Sulong-OpenMP: Implementation with Sulong and Evaluation

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Computer Sciences

2020

Swapnil Laxman Gaikwad
Department of Computer Science
University of Manchester

Contents

Word count 42088

List of Tables

List of Figures

Abstract

It is now popular to use the existing managed language runtimes, such as Java Virtual Machine (JVM) or Microsoft Common Language Infrastructure (CLI), to host implementations of new or existing languages. The benefit of this approach is that the software development efforts are reduced as only one managed language runtime needs to be optimised and maintained. This approach avoids the need for a separate compiler/runtime for each language implementation. The *Truffle* open-source framework has enabled hosting *guest* language implementations on a JVM. Examples of such Truffle hosted guest language implementations are: Ruby (TruffleRuby), JavaScript (GraalJS), R (FastR), and C (Sulong). Truffle guest languages directly benefit from i) JVM features, that include just-in-time (JIT) compilers and Garbage Collection (GC) ii) mature development tools and environments that can support multi-language debugging and instrumentation. In this thesis, we present *Sulong-OpenMP*; the first Truffle hosted implementation that enables execution of the OpenMP parallel programs on a JVM. The current implementation of Sulong-OpenMP supports a sufficient subset of the OpenMP C API to execute the NAS Parallel Benchmark (NPB) suite. The Sulong-OpenMP extends the *Sulong* project, which executes LLVM Intermediate Representation (LLVM IR) for sequential C/C++/Fortran programs on a JVM.

The thesis discusses a novel approach to enable the execution of OpenMP programs on a JVM. We also outline the challenges dealt with during the implementation of OpenMP semantics using the Java Memory Model. This work primarily focuses on achieving the correct execution of OpenMP programs in C on the JVM. Further, we highlight the diminishing performance gap between the native execution (with clang `-O2`) and *Sulong-OpenMP*, on multi-threaded execution. Parallel execution requires supporting sub-systems, such as thread-private stack and thread pools. Although these sub-systems increase implementation efforts, they also aid the execution performance of the implementation. Additionally, we applied more optimisations to minimise the overhead of Sulong-OpenMP. We demonstrate that the single-thread overhead of Sulong-OpenMP (compared to that of Sulong) is on a par with its native equivalent.

Performance is a crucial aspect for both executions and language implementations. While trying to understand the runtime performance of *Sulong*, we lacked the necessary support of tools and guidance. Existing Java profiling tools could not map the profiling information to corresponding C programs executed with Sulong. Execution of the benchmarks written in Ruby and R, using their Truffle hosted implementations, also faced similar issues. A JVM hosted execution utilises multiple services offered by the JVM, such as JIT-compilation and GC. These services may influence the execution behaviour unpredictably. Further, it is non-trivial to relate performance to source-code of the hosted language and determine the source of overhead. The overhead may arise from the program computation, suboptimal implementation of a language feature, or the JVM services.

We describe how to visually analyse the performance of Truffle hosted languages using *Flamegraphs*, by relating execution time to sampled call-stacks. We map sampled call-stacks onto JVM hosted *guest* language source-code using i) Linux tool: *perf*, ii) JVM Tooling Interface (JVMTI) agent: *perf-map-agent*, iii) enhancements to the Graal JIT-compiler. This work demonstrates the ease and flexibility of using these modified tools during execution, with lower overhead. We also illustrate the applicability of the techniques to understand the performance of Polyglot applications.

Declaration

No portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright Statement

Acknowledgements

I would like to thank my supervisors Prof. Mikel Luján and Dr Andy Nisbet for their valuable feedback and academic insights. They have been extremely patient throughout this PhD process.

The labmates of Advanced Processor Technologies (APT) group added *spices* of fun to this journey. Especially, Andrey Rodchenko, Abdullah Khalufa, Anuj Vaishnav, Crefeda Rodrigues, Crystal Wu, Guillermo Callaghan, Josh Lant, Khoa Pham, Moteb Alghamdi, Mark Kynigos, Maxim Schmitt, Merve Şimşek, Nuno Nobre, Osman Seçkin Şimşek, Richard Neil, Salim S. Salim, Serhat Gesoglu and Thanos (the good one) Stratikopoulos. Thank you guys for being there.

I would not have taken this path without a pivotal role of some people. I would like to thank Dr Alin Elena, for guiding me to take this path, and importantly, telling me that *it's the process that matters more than the outcome*. Dr Vidya Durai, for being a constant source of motivation. Gaurav Saxena for being 24x7 stress-relieving helpline over many years. Thank you Gaurav for your advice: *one step at a time!*.

I would like to thank my wife Madhura for her eternal love and rock-solid patience. This journey would have been very tough without you (and certainly without delicious food). I would like to thank my loving parents, Mr Laxman Gaikwad and Mrs Nanda Gaikwad. My parents have always been my strength every step, especially at uncertain times, in this journey.

This research is generously funded by the Kilburn Scholarship and the research donations of Oracle Labs. I would like to thank them both and the school of Computer Science at the University of Manchester for hosting me.

Chapter 1

Introduction

There is no one single programming language suitable for every task. Typically, new languages emerge to solve specific tasks more efficiently than existing languages. Many modern programming languages go through an implementation path, where initially a language interpreter is implemented to execute programs written in that language. As a language evolves (i.e., its specifications and features start getting mature), its performance becomes a priority. To improve the performance of a language, many implementors tend to create a virtual machine (VM) for that language. VM contains one or more compilers that can compile and optimise frequently executing parts of the program. VM uses garbage collectors (GC) to provide a memory management system for a language. The VM model of execution became more popular with the Java language (using Java Virtual Machine (JVM)), that is then adopted by many language implementations, such as JavaScript and .Net Framework. VM implementation requires a significant amount of development effort. Popular JVM implementations, such as the HotSpot JVM, have undergone hundreds of person-years of development efforts for over two decades. Investing such a huge amount of development efforts for building a VM, may not be possible for many programming languages. Alternatively, a language can use the existing VM to execute its programs.

JVM is a popular platform for executing programming languages other than Java. This approach benefits from significantly reduced implementation efforts and competitive runtime performance. JVMs offer these features using their underlying infrastructure, which include Just-in-Time (JIT) compilers and GC. Additionally, use of JVM allows leveraging its extended ecosystem of software support tools, such as, the JVM

profilers for performance analysis and portability to other platforms that are supported by JVM. Groovy, Scala and Kotlin are examples of languages that compile their programs to Java bytecodes and execute them using a JVM. Observing the popularity of hosting language executions on a JVM, Java specifications added the new *invokedynamics* bytecode, as a part of JSR[1]292 [Ora09]. This bytecode aims to increase the ease, and potentially performance, of implementation of the dynamically typed languages on a JVM. Furthermore, the Truffle framework (along with the GraalVM project) [Cor15] and Eclipse OMR (along with Eclipse OpenJ9 project) [Fou16], made it easier to implement a language that then runs on the JVMs.

In this work, we use the language implementation framework: Truffle [HWW+14], which enables us to write an Abstract Syntax Tree (AST) interpreter for a language in Java. These interpreters create AST representation of an input guest language program, which is also in Java; and hence can be executed on a JVM. Currently, Truffle framework offers support for executing languages such as JavaScript (known as Graal.js [Cor16a]), Ruby (known as TruffleRuby [Cor16b]) and R (known as FastR [SWHJ16]). Further, Truffle offers execution of the *polyglot* programs (i.e., the programs written using two or more programming languages), using its language interoperability features[GSS+18].

Although Truffle has been predominantly used to implement dynamically typed languages, statically typed languages can also benefit from JVM hosted execution. The *Sulong* project executes C programs, converted to LLVM Intermediate Representation (IR), on the JVM (discussed in Section 2.5). The *Safe Sulong* project extends Sulong. Safe Sulong can identify bugs and programming errors arising from the *undefined behaviours* in C programs [RSM+18]. It can detect errors such as out-of-bounds access for arrays, `NULL` pointer dereferencing and use-after-free. Safe Sulong implements C data structures using Java data structures. It then uses the ability of JVM to identify the aforementioned bugs.

[1]JSR stands for Java Specification Request (JSR). A JSR request is a formal proposal to add/change the Java language specifications.

1.1 Motivation

This section describes the motivation for the work presented in this thesis, which involves i) extending Sulong to implement support for the execution of OpenMP programs on the JVM, ii) performance analysis approach for the Truffle hosted languages.

1.1.1 OpenMP on a Java Virtual Machine

In the era of multi-core and heterogeneous systems, parallel programming has become an inevitable paradigm for the programming languages. Parallel programming enables software applications to utilise computing capacity of modern systems. Along with its advantages, parallel programs also give rise to programming bugs, such as *data races* that are notoriously difficult to debug. Safe Sulong shows that the JVM hosted implementation can help to find bugs in sequential C programs, which are otherwise difficult to identify using conventional tools and techniques. This motivated us to explore: "*can the parallel executions in statically typed languages, such as C, benefit from the JVM hosted execution?*"

As a first step towards our objective, we decided to build a system that can help identify parallel programming bugs in statically typed languages. To minimise the implementation efforts, we decided to enable execution of parallel programs on Sulong, that already executes statically typed languages on JVMs. Sulong initially supported only the execution of sequential programs. We chose parallel programs written using OpenMP. The OpenMP is one of the popular directive-based parallel programming models for shared memory systems. It is available in C, C++ and Fortran programming languages. OpenMP also offers ways to utilise the heterogeneous system comprising of accelerators. In this thesis, we present our work: Sulong-OpenMP, that extends Sulong to enable execution of OpenMP parallel programs on a JVM. Currently, Sulong-OpenMP support execution of a sufficient subset of C the OpenMP C API to execute the NAS Parallel Benchmark (NPB) suite.

1.1.2 Performance Analysis

A language implementation is typically complemented by the need for its performance analysis. Performance analysis aids the optimisation process by highlighting time-consuming functions. Traditional performance analysis approaches are based on either tracing or profiling. Performance analysis for a managed runtime environment involves monitoring the behaviour of VM services, such as JIT compilation and GC. In the context of guest languages that are hosted on managed runtimes (such as, Truffle hosted languages on the JVM), profiling can help to isolate and measure the performance of specific guest language features. Analysing behaviour of the guest language feature under different use-case scenarios can help to identify, where optimisation is likely to yield larger performance improvements. However, the host language tools face challenges, when used to profile the guest language executions, because they are unaware of non-host language executions. For example, when a sampling profiler for JVM profiles a Sulong-based execution, the profile shows only Java methods. These are the Truffle API methods that are used for implementing Sulong, instead of LLVM IR functions from the guest program. This problem motivated us to explore: "*how can the host language profilers enable to map the collected profiling information to the guest language program's execution?*"

In this thesis, we focus on the call-stack sampling profiling approach for performance analysis for the Truffle hosted guest language implementations. We choose three Truffle hosted languages: FastR (for R), TruffleRuby (for Ruby) and Sulong (for LLVM IR) for evaluating the language implementations and identifying opportunities for performance improvements.

1.2 Contributions

- Chapter 3 demonstrates our approach to execute the OpenMP programs on a JVM. This is accomplished primarily by extending the LLVM IR bitcode interpreter of Sulong and mapping the OpenMP memory model onto the Java memory model. To the best of our knowledge, Sulong-OpenMP is the first approach that enables execution of the OpenMP programs on a JVM.

- Chapter 5 presents evaluation of Sulong-OpenMP using the NAS Parallel Benchmarks (NPB) suite. We discuss performance analysis techniques to identify the bottlenecks, along with the optimisations applied to improve the implementation of Sulong-OpenMP.

- Chapter 6 presents our performance analysis approach that extends the existing Java profiler, to profile executions of Truffle hosted language implementations. We demonstrate the usability of our approach by evaluating the relative performance of three Truffle hosted language implementations: Sulong, TruffleRuby and FastR. Performance of these implementations is compared to that of Java and C, using the shootout benchmarks from *Computer Language Benchmark Game (CLBG)* suite [Guo18]. We also show that the standard tools can be extended to profile Polyglot applications.

Publications

Work presented in this thesis is based on the following peer-reviewed publications.

- **Swapnil Gaikwad**, Andy Nisbet, and Mikel Luján. Performance analysis for languages hosted on the truffle framework. In Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang'18).

- **Swapnil Gaikwad**, Andy Nisbet, and Mikel Luján. Hosting OpenMP programs on Java virtual machines. In Proceedings of the 16th International Conference on Managed Programming Languages and Runtimes (MPLR'19).

1.3 Thesis Structure

Chapter 2 describes various components of Sulong-OpenMP, the underlying system that we extend to execute OpenMP programs. We also describe the use of Truffle framework for language implementation. This prepares a platform for the discussion about extending Sulong to execute OpenMP programs. Execution of Truffle ASTs is crucial to understand the challenges faced by the Java profilers when used to profile the Truffle hosted language execution.

