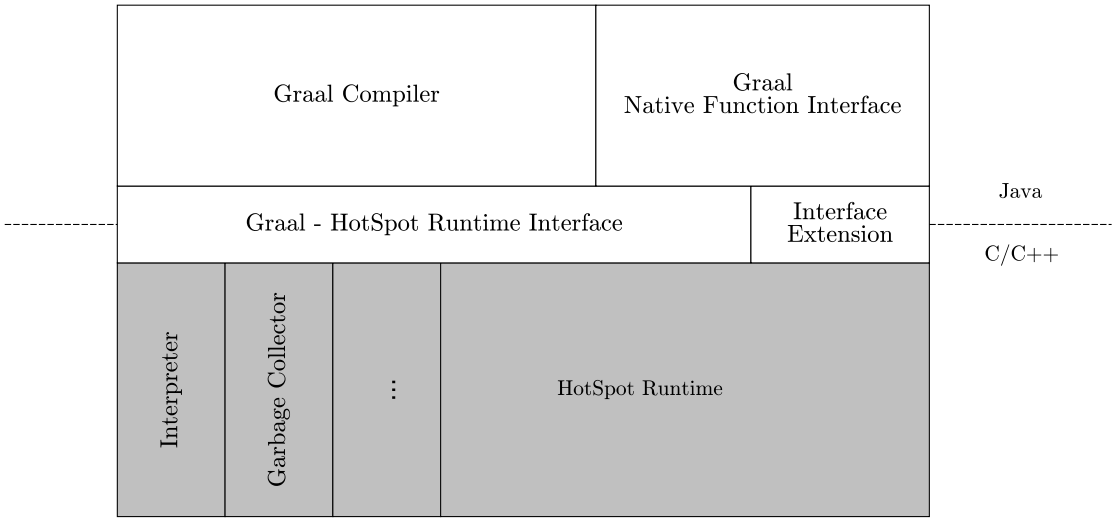


Figure 2.1: System architecture of the Graal VM including GNFI.

Graal parses Java bytecode into the so-called Graal Intermediate Representation (IR) [10, 22]. The Graal IR is a graph that captures the control flow and the data flow of instructions. It gets compiled to executable machine code by the Graal compiler. During compilation, the Graal compiler performs optimizations on this IR and installs the resulting machine code using its interface to the HotSpot™ VM. Besides code installation, this interface also allows the invocation of an installed method by providing a reference to it. Hence Graal can call any compiled graph.

The Graal VM allows the installation of so-called *method intrinsics*. A method intrinsic is a pre-built graph that replaces a Java method's implementation.

We implemented GNFI in the context of the Graal VM. The modifications necessary to implement the interface for the Graal VM are small: Firstly, GNFI extends the Graal IR with new nodes to execute calls to native functions. GNFI also adds a new capability to the public Graal API, namely the *Native Function Interface*. This provides a user-friendly interface for programmers.

To improve the interpreter performance of GNFI, we extend the interface between the Graal compiler and the HotSpot™ VM as described in Section 3.3.

As Figure 2.1 shows, GNFI extends the Graal VM by a new component (Graal Native Function Interface) and also extends the interface between the Graal compiler and the HotSpot™ VM (Interface Extension). GNFI uses the Graal compiler to synthesize and compile Graal IR to machine code. To

install and invoke the machine code, GNFI uses the existing interface between the Graal compiler and the HotSpot™ VM. GNFI also introduces an extension of this interface, whose purpose is to speed up the interpreter performance of GNFI.

2.2 Truffle

The idea behind Truffle [27] is to establish a layered architecture for guest language VMs. VMs are mostly monolithic pieces of software, written in one language (mostly C/C++) and executing another language. VMs offer many benefits for their guest language, but in fact do not utilize these benefits for themselves [27]. The approach of Truffle aims to utilize the benefits of existing VMs. Würthinger et. al describe in [27] how a *Guest VM* (Truffle/C VM) runs on top of a *Host VM* (Graal VM). The Host VM (Graal VM) is written in Java and C++ and executes the language the Guest VM is written in (Java in the case of Graal VM). The Guest VM (Truffle/C VM) is written in Java and executes the code of the guest language (C language). Würthinger et. al mention several advantages of this layered approach that makes implementing a new Guest VM easy. Guest VMs can benefit from existing automatic memory management, exception handling, threads and synchronization primitives as well as an existing well-defined memory model [27]. Figure 2.2 illustrates the layering of Guest VM and Host VM.

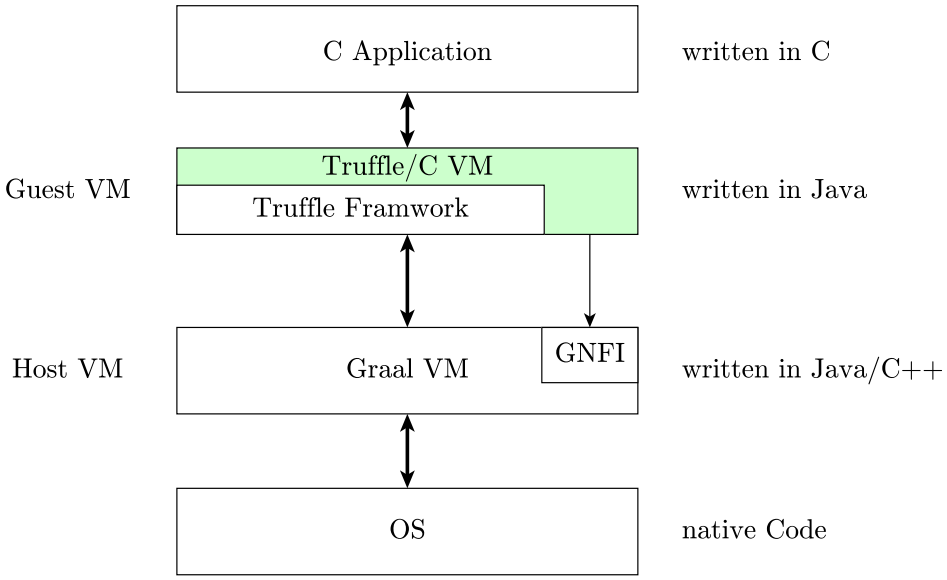


Figure 2.2: System architecture of the Truffle/C VM [27].

Truffle, the Guest VM, is an Abstract Syntax Tree (AST) based framework for implementing guest languages in Java. An AST interpreter models constructs of the guest language as nodes, where each node represents the semantics of one construct of the guest language. These AST nodes build a tree that represents the program to be interpreted. Each node that is part of the AST has an `execute` method. The `execute` method itself implements the semantics of the corresponding language construct. The language implementer's task is to design these AST nodes and provide facilities to parse the guest language code and build the tree.

AST-based interpreters are often considered to be slow in terms of performance. To overcome the performance issue, Truffle offers 2 different types of optimizations: *Self-Optimization* and *Partial Evaluation*.

The following sections describe these types of optimizations.

2.2.1 Self-Optimization

The Truffle framework offers facilities for the AST to do self-optimization. Self-optimization in this context means that the AST can change its structure to gain better run-time performance. The following list describes the parts of Truffle that enable the self-optimization of ASTs [26]:

- **Tree Rewriting:** Tree rewriting describes the possibility for a node to replace itself and its parent with a different node. This rewriting allows the AST interpreter to automatically incorporate profiling feedback while executing. The possibility to rewrite the AST combined with collecting profiling information makes the following self-optimizations possible [26]:

  - **Type Specialization:** For dynamic languages, where there is no type information for variables and operations at compile time, type specialization can rewrite nodes of the AST to a specific type and therefore improve performance. The implementation of the Truffle/C VM will only benefit from type specialization in rare cases because C is a statically typed language where all types are known at compile time.

  - **Polymorphic Inline Caches:** Truffle supports polymorphic inline caches [8]. They work by chaining nodes that represent the entries in the cache. A new entry in this cache always adds a node to the tree. The nodes propagate the operation down the chain until it reaches a node that can execute the operation. This adding of new nodes and entries to the cache

continues until it reaches a certain length. Once it reaches this length, the chain replaces itself with the megamorphic case [26]. As the future work points out, the Truffle/C VM can use this optimization for function pointers.

– **Resolving Operations:** Operations of a guest language might include a resolving step (e.g., function resolving). This resolving can be done by using the tree rewriting facility. Resolving and rewriting nodes comes with the advantage that a rewrite avoids subsequent checks. The Truffle/C VM makes use of this resolving technique to resolve functions.

- **Boxing Elimination:** The standard `execute` method of a node always returns an `Object` value [26]. To avoid the boxing and unboxing of Java, which would be unavoidable when only a generic `execute` method exists, Truffle enables the programmer to define differently typed `execute` methods for nodes. For example, if it is known that a node always returns an `int` value, the node can implement an `executeInt` method. These specialized methods reduce boxing. This technique offers the possibility to remove the boxing of primitive variables for the statically typed language C. In C, types are always known at compile time, therefore it is always possible to derive a type-specific `execute` method.

- **Tree Cloning:** As the Truffle AST aims to collect run-time feedback during execution, different callers can pollute this feedback. Especially for dynamically typed languages this pollution has a big impact on the quality of the profiling information. To avoid this pollution, Truffle is able to clone the AST to create caller specific versions.

2.2.2 Partial Evaluation

The dynamic dispatch between nodes causes a significant loss of performance [26]. Truffle provides *partial evaluation* to remove this overhead and further optimize the Truffle ASTs. Truffle partially evaluates stabilized hot functions, represented as ASTs. The term *stabilized* means that there are no longer rewrites in the AST and the tree can be seen as stable; Truffle assumes that the tree remains constant. A function being *hot* means that the number of executions of the function has exceeded a certain threshold.

These assumptions enable the Truffle framework to convert the virtual dispatch between the `execute` nodes to direct calls. Truffle uses the Graal VM (see Figure 2.2) and its compiler to perform the optimizations as part of the partial evaluation. The partial evaluation first starts with inlining all these direct calls, forming one combined unit of compilation for a whole tree [26]. Having one combined unit allows the Graal compiler to do a wide variety of optimizations.

As Truffle speculates that the AST is constant, the compiler graph of the partially evaluated AST has a deoptimizaion point in all branches where a rewrite would happen. These deoptimization points invalidate the compiled code and continue executing the tree in interpreter mode.

As local variable access is also critical for guest languages [26], Truffle introduces the *Frame*. A guest language reads and writes local variables from and to a Frame. The Frame is represented by an array and contains all local variables. During partial evaluation, Truffle forces an *escape analysis* [15] of this array. This technique enables Truffle to remove all access to the array and to connect the read operation of the variables with the last write operation [26].

The Truffle/C VM is built on top of Truffle (see Figure 2.2). The Truffle/C nodes extend the given `abstract Node` of Truffle and model the semantics of C. Because the VM needs the possibility to access native libraries (e.g., standard library), it also directly uses the Graal extension GNFI, as shown in Figure 2.2.

Chapter 3

Graal Native Function Interface

This chapter explains the Graal Native Function Interface in detail. In Section 3.1 this chapter shows the public interface of GNFI and how to use it. After the description of the public interface, Section 3.2 describes the call stubs, which perform the conversion of calling conventions. Sections 3.3 and 3.4 describe how GNFI efficiently performs native calls from interpreted and from compiled Java methods.

3.1 Public Interface and Execution Modes

Developers using Graal can easily use the *Graal Capabilities* interface [20] to load GNFI and access native target functions. This section compares the usage of GNFI with JNI and points out the advantages of GNFI. It introduces two different modes for calling native target functions via GNFI. These two modes are native calls from *interpreted* Java code and *compiled* Java code.

3.1.1 Public Interface

This section shows an example that demonstrates how a user can acquire and use the `NativeFunctionInterface` capability. The goal of the example is to invoke the `floor` function of a math library written in C. Listing 3.1 shows the signature of the `floor` function.

```
1 double floor(double value);
```

Listing 3.1: Signature of an C function `floor`.

The usage of the `NativeFunctionInterface` follows a simple scheme, as shown in Listing 3.2. First, one has to load the capabilities for the `NativeFunctionInterface`, assuming the Graal VM supports it. The second statement uses the capabilities to load a math C library containing the native target function `floor`. Resolving the library results in a library handle, which can be used to obtain a handle to the native target function, as shown in the third statement. To do so, the user has to provide the library handle, the name of the native target function and the signature. After these three steps, one can invoke the native target function, referenced by the `NativeFunctionHandle`.

```
1 NativeFunctionInterface ffi =
2   Graal.getRequiredCapability(NativeFunctionInterface.class);
3 NativeLibraryHandle libraryHandle = ffi.getLibraryHandle("libMyMath.so");
4 NativeFunctionHandle functionHandle =
5   ffi.getFunctionHandle(libraryHandle, "floor", double.class, double.class);
```

Listing 3.2: Obtaining a `NativeFunctionHandle`.

To invoke the native target function, the `NativeFunctionHandle` provides a method (`call`), which performs the native call.

In order to perform this call, GNFI has to convert the call on the `NativeFunctionHandle`, which is a Java method invocation, to a native call. Thus, the `NativeFunctionHandle` has to convert the calling convention from Java to C. For this calling convention conversion, GNFI defines the *GNFI Java calling convention*, which all calls on `NativeFunctionHandles` have to follow. This calling convention supports all parameter types that the native language requires.

To use the GNFI Java calling convention it is necessary to encode all native types using boxed Java primitive values. In Java, primitive types are not inherited from `Object`. In order to treat them as objects, one has to use wrapper classes. For this thesis I refer to instantiating such a wrapper class for a primitive type as *boxing*. For example, Java provides a class `Integer`, which holds a value of the primitive Java type `int`.

The signature of the call method of a `NativeFunctionHandle` takes an `Object` array as a parameter. The values of this array are supposed to be boxed Java primitive values or one-dimensional Java arrays. Primitive native arguments take one slot in the array. Complex native arguments can take multiple slots. This thesis refers to all native types that are atomic and can be encoded in a single Java primitive type as *primitive native types*. *Complex native types* are native types that do not fit into a Java primitive type or are composite types. Complex native types are represented as a series of boxed Java primitive values.

Native calling conventions, such as the C calling convention, can require arguments to be passed by value or by reference. To pass arguments to a native target function by reference, it is necessary to know the memory address of the parameters. In Java this can be achieved by copying the data into a memory block according to the data layout of the native target language. The Java Unsafe API allows the allocation of memory on the Native Heap as well as loading data from it and storing data into it. The address of the allocated memory block can be stored into a Java `Long` value and passed to the native target function.

An alternative is to pass the address of the data in the Java heap to the native target function without copying it. This can only be applied for one-dimensional primitive Java arrays because their Java memory layout is the same as in C. The user can directly pass a Java array of a primitive type to the native target function. The requirement to pass Java array objects without copying justifies the fact that the GNFI Java calling convention uses an object array for its arguments and boxes primitive types. To perform the call, GNFI takes the reference to the Java array object and determines the base address of the array data inside it. In Section 3.2 we explain why garbage collection is not a problem here. Depending on the application and purpose, the user can decide how to pass arrays by reference.

Native languages like C can also return complex types (e.g., structs) by value. In this case, the GNFI Java calling convention expects an additional parameter at position zero of the Java `Object` array. This parameter represents a pointer to a memory cell (encoded as a Java `Long` value) into which GNFI places the return value.

Listing 3.3 shows an example of how to use the GNFI Java calling convention. This example calls the native target function `floor`. This call passes a `double` value to a native C function. The code allocates an `Object` array with one slot, into which it puts a Java `Double` value and passes the array to the native C function. The return value of the `call` is a Java `long` value, which encodes a `double` value. The method `longBitsToDouble` can be used to convert the `long` value into a `double` value.

```
1 Object[] arg = new Object[1];
2 arg[0] =  new Double(1.5);
3 double result = Double.longBitsToDouble(functionHandle.call(arg));
```

Listing 3.3: Using the `NativeFunctionHandle` to call a native target function.

To accomplish the same with JNI, the programmer first has to declare a method as *native* on the Java side. The following considers the same example as before, namely calling the native target function `floor` of Listing 3.1.

```
1  class MyJavaMath {
2    static {
3      System.loadLibrary("MyJavaMathWrapper");
4    }
5
6    public static native double floorWrapper(double value);
7
8    public static void main(String[] args) {
9      double result = floorWrapper(1.5);
10   }
11 }
```

Listing 3.4: Calling `floor` using JNI (Java side).

