Java interpreter program using Truffle

Abstract

This thesis presents an efficient runtime environment for an interpreter implemented in Java. This interpreter uses the Truffle framework and is called the Truffle/Java . A runtime environment for this one has to bridge Java code, running in a Java Virtual Machine (JVM), and native code. This is necessary because Java code, which runs in the Truffle/Java VM (Java), can call any function present as native code (e.g., Java standard library functions). It is not possible to use the Truffle/Java 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/Java 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/Java VM. This memory model offers sharing of run-time data between the interpreter (written in Java) and native code.

1. Introduction

On the first impression, the programming languages Java is imperative, is statically typed and also share many syntactic elements. 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 goal of the Truffle/Java project is to execute Java code on top of a JVM. It aims to implement Java as a guest language on top of the Truffle framework. Truffle 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 Java benchmarks on top of Truffle and compare the performance to Java code compiled using standard compilers like GCC.

2. Goals, Scope and Results

The different aspects of Java raise several challenges when implementing an An interpreter in Java:

- The function handling: The Truffle/Java VM needs facilities to handle any kind of Java function call. There are two types of Java function calls. A Java function executed on top of the Truffle/Java VM can call another function available as a Truffle/Java 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 memory model: A memory model for the Truffle/Java VM has to provide infrastructure tostore and represent run-time data of a Java 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 Java code because pointers can be shared between the Truffle/Java VM and native functions.

3. Truffle/Java

Compiling a program written in Java involves two main steps: First, the compiler compiles the Java 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 Java code to object files, can be split into several steps. First, the Java compiler performs preprocessing. This preprocessing step is responsible for including header files, expanding macros, handling conditional compilation and doing line control. After preprocessing, the The Truffle/ Java Runtime Environment has to provide an underlying memory model for the VM that handles all kinds of memory allocations. The Truffle/ Java needs to allocate memory whenever the executed program instantiates a primitive variable, array, structure or union. The Truffle/ Java Memory Model manages this run-time data. The Truffle/ Java Memory Model has to solve the following issues:

4. Future Work:

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

- Completeness: The current state of the Truffle/Java VM supports a subset of the C language syntax. The Truffle/ Java 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 Java standard, the VM has to support function pointers and all unsigned primitive types (e.g., unsigned int).
- Memory Safe Java: The Truffle/ Java VM is a great platform for research towards Memory Safe Java. Memory safeness means that the Truffle/Java VM can check array bounds, can do buffer overflow checks and can perform memory access checks at runtime.

5. Conclusion

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

The major issue is a memory model that allows the Truffle/Java VM to share data with native code (e.g., library functions). This thesis presents the Truffle/Java Memory Model that enables sharing data between the VM and a native code. To evaluate the Truffle/Java Runtime Environment, this thesis demonstrates the performance of Truffle/java VM on various C benchmarks. The results show that the execution time of the benchmarks, running on top of the Truffle/java 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/java VM is in average a factor of 2 slower than the performance of GCC with all optimizations enabled The implementation of the Truffle/Java Runtime Environment and GNFI should provide a solid base, including a reusable architecture that makes further work related to the Truffle/Java VM simpler.

6. REFERENCES:

[1]http://cesquivias.github.io/blog/2014/10/13/writing-a-language-in-truffle-part-1-a-simple-slow-interpreter/

[2]http://en.wikipedia.org/wiki/Graal_(compiler)

[3]http://www.ssw.jku.at/Research/Papers/Grimmer13Master/Grimmer13Master.pdf

[4]http://ecollection.library.ethz.ch/eserv/eth:8304/eth-8304-01.pdf

[5]Programming Language Popularity. http://www.langpop.com, 2013.

[6] TIOBE Programming Community Index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2013.