Chapter 3 describes the generation of LLVM IR for the OpenMP programs and the approach for its execution using Sulong-OpenMP. The chapter also dives into describing the implementation of commonly used OpenMP features.

Chapter 4 describes the experimental methodology to measure the performance of Sulong-OpenMP. A JVM contains multiple subsystems providing different services for the hosted execution, which might influence its performance. This chapter discusses the relevant factors for performance evaluation of JVM hosted executions.

Chapter 5 presents the performance evaluation of Sulong-OpenMP using the NAS Parallel Benchmarks (NPB) suite. This chapter also describes the optimisation journey that involves various optimisations applied to improve the implementation of Sulong-OpenMP.

Chapter 6 describes the performance analysis approach for the Truffle hosted language implementations. This chapter describes the challenges faced by the JVM profilers to capture the behaviour of JVM hosted executions accurately. The chapter then presents a profiling approach that addresses these challenges. Case studies demonstrate the applicability of our profiling approach for the guest language executions, as well as their implementation. This chapter also presents the evaluation of three Truffle hosted language implementations on the Computer Language Benchmark Game (CLBG) suite of benchmarks.

Chapter 7 presents the conclusions derived from this thesis, current limitations and the future work for Sulong-OpenMP. This chapter also highlights the next steps to improve the performance analysis methodology, presented previously in Chapter 6.

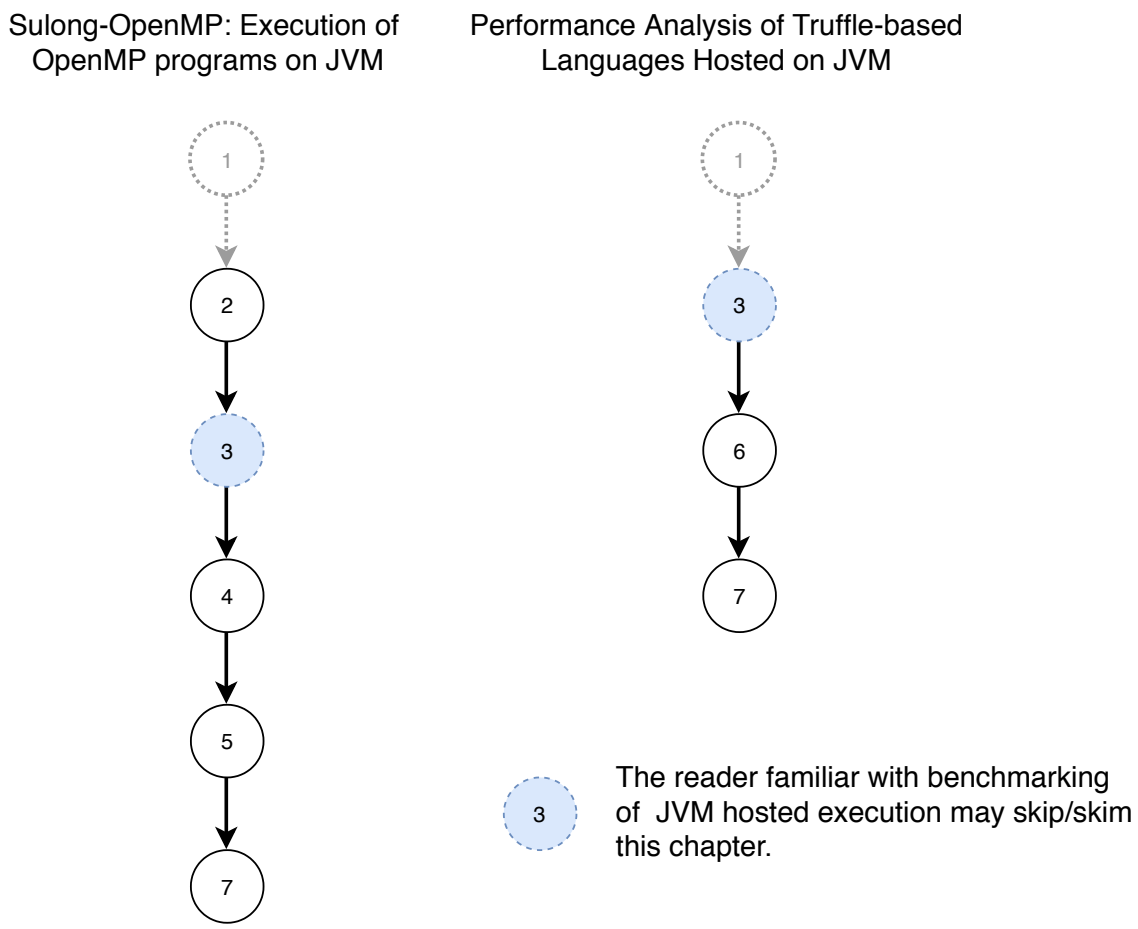


Figure 1.1: Recommended orders for reading this thesis.

Chapter 2

Background

In this chapter, we provide a high-level overview of the underlying system used in this work namely, the Sulong project. This discussion aims to position the work in the ecosystem of the project. We then discuss some of the important components in detail, including a language implementation framework: *Truffle*, the JIT compiler: *Graal*, and the *Sulong* project that is based on Truffle and Graal.

2.1 Overview

The JVM infrastructure has evolved over many years and has hundreds of person-years of effort gone into it. Such a mature infrastructure of JVM can be reused by programming languages other than Java for their execution. JVM offers services such as compilers and garbage collectors, etc., to a *guest language* that it hosts. Examples of such guest languages are Scala [EPFL04] and Groovy[Fou03] that use JVM as a host for their execution. In the case of Scala, the programs are compiled to Java bytecodes that are then executed on a JVM. Here, the JVM is not aware that it is executing a Scala program. As an alternative to the Scala's approach of generating Java bytecodes, one can implement an interpreter for language in Java that then executes the programs on a JVM. However, just as in the case of Scala, the JVM still remains unaware of the guest language that is being executed on it. JVM's unawareness may result in lower performance while executing the guest language programs. This problem can be addressed by using the *Truffle Framework* to write a language interpreter in Java which then executes the guest language programs on JVM [HWW+14]. However, this

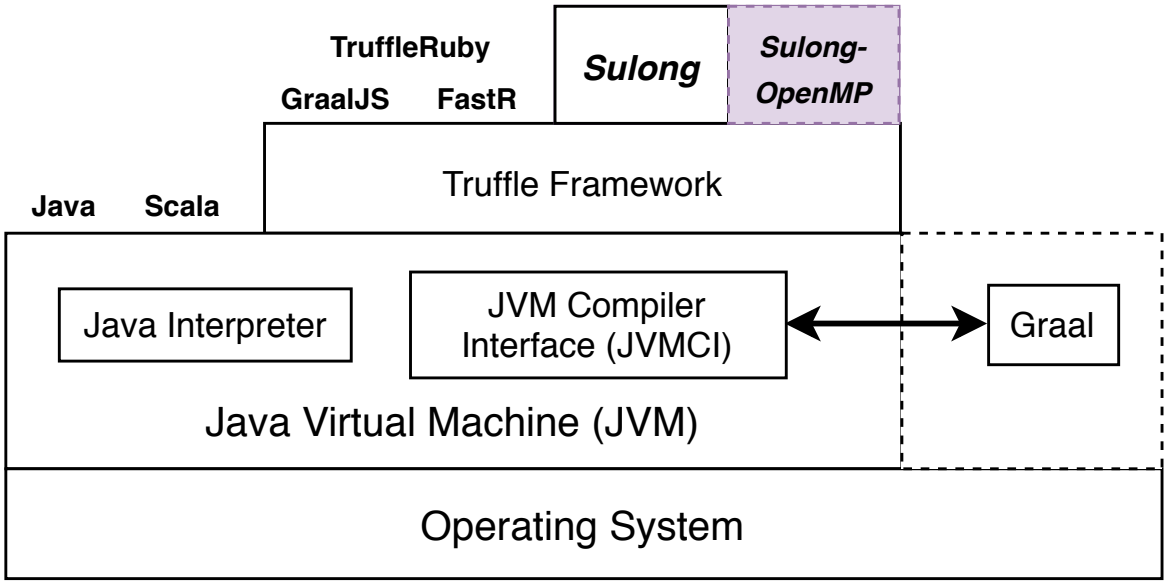


Figure 2.1: Overview of the project ecosystem

time, the JVM can be made aware of the guest language execution and consequently, be advised on how to perform tailored optimisations to improve the performance.

The Truffle framework-based implementations have shown significant performance gains, especially, for the dynamically typed languages such as Ruby (using TruffleRuby), and R (using FastR) [Cor16b, SWHJ16]. The Truffle Framework and the way it helps a language implementation to achieve high performance is explained in Section 2.3.

Sulong is a Truffle framework-based implementation that executes C programs converted into LLVM Intermediate Representation (IR) on JVM [RGW+16]. The internal workings of Sulong are discussed in Section 2.5. In this work, we extend the Sulong project to execute OpenMP parallel programs written in C. The syntax and semantics of OpenMP programs and their execution model are described in Section 2.6. While discussing Sulong, we also cover the schematics of LLVM IR generated for the C programs and the essential associated aspects. Here, we aim to set forth the base for the discussion on how we extend Sulong to execute LLVM IR for OpenMP programs in the chapter that follows.

Figure 2.1 gives an overview of the project undertaken. The highlighted rectangle shows the Sulong-OpenMP project that we present in this work that aims to achieve the execution of the OpenMP parallel programs written in C on top of a Java Virtual

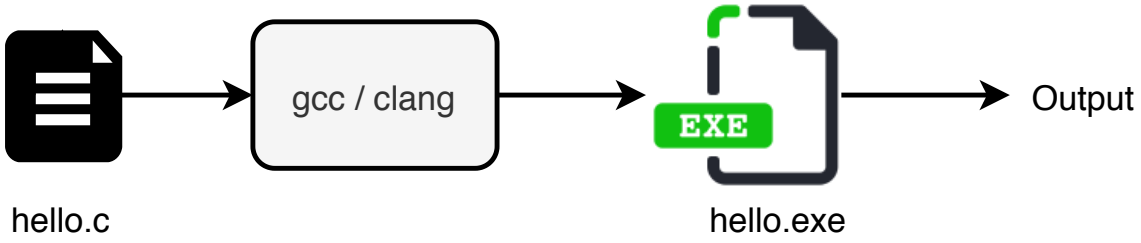


Figure 2.2: Execution based on the ahead-of-time compilation.

Machine (JVM). Rest of the chapter explains the different components of the project in detail. We begin our discussion with the execution of a Java program on a JVM.

2.2 JVM hosted execution

Traditionally, to execute a program, it is first compiled to an executable binary file using a compiler, such as the *gcc* or *clang* as shown in Figure 2.2. Such a binary file would have machine instructions specific to the selected hardware platform so that the binary can be executed on it. Also, it may not be possible to execute the same binary file on other hardware platforms, because the binary contains instructions specific to a certain platform. Therefore, in order to port and run a program on a different platform, the program must be recompiled using a compiler written for that platform.

In contrast, the **JVM** follows a different approach for executing a program. A Java program is first compiled to Java bytecodes - a class file, using a Java compiler *javac* as shown in Figure 2.3. Bytecodes are executed by the JVM, using functionalities provided by a given **Operating System**. The bytecodes are treated as instructions for a JVM, analogous to the machine instructions for particular hardware. However, bytecodes are common across JVMs that enable the class files generated on one machine to be executed on any other machine using the JVM which supports the bytecodes. Thus, the approach makes the compiled Java bytecode files platform-independent and avoids the need for recompiling when migrating to a different platform.

During the execution of a class file, the JVM first executes the program using a bytecode interpreter. The interpreted mode of execution is significantly slower compared to an equivalent program executed using the traditional ahead-of-time compilation mode. The primary reason for such a slow execution is that for each bytecode its

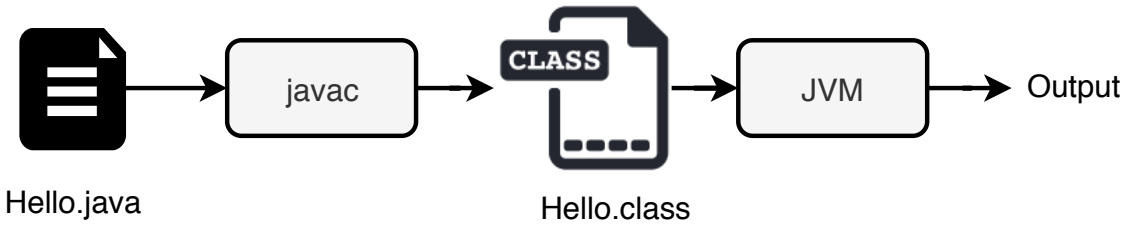


Figure 2.3: JVM hosted execution.

bytecode-handler is executed as an independent function. Further, the optimisation scope is limited to the bytecode-handler. To improve the execution performance of a Java program, a JVM compiles the frequently executed parts of the program during the execution of the program itself. Such compilation is referred to as a Just-in-Time (JIT) compilation.