Listing 3.4 shows a class `MyJavaMath`, which declares a method `floorWrapper` as native and loads the shared library `libMyJavaMathWrapper.so` (derived from `MyJavaMathWrapper` using a naming convention). This library contains the JNI wrapper function that calls the native target function `floor`. Listing 3.5 shows the source code of `libMyJavaMathWrapper.so`. To implement this wrapper library, the programmer first has to implement a function (`Java_MyJavaMath_floorWrapper`) according to an auto-generated header file `MyJavaMath.h` (included in the source file of `libMyJavaMathWrapper.so`, Listing 3.5).

```
1 #include <jni.h>
2 #include "MyJavaMath.h"
3
4 double floor(double);
5
6 JNIEXPORT jdouble JNICALL Java_MyJavaMath_floorWrapper
7   (JNIEnv * env, jclass obj, jdouble value) {
8     return floor(value);
9   }
```

Listing 3.5: Calling `floor` using JNI (C side).

The signature of the function `Java_MyJavaMath_floorWrapper` in Listing 3.5 is derived from the native Java method `floorWrapper`. A call of the Java method `floorWrapper` invokes `Java_MyJavaMath_floorWrapper`. When compiling and linking `libMyJavaMathWrapper.so` one has to link it against the library `libMyMath.so`, which contains the C function `floor`.

This example demonstrates that JNI and GNFI follow different approaches for handling parameters. GNFI abstains from implicit parameter conversions (i.e., from Java to C/C++) and requires the user to do the parameter preparation explicitly on the Java side. This removes the need for native JNI wrappers. However, it requires that the programmer aligns the data as expected by the native code. For example, the programmer is responsible for providing a memory block that represents a C structure before using a pointer to this structure as an argument. This arguments preparation on the Java side provides several advantages:

The first advantage in terms of performance is that it widens the compilation span of the JIT compiler. A JIT compiler, like Graal, benefits from this because the only compilation barrier is the call to the native target function itself. By bundling the calling convention-specific code in a Java library, developers can use the interface without having to care about native code specifics. Developers will benefit from all advantages of GNFI plus the advantages of Java compiler optimizations such as method inlining.

The second advantage is that users do not have to maintain, compile and link C/C++ wrapper code when calling new native target functions. GNFI only requires the modification of Java code.

The third advantage, which is related to the Truffle/C VM, is that GNFI improves flexibility. This means that it is possible to load libraries and resolve function handles at runtime. A further advantage is that the VM also uses the same data padding and alignment as native C code and therefore we can directly exchange pointers between the VM and a native execution.

3.1.2 Modes of Execution

To convert the GNFI Java calling convention to a native calling convention it is necessary to distinguish between two modes of the caller: the caller can either run in *interpreted mode* or in *compiled mode*. When it runs in interpreted mode GNFI has to bridge two gaps on different levels (see Figure 3.1).

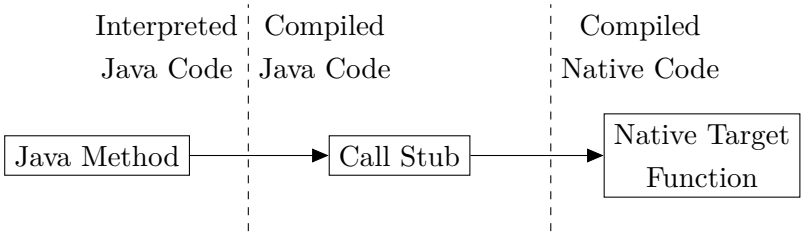


Figure 3.1: Layer gaps between the interpreted Java code and the native target function.

The first gap, on the left-hand side of Figure 3.1, is the call from interpreted Java code to a call stub, represented as installed code. For this thesis we define a *call stub* to be an installed machine code method that converts the GNFI Java calling convention to the native calling convention and performs the actual call to the native target function. The call stub bridges the second gap on the right-hand side of Figure 3.1, and is described in more detail in Section 3.2. In order to call the call stub's machine code from interpreted Java code, GNFI forwards the call to the HotSpot™ VM. The HotSpot™ VM bridges interpreted and compiled Java code and invokes the call stub code. The `NativeFunctionHandle` provides an interface for this installed call stub and uses the HotSpot™ VM facilities, which is hidden from the users. Finally, one uses the `NativeFunctionHandle` to call the native target function following the GNFI Java calling convention.

If the calling Java method runs in compiled mode GNFI can directly invoke the call stub because there is no need to bridge the gap between interpreted and compiled Java code.

3.2 The Callstub

Whenever GNFI resolves a function handle, it is necessary to create a call stub. This call stub converts from the GNFI Java calling convention to the native calling convention (e.g., the C calling convention), which requires the allocation of registers and the reservation of stack space according to the native calling convention.

Only machine code can do this allocation, hence it is not possible to implement the call stub method in Java. To overcome this issue, GNFI creates a Graal IR graph [10, 22] at run time and makes it the implementation of the call stub method. The signature of this Java method has an `Object` array as its argument and returns a `long` value. To call such a stub method, one has to follow the GNFI Java calling convention as described in Section 3.1.

GNFI creates machine code by first synthesizing an IR graph and then compiling it into machine code using the Graal VM's facilities. The IR graph of the call stub contains nodes that load the parameters from the supplied `Object` array. Then it unboxes these parameters or resolves their addresses (array parameters). Finally, the call stub contains a special IR node that represents the native target function call.

Since GNFI creates the IR for the call stubs, it is possible to decide at which point garbage collection (GC) can happen in the Java VM. GCs can only happen when all threads running on the Java VM have reached a safepoint, i.e., a position where the Java VM can safely stop them before a GC is initiated.

Neither the graph of the call stub nor the code of a native target function contain a safepoint at which a GC can happen. Hence passing Java object references to native target functions is safe because the VM cannot do a GC while the native target function is executed. The fact that other Java threads have to wait for the native call to return when a GC is pending might be seen as a restriction, but it also comes with the advantage that native target functions can safely access Java objects. If the blocking of the GC during the execution of a native target function is not acceptable, one has to resort to JNI.

Figure 3.2 shows the Graal graph of the call stub of our C function `floor`. In this graph, thin edges denote the data flow, while thick edges denote the control flow. The `arg` node is the parameter of this `callstub` method. It is the reference to the `Object` array that contains the parameters for the `floor` function. The *LoadIndexed* loads the first element of the argument array. The *Load* node loads the `double` value from the `Double` object and passes the value to the native call node. Beside the argument, the native call node also has a pointer to the function `floor` as an input.

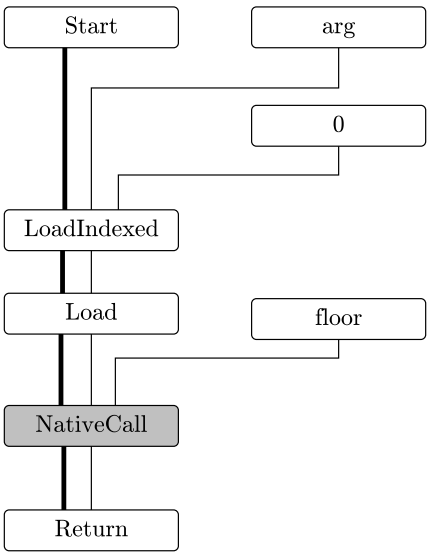


Figure 3.2: IR graph of the call stub for the native target function `floor`.

The native call node is compiled to machine code, which performs the platform-specific call to the native target function. The machine code stores the parameters into the right registers and stack slots, according to the native calling convention, and finally calls the native target function. The implementation, described in this thesis and used for the Truffle/C VM, produces machine code according to the AMD64 Application Binary Interface (ABI). Of course this ABI can easily be exchanged for different platforms.

To install the machine code of this call stub in the VM, GNFI uses the interface between the Graal compiler and the HotSpot™ VM. Every subsequent call to a `NativeFunctionHandle` will invoke this installed machine code.

The call stub only depends on the signature of the native target function. The programmer has to specify this signature. If the native target function has a fixed signature, i.e., it has a fixed number of argument types and not a variable number like the C function `printf`, the signature can be seen as a compile-time constant. Therefore, the call stub and its IR graph are also compile-time constants and it is possible to create them statically. As most target functions have a fixed signature, their call stubs are compile-time constants. A constant call stub has the advantage that GNFI can inline the call stub's IR graph into the Java application. Inlining reduces the call overhead to a minimum and creates opportunities for further optimizations. Section 3.4 explains how call stubs are inlined.

3.3 Native Function Calls in Interpreted Mode

When an interpreted part of a Java application calls a native target function, there are two gaps that GNFI has to bridge. As Figure 3.1 shows, the first gap is between the interpreted Java application and the compiled Java call stub. The second gap is between the Java calling convention and the native calling convention, which Section 3.2 describes. This section describes the first gap and how GNFI accesses compiled Java code from interpreted Java code.

Whenever the user of GNFI executes a `NativeFunctionHandle` by using the GNFI Java calling convention and the caller runs in interpreted mode, GNFI has to invoke the call stub. The call stub is represented by an installed code. To invoke it, GNFI uses the interface between Graal and the HotSpot™ VM. This interface offers a method that tells the HotSpot™ VM to execute an installed code, which in our case is the call stub.

To speed up communication between Graal and the HotSpot™ VM, GNFI extends the interface by an additional native-declared execute method (Java side of the interface). It has a machine code implementation on the HotSpot™ VM side (native side of the interface) that converts the Java interpreter calling convention to the Java compiled calling convention before invoking the actual call stub.

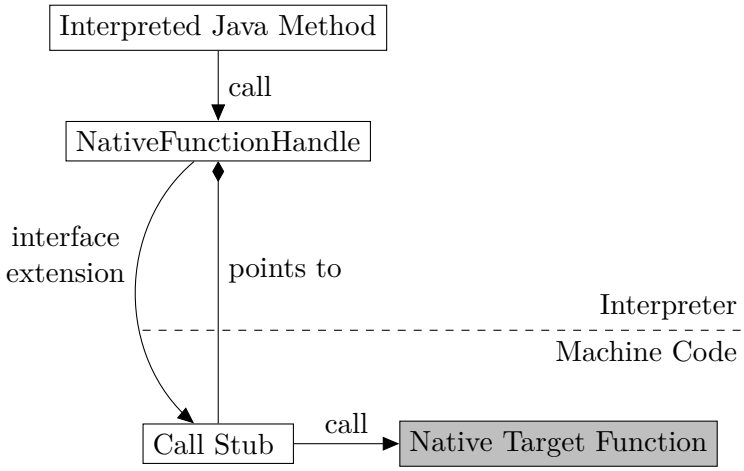


Figure 3.3: Execution in interpreter mode.

Figure 3.3 shows an interpreted Java method that calls a `NativeFunctionHandle`. The `NativeFunctionHandle` refers to the installed call stub. As described, GNFI invokes the call stub by using the interface extension of the HotSpot™ VM. This extension bridges from interpreted Java code to compiled Java code. The call stub finally performs the call to the native target function.

3.4 Native Function Calls in Compiled Mode

A `NativeFunctionHandle` can also be called from compiled code. When the Graal VM compiles the caller method it is possible to make use of Graal *intrinsifications*. Instead of invoking the call stub by using the Graal to HotSpot™ VM bridge (from interpreted Java code to compiled Java code; see Section 3.3), GNFI intrinsifies the call method of the `NativeFunctionHandle` object.

This intrinsification inserts a special node (*FunctionHandleIntrinsification*) into the IR of the caller method. The *FunctionHandleIntrinsification* node replaces the call of the `NativeFunctionHandle`. To illustrate this intrinsification this section again uses the native C function `floor`. Listing 3.6 introduces a new example where it adds 1.0 to the result of the native call to the `floor` function. The argument for this call is the value 1.5. To prepare the arguments array `arg` the example boxes the value 1.5 and stores it at position zero of the array.

```
1 Object[] arg = new Object[1];
2 arg[0] = new Double(1.5);
3 double c = 1.0 + Double.longBitsToDouble(functionHandle.call(arg));
```

Listing 3.6: Addition example using the native function `floor`.

Figure 3.4 shows the IR graph of this example. As we can see, the intrinsification of the `NativeFunctionHandle` replaces the call on its `call` method with a *FunctionHandleIntrinsification* node. The *StoreIndexed* node prepares the arguments by storing the boxed value 1.5 at position zero of the array.

The key observation in this graph is that it no longer contains the call of a `NativeFunctionHandle`; the *FunctionHandleIntrinsification* node replaces the call. Finally, the graph shows the addition of the result of `floor` and the value 1.0.

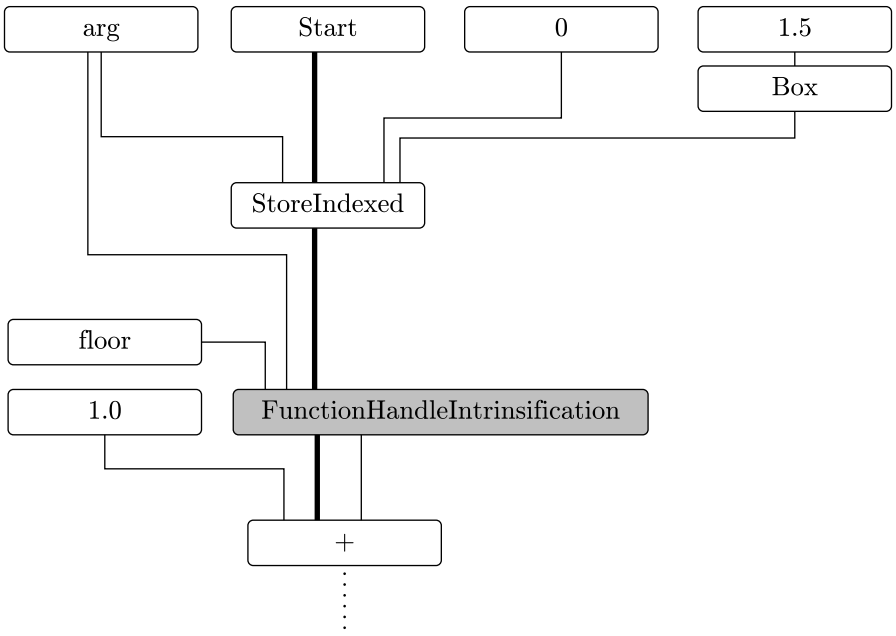


Figure 3.4: IR graph for Listing 3.6 containing a *FunctionHandleIntrinsification* node.

The *FunctionHandleIntrinsification* node exhibits two different behaviors, depending on the call stub it represents. Call stubs for variadic functions are run-time variables, because the number of arguments (and therefore the call stub) can vary. To call a variadic native target function it is necessary to invoke the call stub, resulting in two calls — one from the compiled code to the call stub and one from the call

stub to the native target method. Call stubs for non-variadic functions are compile-time constants, because the call stub does not change. If the call stub is a compile-time constant, it is possible to reduce the number of calls to one.

In the first case, where the call stub is a run-time variable, the *FunctionHandleIntrinsification* node calls the installed code that represents the call stub. This means that the node replaces itself with a direct call to the installed call stub. For this call it is not necessary to convert the arguments from the Java interpreter calling convention to the Java compiled calling convention. Instead, all arguments are already in place. Therefore, it is possible to directly perform the call to the call stub.

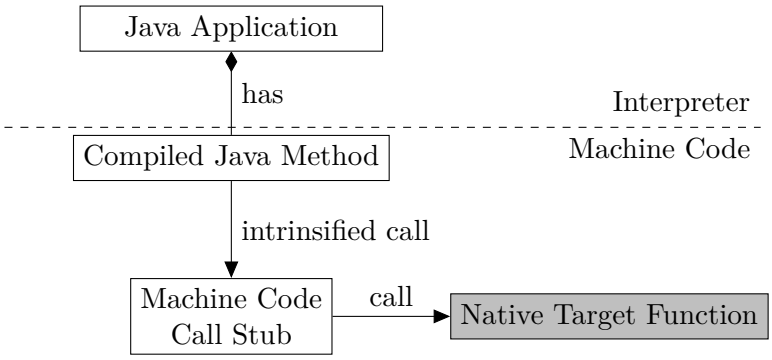


Figure 3.5: Execution in compiled mode: call to the call stub.

Figure 3.5 shows the execution of a compiled Java method using GNFI where the call stub is a run-time variable. The compiled Java method does not contain code of the `NativeFunctionHandle` anymore. Instead it does a direct call to the call stub.