A typical JVM contains a bytecode interpreter and one or more JIT compilers. The Oracle's HotSpot JVM contains two JIT compilers named as the *Client* and the *Server* compiler [Wik16]. The Client compiler trades-off performance for a faster compilation while the Server compiler does vice-versa. In the case of HotSpot JVM, when a function is executed frequently, it is first compiled by the Client compiler and the compiled version is used for further invocations. The function compiled by the Client compiler is usually faster than the interpreted mode of execution because of its larger scope for compilation. The Client compiler has scope spanning across a Java method, much larger compared to the scope limited to bytecodes for the interpreted version. When the compiled version from the Client compiler is executed frequently, then it is compiled again with the Server compiler. The Server compiler does aggressive optimisations that may take longer compilation time but yield usually the faster version of the function compared to the Client compiler. Using multiple levels/tiers for compilation optimised for different trade-offs is referred to as *Tiered compilation*. The Graal compiler is one of the top-tier JIT compilers that trades-off compilation time for better performance [WWW+13]. Section 2.4 contains a detailed discussion about the Graal compiler and how it optimises the execution of non-Java *guest* languages on a JVM.

2.2.1 Executing non-Java languages on a JVM

Executing non-Java languages on top of a JVM has been a popular approach. Before Java 7, there were over 200 language implementations that used JVM for their execution [Tol19]. As the JVM was originally designed for Java that made it difficult to use it for other language implementations. Rose, *et al.* extensively listed the pain points for implementing a *guest* language on top of a JVM [Ros09]. The difficulties are mainly associated with implementing method invocation mechanisms needed by the guest languages. Considering the popularity of the JVM and the issue for implementing a guest language, Java 7 introduced a new bytecode *invokedynamics* [Ros09]. The new bytecode especially benefited the implementation of dynamically typed languages.

One of the quicker ways to implement a programming language is to build an Abstract Syntax Tree (AST) interpreter for a language. However, the interpreters trade performance for the simplicity and low implementation efforts. The simplicity of interpreters makes it easier to incorporate changes to language behaviour in the early stage of a programming language. Later, when the language starts getting popular and used more, the need for better performance increases. One of the common ways adopted by the language implementors to improve the performance is to embed the interpreters in a virtual machine that has JIT compilers to address the performance needs. On observing such language implementation approaches, Oracle Labs introduced the Truffle framework, explained in the next Section 2.3, to make guest language implementations on a JVM easier.

2.3 The Truffle framework

Truffle is a language implementation framework, written in Java, that can be used to write AST an interpreter for a *guest language* [WWW+13]. One of the examples of such a Truffle-based interpreter is Sulong, which is a building block of the underlying system used in this work. This section explains how a language interpreter is implemented using the Truffle framework.

The interpreters built using the Truffle framework take a guest language program as input, create an AST representation for that program. The AST representation of a program is in Java that can then be executed using a JVM. Each node in the

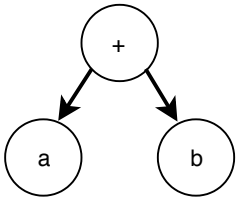


Figure 2.4: AST for simple addition

Truffle-AST contains an `execute()` method that defines the computation associated with the particular node. Therefore, execution of a node is performed by calling the `execute()` method of the node. During the execution of the AST, child nodes are executed before the parent node. A simple example of adding two variables (a and b) is shown in Figure 2.4. The `execute()` methods of the child nodes a and b are evaluated before the `execute()` method of the addition operator. The add operation uses these values to perform addition.

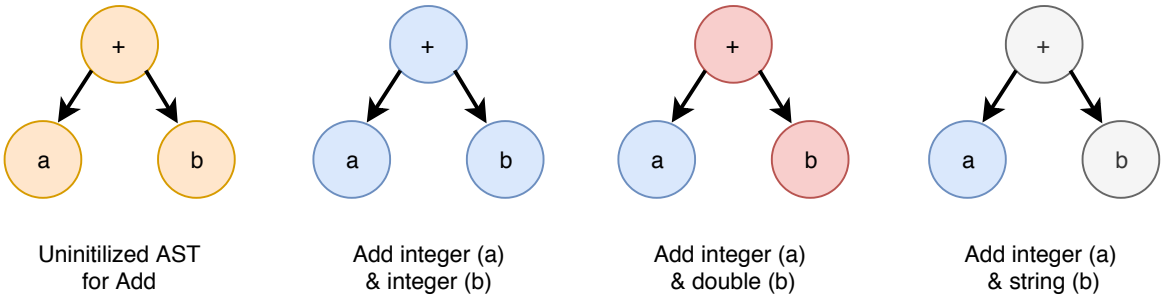


Figure 2.5: Specialisation of Truffle AST nodes based on the data types of the operands

The Truffle framework helps to build self-optimising AST interpreters where the nodes of an AST are specialised based on the observed behaviour of a program, such as the data types of variables. The add operation is usually polymorphic in dynamically typed languages that can add two integers, floating-point numbers, strings or any combination of them. Thus, the Truffle based implementation of an addition operator provides mechanisms for adding all the permissible combinations of the data types. When the Truffle AST showed in Figure 2.4 is executed for the first time, the nodes of the AST are specialised based on the data types of the variables a and b. The add operator is then specialised based on the data types of the children nodes as shown in Figure 2.5. When both the children are of type integer, the add node is specialised with the implementation that adds two integers. Similarly, when b is of type double, the add

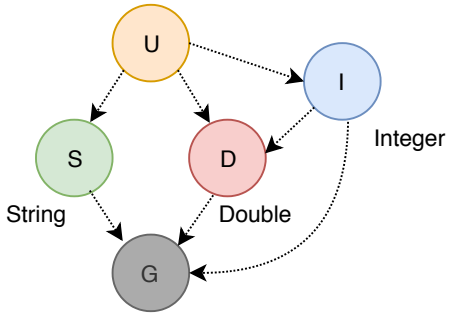


Figure 2.6: Type specialisation lattice where the re-specialisation moves towards more generic type.

node selects the implementation that performs addition by typecasting a to a double value and returns the result as with type double. Such specialisation of nodes stores the observed behaviour of the program in the AST itself. The specialised implementation keeps track of the previously observed types and its corresponding implementation. This helps the interpreter to select a suitable implementation of an operation when the same expression is evaluated again. Further, the observed behaviour of a program acts which is crucial for optimising the AST during its JIT compilation (explained in Section 2.4).

During the interpretation phase, nodes of the Truffle-AST are re-specialised when the observed behaviour of a program changes. In the above addition example, the node for variable a gets re-specialised when it holds a value of type double which previously held the value of integer type. To avoid continuous re-specialisation, the specialisations are implemented in a way that converges. For example, in the case of type specialisation, the re-specialisation moves towards a more generic type. Figure 2.6 shows such a type re-specialisation lattice where a node may be re-specialised from an uninitialised state to an integer, double or string type. The node specialised to a double type would not be re-specialised to an integer type even when the later invocation is performed with the integer type. The generic specialisations are implemented to handle more cases including all the previous specialisations.

To make a language implementation easier, Truffle provides a Java annotation-based Domain Specific Language (DSL). The annotations are used to specify the classes implementing nodes of an AST and methods implementing node specialisations. Truffle DSL also provides annotations for creating inline caches (IC) that allows storing

the observed method implementations in a localised look-up table at the call site to accelerate the selection of the correct implementation. ICs are useful, especially, for languages such as Ruby where the method implementation could be overridden during the runtime. The use of annotations significantly reduces the amount of boilerplate code that needs to be written by the language implementor. Truffle includes an annotation processor that creates new classes that extend the annotated classes along with the required boilerplate code to handle tasks such as handling specialisation of a node and adding IC for a node. There are many useful functionalities provided by the Truffle framework for a language implementation that are not discussed further in this thesis.

Typically, during the execution of an AST for a function, the AST *stabilises* for the given input, i.e., it stops re-specialising after a few invocations. A stabilised AST is then fed to the optimisation pipeline of a JVM where one of the JIT compilers can optimise the AST. JVMs usually contain multiple JIT compilers that focus either compilation time or performance of the generated code. The next section describes a performance-focused JIT compiler that we use in this work — Graal.

2.4 The Graal Compiler

In this section, we discuss the optimising JIT compiler Graal and some of the optimisations techniques that are important for this work. The *Graal* compiler is a top-tier JIT compiler similar to the Server compiler in the HotSpot JVM that trades off compilation time for higher performance. In fact, Graal can be used in place of the Server compiler in the HotSpot JVM. Graal originated from the C1X compiler of the Maxine project [WHVDV+13]. The Maxine VM is an implementation of the JVM in Java itself. It is a *metacircular* virtual machine, i.e., the VM is written using the same languages that it implements. Having a JIT compiler written in Java made it possible for a Java program to communicate with the compiler. In the case of Graal, it implements the JVM Compiler Interface (JVMCI) introduced using the JDK Enhancement Proposal-243 (JEP-243) [Ros19]. The JVMCI enables Truffle-generated ASTs to use the compiler as a service.

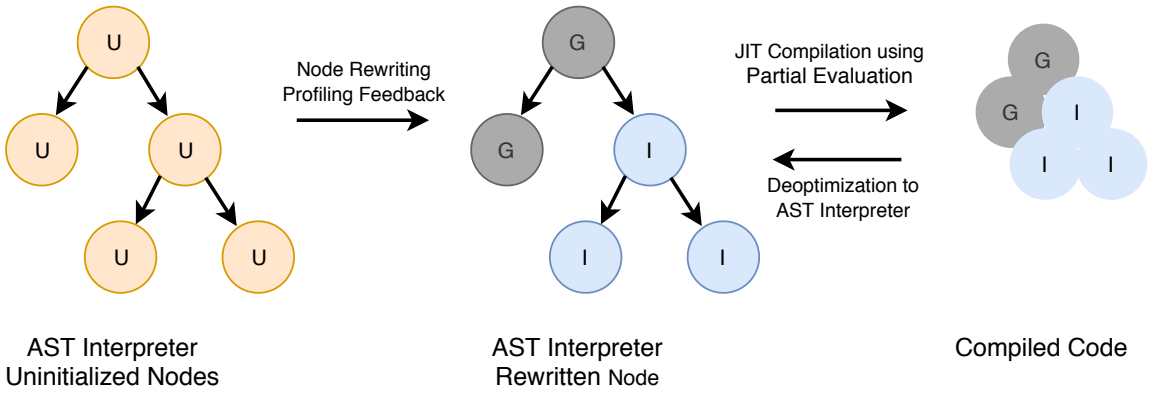


Figure 2.7: AST specialisation. Image reproduced from [HWW+14]

Using Graal, the compiled version of the Truffle-based ASTs can achieve higher performance compared to the compilation using other JIT compilers [WWW+13]. Not only the Truffle-ASTs, but Graal can also be used to compile the Java programs as well. However, while optimising the specialised Truffle-based ASTs as shown in Figure 2.7, Graal uses an additional optimisation technique called *partial evaluation*. During the partial evaluation, usually, all the methods implementing the nodes of the AST of a function are inlined into a single function that is then fed to the compilation pipeline. This extent of inlining, measured in inlining depth, performed by Graal is much higher than the inlining depth used by typical JIT compilers that allows Graal to prepare a larger compilation unit for the further optimisation stages [WWW+13]. The larger compilation units allow Graal to generate highly optimised code compared to other JIT compilers. Along with inlining, Graal also executes code as much as possible using the profiling information collected during interpretation. The execution involves applying constant folding, dead code elimination and converting virtual method calls to direct calls to a known method name [WWH+17]. This process gives the optimisation its name — *partial evaluation*.

In addition to the partial evaluation, another notable optimisation that Graal performs is *Partial Escape Analysis*[SWM14]. Escape Analysis is a well-known optimisation performed by compilers where the allocation of the object is avoided if it is not escaping the function scope [Bla03]. In the conventional escape analysis, when an object is considered as escaping then it is allocated, even if the object is escaping only when one of the branches is taken during the execution flow. Graal extends the

conventional escape analysis and allocates objects on the branches so that the allocation is avoided on the branches where the object does not escape. This optimisation is effective, especially, when a significantly large number of objects are allocated.

When Graal receives a request to compile a method, first, the bytecode interpreter generates a graph in the sea-of-node representation for the method. The sea-of-node representation simplifies the intermediate representation and the compiler implementation [CP95]. Therefore, it is used by modern JIT compilers such as the Server compiler in HotSpot and the V8 in Google Chrome. After partial evaluation, the Truffle AST goes through the optimisation pipeline performing conventional optimisations to generate platform-agnostic representation named as High-level Intermediate Representation (HIR). Subsequently, the platform-specific optimisations are performed when the HIR is converted to a platform-specific Low-level Intermediate Representation (LIR). At the time of writing this thesis, the performance of the code generated by Graal is at par with or exceeding the performance of the code generated using the HotSpot's Server compiler [Cor20].

Graal applies speculative optimisations to generate efficient machine code that gets *deoptimised* when the underlying assumptions no longer hold. Deoptimisation switches execution of a method from the fast compiled mode to the slow interpreted mode. During optimisation, Graal assumes the AST of a method is stable and generates code only for the specialised AST. For example, when an add operation in a dynamically typed language is always performed on two integers, the Truffle AST for that operation gets specialised for add operations on integers, and Graal generates machine code for the integer add operation. Graal also puts guard conditions to ensure that the operands are of integer type. When the addition is performed with one of the operands with non-integer type, such as float, the underlying assumption about both operands being integer fails. This makes the optimised version of method invalid and the execution can no longer continue using it. Thus, JVM switches execution to the interpreted mode by triggering deoptimisation. While executing in the interpreter mode the AST for the method gets re-specialised to accommodate the floating-point parameter. The re-specialised method gets optimised again when it is called more times than the compilation threshold. Deoptimisation is an expensive operation, it needs to construct the interpreter frame representing the current execution state of the method in the

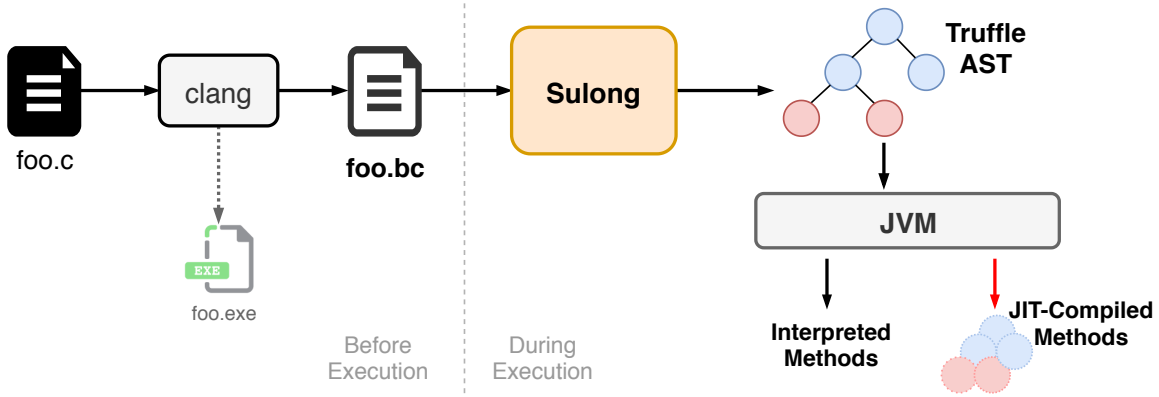


Figure 2.8: Execution workflow of Sulong.

compiled mode and resume execution in the interpreter mode in a transparent way to the user.

We do not make changes to internal workings of Graal to benefit the Sulong-OpenMP implementation. Thus, we limit our discussion on Graal in this chapter. However, we will discuss on the compilation of Truffle-hosted guest language functions in Chapter 6.

2.5 Project Sulong

Sulong is an interpreter implemented using the Truffle framework that runs LLVM Intermediate Representation (IR) on JVM. In this section, we begin by describing LLVM IR that is necessary for the subsequent discussion on the Sulong project.

2.5.1 LLVM IR

LLVM IR is an intermediate representation used by the compilers from LLVM project, such as *clang* for C and *clang++* for C++ programs. These compilers contain a language-specific front-end that generates the language-agnostic intermediate representation (LLVM IR). LLVM IR generated by a compiler front-end is optimised using the platform-agnostic optimisations. The compiler back-end generates a platform-specific executable binary code from the optimised LLVM IR. The intermediate representation separates compiler front-ends from their back-ends. The clang compiler can generate optimised LLVM IR as well as a binary executable for C programs. Henceforth, whenever we refer to clang in the context of LLVM IR or Sulong, we refer to its

C front-end that genereates LLVM IR. Listing 2.1 and Listing 2.2 shows the C code and its LLVM IR respectively.

```
1  int foo(int i) {
2      return (i * i) + 42;
3  }
```

Listing 2.1: A simple C function `foo` that returns the square of the input argument plus the constant number 42.

```
1  define i32 @foo(i32) {
2      %2 = mul i32 %0, %0
3      %3 = add i32 %2, 42
4      ret i32 %3
5  }
```

Listing 2.2: An LLVM IR for the function `foo` in Listing 2.1.

The LLVM IR for a program is a sequence of basic blocks which consists of instructions in an assembly-like language. Each basic block is a sequence of instructions where control flow enters at the first instruction and exits after executing the last instruction. There can be no branch instructions in the middle of the basic block. Listing 2.2 has only one basic block that starts on line 2 and ends on line 4 with the return instruction. The `mul` instruction squares the first argument and `add` instruction increments its result by the constant 42.

LLVM IR uses the Static Single Assignment (SSA) form where value is assigned to a variable exactly once [TC07]. The `%2` and `%3` in Listing 2.2 are such stack-allocated local variables that store the intermediate results. Use of the SSA form simplifies applying compiler optimisations, such as dead code elimination, constant propagation and global value numbering [ALSU06]. LLVM IR abstracts many source language and target-specific details, such as the calling convention.

2.5.2 Sulong

A high-level execution workflow Sulong is shown in Figure 2.8. Sulong takes LLVM IR as an input and generates the Truffle AST for the program that is then executed

on a JVM. By targetting LLVM IR, theoretically, Sulong can run languages that have compiler front-ends to generate LLVM IR without writing a separate parser and interpreter for each of those languages.

Use of LLVM IR allows Sulong to benefit from the static optimisations provided by the LLVM infrastructure as well as the dynamic optimisations from the Graal compiler. Static optimisations can be applied to LLVM IR using the *opt* tool [LLV03]. Sulong benefits from the following dynamic optimisations performed by Graal [RGW+16]:

- **Polymorphic Inline caches:** Sulong uses polymorphic inline caches to optimise calls using function pointers. IC for a function call using a function pointer is a local cache of the observed values for that function pointer. They are stored in an AST node at the call-site that performs the call. At the next execution, the call-site checks the target function address with cached addresses and makes a direct call to the function when a match is found. If the call-site exceeds the specified inline cache limit, the indirect call is performed.

- **Value profiling:** The value profiling technique is used to identify the run-time invariants that cannot be identified at compile time [CFE97]. For example, global configuration variables that are initialised at the beginning of the program and do not change later. Sulong uses value profiling to check the values loaded from memory that can be a primitive or a pointer. If the loaded value is unchanged then the load node is replaced by a node that performs a check if the loaded value is still the same and returns the cached constant. While compiling such a node, Graal can speculate that the value would remain constant.

- **Dynamic dead code elimination:** Sulong maintains the probability for the successor basic blocks in a given basic block. Graal uses the successor probability to avoid compiling basic blocks that are never executed. This enables dead code elimination during execution for the untraversed branches.

- **Inlining at run-time:** Inlining done by Graal benefits from the profiling feedback collected at run-time, such as function call counts, that can help to perform better inlining decisions. Thus, one may also defer function inlining to the run-time by disabling it during static compilation using LLVM.

The LLVM IR can be generated in the textual (as .ll files) or binary bitcode format (as .bc files). Sulong uses the bitcode parser to parse the LLVM IR in the binary format. In the binary format, the .bc file contains a bitstream for an LLVM IR. LLVM IR bitstreams are self-defined, i.e., how a stream of bits should be interpreted is defined by the stream itself. For example, the first two bits would decide the data type of the following content which would then determine the number of bits of the remaining bitstream that represent the actual data of that type.

At the beginning of this PhD project, I extended the Sulong bitcode parser to process records of binary `BLOB` type. An LLVM bitcode file is organized as a sequence of nested blocks containing records. The bitcode parser reads these blocks of records while parsing an LLVM IR file. Records are stored in either textual (.ll files) or in binary large object (`BLOB`) format (.bc files). The `BLOB` records are used to store the debugging information at the end of the IR. This work was added to support the latest LLVM 3.9 bitcode format available at that time.

We will continue our discussion on the parts of Sulong relevant for the implementation and performance improvements of Sulong-OpenMP in Chapter 3 and Chapter 5 respectively.

2.6 OpenMP

OpenMP is a popular directive-based parallel programming approach for shared memory systems available in C, C++ and Fortran languages. In this PhD work, we have extended the Sulong project to execute OpenMP parallel programs on a JVM. OpenMP provides two types of parallel programming models namely work-sharing model and task-based model. In the case of the work-sharing model, iterations of a for-loop are collaboratively executed by the specified number of OpenMP threads. In the case of the task-based model, one or more OpenMP threads create *OpenMP tasks* that are then executed by the OpenMP threads. We would focus on the work-sharing model of OpenMP, primarily for-loops.

The OpenMP work-sharing model is based on the fork-join parallel pattern, as shown in Figure 2.9. Here, threads are forked at the beginning of the program block annotated with OpenMP pragma directives and they are joined at the end of the

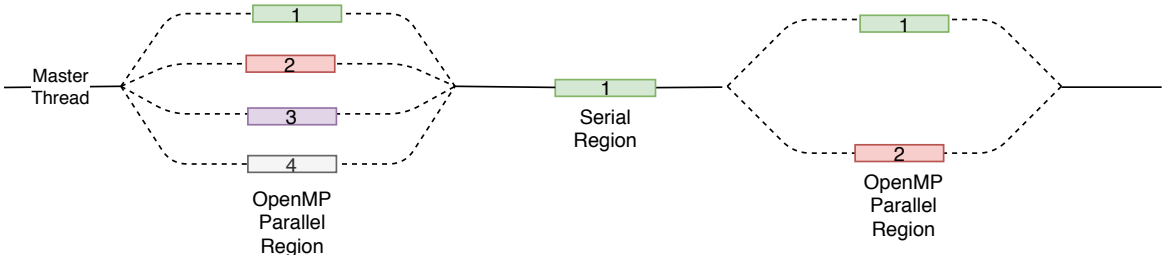


Figure 2.9: OpenMP Fork-Join model.

block. Listing 2.3 shows a simple OpenMP program that contains an OpenMP block annotated with the pragma on Line 3. Each thread executes the OpenMP annotated block in parallel and prints the greeting message. OpenMP provides functions to retrieve runtime information, such as the current thread ID, total number of OpenMP threads, etc., that can be used to control the behaviour of a thread. A program may contain one or more OpenMP parallel blocks.

```
1  void main() {
2    // Code before OpenMP block ...
3    #pragma omp parallel
4    {
5        printf("Hi thread %d\n", omp_get_thread_num());
6    }
7    // Code after OpenMP block ...
8  }
```

Listing 2.3: A simple OpenMP C code where each thread prints a welcome message and its thread ID.

`#pragma omp for` distributes the iterations of the loop amongst the threads in the enclosing parallel region. There are three main ways, referred to as *OpenMP schedules*, to distribute loop iterations amongst OpenMP threads: static, dynamic and guided schedule. Static schedule divides the iteration space equally amongst given OpenMP threads. Dynamic schedule assigns an iteration to each thread and whichever thread finishes first, gets the next iteration assigned. Guided schedule divides the iteration space based on the number of unassigned iterations and the available number of threads. Subsection 3.3.2 discusses the OpenMP schedules in more details. Table A.1

OpenMP Version	Release Date	Comments / Key Features
1.0	Oct 1997	Support for only Fortran
1.0	Oct 1998	Added suppport for C and C++
2.0	Nov 2000	Support for only Fortran
2.0	Mar 2002	Added suppport for C and C++
2.5	May 2005	Support for C, C++ and Fortran
3.0	May 2008	Introduced support for Task construct
3.1	Jul 2011	Improved support for reduction and atomic constructs. Added support for optimisations within the OpenMP tasking model
4.0	Oct 2013	Added support for accelerator constructs, error handling, actomics and SIMD constructs. Extended support for Tasks
4.5	Nov 2015	Improved support for accelerator constructs
5.0	Nov 2019	Added support for C11, C++14/17, and Fortran 2008. Improved support accelerator constructs and Task dependencies

Table 2.1: Description of OpenMP specification releases dates and the notable features offered by them [ARB20].

and Table A.2 in Appendix A contains the list of LLVM's OpenMP runtime library functions that are implemented in Sulong-OpenMP.