Figure 3.6 shows the resulting graph for the example of Listing 3.6. To illustrate the first case this example treats the call stub of the native target function `floor` as if it was a run-time variable and calls it. In Figure 3.6 a call replaces the *FunctionHandleIntrinsification* node. The other parts of the graph (the call stub and the graph of Listing 3.6) remain unchanged.

In the second case, where the call stub and its graph are compile-time constants, we can directly inline this graph. Instead of a call to the installed code of the call stub, the *FunctionHandleIntrinsification* node replaces itself with the IR graph of the call stub. After compilation, the machine code only has one call site left, namely the site which directly calls the native target function. Thus, inlining the IR graph of the call stub enables the Graal compiler to do optimizations on the whole range up to the native call itself.

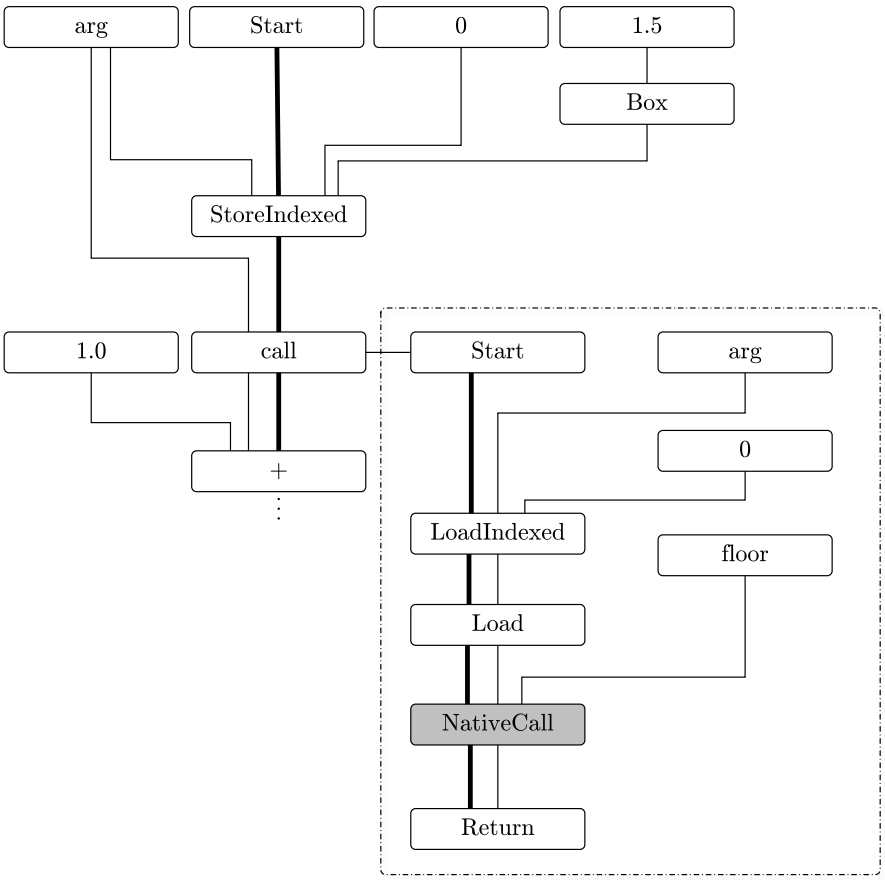


Figure 3.6: IR graph for Listing 3.6; the intrinsified function handle node was replaced by a call to the call stub.

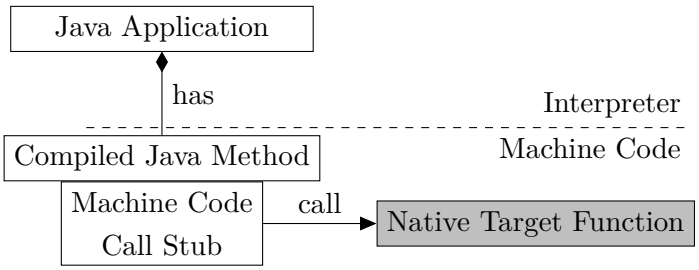


Figure 3.7: Execution in compiled mode: inlined call stub.

Figure 3.7 shows the execution of a compiled Java method using GNFI where the call stub is a compile-time constant that could be inlined. The call site from the compiled Java method to the call stub is gone. The only call left is to the native target function.

In the example of Listing 3.6 the stub can be inlined because `floor` has a static signature. Figure 3.8 shows the graph that results from replacing the *FunctionHandleIntrinsification* node by the actual call stub graph. In this graph all call sites, except the native call to the `floor` function, are gone. After inlining the call stub, the Graal compiler is able to apply its optimizations up to the point where the call to the native function appears.

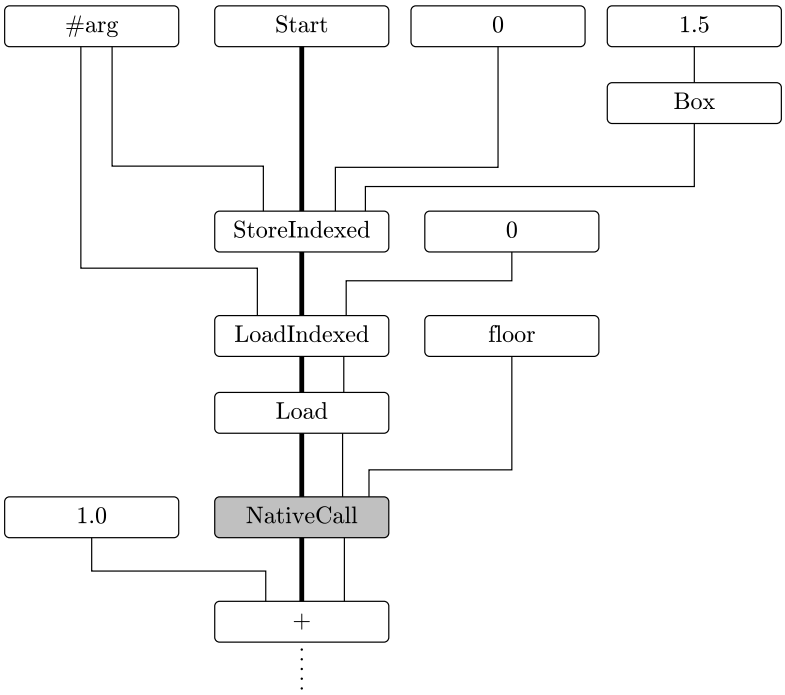


Figure 3.8: IR graph for Listing 3.6 with the call stub inlined.

Removing all extra calls, and therefore widening the compilation span, is the biggest benefit of GNFI. If Graal optimizes the graphs from Figure 3.8, it can remove the `Object` array for the arguments. It can also remove the boxing of the primitive value 1.5. Figure 3.9 shows the final graph after all optimizations. All argument manipulations could be eliminated and only the native call itself remains.

One might argue that it is unlikely that an installed code is a compile-time constant. However, for call stubs of native target functions, the installed code is nearly always a compile-time constant.

The call stub only depends on the signature of the native target function. If the signature is known at compile time, which is the case for most native target functions, the call stub graph can indeed be seen as a compile-time constant. The only case where a call stub is not a compile-time constant is when the signature contains a variable number of parameters. In this case GNFI has to create separate call

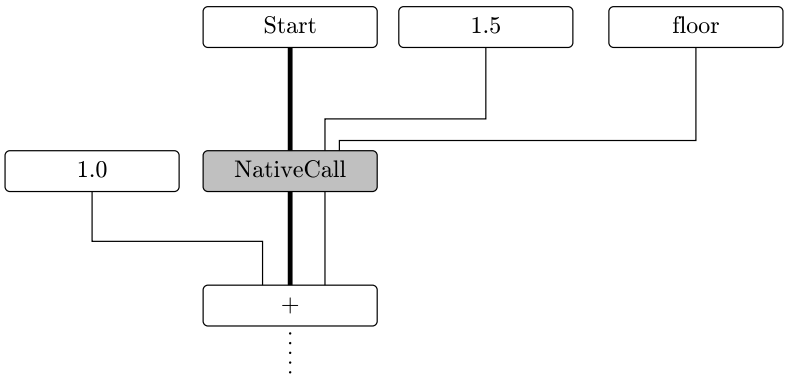


Figure 3.9: IR graph for Listing 3.6 after optimizations.

stubs for different calls at runtime. In all cases with respect to the Truffle/C VM, the callstub is a compile-time constant. This is due to the fact that the VM resolves a function handle for each call site. A call site knows how many parameters are passed, therefore call stubs are always compile-time constants.

The main advantage of our approach compared to other approaches such as JNI or JNA is that GNFI can remove all extra call sites, enabling the JIT compiler to optimize the argument conversion code.

Chapter 4

Truffle/C VM

The first part of this chapter provides an overview of all pieces of the Truffle/C VM that are necessary to execute C code on top of the VM. It describes all steps necessary to compile and execute a C file using a standard compiler like GCC, and then compares how the Truffle/C VM performs these steps. It also gives an overview of the Truffle/C nodes.

The rest of this chapter then introduces the two main parts of the Truffle/C Runtime Environment. Section 4.2 explains the Truffle/C Memory Model. It explains how the Truffle/C Memory Model distinguishes between two different storage locations of run-time data, which are the Frame and the Native Heap. Furthermore, this chapter explains how the Truffle/C Memory Model shares run-time data between the Truffle/C VM and native code. Section 4.3 explains the Truffle/C Function Handling. It describes how the Truffle/C VM establishes linking at runtime by node rewriting. Besides linking, this section shows the differences between Truffle/C calls and native calls and explains the calling procedure of each call type.

4.1 Overview

Compiling a program written in C involves two main steps: First, the compiler compiles the C files to object files (a relocatable machine code representation). Second, the linker generates an executable program out of the linked object files and libraries.

The first task of this procedure, compiling C code to object files, can be split into several steps. First, the C compiler performs *preprocessing*. This preprocessing step is responsible for including header files, expanding macros, handling conditional compilation and doing line control. After preprocessing, the

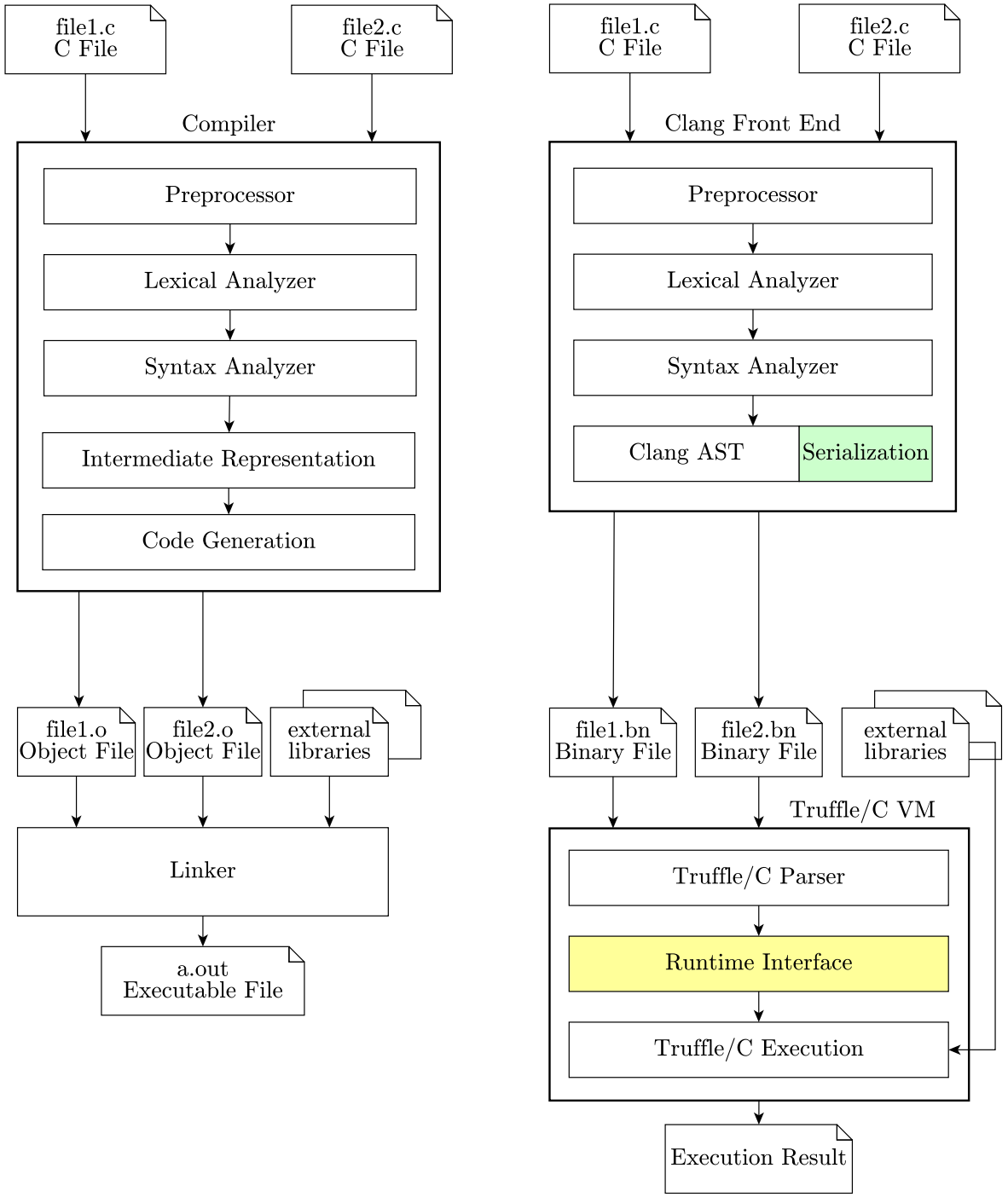
compiler performs *lexical analysis*, which converts a sequence of characters into a sequence of tokens. *Syntactic analysis* parses these tokens according to the rules of a formal grammar (C grammar). The result of syntactic analysis or parsing is a so-called *parse tree*, which can be transformed into an *intermediate representation* (IR). The IR is an abstracted representation of the source code, which the compiler uses to optimize the code. The compiler finally compiles the optimized IR to machine code, stored in an object file.

The second major task, linking, combines given object files and libraries by relocating their data and patching symbol references (e.g., patching function call target addresses). Figure 4.1(a) illustrates the process of compiling and linking C files to produce an executable file.

4.1.1 Creating the Truffle/C AST

The compilation chain of a C program is complex. Reimplementing these steps for the Truffle/C VM would not be efficient or feasible. Instead of reimplementing the compilation chain for the Truffle/C project we modify *clang* [1], a compiler front end for C/C++. Clang comes with a preprocessor, performs lexical and syntactic analysis and has an Abstract Syntax Tree (AST) as an intermediate representation for functions. It also provides a symbol table. For the Truffle/C VM we extend clang and serialize the ASTs of all functions of a C file, including the symbol table. The serialization writes *Truffle/C binaries* (`*.bn`). These Truffle/C binaries replace the object files in our implementation. After creating the Truffle/C binaries, the Truffle/C VM uses them as input. The VM has a parser that reads these binary files and creates the Truffle/C ASTs.

One big difference between the GCC compilation queue and the Truffle/C VM is that the VM does not have an explicit linking step. The VM resolves the targets of the symbolic references, like variables and functions, at runtime and links them on demand. Instead of patching addresses, the Truffle/C VM rewrites nodes after resolving the target (see Section 2.2.1 and Section 4.3). The Truffle/C VM can access all objects, stored in different binary files, at runtime by using the *Truffle/C Runtime Interface* (see Figure 4.1(b)). The Truffle/C Runtime Interface allows resolving objects by their symbolic reference. It looks for the requested object in all binary files and returns an AST representation of it (assuming such an object exists). Figure 4.1(b) shows the clang extension for serialization and the schematic overview of the Truffle/C VM as described in this section.



(a) Components that compile and link a C program.

(b) Components of the Truffle/C VM to compile and execute a C program.

Figure 4.1: Architectural overview of the components of a normal C compiler and of the Truffle/C VM that compile and link a C program.

4.1.2 Truffle/C Nodes

The Truffle/C VM is an AST interpreter and thus represents C functions as ASTs. The nodes of the Truffle/C ASTs model the semantics of the target language. The following list summarizes the most important nodes that are necessary to understand the Truffle/C Runtime Environment.

- **Variables:** The Truffle/C VM distinguishes between two different places where it stores run-time data (Frame and Native Heap, see Section 4.2). *FrameSlot* nodes represent variables that the VM stores in the Frame. *MemoryAddress* nodes represent the addresses of variables, located on the Native Heap.