The front-end for C programs, clang, can generate LLVM IR for OpenMP programs. From the version 3.8.0 of clang (released in 2016), support for OpenMP 3.1 (released in 2011) is available [LLV18]. The latest version 9.0.0 of clang (released in 2019), fully supports OpenMP 4.5 features (released in 2015), with limited support for offloading to the accelerators. Table 2.1 provides the release history of OpenMP and their key features. The subsection describes the schematics of LLVM IR for OpenMP programs generated by clang. The OpenMP specification provides many features. The implementation of Sulong-OpenMP currently supports a subset of features available in OpenMP 2.0 specialisations. Some of the notable unsupported features are OpenMP tasks, sections, offload directives and nested parallelism.

2.7 Summary

This chapter introduced important components of the Sulong-OpenMP project including the Java Virtual Machine (JVM), the Truffle Framework, the Graal compiler and the Sulong project. Then we discussed the directive-based parallel programming approach, OpenMP.

We enable execution of OpenMP parallel programs on the JVM using the Sulong project. The Sulong project takes an LLVM Intermediate Representation (LLVM IR) of a C/C++/Fortran program as input and executes on a JVM. Sulong is an interpreter for LLVM IR built using the Truffle framework that can be used to implement an Abstract Syntax Tree (AST) interpreter for a guest language. Truffle is written in Java so the guest language ASTs built using Truffle are executed using a JVM. The Truffle-AST is specialised using run-time information, such as observed data type, which helps the JIT compiler to generate optimised code for only the observed behaviour of the program. When the Truffle-AST is executed using a JVM containing Graal as a top-tier JIT compiler, it can achieve higher performance compared to a JVM without Graal. One of the main reasons why Graal can produce code to achieve high performance is because it performs partial evaluation on the Truffle-AST that involves aggressive inlining of the AST nodes comprising a guest language function.

In the second part of the chapter, we discussed OpenMP's fork-join based work-sharing model. OpenMP achieves work-sharing by distributing iterations of a loop amongst multiple threads for execution. In the next chapter, we will continue the discussion on OpenMP, its commonly used features and their implementation on Sulong.

Chapter 3

Implementation of OpenMP on Sulong

In the previous chapter, we discussed the generation of LLVM IR for a sequential C program and its execution using Sulong. This chapter discusses how Sulong-OpenMP extends the Sulong project to enable the execution of LLVM IR for OpenMP parallel programs on top of the JVM. We begin our discussion by describing the LLVM IR generated for an OpenMP program, how it differs from the sequential version of the same program (when the OpenMP support is not enabled), and how the changed LLVM IR achieves parallel execution of the OpenMP blocks in the source program. This discussion highlights the functionalities required to enable the execution of LLVM IR for OpenMP parallel programs using Sulong-OpenMP. We then discuss the potential approaches to implement the OpenMP support to Sulong, their trade-offs, and the approach implemented by Sulong-OpenMP. We then describe the implementation of some of the important OpenMP features. Finally, we discuss other implementation approaches that provide OpenMP-like directives for Java programs that are then run on a JVM.

3.1 LLVM IR for OpenMP

To enable execution of LLVM IR for OpenMP programs using Sulong, it is important to understand how they execute natively. Clang generates LLVM IR for OpenMP program that contains calls to its OpenMP runtime library. Furthermore, the generated

LLVM IR is tightly coupled to the functionality of those runtime functions. Therefore, we first discuss the layout of the LLVM IR generated for OpenMP programs to achieve desired parallel execution. In this section, we take a look at the role of the OpenMP runtime to achieve parallel execution of OpenMP blocks.

```
1  void main() {
2    // Code before OpenMP block ...
3    #pragma omp parallel
4    {
5        printf("Hi thread %d\n", omp_get_thread_num());
6    }
7    // Code after OpenMP block ...
8  }
```

Listing 3.1: A simple OpenMP code (in C) where each thread prints a welcome message and its thread ID.

```
1  define i32 @main() {
2    ; LLVM IR for code before OpenMP block
3    call @__kmpc_fork_call(...,
4              void (...)* @.omp_outlined), ...)
5    ; LLVM IR for code after OpenMP block
6  }
7  define void @.omp_outlined() {
8    ; LLVM IR for actual OpenMP block
9    call @printf()
10   ...
11 }
```

Listing 3.2: A simplified snippet of LLVM IR generated for the OpenMP program from Listing 3.1 using clang.

Listing 3.2 shows simplified LLVM IR for the OpenMP program in Listing 3.1. An LLVM IR for the @main() function (Line 1-6 in Listing 3.2) resembles the `main()`

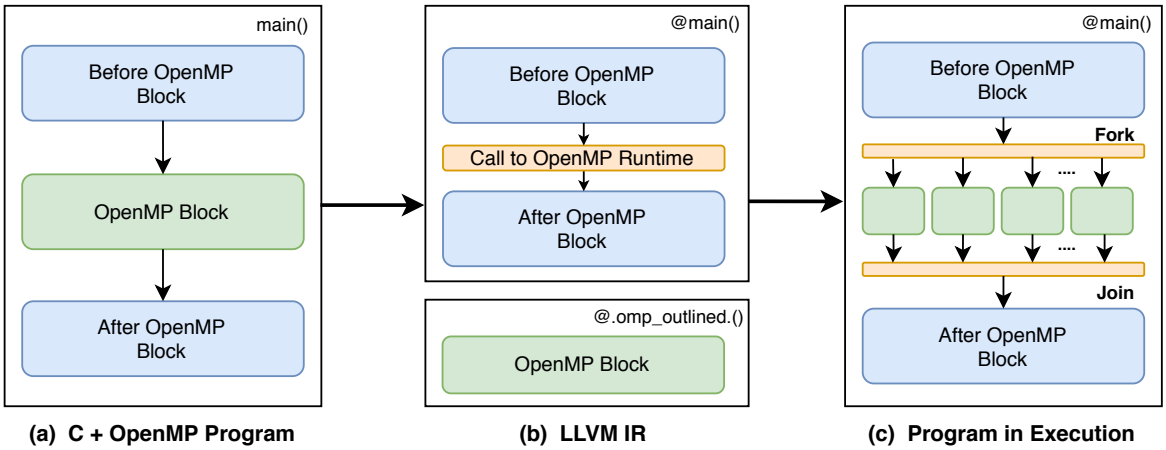


Figure 3.1: Block diagram of an OpenMP program: (a) source code, (b) LLVM IR, and (c) execution model. Calls to OpenMP runtime functions are in orange.

function (Line 1-8 in Listing 3.1) in the source program. However, instead of having IR for the OpenMP block within the main function, it is replaced by a call to the OpenMP runtime function `@__kmpc_fork_call` (Line 4 in Listing 3.2). This runtime call takes a pointer to the `@.omp_outlined` function as an argument that contains the actual LLVM IR for the OpenMP block, including a call to the `printf` function (Line 9 in Listing 3.2). LLVM IR for the OpenMP blocks is wrapped in the functions with their names starting with *omp_outlined*. This process of extracting a part of the function, is opposite of the function inlining transformation done by compilers.

Figure 3.1 shows a block diagram of the various sections the OpenMP program and their interactions: (a) in source format, (b) in LLVM IR format, and (c) during the execution. Figure 3.1(a) represents the block structure of `main()` function in Listing 3.1, where the green rectangle represents an OpenMP block and the outer black boarder represents the `main()` function. Figure 3.1(b) shows LLVM IR generated for the `main()` function in (a). The LLVM IR for the OpenMP block is wrapped into a separate function, shown as a separate black rectangle representing the outlined function. The OpenMP block in the `main()` function is now replaced by the OpenMP runtime function, shown as the orange rectangle. Figure 3.1(c) shows the execution of the LLVM IR. OpenMP runtime function (in orange), creates required number of OpenMP threads. Each of these threads then execute the parallel block (in multiple green rectangles). OpenMP runtime function (in orange) ensures merging of threads after they finish execution of the parallel block.

3.2 Implementation Approach

Now we discuss three main approaches, that we considered to implement Sulong-OpenMP: PThread-based approach, Function Morphing approach, and the Hybrid approach. These approaches take the LLVM IR for an OpenMP program as shown in Figure 3.1(b), and execute it on a JVM as shown in Figure 3.1(c).

The sequential version of Sulong (without OpenMP support), fails to execute the LLVM IR for OpenMP programs. Sulong cannot find implementation of the runtime function `@__kmpc_fork_call`. If the implementation of this function is made available to Sulong, it can execute the OpenMP program. To achieve the expected behaviour, Sulong needs implementation of the `@__kmpc_fork_call` function that matches the implementation of the corresponding OpenMP runtime library function of clang. In this case, the implementation is expected to behave as shown by orange rectangles in Figure 3.1(C).

Therefore, the implementation approaches primarily differ in ways the OpenMP runtime functions are implemented and made available to the Sulong, during the execution. We begin our discussion with the Pthreads-based approach in the section that follows.

3.2.1 Pthreads-based approach

POSIX threads, also referred to as pthreads, is an implementation of the standardised thread API specified by IEEE POSIX 1003.1c standard [IEE95]. Programs written using the pthreads library functions can be executed on any other hardware platform using the platform-specific pthreads library implementation. The pthreads API offers functions to perform thread management and synchronisation.

The OpenMP runtime library of clang uses `pthreads` API for implementation, e.g., the implementation of fork function uses the pthreads API to spawn the required number of threads, that are then assigned the outlined OpenMP function for execution. The synchronisation constructs in OpenMP use the underlying pthreads API-based functionalities, such as mutexes and barriers, for their implementation. Therefore, if Sulong provides the implementation for the pthreads functionalities, we can use the OpenMP runtime library implementation of clang. This would require the OpenMP

runtime library compiled to its LLVM IR format, to execute OpenMP programs. The compiled version of the runtime library can be provided as an external library using the existing mechanism in the Sulong. Thus, the Pthreads-based approach would use the existing implementation of clang for the OpenMP runtime functions, such as `@__kmpc_fork_call`. Consequently, this would enable executions using Sulong to match the behaviour of native executions. The reuse of OpenMP runtime library would provide complete feature compatibility for OpenMP with the specific version of clang.

We did not opt for the Pthreads-based approach to implement Sulong-OpenMP because of three challenges. i) Sulong did not support pthreads at the time of implementation of Sulong-OpenMP. Therefore, we would have required to add support for pthreads to Sulong. We found that implementing support for a subset of pthread functionalities necessary to run a simple OpenMP program, such as Listing 3.1, would require significantly large efforts compared to the alternative approaches discussed in the following subsections. ii) This approach would have involved identifying and compiling the OpenMP runtime library source files along with their dependencies to LLVM IR, that are required to utilise pthreads support. We considered this task cumbersome compared to alternative approaches. iii) We considered the Pthreads-based approach less flexible for the future experiments with the OpenMP runtime, e.g., duplicating the Truffle-AST of an OpenMP block to achieve thread-behaviour specific JIT-compilation, that may benefit the programs exhibiting the master-slave or the producer-consumer pattern.

3.2.2 Function Morphing approach

The function morphing approach originates from the necessity to determine the sequence of OpenMP features to implement in Sulong-OpenMP. The OpenMP specification offers multiple sets of features, such as loop parallelism, tasks, and offloading to accelerators directives. Supporting all the features is a huge task, that requires to determine the order of implementation of the OpenMP features beforehand. We decided the order of implementation by focusing on the subset of OpenMP features that are necessary to run the benchmarks from NAS Parallel benchmarks (NPB) suite. Further, we plan to continue this process to increase completeness incrementally. The total number of OpenMP runtime library functions that are used in the entire suite

is significantly small which made the Function Morphing based approach easier to implement compared to the Pthreads-based approach.

The Function Morphing approach is based on the way Sulong executes an LLVM IR function. Sulong maintains a registry of the function definitions that are observed while parsing the LLVM IR. This registry is a map of key-value pairs, where the key is a name of an LLVM IR function and its value is the Truffle-based AST for the function. When a function is invoked, the map is searched and the corresponding Truffle-AST is called. If the function is not found in the map, the `LLVMLinkerException` exception is thrown. When an OpenMP program is executed on Sulong (without OpenMP support), the same exception is thrown while calling the OpenMP runtime library functions. Thus, we can specify a different function, our implementation for the function, to be invoked when a missing OpenMP runtime library function is called. When the custom function behaves the same as the original function, execution of the new program matches the original program. We refer to such custom function as the *morphed* function and the approach based on this technique as the *Function Morphing approach*. We implement the OpenMP runtime functions, in Java, and add such implementations to the registry. Therefore, when a runtime function is called, the *morphed* implementation gets invoked.

The Function Morphing approach has several advantages over the PThread-based approach. First, it is much simpler to implement. For example, the approach required to implement just two OpenMP runtime library functions, namely `@__kmpc_fork_call` and `@omp_get_thread_num`, to execute the simple OpenMP program in Listing 3.1. Also, this approach provides the OpenMP implementation with additional flexibility because it provides control over creating OpenMP threads and assigning the Truffle-AST of the function representing the OpenMP block.

3.2.3 Hybrid approach

A Hybrid approach is a combination of the Pthreads-based and Function Morphing approach. The hybrid approach uses the OpenMP runtime library functions written in both, C and Java, similar to the Pthreads-based and Function Morphing approach respectively. The C functions are converted into LLVM IR and made available as external library functions.

The Function Morphing approach works for most of the runtime functions, that are used to implement the OpenMP directives. However, some of the OpenMP runtime functions need to read and/or write to local variables of the program. This becomes cumbersome to implement using the Function Morphing based approach. We explain the challenge for Function Morphing with the parallelised for-loop example shown in Listing 3.3.

```
1       void main() {
2         #pragma omp parallel for
3         for(int i = 0; i < N; i++ ) {
4            // code in the parallel block
5         }
6       }
```

Listing 3.3: A for-loop in C where iterations of the loops are divided amongst OpenMP threads for execution.

```
1    define i32 @main()  {
2        ...
3        call @__kmpc_fork_call(..., @.omp_outlined(...)), ...)
4        ...
5    }
6
7    define void @.omp_outlined(...) {
8        ...
9        call @__kmpc_for_static_init_4(...)
10       ...
11       ; LLVM IR for the parallel block
12       ...
13   }
```

Listing 3.4: Simplified LLVM IR for the OpenMP program in Listing 3.3.

The parallelised for-loop in Listing 3.3 divides the iteration space of a for-loop nearly-equally amongst the OpenMP threads. Listing 3.4 shows the simplified LLVM IR for the C code in Listing 3.3. During the execution, each thread queries the OpenMP runtime using the init() function (Line 9 in Listing 3.4), to obtain the local iteration bounds to execute the loop-body. This runtime function takes the global iteration space bounds as input and populates the local bounds for a thread using its thread ID. The variables holding the local bounds are passed-by-reference. These variables are then *written-to* in the runtime function. The runtime function performs the following tasks: i) reading global iteration bounds, ii) calculating local bounds using the thread ID and global bounds, iii) updating the variable holding local bounds. A hand-written implementation approach for such a sequence of operations in Java becomes cumbersome. This approach requires to write Truffle-AST that matches the execution behaviour of the runtime function. The hand-made Truffle-AST needs to be equivalent to one that the bitcode parser of Sulong would have generated from LLVM IR for the function. A simple 32-bit integer-add operation in LLVM IR contains more than ten Truffle-AST nodes in Sulong. Thus, creating the Truffle-AST for the `@__kmpc_for_static_init_4()` function consisting of about 35 lines of LLVM IR, including branches and function calls, would require creating a significantly large number of AST nodes by hand. These hand-written implementations are error-prone and difficult to maintain.

For the majority of the OpenMP runtime library functions, we can implement them in Java without using their hand-written Truffle-AST. Examples of these runtime library functions are i) functions retrieving the runtime information, such as the `@omp_get_thread_num()` that retrieves the ID of the current thread ii) functions synchronising the OpenMP threads, such as `@__kmpc_barrier()` that waits for other OpenMP threads to reach the synchronisation point. These functions can use purely Java-based implementations and avoid the need for using the Truffle API. Therefore, we use a *Hybrid approach* that implements the OpenMP runtime functions, or part of their functionality, in C. Sulong-OpenMP uses these C implementations when the runtime functions need to interact with internal data representation of Sulong. We convert the C functions to LLVM IR using clang and add them as an external library. Sulong-OpenMP parses these external libraries using the existing bitcode parser and

registers the functions defined in them. Sulong-OpenMP can invoke these registered functions like other functions defined in the program. We tried to minimise the need of such Hybrid approach, and so far we have a single function implemented in C: `@__kmpc_for_static_init_4()`.

The Hybrid approach leverages the benefits of both: the Pthreads-based and Function Morphing approach. The Hybrid approach was helpful for faster prototyping during the implementation of OpenMP runtime functions. It allowed us to add a placeholder implementation for the actual runtime functions in C and observe the execution behaviour of the program. This approach enabled Sulong-OpenMP to support for the OpenMP features incrementally.

3.3 Implementation of OpenMP features

This section describes the implementation of some of the most commonly used OpenMP features using the Hybrid approach (discussed in the previous section).

3.3.1 Fork-Join Execution Model

```
1  // OMP_NUM_THREADS = 4
2
3  void main() {
4     #pragma omp parallel
5     {
6        // OpenMP block
7     }
8  }
```

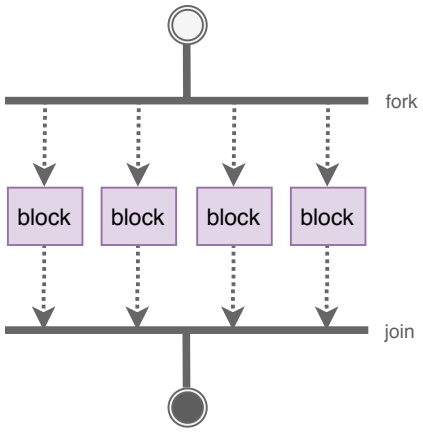


Figure 3.2: Block diagram of a simple OpenMP program in C, and its execution using a fork-join model.

OpenMP supports the fork-join model that was introduced in Section 2.6 and illustrated in Figure 3.2. Environment variable `OMP_NUM_THREADS` can be used to control the number of threads to fork/spawn at the beginning of an OpenMP region. Each thread then executes the region in parallel and joins/merges at the end of the region. The environment variable `OMP_NUM_THREADS` is one of the values (referred to as *control*

variables) that is used to determine the number of threads for executing a parallel region. The detailed steps to determine the number of OpenMP threads are specified in Algorithm 2.1 in [ARB19]. However, in the absence of other control variables, `OMP_NUM_THREADS` specifies the number of threads used for executing a parallel region. Comma-separated values for the environment variable specify the number of threads to be used for each nested level of the parallel regions.

Section 3.1 explained the implementation of the fork-join model by clang. When clang generates LLVM IR, the OpenMP block is wrapped into a separate `outlined` function. Each OpenMP thread then invokes this `outlined` function. The `@__kmpc_fork_call()` function replaces the OpenMP block in the LLVM IR of `@main()` function.

Sulong-OpenMP provides a *morphed* implementation of the `@__kmpc_fork_call()` function. This morphed implementation calls the `outlined` OpenMP function on the specified number of OpenMP threads. The runtime function receives a pointer to the `outlined` as an argument. Sulong-OpenMP treats call to the `outlined` function as a computational task and assign it to the desired thread in the pool of OpenMP threads. Each thread in the thread-pool takes a `outlined` function and the stack-frame of the calling function (i.e., `@__kmpc_fork_call()` in this case). The stack-frame is necessary to evaluate the arguments to the `outlined`. Implementation of the runtime function synchronises all the Java threads on finishing the execution of the assigned task. This implements the implicit synchronisation when the parallel block exits.

Initially, Sulong-OpenMP used to spawn/fork new Java threads for each call to the `@__kmpc_fork_call()` function and merge/join them at the end of the region. This approach is inefficient (more performance analysis details on this are in Subsection 5.4.4). Thus, Sulong-OpenMP implemented a thread-pool using Java threads where each member thread represents an OpenMP thread. The thread-pool is initialised at the beginning of the program and destroyed when the program finishes execution. Sulong-OpenMP chooses the hand-written implementation of the thread-pool to control the assignment of work to a specific thread in the pool. The ability to assign work is crucial to implement the work-sharing constructs, such as the static schedule discussed in the following section.

3.3.2 Work-sharing Constructs

In this section, we discuss the implementation of work-sharing constructs offered by OpenMP. This includes OpenMP master and single pragmas. This section explains the sharing of work using OpenMP schedules, and then describes the implementation of the static schedule in detail.

Master

The master thread (i.e., a thread with an ID 0), executes a block enclosed with an OpenMP master pragma, as illustrated in Figure 3.3. The OpenMP threads do not synchronise at the end of the master block.

```
1  void main() {
2     #pragma omp parallel
3     {
4        ...
5        #pragma omp master
6        {
7           // master block 1
8        }
9        ...
10       #pragma omp master
11       {
12          // master block 2
13       }
14       ...
15    }
16 }
```

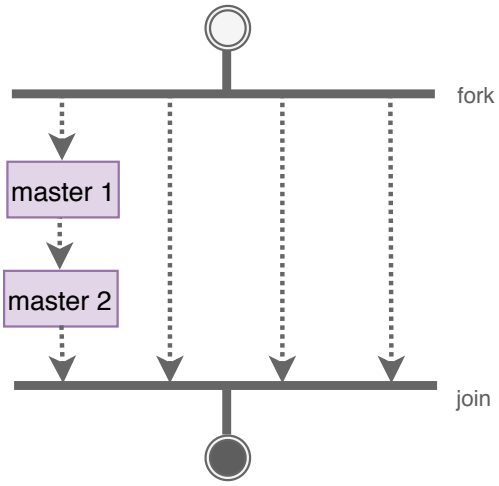


Figure 3.3: Only the master thread executes the OpenMP parallel regions with a master pragma. There is no synchronisation point at the end of the master block.

When clang generates LLVM IR for the master block, it is surrounded by `@__kmpc_master()` and `@__kmpc_master_end()` functions as shown in Listing 3.5. All the OpenMP threads execute the `@__kmpc_master()` function and the value returned by the function determines whether the thread would execute the master block. The function returns value 1 for the master thread and value 0 for the remaining threads.

```
1  define void @.omp_outlined.(...) {
2     ...
```

```
3     call @__kmpc_master(...)
4     ; LLVM IR for code in the master block
5     call @__kmpc_end_master(...)
6     ...
7   }
```

Listing 3.5: Simplified LLVM IR for a master block in the OpenMP parallel region.

Sulong-OpenMP supports the master directive using the *morphed* implementation of the `@__kmpc_master()` function. Sulong-OpenMP maintains a map of Java and OpenMP thread IDs, where the key is a Java thread ID, and the value is its corresponding OpenMP thread ID. Sulong-OpenMP queries this map to determine whether the current Java thread is the master thread. The runtime function `omp_get_thread_num()` uses the same map to retrieve the OpenMP thread ID of the current thread. For the most common scenario, this setup works as expected, when the master block is not orphan (i.e., present inside the OpenMP parallel block). However, some of the benchmarks from the NPB suite contain orphan master blocks (i.e., the master block that is present outside the OpenMP parallel block). Therefore, we need to put an additional mechanism to ensure the correct execution of orphan master blocks. The `@__kmpc_master()` function returns value 1 even when executed by the main thread of program from an orphan block.

Single

The OpenMP single directive is very much similar to the master directive. Only one of the OpenMP threads executes the block enclosed with OpenMP single pragma. This thread is not necessarily a master thread as illustrated in Figure 3.4. Additionally, there is an implicit barrier at the end of the single block, unless the `nowait` clause is present along with the single pragma. Similar to the OpenMP master pragma, clang surrounds the OpenMP single block using the `@__kmpc_single()` and `@__kmpc_end_single()` functions.

To implement the OpenMP single pragma, the morphed implementation marks that the single block has been *taken* by a thread that executes the `@__kmpc_single()` function first, and we store the thread ID of a *winner* thread. When the other threads

```
1  void main() {
2     #pragma omp parallel
3     {
4        #pragma omp single
5        {
6           // single block 1
7        }
8        ...
9        #pragma omp single
10       {
11          // single block 2
12       }
13    }
14 }
```

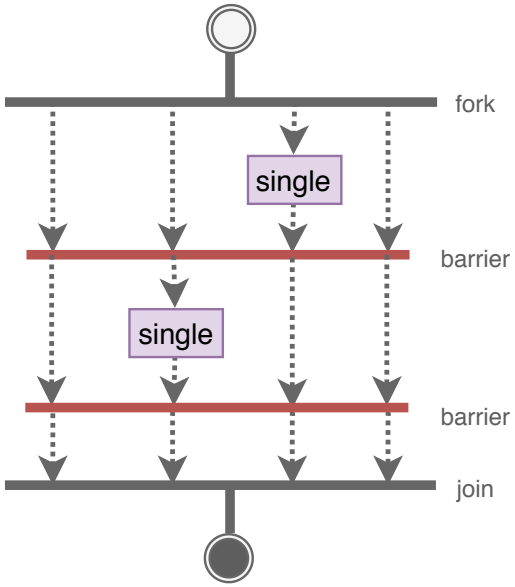


Figure 3.4: An OpenMP parallel region with a single block that is executed by one of the OpenMP threads. There is an explicit barrier at the end of the single block.

execute the `@__kmpc_single()` function, they see that the block is no longer available for the execution. The other threads then continue without executing the single block. Sulong-OpenMP checks whether the block is available for execution in a synchronised Java block. Similar to an OpenMP master pragma, the `@__kmpc_single()` function returns value 1 when the current thread is the winner, otherwise returns value 0. This value decides whether the thread should execute the block marked with the OpenMP single pragma or not. We do not need to implement the implicit synchronisation at the end of the single block. Clang generates a call to the barrier function after the single block unless the `nowait` clause is not specified.