- **Frame Read/Write:** The Truffle framework offers the Frame (see Section 2.2), where an execution can store run-time data. To access the Frame, the Truffle/C VM uses *FrameRead* and *FrameWrite* nodes, which in turn reference the aforementioned *FrameSlot* nodes.

- **Memory Read/Write:** Besides the Frame, the Truffle/C VM also keeps data on the Native Heap. The VM uses *MemoryRead* and *MemoryWrite* nodes (to read from and write to the Native Heap), which in turn have one child that can be either a *MemoryAddress* node or a computed address value.

- **Function Calls:** The Truffle/C VM represents a function call as a *FunctionCall* node. The *FunctionCall* node has a variable number of children, which are the arguments. An argument itself can be an arbitrary subtree that evaluates one argument of the call. Besides the arguments, the *FunctionCall* node also knows the root node of the target function. After evaluating the arguments, the call node calls this root node (see Section 4.3).

4.2 Truffle/C Memory Model

The Truffle/C Runtime Environment has to provide an underlying memory model for the VM that handles all kinds of memory allocations. The Truffle/C VM needs to allocate memory whenever the executed program instantiates a primitive variable, array, structure or union. The Truffle/C Memory Model manages this run-time data. The Truffle/C Memory Model has to solve the following issues:

- **Pointers:** A C application can retrieve the address of variables (e.g., by using the `&` operator). It is important that the Truffle/C Memory Model is able to provide addresses for variables. The garbage collector of the JVM can move objects within the Java heap, hence pointers must not point into it. The Truffle/C Memory Model stores variables which are referenced by a pointer on the Native Heap.

- **Arrays, Structures and Unions:** C objects like arrays, structures and unions are represented as blocks of memory that contain the data of the objects. C programs rely on the fact that these objects never move inside the memory, hence the Truffle/C Memory Model of the VM has to guarantee that C programs running on top of the VM can rely on this fact. Therefore, the Truffle/C Memory Model keeps these objects on the Native Heap and stores them with the same alignment and padding as native code. A common alignment and padding of data allows the Truffle/C VM to exchange these objects with native code via pointers.

All run-time data of the Truffle/C VM on the Native Heap has the same alignment as allocations of native code and therefore the VM can exchange this data freely with native code. The programmer can access the Native Heap on a byte level from Java using the Unsafe API. This API allows allocating and deallocating memory and offers read and write operations on a byte level. Using this API, the Truffle/C VM is able to manage run-time data that is also accessible from native code.

The Truffle framework itself offers the *Frame* for run-time data. The Truffle optimization *partial evaluation* (see Section 2.2.2) performs an *escape analysis* on the Frame and is able to remove all accesses to it in the resulting optimized code. For the Truffle/C VM, this Frame optimization potential should not be neglected. Therefore, the Truffle/C Memory Model aims to only store variables on the Native Heap that actually could be accessed by native code or that are referenced by pointers. To benefit from the Frame optimization and provide interoperability with native code, the Truffle/C VM distributes the run-time data into two types of storage. The first type, referred as Frame, keeps all function local and primitive typed data that is never referenced by pointers. For this thesis we define primitive types to be all primitive types of the C language (e.g., `int, double, ...`). The second type, referred as Native Heap, keeps all data that is referenced by a pointer, variables that live during the execution of the whole C program (*global* and *static local* variables) and complex typed data. Complex types are types that are constructed out of multiple primitive types. With respect to the C language, structures, arrays and unions are complex types. The following Sections 4.2.1 and 4.2.2 explain and illustrate the two locations of run-time data with an example program.

4.2.1 Frame

The Frame is part of the Truffle framework and guest languages can use it store their run-time data. It is not possible to derive a memory address for any variable that the Truffle/C VM stores in the Frame, which limits the usage of the Frame by the VM. Nevertheless, the VM creates a new Frame whenever the VM calls a Truffle/C function and keeps all primitive variables and arguments that do not escape by their address inside the Frame.

```
1  void nativeFoo(struct ExampleStruct*, int*);
2
3  struct ExampleStruct {
4    char c;
5    int  *p;
6  };
7
8  int global = 23;                          // global variable
9
10 int main(int argc, char* argv[]) {
11   int escaped = 42;                       // escaping variable
12
13   struct ExampleStruct *s =               // local variable
14     malloc(sizeof(ExampleStruct));
15   char c = 'a';                           // local variable
16   int a[3] = {1,2,3};                     // complex type variable
17
18   s->c = c;
19   s->p = a;
20
21   nativeFoo(s, &escaped);
22   free(s);
23   return 0;
24 }
```

Listing 4.1: A small C program to illustrate the different types of storage.

To illustrate the distinction between the Frame and the Native Heap, we consider the example in Listing 4.1. This example uses a native function `nativeFoo`, which takes an instance of `ExampleStruct` and an integer by reference. As `nativeFoo` is a native function for which no C source code is available, its arguments have to point to data that is located in the Native Heap and aligned as expected by the native code.

The `main` function declares an integer variable `escaped`, which `main` passes to `nativeFoo` by reference. Besides this integer variable `escaped`, `main` also allocates memory for a structure `s`, which the program passes to `nativeFoo` after setting its members.

With respect to the example of Listing 4.1, the VM creates a Frame before it enters the `main` function. The call to `main` includes placing the arguments `argc` and `argv` into slots in the Frame (see Section 4.3).

Besides Frame slots for the arguments, the Truffle/C VM also reserves slots inside the Frame for all primitive non-escaping local variables. As the Truffle/C VM knows at compile time which variables it can place in the Frame, it reserves a fixed number of Frame slots for these variables. In the example, the VM needs four Frame slots, two for the arguments (`argc` and `argv`), one for the character (`c`) and one for the pointer to the structure `ExampleStruct` (`s`). Figure 4.2 illustrates the Frame of the function `main` where the Truffle/C VM uses slot 0 and 1 for the arguments and slot 2 and 3 for the pointer `s` and the character `c`. All other slots are unused. As the example never resolves the addresses of `c` and `s` and both variables have a primitive type, the Truffle/C VM can store them inside the Frame. The VM stores all other local variables of `main` on the Native Heap (see Section 4.2.2).

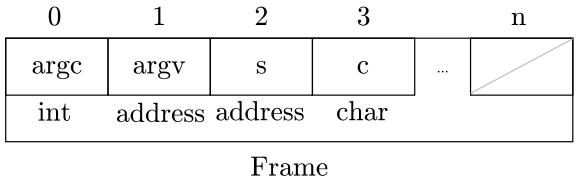


Figure 4.2: Frame allocation of function `main` in Listing 4.1.

4.2.2 Native Heap

Besides the Frame, the Native Heap is the second place where the Truffle/C VM can store run-time data. The VM can access run-time data on the Native Heap via pointers and can also provide addresses of elements stored in this area.

The Truffle/C VM organizes the Native Heap in blocks. A memory block is an allocation performed by the VM, using the Unsafe API, or the programmer. There are three kinds of blocks:

- **Static Block:** The Truffle/C VM allocates one block that contains all static variables. This allocation happens when the VM starts up.

- **Function Block:** For each function call, the Truffle/C VM creates a Function Block and deallocates it when the function returns. A Function Block contains all local variables or arguments of a function whose address is needed, as well as all complex local variables.

- **User Allocation:** User allocations are memory regions allocated by the programmer. The programmer can allocate memory using native functions like `malloc`.

Figure 4.3 shows the allocation of the VM on the Native Heap for the example in Listing 4.1. In this example there is one global variable (`global`). Therefore, the Static Block has a size of 4 bytes and contains this integer variable.

At the point where the Truffle/C VM calls the `main` function, it allocates a Function Block. This Function Block has a size of 16 bytes. It contains the escaping integer (`escaped`, 4 bytes) and the integer array (`a`, 3×4 bytes). This block is alive and accessible by the VM or native code from the entry point of the function `main` until it returns.

The example of Listing 4.1 also performs an allocation using the native function `malloc`. The function `malloc` allocates memory on the Native Heap. This allocation reserves 16 bytes that contain the data of the structure `ExampleStruct`. The deallocation of this memory block is not the responsibility of the Truffle/C VM but rather has to be performed by the user code.

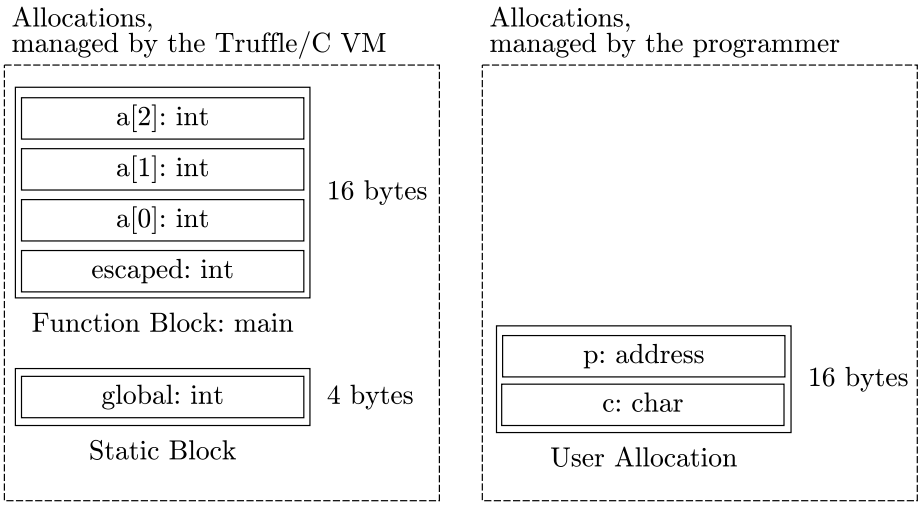


Figure 4.3: Native Heap allocations of function `main` in Listing 4.1.

4.2.3 Arrays, Structures and Unions

As Section 4.2 has already stated, C programs rely on the fact that C objects (arrays, structures and unions) are located in the memory and never change location. Also, C code assumes a well-defined alignment of the data so that the program can access it via pointers. The Truffle/C VM has to satisfy these assumptions for all code (Truffle/C functions and native C functions) that can access these C objects. To establish an interoperability between Truffle/C functions and native C functions the VM keeps these C objects on the Native Heap and also stores them with the same data alignment and padding as native code. The following two bullets illustrate how the Truffle/C VM incorporates these invariants, induced by native code, for arrays, structures and unions.

Arrays: C code assumes that arrays are stored as a sequence of variables. Each element of the array takes n bytes, where n is the size of the array element type (e.g., $n = 4$ bytes per element for `int a[]`). Figure 4.4 shows an example integer array with 4 elements. The Truffle/C VM incorporates this alignment and stores arrays inside the Static Block or the Function Block depending on if the variable is global or local. *MemoryAddress* nodes represent arrays by pointing to the base address of the array data.

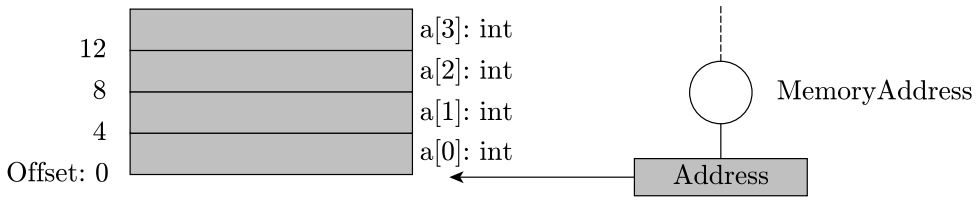


Figure 4.4: Memory representation of an array.

A result of a *MemoryAddress* node can be seen as a pointer. It is possible to pass this address value to any native target function. The native target function itself can then access the array because the VM uses the same alignment as native code.

Figure 4.5 illustrates an array access (e.g., `base[index]`) of the Truffle/C VM. On the right side Figure 4.5 shows the variable node (*MemoryAddress*), which points to the array data. The parent of this node is an *ArrayElementAddress* node. Besides the array base address, the *ArrayElementAddress* node has a second child. The second child can be an arbitrary subtree that calculates the index of the array access. This index value is multiplied by a stride which

converts the index value to an offset within the array data. The result of this calculation, which is the return value of the *ArrayElementAddress* node, is an address value that the Truffle/C VM can read. The blue labels of Figure 4.5 show the return values of all nodes.

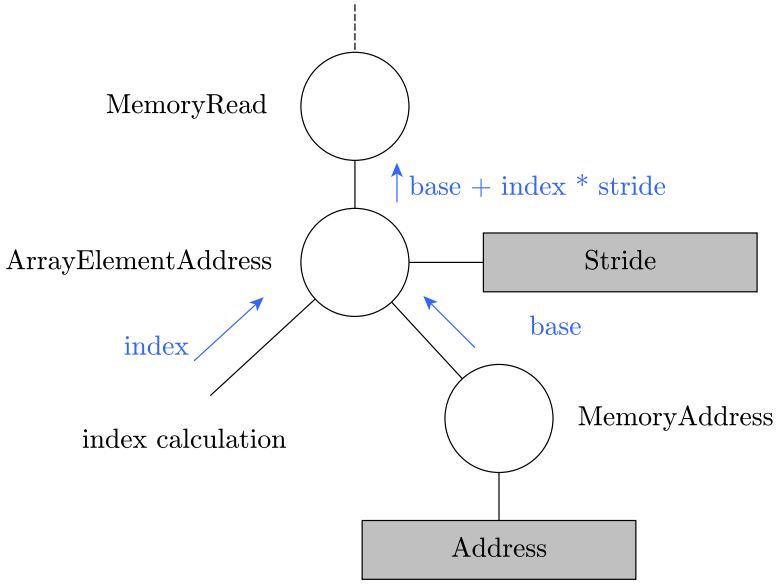


Figure 4.5: Truffle/C nodes to read from an array.

Structures and Unions: To store the data of a record variable (structure or union), the Truffle/C VM has to reimplement the alignment and padding expected by native code.

According to the native alignment rules, the first member of a structure has the offset $O_0 = 0$ from the base address. All offsets of the following members (O_i) are calculated as follows:

$$O_0 = 0$$
$$O_i = Align(O_{i-1} + Size_{i-1}, Size_i)$$

O_i, the offset of member i, is derived from the offset of the previous member O_{i-1} plus its size ($Size_{i-1}$) in bytes and the size of the member i ($Size_i$). The alignment makes sure that the structure's members are aligned to platform specific address boundaries. For instance, `int` members have an offset O_i mod $4 = 0$, where 4 is the size of an `int` in bytes.

Figure 4.6 shows the memory representation of `ExampleStruct` (see Listing 4.1). The character `c` has an offset of 0 because it is the first member ($O_0 = 0$). The platform dependent offset of `p` is $O_1 = 8$. $O_1 = Align(0 + 1, 8)$, where 0 is the offset of the first member `c`, 1 the size of the first member in bytes ($Size_0$) and 8 the size of the second member ($Size_1$).

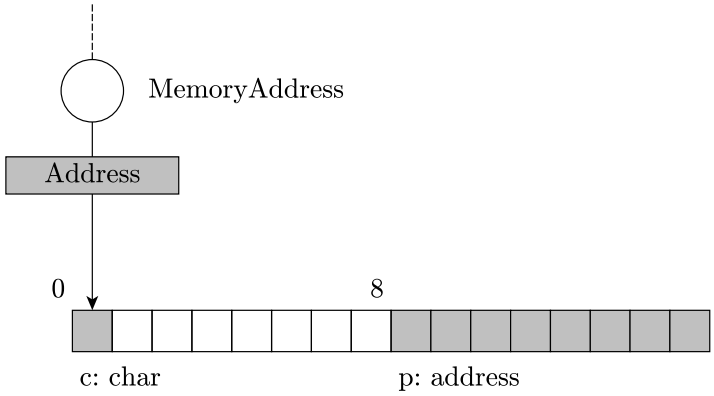


Figure 4.6: Memory representation of a struct.

Similar to arrays, the Truffle/C VM uses a *MemoryAddress* node to represent the base address of a structure instance. Figure 4.6 illustrates a *MemoryAddress* node pointing to an instance of `ExampleStruct`. To access a member of a structure, the Truffle/C VM calculates the address of the member using a *StructElementAddress* node (see Figure 4.7). The child of the *StructElementAddress* node is a *MemoryAddress* node. The *MemoryAddress* node produces a pointer to the base address of a structure. The *StructElementAddress* node then adds the offset of a structure member to the base address. The blue labels of Figure 4.7 show the return values of all nodes.