OpenMP-For

OpenMP allows distributing iterations of a for-loop amongst the OpenMP threads to execute them collaboratively. The OpenMP-For construct is one of the commonly used features of OpenMP. OpenMP-For construct enables a program to leverage multiple threads to perform loop computation. A for-loop within the parallel block can specify the OpenMP-For construct using `#pragma omp for` on the previous line. Figure 3.5 shows an OpenMP-For pragma that is combined with the pragma for the parallel block. The merged pragma limits the parallel block to the for-loop.

```
1  void main() {
2
3     #pragma omp parallel for schedule(static)
4     for(int i = 0; i < N; i++ ) {
5
6       printf("Thread %d -> iteration %d\n",
7                 omp_get_thread_num(), i);
8
9     }
10 }
```

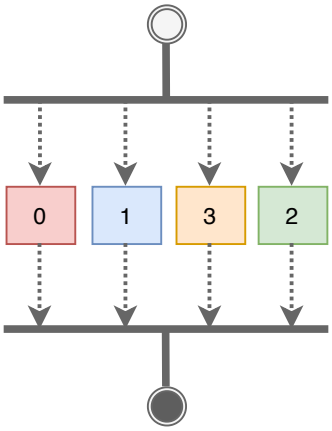


Figure 3.5: An OpenMP-For loop in C where each thread prints its ID and the iteration number that it executes. The block diagram on the right represents the execution of the program with four OpenMP threads and N=4.

Figure 3.5 shows an OpenMP program with a for-loop whose iterations are collaboratively executed by OpenMP threads. The diagrammatic representation shows the execution of a program with four OpenMP threads where each thread executes one iteration of the for-loop.

OpenMP allows controlling the distribution of iterations amongst the threads using the combination of OpenMP schedules and chunk-size. OpenMP schedule specifies the order assignment of the units of work. The chunk-size is an optional parameter, a positive integer, that controls the minimum unit of work. There are five *schedule kinds* specified in OpenMP: static, dynamic, guided, auto and runtime. Similar to other major implementations, clang uses the static schedule as a default OpenMP schedule.

- The **static schedule** divides the iteration space statically amongst the OpenMP threads. The static schedule divides the iteration space into the chunks of the size specified by the chunk-size, except the last chunk which contains the last iterations. When no chunk-size is specified, the static schedule divides iteration space into approximately equal-sized chunks. Here, the rationale is to distribute the work done in the loop as evenly as possible. Chunks are then assigned to threads in a round-robin fashion in the order of the thread number. The static schedule is useful when all the iterations of the loop contain a similar amount of work.

- The **dynamic schedule** assigns iterations specified by the chunk-size to each thread. The dynamic schedule uses the default chunk-size of 1. When a thread finishes the assigned work, it requests the next chunk of work from the OpenMP runtime. This process continues until there are no more chunks left for execution. The dynamic schedule has an additional runtime overhead compared to the static schedule, because of the additional interactions with the runtime to acquire the next chunk of iterations. However, the dynamic schedule is helpful to achieve better load-balance of work amongst the OpenMP threads when the iterations of a loop do not have a uniform distribution of work.

- The **guided schedule** determines the chunk-size based on the number of unassigned iterations. Thus, the threads get larger chunks at the beginning. The chunk-size reduces as the portion of allocated iterations increases. In the case of guided schedule, the chunk-size plays a different role compared to the remaining two schedules. The chunk-size specifies the smallest size of the chunk that the schedule assigns to a thread.

- The **auto schedule** delegates the scheduling decision to the compiler and/or the runtime. The behaviour of the auto schedule is implementation-specific.

- The **runtime schedule** allows a program to defer the choice of OpenMP schedule until runtime. The choices of schedule and chunk-size can be specified at runtime, e.g., using environment variables.

The primary use of the OpenMP schedules is to reduce the imbalance of workload amongst the threads. The use of chunk-size is to tune the scheduling overhead by controlling the number of interactions with the OpenMP runtime.

The Sulong-OpenMP currently supports only the static schedule with the default chunk-size. Implementation of the static schedule uses the Hybrid approach (as explained in Section 3.2). Each thread queries the local iteration space at the beginning of executing the OpenMP parallel region using the `@__kmpc_for_static_init_4()` function. Sulong-OpenMP uses the C implementation of this function.

Data-sharing attributes

OpenMP provides a mechanism to control the sharing of data amongst the OpenMP threads using the attributes, such as private, shared, firstprivate and lastprivate. The data-sharing attributes are typically specified along with the OpenMP parallel pragma and contain a comma-separated list of variable names to which the attribute is applied.

- The **private** attribute creates separate instances of the variables for each thread and their scope is limited to the parallel region. The private instances are not initialised when created, thus may contain a garbage-value.

- The **threadprivate** clause creates separate instances of the variables for each thread. The threadprivate instances of a variable are initialised to the value held by that variable before entering the parallel region.

- The **firstprivate** is a superset of the private attribute that specifies the initial value of a variable's private instance. In addition to the functionality of the private clause, the firstprivate initialises the private instances of a variable with the value in the original version of the variable.

- The **lastprivate** is also a superset of the private-clause. It is specified along with the OpenMP-For clause. Similar to the private-clause, the lastprivate clause creates the private instances of the specified variables for each thread. Additionally, the lastprivate clause updates the original values of the variables with the values from the thread that executes the sequentially last iteration. In a nutshell, after executing the parallel region, the lastprivate variables will contain the values equivalent to the values on sequential execution of the for-loop.

- The **shared** attribute instructs threads to share the local instances of the specified list of variables. This attribute does not create new instances of a shared variable for threads. Each thread shares the same copy of a shared variable. Consequently, a shared variable preserves its value before entering and after exiting the parallel region. Importantly, updates to the shared variables need explicit synchronisation to avoid races. If the shared variable can be an array, the array elements can be updated by multiple threads individually.

- The **reduction** attribute specifies a form of recurrence calculation that is performed in parallel. The clause specifies *reduction-identifier*, an operation such as + (for aggregation), min, max, and a list of identifiers to which that operation is applied. We do not discuss the reduction attribute at length. The NAS Parallel Benchmarks (NPB) suite uses a simple form of a reduction attribute to aggregate partial sums across the OpenMP threads.

The data-sharing attributes topic in the OpenMP specification covers additional constructs, such as the `linear` clause, that we do not discuss here. We focus our discussion on the attributes used in NPB suite, that is limited to the attributes specified along with the parallel region or OpenMP-For pragmas.

The LLVM IR generation of clang and allocation of local variables in Sulong simplified the implementation of the data-sharing attributes. For `private`, `firstprivate` and `threadprivate` variables, clang generates LLVM IR to create their corresponding local variables in the `outlined` OpenMP function. The `outlined` function uses the same local/private copies for its computation. As clang allocates variables locally, their scope is limited to the `outlined` function. Sulong-OpenMP treats those private copies similar to any other local variables defined in the `outlined` function. For the firstprivate variables, clang generates instructions to initialise them to zero after their allocation.

The `shared` variables are passed-by-reference, as arguments to the OpenMP runtime function `@__kmpc_fork_call()` in the generated LLVM IR. This runtime function then passes the same arguments to the `outlined` OpenMP function. Therefore, accesses to the `shared` variables in the parallel region are performed to the same location in the memory. The LLVM IR allocates `shared` variables using the `alloca` instruction outside the parallel region. These allocations on Sulong-OpenMP use the existing implementation of Sulong that built using UNSAFE API from Sun [MPM+15]. The UNSAFE API allocates a chunk of memory off-the-heap of JVM and returns its native pointer as a long value. Sulong uses this pointer to access the content at that memory location. The generated LLVM IR from Clang passes `shared` variables to the `outlined` function using such pointers to their locations. The memory allocation mechanism in Sulong already supports all these operations. Thus, Sulong or Sulong-OpenMP did not require any modifications to support the OpenMP `shared` attribute.

In the case of `reduction` clause, the LLVM IR corresponding to the reduction operation is surrounded by `@__kmpc_reduce_nowait` and `@__kmpc_end_reduce_nowait` functions. In this LLVM IR block, the result variable is updated synchronously by using the pointer to its address. Sulong-OpenMP mapped these functions to the function performing enter and exit from the critical region. Consequently, the reduction is performed synchronously before exiting the parallel region. Therefore, we did not require an additional mechanism to support OpenMP reduction clause.

3.3.3 Synchronisation constructs

This section describes the OpenMP synchronisation constructs. This includes the implementation of the barrier, critical and flush pragmas.

Barriers

A barrier is a synchronisation point where a thread waits until all the threads reach that point, and then all the threads continue. OpenMP has implicit and explicit types of barrier. The implicit barriers are present at the end of the parallel and single region. The explicit barriers can be specified using `#pragma omp barrier` as shown in Figure 3.6.

```
1 void main() {
2    #pragma omp parallel
3    {
4       // code before barrier
5       #pragma omp barrier
6       // code after barrier
7    }
8 }
```

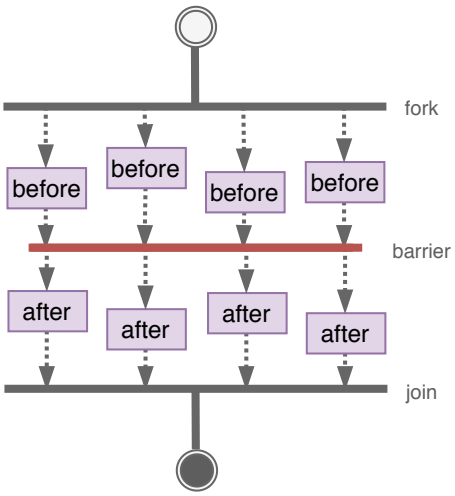


Figure 3.6: A block diagram of a parallel block containing the OpenMP barrier. All the OpenMP threads synchronise at the barrier. However, the code blocks before and after the barrier execute without synchronisation. The red line shows an explicit barrier while the gray line, where threads join, shows an implicit barrier.

In the generated LLVM IR, the `@__kmpc_barrier()` function call replaces the barrier directive in the parallel block. Every OpenMP thread calls this function and blocks itself until the remaining threads to call the same function. We provide a simple implementation of OpenMP barrier using the wait-notify mechanism of Java threads. A Java thread can invoke the `wait()` method on an object and suspends itself until any other thread *interrupts* or *notifies* it. We use this behaviour to implement barrier. All the threads except the last thread invoke the `wait()` method on the specified object in the *morphed* implementation of the `@__kmpc_barrier()` (i.e., when a thread reaches barrier). When the last thread reaches the barrier, it wakes up all the waiting threads.

Critical

The code block marked with an OpenMP critical directive creates a mutually exclusive region, where only one thread can be present at a given time. For a particular critical block, if one thread is executing it, other threads have to wait for that thread to exit the block then the next thread may enter the block. When the LLVM IR is generated for an OpenMP critical block, it is enclosed by the `@__kmpc_critical()` and `@__kmpc_critical_end()` OpenMP runtime functions. Figure 3.7 shows that the generated LLVM IR is similar to the master and single directive.

```
void main() {
  #pragma omp parallel
  {
     #pragma omp critical
     {
        // Critical region
     }
  }
}
```

```
define void @.main.(...) {
    ...
    call @.omp_outlined.(...)
    ...
}

define void @.omp_outlined.(...) {
    ...
    call @__kmpc_critical(...)
    ; LLVM IR for the critical region
    call @__kmpc_end_critical(...)
    ...
}
```

Figure 3.7: An OpenMP C program containing the critical region is shown on the left. Simplified LLVM IR for the critical region is shown in on the right side. The LLVM IR for the critical region is surrounded by the runtime function calls.

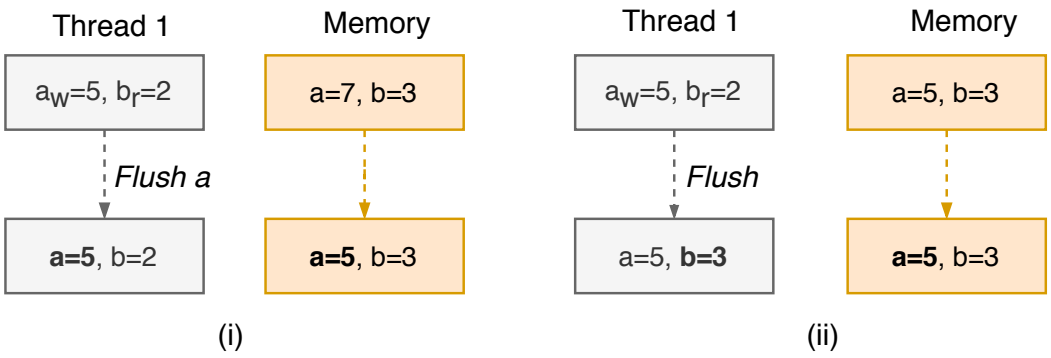


Figure 3.8: Examples of flush operations: (i) flush-set contains variable `a`, (ii) when flush-set is empty. When the flush-set is empty, the flush operation makes the entire temporary view of the thread, consistent with the memory. a_w specifies that the thread has updated value of `a`. Similarly, b_r specifies read operation on `b`.