The second type of records (unions) follow the same principle as structures, except that the offset is always zero and therefore the VM does no offset calculation.

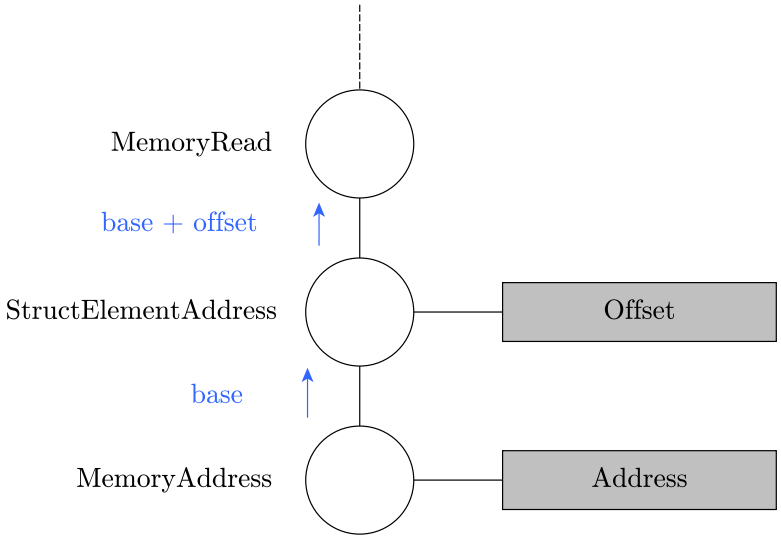


Figure 4.7: Truffle/C nodes to read from a structure.

4.3 Truffle/C Function Handling

The Truffle/C Function Handling is the second part of the Truffle/C Runtime Environment. It has to provide infrastructure to efficiently resolve, link and call C functions at runtime. The Truffle/C Function Handling provides the following functionality:

- **Function Resolving:** An execution of a C program on top of the Truffle/C VM can call available C functions. The VM does linking at runtime and therefore resolves the target function to provide a call target on demand (see Section 4.3.1). The VM distinguishes between two different kinds of call targets:
  - *Truffle/C functions:* If source code of the target function is available and accessible by the Truffle/C VM, the VM resolves a Truffle/C function.
  - *Native functions:* If the requested function is present as a native function inside a linked native library, the Truffle/C VM uses GNFI and resolves a `NativeFunctionHandle`.

- **Function Calling:** Depending on the kind of the call target, the Truffle/C VM has to apply different call strategies:

- *Truffle/C functions:* If the call target is a Truffle/C function, the VM uses existing Truffle facilities to perform the call (see Section 4.3.2).
- *Native functions:* If the call target is a native target function, the VM performs a native call using GNFI. The VM executes the `NativeFunctionHandle` of the target function, using the GNFI Java calling convention (see Section 4.3.3).

4.3.1 Linking Functions

Normally, a C program is compiled and linked completely before executing it. The Truffle/C VM follows a different approach: The VM performs linking at runtime, so there is no need to link files before starting execution. The VM takes a list of Truffle/C binaries and also a list of native libraries as input and immediately starts execution. The linking is opaque to the user.

When the VM starts the execution of a C program, it first resolves the `main` function. For this purpose it uses the Truffle/C parser and builds up the function AST of `main` and uses it as an entry point (the `main` function has to be present as a Truffle/C function).

All call sites inside a function AST are initially represented as *unresolved FunctionCall* nodes. An unresolved *FunctionCall* node denotes a call to a C function that only knows the call target's name, not the call target itself. A *FunctionCall* node remains unresolved until the VM executes it the first time. When the VM tries to execute an unresolved *FunctionCall* node, the node first resolves an executable call target for the call target name. The node uses the Truffle/C Runtime Interface (see Figure 4.8) to get an executable call target. The Truffle/C Runtime Interface first tries to resolve a Truffle/C call target. It uses the Truffle/C parser and looks in all Truffle/C binaries for a function that matches the call target name. If a Truffle/C function matches, the Truffle/C Runtime Interface returns a Truffle/C function to the unresolved *FunctionCall* node. The unresolved *FunctionCall* node then replaces itself with a *ResolvedTruffleCall* node.

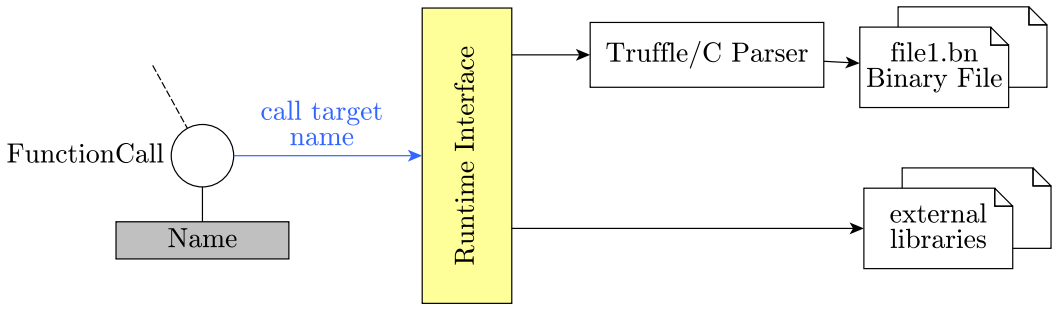


Figure 4.8: Truffle/C function resolving.

If the Truffle/C Runtime Interface did not find a Truffle/C function, it tries to resolve a `NativeFunctionHandle`. It looks for the requested function in all given native libraries (see Figure 4.8). If the Truffle/C Runtime Interface was able to resolve a `NativeFunctionHandle` the interface returns it and the unresolved *FunctionCall* node replaces itself with a *ResolvedNativeCall* node. If the Truffle/C Runtime Interface can neither resolve a Truffle/C function nor a `NativeFunctionHandle`, the Truffle/C VM raises a linking error and stops execution. Figure 4.9 illustrates the procedure of a *FunctionCall* node rewrite. An unresolved *FunctionCall* node replaces itself with either a *ResolvedTruffleCall* node or a *ResolvedNativeCall* node.

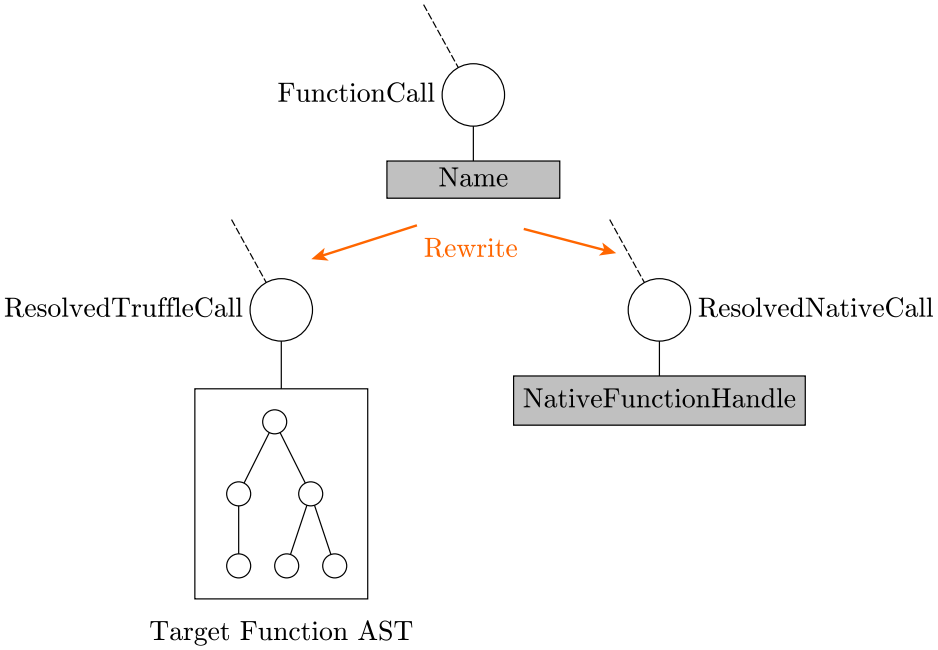


Figure 4.9: Truffle/C call node rewriting.

4.3.2 Calling Truffle/C Functions

Once a call node is resolved, the Truffle/C VM is able to call the target function. Listing 4.2 shows an example to illustrate the function call.

```
1 int foo(double, int);
2
3 void main() {
4   int result = foo(3.0, 1 + 2);
5   // ...
6 }
```

Listing 4.2: A call statement in C.

This section assumes that the function `foo` (Listing 4.2) is a Truffle/C function. Calling the Truffle/C function `foo` involves the following steps, which apply to all Truffle/C function calls, as shown in Figure 4.10:

- **Argument Evaluation:** First, the Truffle/C VM evaluates all arguments. The arguments of the function `foo` (Listing 4.2) are the literal node of the value 3.0 and the addition of the values 1 and 2, which the VM has to evaluate.
- **Pack Arguments:** The VM then stores the evaluated arguments into a representation that is independent from the function signature. The Truffle/C VM uses an `Object` array as such an independent representation.
- **Call:** After packing the arguments, the VM performs the call to the target Truffle/C function. It passes the arguments, packed into the `Object` array, to the callee.
- **Copy Arguments to Frame:** On the callee side, the VM first stores all arguments that do not escape by their address and have a primitive type into the Frame.
- **Open Function Block:** Before executing the function AST, the Truffle/C VM creates a Function Block. As Section 4.2 explains, whenever the VM calls a function, it creates a Function Block on the Native Heap where it stores escaping or complex variables of a function.
- **Copy Arguments to Native Heap:** After creating a Function Block, the Truffle/C VM copies all complex arguments or arguments that escape by their address.
- **Execute Function:** The VM evaluates the function body.
- **Close Function Block:** When the function returns, the Truffle/C VM closes the Function Block on the Native Heap.

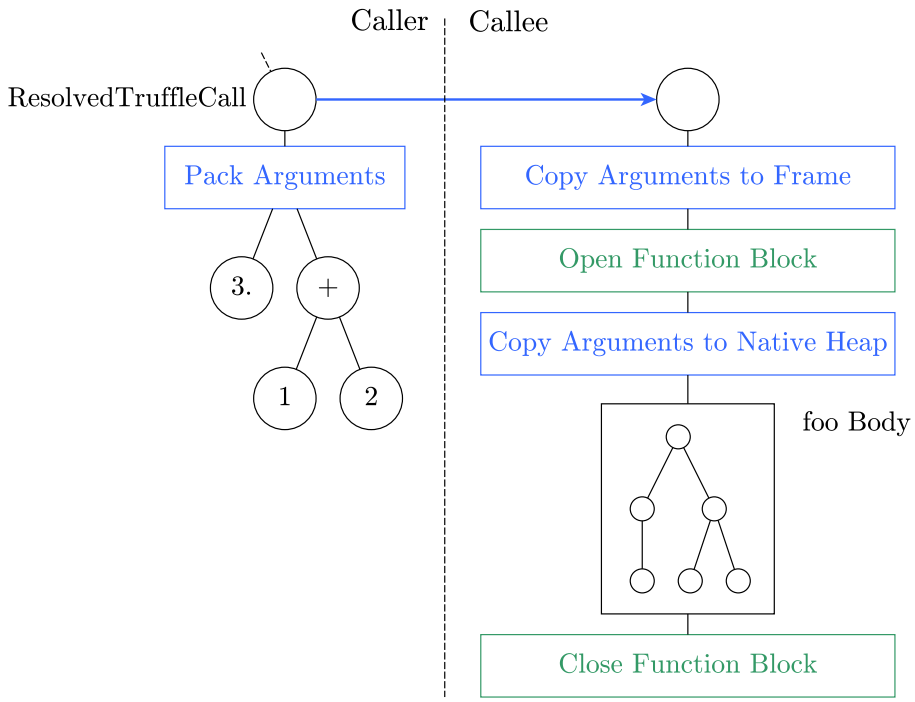


Figure 4.10: Call to a Truffle/C function.

After these steps, the Truffle/C function returns the result to the call node and the caller continues.

4.3.3 Calling Native Library Functions

Again, we consider the call to the function `foo` in Listing 4.2. For this section we assume that the function is a native function. The Truffle/C VM has already resolved a `NativeFunctionHandle` (see Chapter 3) and can call `foo`. The call to this native C function involves the following steps which apply to all native function calls, as shown in Figure 4.11:

- **Argument Evaluation:** First, the VM evaluates all arguments, similar to Section 4.3.2.
- **Pack Native Arguments:** The VM stores the evaluated arguments into an `Object` array as required by the GNFI Java calling convention.
- **Call:** After packing the arguments, the VM executes the `NativeFunctionHandle`.

After these steps, the call node returns the result of the native function, and the caller continues.

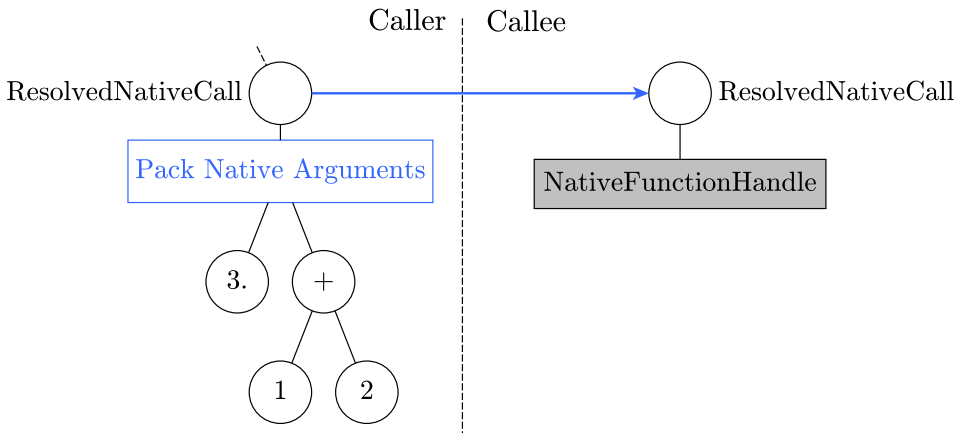


Figure 4.11: Call to a native function.

Chapter 5

Case Study

This chapter presents a case study which exercises a C program executed on top of the Truffle/C VM. The case study demonstrates how the Truffle/C VM calls Truffle/C functions and native functions. Furthermore, it illustrates the states of the Frame and Native Heap at important points of the C program execution. This example uses the native library cblas, which is a library that offers highly optimized matrix operations.

The program of the case study consists of one file, `caseStudy.c` (see Listing 5.1). It performs two matrix multiplications, one of which is executed using a Truffle/C function, while the other one uses a native function from the native library cblas.

In order to start the Truffle/C VM and execute `caseStudy.c`, the VM requires two arguments. The first argument is the path to the C file it should execute (containing the `main` function, `caseStudy.c`). Besides the C file, it is necessary to tell the VM all libraries that are used. While one would set the linker option `-lcblas` using GCC, the Truffle/C VM needs the path to the library object. Therefore, the second argument is the path to the cblas library (`libcblas.so`). Given these arguments, the Truffle/C VM can execute `caseStudy.c`. The VM automatically compiles `caseStudy.c` to a Truffle/C binary and starts executing the `main` function of the program.

```
#include <cblas.h>

struct matrix {
  int rows;
  int cols;
  double* data;
};

void myMult(struct matrix * m1, struct matrix * m2, struct matrix * m4) {
  int i, j, k;
  double sum = 0;                          // ||Point 3||
  for (i = 0; i < m1->rows; i++ ) {
    for (j = 0; j < m2->cols; j++ ) {
      for (k = 0; k < m1->cols; k++ ) {
          sum = sum +
            m1->data[i * m1->cols + k] *  // ||Point 4 (i = 1, j = 1, k = 1)||
            m2->data[k * m2->cols + j];
      }
      m4->data[i * m3->cols + j] = sum;
      sum = 0;
    }
  }
}

int main() {
  struct matrix m1; // initialize matrix 1:
  double m1_data[] = {
    1.0, 2.0, 3.0,
    4.0, 5.0, 6.0};
  m1.data = m1_data;
  m1.rows = 2;
  m1.cols = 3;

  struct matrix m2; // initialize matrix 2:
  double m2_data[] = {
    7.0, 8.0,
    9.0, 10.0,
    11.0, 12.0};
  m2.data = m2_data;
  m2.rows = 3;
  m2.cols = 2;

  struct matrix m3; // initialize matrix 3:
  double m3_data[] = {
    0.0, 0.0,
    0.0, 0.0};
  m3.data = m3_data;
  m3.rows = 2;
```

```
49   m3.cols = 2;
50
51   struct matrix m4; // initialize matrix 4:
52   double m4_data[] = {
53     0.0, 0.0,
54     0.0, 0.0};
55   m4.data = m4_data;
56   m4.rows = 2;
57   m4.cols = 2;
58
59   double alpha = 1.0, beta = 0.0;
60                                          // ||Point 1||
61   //BLAS CALL m3 = m1 x m2:              // ||call cblas_dgemm||
62   cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, m1->rows, m2->cols, m1->cols,
63   alpha, m1.data, m1.cols, m2.data, m2.cols,
64   beta, m3.data, m3.cols);
65                                          // ||Point 2||
66   //CALL IMPLEMENTATION m4 = m1 x m2:        ||call myMult||
67   myMult(&m1, &m2, &m4);
68   return 0;                              // ||Point 5||
69 }
```

Listing 5.1: A program (`caseStudy.c`) that multiplies matrices using a Truffle/C function and the `cblas_dgemm` function (blas library).

5.1 Problem Statement

The program of `caseStudy.c` (see Listing 5.1) declares a structure `matrix`. This structure represents a matrix of `double` values. It contains a pointer to a one dimensional `double` array (`data`). This array contains the values of the matrix. Furthermore, this structure declares the number of rows (`rows`) and the number of columns (`cols`). These values (`rows` and `cols`) are necessary for the multiplication algorithms to interpret the one dimensional `double` array as a matrix.

The `main` function first initializes 4 matrices (stored in `struct matrix m1, m2, m3` and `m4`). For matrix `m1`, `main` initializes a 2×3 matrix. Afterwards, `main` initializes a 3×2 matrix (`m2`). Finally, `main` creates two 2×2 matrices (`m3` and `m4`).

Having initialized these matrices, `main` calls the native blas function `cblas_dgemm`. This function multiplies two `double` precision matrices ($m3 = m1 \times m2$). Listing 5.2 shows and describes the signature of `cblas_dgemm`.

```
void cblas_dgemm (
  // Specifies row-major (C) or column-major (Fortran) data ordering.
    const enum CBLAS_ORDER Order,
    // Specifies whether to transpose matrix A.
    const enum CBLAS_TRANSPOSE TransA,
    // Specifies whether to transpose matrix B.
    const enum CBLAS_TRANSPOSE TransB,
    // Number of rows in matrices A and C.
    const int M,
    // Number of columns in matrices B and C.
    const int N,
    // Number of columns in matrix A; number of rows in matrix B.
    const int K,
    // Scaling factor for the product of matrices A and B.
    const double alpha,
    // Matrix A.
    const double* A,
    // The size of the first dimension of matrix A;
    // if you are passing a matrix A[m][n], the value should be m.
    const int lda,
    // Matrix B.
    const double* B,
    // The size of the first dimension of matrix B;
    // if you are passing a matrix B[m][n], the value should be m.
    const int ldb,
    // Scaling factor for matrix C.
    const double beta,
    // Matrix C.
    double* C,
    // The size of the first dimension of matrix C;
    // if you are passing a matrix C[m][n], the value should be m.
    const int ldc
)
```

Listing 5.2: Signature of `cblas_dgemm`.

After the native call to `cblas_dgemm`, the case study performs a call to the Truffle/C function `myMult`. The function `myMult` takes three matrices, represented as pointers to instances of the structure `matrix`. The function computes $m4 = m1 \times m2$.

This example is well-suited to explain the Truffle/C Runtime Environment, because it makes use of the different types of storage, namely the Frame and the Native Heap, and therefore allows us to extensively investigate the Truffle/C Memory Model. The example also calls a Truffle/C function and a native function, hence this case study can explain the different procedures of resolving and calling functions.

5.2 Analysis

To analyze the program (see Listing 5.1), this chapter will focus on seven points of the execution (denoted by comments like `"|| Point 1 ||"`) in the source code of Listing 5.1. At these points of execution, this chapter analyzes the content of the Frame and the Native Heap. Also, this analysis focuses on function calls and linking. Therefore, this section describes how the VM resolves the functions and how it performs the calls.

Point 1: At `Point 1` of the execution, the structures (`m1, m2, m3` and `m4`) and arrays (`m1_data, m2_data, m3_data` and `m4_data`) are already initialized and stored on the Native Heap. The Truffle/C VM places these variables in the Function Block of the `main` function. To store the structures, the VM applies the same alignment (see Section 4.2) as native code. Figure 5.1 illustrates the state of the Native Heap at `Point 1`. The VM allocates and stores the data in the same order in which they are declared in the program. There is no Static Block because the example does not use global variables or static local variables.

The Frame at `Point 1` contains only two values, `alpha` and `beta`, as Figure 5.2 shows. These variables have a primitive type and they are never referenced by their address, therefore the VM can place them inside the Frame.

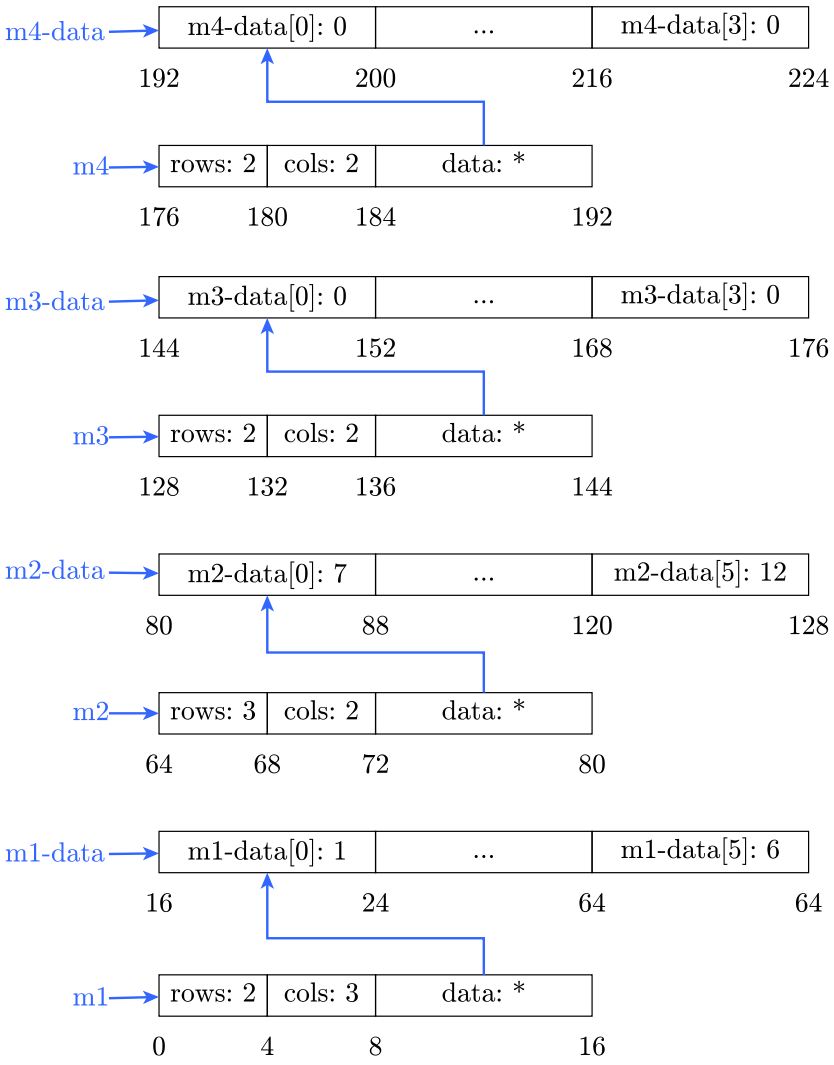


Figure 5.1: Native Heap at Point 1.

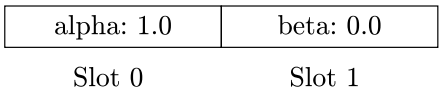


Figure 5.2: Frame of `main` at Point 1.

Call cblas_dgemm: When the execution reaches the call to `cblas_dgemm`, the call is initially unresolved. To resolve this call, the call node asks the Truffle/C Runtime Interface for a call target, providing the name `cblas_dgemm`. The Truffle/C Runtime Interface first tries to find a Truffle/C function with the name `cblas_dgemm` by using the Truffle/C Parser. The parser will not

find a Truffle/C function with the name `cblas_dgemm`, hence the Truffle/C Runtime Interface tries to resolve a `NativeFunctionHandle`. As there is a native `cblas_dgemm` function, the Truffle/C Runtime Interface returns a `NativeFunctionHandle` as a call target. The call node replaces itself with a *ResolvedNativeCall* node and performs a native call. To perform the call, the VM first evaluates all arguments and stores the results in an `Object` array as required by the GNFI Java calling convention. Table 5.1 shows the content of the `Object` array that contains the arguments for the `NativeFunctionHandle`.

index	Content	Description
0,1,2	**101, 111, 111**	enum values of CblasColMajor , CblasNoTrans , CblasNoTrans
3,4,5	**2, 2, 3**	row/column information for m1, m2, m3
6	**1.0**	double value 1.0
7	**139998154615616**	the address of m1.data
8	**3**	size of first dimension of m1
9	**139998154615680:**	the address of m2.data
10	**2**	size of first dimension of m2
11	**0.0**	double value 0.0
12	**139998154615744**	the address of m3.data
13	**2**	size of first dimension of m3

Table 5.1: Content of the arguments array for the native call to `cblas_dgemm`.

After preparing the arguments and executing the `NativeFunctionHandle`, the execution of `main` continues after the call.

Point 2: The native function `cblas_dgemm` updated the data of array `m3_data` on the Native Heap. The new content of `m3_data` is the result of the matrix multiplication. The Frame did not change and still contains the values for `alpha` and `beta`.

Call myMult: When the execution reaches the call to `myMult`, the call node is also initially unresolved. As previously stated, the call node asks the Truffle/C Runtime Interface for a call target. Again, it first uses the Truffle/C Parser and tries to resolve a Truffle/C function. This time the parser succeeds and finds a Truffle/C function and returns it as the call target of `myMult`. After the call node rewrites itself to a *ResolvedTruffleCall*, the VM performs a Truffle call.

The call first evaluates the arguments. The arguments for the `myMult` function are the 3 pointers to `m1, m2` and `m4`. The call node boxes the 3 addresses into `Long` Objects and packs them into an `Object` array, as shown in Table 5.2.

index	Content	Description
0	**139998154615600**	the address of m1
1	**139998154615664**	the address of m2
2	**139998154615776**	the address of m4

Table 5.2: Content of the arguments array for the Truffle call to `myMult`.

After packing the arguments, the Truffle/C VM calls `myMult`. Before the function executes the body, it copies the arguments to the Frame of `myMult`. Besides copying the arguments, the VM also opens a Function Block where it can store complex or escaping local variables of `myMult`. In the case of `myMult`, there are no complex or escaping local variables, therefore the Function Block is of size zero. As there are no complex arguments or arguments that are referenced by their address, there is no need to copy arguments to the Native Heap. Finally, the VM executes the function body.

Point 3: The VM only creates a new Frame for `myMult`. This Frame contains the three arguments and the local variables `i`, `j`, `k` and `sum`, which are needed for the matrix multiplication. Figure 5.3 shows the Frame of function `myMult`.

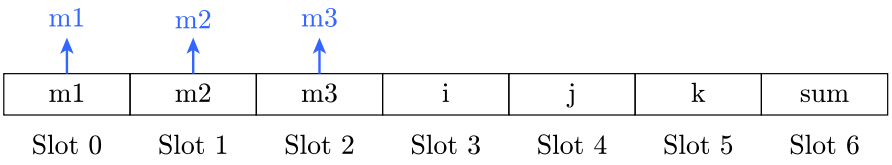


Figure 5.3: Frame of `myMult` at Point 3.

Point 4: The function `myMult` reads the elements of the matrices `m1` and `m2` to finally calculate the product and store it in the matrix `m4`. The expression `m1->data[i * m1->cols + k]` at `Point 5` reads the element at position $column = i$ and $row = k$ of the matrix `m1`. Figure 5.4 shows the AST representation of this expression. On the top of this tree there is a *MemoryRead* node which reads the element of the matrix. The *ArrayElementAddress* node provides the address for this read operation, which points to the Native Heap. To provide this address, the *ArrayElementAddress* node has two children. The first child computes the array index (*ArrayElementAddress* - Left child) while the second child (*ArrayElementAddress* - Right child) computes the base address of the array.

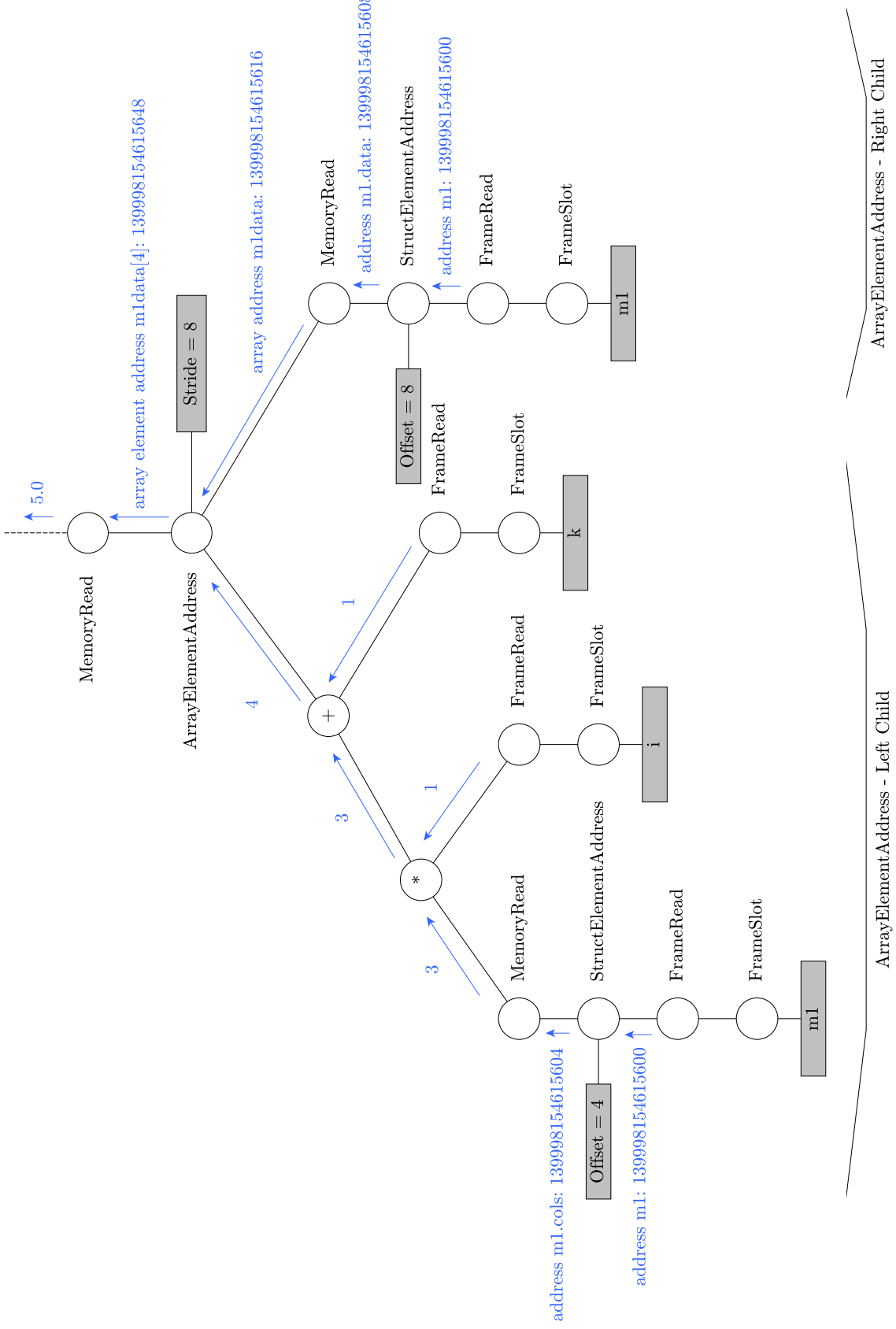


Figure 5.4: Truffle AST for the expression `m1->data[i * m1->cols + k]`.

ArrayElementAddress - Left child: The index calculation (`i * m1->cols + k`) first multiplies the value `i` (located in the Frame) and the number of columns of `m1` (`m1->cols`). A *MemoryRead* node reads the number of columns (`m1->cols`) from the Native Heap. To do so, the AST has to provide the address of the member `cols`. It computes this address by first reading the base address of `m1` from the Frame and afterwards adding the offset of the member `cols` to the base address (*StructElementAddress* node). After the multiplication of `i * m1->cols`, the tree adds the value of `k` to this product.

ArrayElementAddress - Right child: To provide the base address of the array (`m1->data`), the tree first reads the base address of `m1` from the Frame. Afterwards, it adds the offset of the member `data` (*StructElementAddress* node) to the base address of `m1`. The *MemoryRead* reads the content of this member (`data`) and returns the address of the array `m1_data`.

After evaluating the right and left child, the *ArrayElementAddress* can provide the address of the matrix element.

Blue edges in this tree denote the data that the different nodes produce. We assume that the local variables `i, j` and `k` have the value 1 (`i = 1, j = 1` and `k = 1`).

Point 5: After executing `myMult`, the VM closes the Frame of `myMult` and it is no longer accessible. As `myMult` changed values, stored in the Function Block of the function `main` (structure `m4`), the content of the Native Heap changed. The function `myMult` placed the result of the multiplication of the matrices `m1` and `m2` into `m4`. At the end of this program, the matrices `m3` and `m4` both contain the product of $m1 \times m2$.

This case study has shown how the Truffle/C VM distinguishes between the different locations of run-time data, namely the Frame and the Native Heap. Furthermore, it illustrated how the VM aligns the data on the Native Heap to be in accordance with native code. Besides investigating the Truffle/C Memory Model, this case study also shows how the VM resolves functions and how it performs Truffle calls as well as native calls.

Chapter 6

Evaluation

This chapter evaluates GNFI and the Truffle/C VM independently from each other. The evaluation of GNFI uses micro benchmarks and a jblas benchmark. To evaluate the Truffle/C Runtime Environment, this chapter presents performance numbers of the Truffle/C VM compared to GCC on various C language benchmarks.

The benchmarks were executed on an Intel Core i7 3520M dual-core 2.9 GHz CPU running 64 Bit Fedora 17 (Linux3.6.9-2) with 8 GB of memory. GNFI and the Truffle/C VM are built on Graal revision `ee8cd087a731` (including the Truffle framework) from the official OpenJDK Graal repository[1]. The evaluation uses OpenJDK 1.7.0_09 for JNI, and JNA version 3.5.2.

6.1 Evaluation of GNFI

This section presents an evaluation of GNFI for standard Java applications. It presents a comparative performance evaluation of GNFI, JNI and JNA. The first benchmark is a micro benchmark that determines the cost of a call. It invokes an empty C function (with 0, 3 and 5 arguments) in a loop. This benchmark measures the call overhead of the three approaches.

The evaluation also benchmarks an application that uses a native numerical library, accessed by JNI, JNA and GNFI. This benchmark is a good choice because researchers in the past extensively investigated aspects such as efficiency and usability when calling native numerical libraries [5, 6, 7, 11]. None of these investigations considered the extra calling overhead caused by JNI and JNA.

[1] available at `http://hg.openjdk.java.net/graal/graal`

As a concrete application that uses a native numerical library, the evaluation uses jblas [17], a Java wrapper for BLAS, which is the de-facto standard for matrix operations. For the evaluation, all usages of JNI within jblas were replaced by GNFI or JNA.

The charts in this section are all arranged on a linear, unbiased, higher-is-better scale. The y-axis shows the different approaches that were evaluated. The x-axis shows the number of executed calculations performed within 10 seconds. For all of the benchmark results in this section, the benchmarks were executed 10 times with the same parameters. The charts show the arithmetic mean for each benchmark result, along with the minimum and the maximum of 10 runs.

The evaluation of each benchmark contains results for several different configurations: In interpreted mode (*int*) the benchmarks run with the compilation disabled, thus with all code running in the interpreter. This mode uses GNFI as described in Section 3.3. The JNI and JNA versions of the benchmark are also running in the interpreter mode with compilation disabled.

In compiled mode we measure the performance after an initial warm-up with compilation enabled. For GNFI there are two modes of operation: In the first mode (*runtime callstub*), the evaluation uses a configuration that makes sure that the optimizations cannot inline the call stub. Therefore, GNFI has to call it (see Figure 3.5). In the second mode (*static callstub*), GNFI inlines the call stub. Section 3.4 describes the configuration for GNFI in detail. For the JNI and the JNA versions of the benchmarks the evaluation also measures the performance after an initial warm-up with compilation enabled.

The following sections briefly describe each benchmark and evaluate it for the modes *interpreted*, *runtime callstub* and *static callstub*.

6.1.1 Microbenchmarks

To measure the pure call overhead, the evaluation uses a shared library that contains an empty function `arg0` as shown in Listing 6.1. Besides `arg0`, this library contains two functions (`arg3` and `arg5`) with 3 and 5 arguments, respectively.

```
1 void arg0() {}
2 int arg3(int a, int b, int c) { return 0; }
3 int arg5(int a, int b, int c, int d, int e) { return 0; }
```

Listing 6.1: Native functions `arg0`, `arg3` and `arg5` to evaluate the pure call overhead.

Figure 6.1 shows the comparison of GNFI, JNA and JNI when calling `arg0` in interpreted mode. We can see that GNFI is slower than JNI by a factor of 2.7. However, it is still faster than JNA by a factor of 3.4.

Figure 6.2 shows that passing the arguments only has a minor effect on the speed of GNFI and JNI. GNFI is still slower than JNI by a factor of 2.7. For JNA, additional arguments cause a slowdown. For the function with 5 arguments (`arg5`), GNFI beats JNA by a factor of 41.

Compared to JNI, GNFI has to go through more layers to perform the call. As Section 3.3 describes, GNFI has to bridge interpreted and compiled code, which impedes the performance. For the performance of an application, interpreted code plays a minor role. Hence, the implementation of GNFI neglects the performance in interpreted mode and focuses on performance in compiled mode.

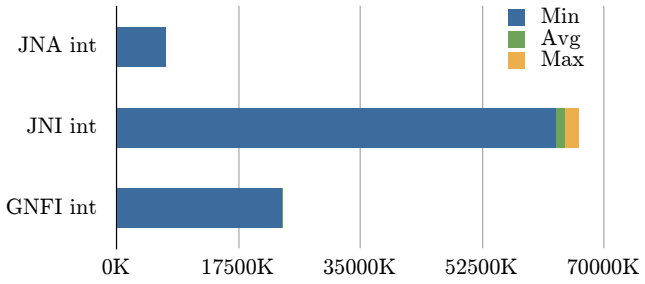


Figure 6.1: Performance of `arg0` (interpreted mode).

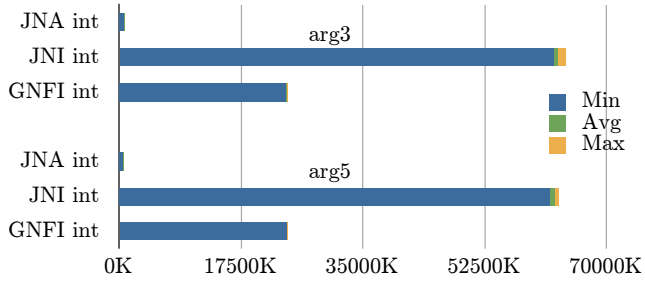


Figure 6.2: Performance of `arg3` and `arg5` (interpreted mode).

Figure 6.3 shows the comparison of JNI, JNA and GNFI on `arg0` in compiled mode. GNFI outperforms JNA by a factor of 10.3. It is now also faster than JNI by a factor of 1.2 with the *static callstub*. The performance of the *static callstub* is 6% better than the performance of the *runtime callstub*.

Again, the evaluation of Figure 6.4 shows that additional parameters only have a minor effect on the performance of JNI and GNFI. GNFI with *static callstub* remains faster than JNI by a factor of 1.2. The performance of JNA decreases with additional arguments. For the function with 5 arguments (`arg5`), GNFI beats JNA by a factor of 74.

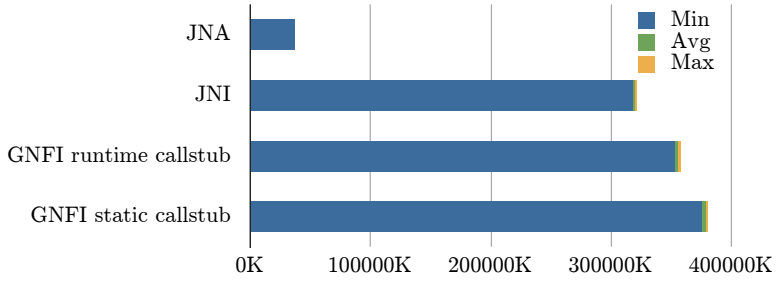


Figure 6.3: Performance of `arg0` (compiled mode).

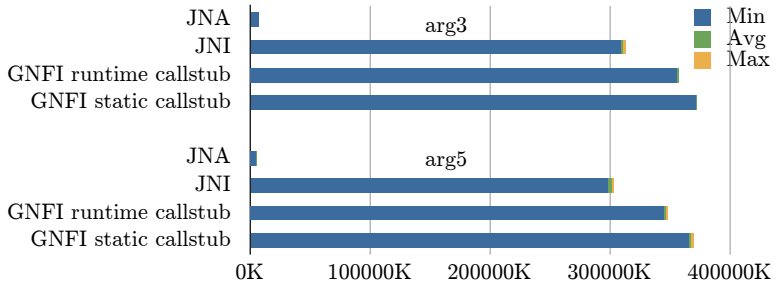


Figure 6.4: Performance of `arg3` and `arg5` (compiled mode).

For the micro benchmarks, GNFI with *static callstub* can inline the callstub (described in Section 3.4) and thus remove one call site. The JIT compiler can also remove the `args` array. This inlining explains the 6% speedup of the *static callstub* compared to the *runtime callstub*. GNFI with *runtime callstub* is faster than JNI because GNFI does a direct call to the call stub. Compared to JNI, GNFI does not set up any environment parameters, which explains the differences in performance.

6.1.2 Jblas Matrix Multiplication Benchmark

The jblas benchmark measures how often random double matrices of different sizes can be multiplied within 10 seconds. The matrix dimensions are $n \times n$ where $n \in \{10, 100, 1000\}$.

In interpreted mode (*int*) for $n = 10$ (Figure 6.5, upper three bars), GNFI is slower than JNI by a factor of 2.6 and 12 times faster than JNA.

We explain the slowdown of GNFI compared to JNI in interpreted mode by the fact that GNFI has to go through more layers to perform the call, as explained in the evaluation of the micro benchmarks. However, when increasing the problem size, GNFI performs better than JNI even in interpreted mode. The benchmark results in Figure 6.6 show the comparison for $n = 100$, where GNFI outperforms JNI

by 70%. GNFI is a factor of 2 faster than JNA for $n = 100$. The explanation of the smaller performance gap between JNA and GNFI is that the constant call overhead of JNA has less impact when more time is spent inside the native method.

Similarly for $n = 1000$ (Figure 6.7) GNFI beats JNI by a factor of 1.9. GNFI is also faster than JNA by a factor of 1.9. The JNI jblas wrapper and the JNA interface copy the elements of the matrices to perform the call to the native BLAS function. GNFI directly passes a reference to the matrices on the Java heap. This is achieved by passing the Java array object to the call stub. The call stub resolves the address of the array data and passes this pointer to the native target function. GNFI benefits from the fact that jblas represents matrices as one-dimensional Java arrays.

In compiled mode GNFI is faster than JNI and JNA for all problem sizes. In mode *runtime callstub* GNFI intrinsifies the `NativeFunctionHandle` as described in Section 3.4. The caller method makes a direct call to the call stub. Thus, GNFI is faster than JNI by a factor of 1.14 for $n = 10$ (Figure 6.5), by a factor of 1.7 for $n = 100$ (Figure 6.6), and by a factor of 1.9 for $n = 1000$ (Figure 6.7). GNFI is faster than JNA by a factor of 17 for $n = 10$, by a factor of 1.8 for $n = 100$, and by a factor of 1.9 for $n = 1000$. Again, the performance gap between JNA and GNFI decreases as n increases because the constant call overhead of JNA has less impact on performance.

In the mode *static callstub*, the machine code of the benchmark can even call the BLAS function directly. However, the optimization from *runtime callstub* to *static callstub* only has an effect on $n = 10$ where the call overhead is still important. For $n = 100$ and $n = 1000$ the speedup is not noticeable, because the constant call overhead becomes negligible, compared to the time spent in the BLAS function. The performance advantage of GNFI results from not having to copy the matrices.

Although there was still a performance gap between compiled and interpreted Java code for $n = 10$, this performance gap becomes almost non-existent for $n = 100$ and $n = 1000$.

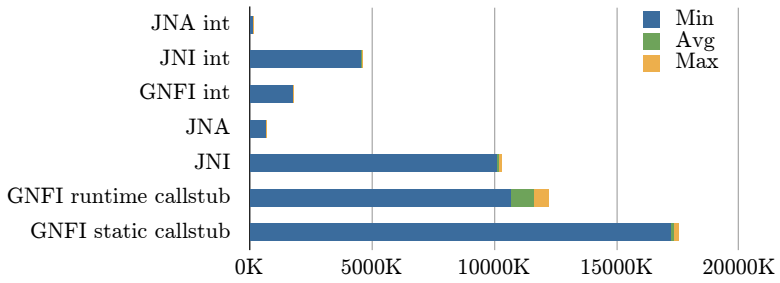


Figure 6.5: Performance of jblas for $n = 10$.

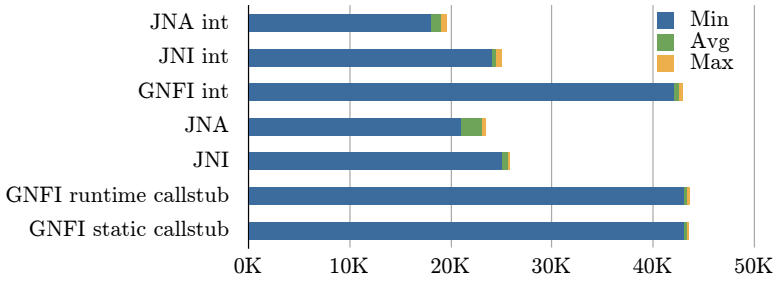


Figure 6.6: Performance of jblas for $n = 100$.

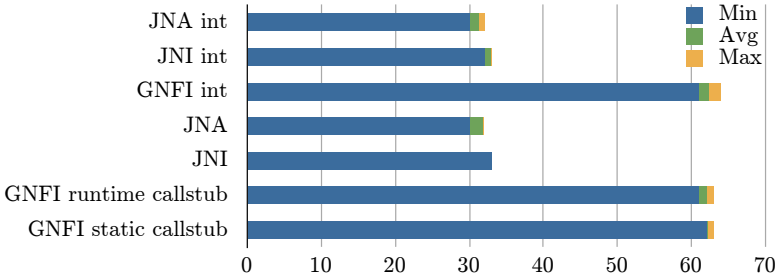


Figure 6.7: Performance of jblas for $n = 1000$.

6.2 Evaluation of the Truffle/C VM

This section presents a comparative performance evaluation of the Truffle/C VM and machine code compiled using GCC. This evaluation aims to determine the efficiency of the Truffle/C Runtime Environment. The benchmarks used for this evaluation are *Mandelbrot*, *Fannkuch Redux* and the *Spectral-norm* benchmark, all of which are part of the *Computer Language Benchmarks Game*[2] collection.

The first benchmark, Mandelbrot, contains no function calls and performs many computations with variables that the Truffle/C VM will keep in the Frame. Therefore, this benchmark demonstrates how the Truffle/C VM benefits from the Truffle optimizations for the Frame. This evaluation justifies the design decision of having two different locations for run-time data, Frame and Native Heap.

The second benchmark, Fannkuch Redux, performs many array permutations and therefore makes heavy use of the Native Heap. The Truffle/C VM keeps most variables of this benchmark on the Native Heap and only a few variables are in the Frame and will benefit from the Frame optimizations. Other than for Mandelbrot, the heavy use of the Native Heap now has a major influence on performance. Hence, this benchmark is well-suited to determine the efficiency of the VM using the Native Heap.

[2] `http://benchmarksgame.alioth.debian.org/`

The third benchmark, Sectral-norm, does many function calls (Truffle/C function calls and native function calls), so that it measures the efficiency of the function calling procedure of the Truffle/C VM. The location of run-time data of the benchmark is well balanced between the Frame and the Native Heap. As a result, this benchmark is well-suited to evaluate the Truffle/C Memory Model and Truffle/C Function Handling.

The charts in this section are arranged on a linear, unbiased, lower-is-better scale. The y-axis shows the different approaches that were evaluated (Truffle/C VM and GCC with different levels of optimizations). The x-axis shows the score of each benchmark. The evaluation of this section executes the benchmarks using a benchmark harness. This harness runs the benchmarks 10 times with the same parameters after an initial warm-up and computes a score. This score represents the arithmetic mean of the execution time in milliseconds. The charts show the scores of the Truffle/C VM and GCC for each benchmark.

The evaluation of the benchmarks contains results for the following configurations:

- **Truffle/C VM:** Truffle/C VM performance after an initial warm up. The warm up compiles and optimizes the Truffle ASTs of the benchmarks.
- **GCC performance:** Performance of binaries, compiled using GCC with different levels of optimizations (`O0`, `O1`, `O2`, `O3` and `Ofast`).

The benchmark results of Mandelbrot (Figure 6.8) show that the Truffle/C VM outperforms GCC with the optimization level of `O0` by a factor of 1.7. The Truffle/C VM performance is only 15% slower than the best GCC performance (optimization level `Ofast`). These numbers show that the VM has a reasonable performance compared to binaries compiled using GCC. Furthermore, these numbers demonstrate that the distinction between Frame and Native Heap improves the Truffle/C VM performance because the VM benefits from all Frame optimizations by the Truffle framework.

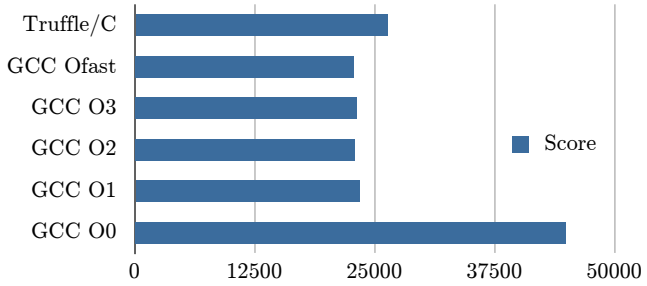


Figure 6.8: Mandelbrot performance of the Truffle/C VM and GCC.

The benchmark results of Fannkuch Redux (Figure 6.9) show that the VM outperforms GCC with the optimization level of `O0` by 4%. The Truffle/C VM performance is 2.4x slower than the best GCC performance (optimization level `O2`). For this benchmark, binaries compiled using GCC with the

optimization level of `O2` give the best performance. This optimization level outperforms the levels `O3` and `Ofast`. We explain this slowdown of `Ofast` by optimizations of GCC that have a bad impact on the performance (optimizations that might cause slowdowns because of, e.g., cache misses or a greater code size). The Truffle/C VM numbers on Fannkuch Redux point out that the distinction between Frame and Native Heap greatly improves the performance of the VM. If we compare the Mandelbrot numbers and the Fannkuch Redux numbers we can see that keeping variables in the Frame (like for Mandelbrot) speeds up the performance of the VM significantly. When the Truffle/C VM has to resort to the Native Heap for many variables, the gap between the best performance of GCC and the VM increases.

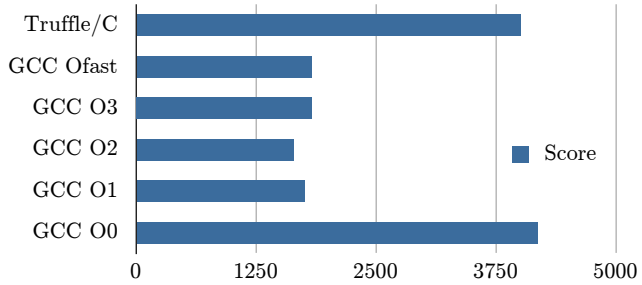


Figure 6.9: Fannkuch Redux performance of the Truffle/C VM and GCC.

The benchmark results of Spectral-norm (Figure 6.10) show that the Truffle/C VM outperforms GCC with an optimization level of `O0` by 30%. The performance of the Truffle/C VM is $4.7x$ slower than the best GCC performance (optimization level `Ofast`). We explain the big performance gap between the GCC with an optimization level of `Ofast` and the other GCC optimization levels by the fact that `Ofast` sets the GCC `ffast-math` option. This option gains performance by using imprecise floating point operations that are faster. Besides GCC with the `Ofast` option, GCC with `O1` gives the best GCC performance. For the Spectral-norm benchmark, the GCC optimizations of level `O2` and `O3` cause a slowdown compared to `O1`. Again, we explain this slowdown by optimizations that might cause, e.g., cache misses or a greater code size. Besides the different GCC performance results, these benchmark results show that the Truffle/C VM, using the Truffle/C Function Handling and the Truffle/C Memory Model, is in average 2.5 times slower than the best performance of binaries compiled with GCC.

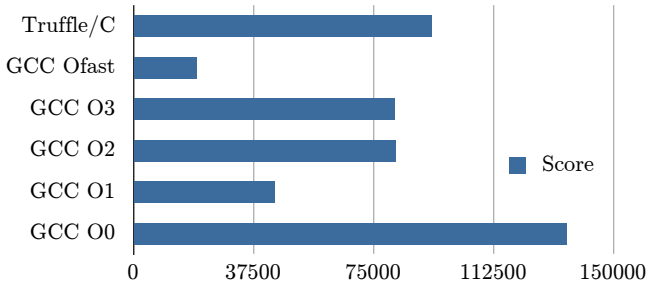


Figure 6.10: Spectral-norm performance of the Truffle/C VM and GCC.

Chapter 7

Related Work

This chapter lists the most important work related to GNFI and the Truffle/C VM. With respect to the Truffle/C VM, this chapter also points out former research towards memory safe C.

GNFI: From the user's perspective, JNA [3] is similar to GNFI. It allows the programmer to dynamically invoke native target functions without having to write native code — the programmer can simply dispatch the call from Java. As with GNFI, one loads a library and can then call arbitrary native target functions contained within. JNA offers the same flexibility as GNFI in terms of dynamically calling native target functions. It uses a native library stub, accessed via JNI, to dynamically invoke the native target code. However, as with JNI, the JIT compiler cannot inline this native code. Therefore, JNA cannot compete with GNFI in terms of performance, as the evaluation in Section 6.1 shows.

Hirzel et al. [14] propose another approach which tries to simplify the invocation of C code from Java, called Jeannie. It is both a language and a compiler to allow both Java and C code in the same file. The programmer can easily combine the two languages without the boiler-plate code of JNI. However, since it is translated to JNI code, it suffers from the same performance problems as the JNI.

Stepanian et al. [23] try to widen the compilation span by letting the JIT compiler directly inline native code. Their approach converts the native code to the compiler's intermediate language, on which the JIT compiler then performs optimizations. Following their approach of decompiling and inlining native JNI wrapper functions, they could gain a significant speedup compared to JIT-compiled code without inlining. However, their approach differs from GNFI because they do not provide an alternative to JNI. The approach of Stepanian et al. speeds up

JNI while preserving the portability properties of JNI. The programmer still uses JNI to access native target functions, though their JIT compiler is able to inline the native code of JNI. GNFI provides a way to invoke native target functions from Java directly while improving performance and usability.

The approach used in the Compiled Native Interface (CNI) [2] is based on the idea of making GNU Java compatible with GNU C++. CNI allows writing Java native methods in C++ with zero overhead. CNI uses hooks in G++ (the GNU C++ compiler) so that C++ code can access Java objects as naturally as native C++ objects. The approach of CNI is faster than JNI, but less portable. It differs from GNFI because GNFI provides facilities to call native target functions from Java, thus GNFI completely avoids writing native code.

Truffle/C VM: There are other language implementations on top of Truffle besides the Truffle/C VM. The most sophisticated is the JavaScript implementation [26]. Truffle is designed for dynamic languages and the JavaScript implementation is used as Truffle's proof-of-concept. The performance of the JavaScript implementation is outstanding and it outperforms other JavaScript interpreters on top of the JVM [26].

CINT [9] is related to the Truffle/C VM in that it also is an interpreter for C/C++ code. It is designed for development, where saving compile and link time is more important than run-time performance. The major difference between CINT and the Truffle/C VM is that CINT is a standalone interpretation system while the Truffle/C VM reuses facilities (like GC and memory management) from the underlying JVM.

The Truffle/C VM is well-suited for research towards memory safe C. Nagarakatte et al. [19] present an approach called *SoftBound* to establish spatial safeness of C. SoftBound is based on compile-time transformation to enforce spatial safety of C. This approach records base and bound information for every pointer as disjoint metadata. This enables checking whenever pointer values are loaded or stored. Similar to this approach is the *Memory Save C Compiler (MSCC)* [4]. This compiler ensures both temporal and spatial memory safety in C programs through a source-to-source transformation [4]. While these approaches are based on code transformations at compile time, the Truffle/C VM is able to ensure memory safeness at runtime because the C code is interpreted by a Java application running on top of the JVM. Such memory safeness checks can easily be added to the Truffle/C AST nodes.

Chapter 8

Future Work

So far, the Truffle/C VM supports a subset of the C language syntax as a proof-of-concept. GNFI supports calling arbitrary native functions from Java by providing the GNFI Java calling convention. This chapter mentions potential future work of GNFI and the Truffle/C VM.

Platform: At the moment, GNFI only supports Unix systems and the *AMD64 Application Binary Interface*. Nevertheless, we designed GNFI in a way that makes platform dependent code easily exchangeable.

Library: To improve the usability of GNFI, the GNFI Java calling convention should be hidden from the user. The GNFI Java calling convention is platform dependent, therefore a user-friendly library that wraps it would improve the usability.

Callbacks: One possible extension of GNFI is to support callbacks. Supporting callbacks means that it is possible to pass references to Java methods to native code. The native code can then use these references (function pointers) to perform callbacks, which execute Java code.

Function Pointers: Function pointers are not supported by the Truffle/C VM at its current state. The VM cannot resolve a function pointer of a Truffle/C function that one can pass to native C functions.

Completeness: The current state of the Truffle/C VM supports a subset of the C language syntax. The Truffle/C VM supports six primitive types (`int`, `char`, `long`, `short`, `float` and `double`). Besides these primitive types, the VM also supports structures, unions, arrays and pointers. To support all concepts of the *ANSI C* standard, the VM has to support function pointers and all `unsigned` primitive types (e.g., `unsigned int`).

Memory Safe C: The Truffle/C VM is a great platform for research towards *Memory Safe C*. Memory safeness means that the Truffle/C VM can check array bounds, can do buffer overflow checks and can perform memory access checks at runtime.

Chapter 9

Conclusion

This thesis describes the runtime environment of a virtual machine for the C language on top of the Truffle framework, called Truffle/C VM. The VM offers an efficient way to execute C programs on top of a JVM. It solves two major issues that arise when C code is interpreted on top of a JVM:

The first issue is the need to call native functions (e.g., library functions) efficiently. For this purpose this thesis proposes a new native function interface, called GNFI. Compared to similar approaches, GNFI has advantages in terms of usability and performance. In terms of usability, the programmer does not have to write C/C++ glue code to call native target functions. Thus, the programmer does not have to compile such source files or to link them to libraries. To evaluate GNFI this thesis demonstrates the performance of GNFI for the jblas library, which is a Java wrapper for the native vector and matrix operation library BLAS. The evaluation uses a matrix multiplication benchmark of jblas. The results show that for calling native code from Java, GNFI performs better than JNI and JNA in all relevant cases.

The second issue is a memory model that allows the Truffle/C VM to share data with native code (e.g., library functions). This thesis presents the Truffle/C Memory Model that enables sharing data between the VM and a native code. To evaluate the Truffle/C Runtime Environment, this thesis demonstrates the performance of Truffle/C VM on various C benchmarks. The results show that the execution time of the benchmarks, running on top of the Truffle/C VM, is faster than the execution time of machine code produced by GCC without optimizations. The numbers also show that the performance of the Truffle/C VM is in average a factor of 2 slower than the performance of GCC with all optimizations enabled.

The implementation of the Truffle/C Runtime Environment and GNFI should provide a solid base, including a reusable architecture that makes further work related to the Truffle/C VM simpler.

Acknowledgement

First and foremost, I would like to thank Hanspeter Mössenböck and Lukas Stadler for their constant and valuable feedback on my work. I would also like to thank my colleagues in the Oracle project, especially Thomas Würthinger and Roland Schatz, for countless discussions and for all the feedback on my work.

This work was performed in a research cooperation with, and supported by, Oracle.

List of Figures

Listings

Bibliography

[1] clang: a C language family frontend for LLVM. `http://clang.llvm.org/`, 2013.

[2] cni, the compiled native interface. `http://gcc.gnu.org/onlinedocs/gcj/About-CNI.html`, 2013.

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

[4] MSCC, Memory Safe C Compiler. `http://www.seclab.cs.sunysb.edu/mscc/`, 2013.

[5] M. Baitsch, N. Li, and D. Hartmann. A toolkit for efficient numerical applications in java. *"Advances in Engineering Software"*, 41(1):75 – 83, 2010.

[6] Aart J. C. Bik and Dennis B. Gannon. A note on native level 1 blas in java. *Concurrency: Practice and Experience*, 9(11):1091–1099, 1997.

[7] Brian Blount and Siddhartha Chatterjee. An evaluation of java for numerical computing. In *Computing in Object-Oriented Parallel Environments*, pages 35–46. Springer, 1998.

[8] Craig Chambers, David Ungar, and Elgin Lee. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. pages 21–38. Springer-Verlag, 1991.

[9] J. W. Davidson and J. V. Gresh. Cint: a risc interpreter for the c programming language. *SIGPLAN Not.*, 22(7):189–198, July 1987.

[10] 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.

[11] Vladimir Getov, Susan Hummel, Flynn, and Sava Mintchev. High-performance parallel programming in java: Exploiting native libraries. *Concurrency: Practice and Experience*, 10(11-13):863–872, 1998.

[12] Rob Gordon. *Essential JNI: Java Native Interface*. Prentice-Hall, Inc., 1998.

[13] 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.

[14] 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.

[15] 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*, VEE '05, pages 111–120, New York, NY, USA, 2005. ACM.

[16] Dawid Kurzyniec and Vaidy Sunderam. Efficient cooperation between java and native codes–jni performance benchmark. In *The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*. Citeseer, 2001.

[17] L. Braun, Mikio. jblas. linear algebra for java. `http://jblas.org/`, 2013.

[18] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. 1999.

[19] 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.

[20] Oracle. Graal openjdk project documentation. `http://lafo.ssw.uni-linz.ac.at/javadoc/graalvm/all/index.html`, 2013.

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

[22] 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.

[23] Levon Stepanian, Angela Brown, Demke, 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.

[24] Gang Tan and Jason Croft. An empirical security study of the native code in the jdk. In *Usenix Security Symposium (SS)*, 2008.

[25] Christian Wimmer and Thomas Würthinger. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 13–14. ACM, 2012.

[26] 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, 2013.

[27] 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*, pages 73–82. ACM, 2012.

Curriculum Vitae

Personal Information

Name	**Grimmer, Matthias**
Address	Schloßstraße 6, 4971 Aurolzmünster
Telephone	+43 664 7842152
Email	grimmer_m@gmx.at
Nationality	Austrian
Nationality	March 2 1989
Gender	male

Professional Experience

2012–today	**Student Assistant, Institute for System Software, Linz.**
2009–2012	Tutor of mathematics, Schülerhilfe, Ried im Innkreis.

Education

2009–2012	**BSc in Computer Science, Johannes Kepler University, Linz, Austria.**
2003–2008	Matura, Höhere Technische Bundeslehranstalt Braunau am Inn, Austria.
1999–2003	Bundesrealgymnasium Ried im Innkreis, Austria.
1995–1999	Volksschule Aurolzmünster, Austria.

Other Interests

Research	Compilers and Virtual Machines
Hobbies	Red Cross Austria, Workout, Hiking, Photography

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, am 4. November 2013

Matthias Grimmer, BSc.