To support the critical pragma, Sulong-OpenMP provides a `morphed` implementation for the runtime functions, using an instance of a `Semaphore` class in Java. When a thread enters a critical region, it calls the `@__kmpc_critical()` function where it acquires the lock. When the thread exits the critical region, it calls the `@__kmpc_end_critical()` function and releases the lock. It ensures that only the thread executing the critical region has exclusive access to it.

Flush

OpenMP uses the relaxed-consistency memory model [HDS05]. This model allows threads to have a view of the memory, that can be temporarily inconsistent with that of the other threads. The flush operation on a thread makes its temporary view of the memory consistent with the actual memory. Thus, the flush operation is analogous to a `memory fence` operation. The flush operation is performed on a set of variables called *flush-set*. This operation ensures that a temporary view of the variables in the flush-set is consistent with the memory [ARB19]. Figure 3.8(i) shows an example of flush operation, where a flush-set consists of only variable `a`. In this case, thread 1 writes variable `a` and reads variable `b`. Flush operation guarantees that the view of thread 1 of `a` consistent with memory but does not guarantee the same about thread 1's view of variable b, because the flush-set only contains `a` not `b`. Further, the flush operation restricts the OpenMP implementations from reordering the memory operations on the flush-set variables. Otherwise, OpenMP implementations can reorder these operations.

The flush pragma specifies the flush-set as a list of comma-separated variable names as `#pragma omp flush (var1, var2, ...)`. When a flush pragma does not specify a list of variables, the flush operation applies to the entire temporary view of a thread that includes all the variables, as shown in Figure 3.8(ii).

On the other hand, Java uses the happens-before consistency memory model [Pug19]. In Java, when a write operation is performed on a `volatile` variable, this write operation along with all the previous write operations to any other variables performed by the thread are visible to the other thread; when that thread performs a read/write operation on the same volatile variable. Sulong-OpenMP uses this behaviour to support the OpenMP flush operation in Java. The LLVM IR for the flush pragma replaces it by a call to the `@__kmpc_flush()` function. Sulong-OpenMP provides a *morphed* implementation of this runtime function, where it increments a value of the predefined volatile variable. This increment performs a read and a write operation on the volatile variable. Therefore, when a thread executes OpenMP flush, all the writes before calling the OpenMP flush become visible to the other OpenMP threads, when they execute the flush function. However, this limits our ability to perform flush on a subset of variables. Thus, every flush operation irrespective of whether it is on a subset of variables, Sulong-OpenMP performs flush on the entire temporary view that includes all the shared variables. The OpenMP specification permits implementations to ignore the flush-set, and perform every flush operation to all the shared variables [ARB19].

OpenMP specifies implicit flushes at places such as exiting from a critical region, end of the parallel regions and during barriers. In the implementation of these constructs, Sulong-OpenMP uses the common pre-defined object in Java `synchronize` construct to achieve the desired synchronisation amongst OpenMP threads. The Java memory model guarantees that the unlock operation on an object happens before the lock operation on the same object. Therefore, operations that follow the aforementioned synchronisation points are guaranteed to see the write operations from other threads that happened before those points.

3.4 Related work

This section discusses some of the implementations that run on a JVM and offer OpenMP-like features. We also discuss an implementation that executes LLVM IR on the JVM. We discuss the other implementations in this chapter because they mainly differ from Sulong-OpenMP in their implementation approach. For completeness, we will briefly mention the native implementations from different compiler vendors which is the most popular way to execute the OpenMP programs.

3.4.1 OpenMP-like implementations in Java

JOMP

The JOMP is a prototype implementation published in 2000 after the release of OpenMP 1.0 in 1997 [BK00]. The primary motivation behind JOMP was to bring directive-based shared-memory parallelism to Java because OpenMP is considered to be less error-prone, easier to write and have better maintainability. JOMP allows writing a subset of OpenMP directives as comments in a Java program. The JOMP compiler processes these directives using the source-to-source translation. This translation converts the OpenMP pragmas to corresponding parallel Java code that contains calls to the JOMP runtime library. This process is analogous to the generation of LLVM IR with calls to the OpenMP runtime library of clang that we discussed earlier in Section 3.1. The JOMP compiler extends the Java 1.1 parser that is bundled with the JavaCC utility to parse of OpenMP directives. This parsed Java code can then be compiled using a regular Java compiler and needs the JOMP runtime library during its execution.

JOMP has a few limitations while mapping OpenMP features to Java, e.g., the types of variables permitted in the data sharing constructs, such as private, firstprivate and shared. The JOMP implementation allows using only the local variables (not the instance variables), in the data sharing constructs. Instance variables are the variables that are declared within the Java class as its fields. Threads always shared the instance variables. Further, the exceptions are more common in Java compared to C++. The OpenMP specification available at that time did not provide clear guidelines to handle exceptions. This lack of guidelines on how to handle exceptions

leads implementation to ambiguous scenarios, e.g., how to handle the exception that occurs in the parallel region but is caught outside the region. In this case, the expected behaviour may be to interrupt the master thread and handover the exception to it. However, the Threading API of Java prohibits to interrupt a running thread. The `Thread.interrupt()` only allows interrupting a waiting thread. This scenario becomes more complex when the exception arises in the work-sharing construct, e.g., the exception occurs in an OpenMP-For loop, and is caught outside the loop but inside the parallel region. In this case, the specification needs to define the behaviour of the synchronisation constructs, e.g., the implicit barrier at the end of an OpenMP-For loop. OpenMP did not support tasks when JOMP was released. Thus, JOMP did not support OpenMP tasks.

omp4j

The omp4j project [BS19] uses an approach similar to that of JOMP. The omp4j expects OpenMP directives as comments in the Java program; it then pre-processes them using a source-to-source translation tool. This translated version of the program expands the directives in the original Java program with the equivalent parallel code. This translated code contains calls to the omp4j runtime library functions. The omp4j is a relatively new project (v1.0 was released in 2015) that supports Java programs written using Java 8. It claims better scalability on more than 24 CPU cores compared to JOMP[BS19]. The pre-processor of the project is built on top of the ANTLR-based [1] grammar for Java 8 (newer than Java 1.1 used by JOMP). The omp4j shares limitations of the JOMP project, e.g., limited support for exception handling, because both the projects bring OpenMP-like features to Java.

JaMP

The JaMP [KBVP07] is the adoption of OpenMP features to Java. Unlike JOMP and omp4j, JaMP targets distributed memory clusters. JaMP supports a much larger subset of OpenMP features compared to JOMP and omp4j. JaMP supports all OpenMP

[1] ANTLR is a popular tool that can generate a parser for a given grammar in various target programming languages [Par13]

2.0 features and some of the OpenMP 3.0 features, such as OpenMP tasks. Additionally, JaMP also supports the execution of OpenMP parallel loops on CUDA-enabled graphics cards. JaMP uses the research compiler: Jackal [VBB01]. Jackal provides a software-based Distributed Shared Memory (DSM) implementation [2] for Java programs. The DSM enables JaMP programs to execute on multiple nodes of a cluster. The JaMP directives closely follow the OpenMP standard. They are put as comments in a Java program. However, JaMP uses a different approach to process directives compared to that of JOMP and omp4j. They are translated to intermediate code of Jackal: LASM. JaMP extends LASM with special instructions to capture information from its directives. This makes the optimisation phase of Jackal aware of the JaMP directives. Further, this enables Jackal to apply JaMP-specific optimisations, e.g., passing only live `shared` variables to worker threads [KBVP07]. However, JaMP does not support execution of the orphaned regions (i.e., a region with OpenMP directive but outside the parallel region). The OpenMP compliant implementations are required to dynamically determine if the code is executed from the parallel region.

Comparision with Sulong-OpenMP

Sulong-OpenMP uses a different approach to execute OpenMP programs. It interpreters the LLVM IR of the OpenMP programs on a JVM. Sulong-OpenMP provides the implementation for the OpenMP runtime library functions of clang. This approach is different from the previously discussed approaches that are required to i) parse the OpenMP directives, ii) generate calls to an implementation-specific runtime library, iii) implement the runtime library. Our approach benefits from the translation mechanism of clang for the OpenMP directives. However, the use of clang also restricts Sulong-OpenMP, to match the implementation of its OpenMP runtime library functions. Sulong-OpenMP aims to support OpenMP tasks and ability to execute OpenMP parallel loops on accelerators in the future. Sulong-OpenMP does not support these features yet. As Sulong-OpenMP is an interpreter in Java, it does have the limitations of mapping OpenMP features to Java as the previous approaches. Further,

[2] A software-based DSM system provides a global address-space to the nodes of the cluster using their separate memories. DSM performs a check for each memory access, to determine if the object is in the local memory. DSM requests the runtime system to bring the remote objects in the local memory.

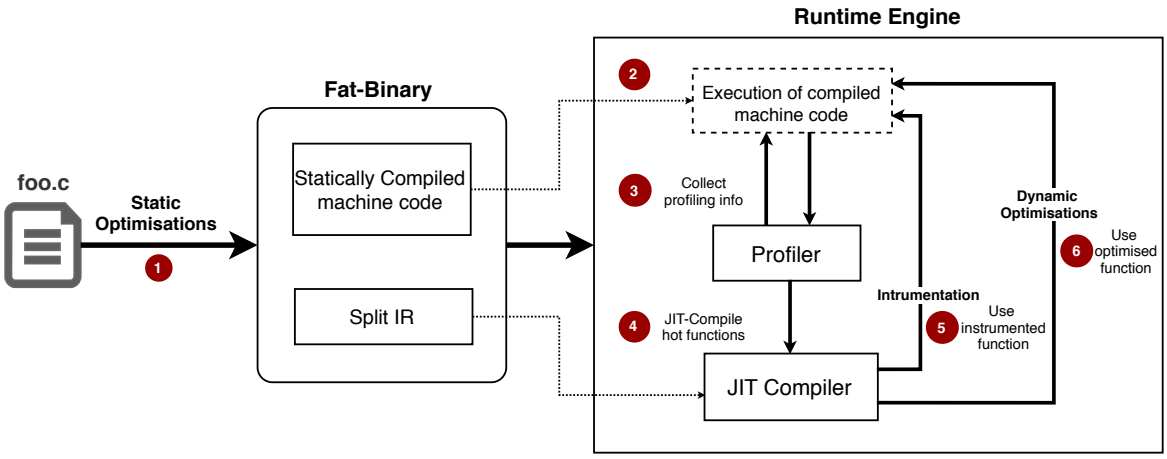


Figure 3.9: Execution flow using the fat-binary approach. Important steps during the execution and their sequence are numbered in the diagram.

Sulong-OpenMP also supports execution of the orphaned regions.

3.4.2 Execution of IR of C programs on JVMs

Sulong is not the only system that achieves execution of a C program by using its intermediate representation generated by a static compiler and then executing that on a JVM. This approach aims to leverage benefits from both static and dynamic optimizations from Ahead-of-Time (AOT) as well as Just-in-Time (JIT) compilers respectively. This subsection describes the fat-binary approach that uses the proprietory intermediate representation of C/C++ programs and executes it using the JVM [NED+13]. The fat-binary approach also aims to offer low start-up time similar to the AOT compiled executions. Although the fat-binary approach uses IBM's proprietory *split IR*, it is also applicable to LLVM IR.

Fat-Binary Approach

Figure 3.9 depicts execution of a C program using the fat-binary approach. During the execution, first, the fat-binary for a program is generated (Step 1 in Figure 3.9) that bundles the AOT compiled binary executable the program along with its IR. The runtime engine begins execution of a program using its binary executable (Step 2) which avoids the slow start-up phase of a typical JVM-based execution. The runtime samples the execution using an event-based profiler to identify the frequently called

C functions (Step 3). The frequently called *hot* functions are then passed to the JIT-compiler (Step 4). Compilation of the hot functions occurs in two stages: i) The JIT-compiler creates instrumented versions of the hot functions. The execution switches to the instrumented version where the runtime collects profiling information (Step 5). The instrumented version is slower as it performs costly profiling operations. ii) The JIT-compiler uses the profiling information collected from the instrumented version, to aggressively optimise hot functions. The optimised version of the functions is then used for further invocations (Step 6). While the JIT-compilation is in progress, the execution switches back to the original AOT-compiled version of that method.

Use of AOT-compiled executable reduces the start-up time compared to using a purely JVM hosted approach. The fat-binary-based approach uses a repurposed JVM to create instrumented and JIT-compiled version using the IR of hot methods. Thus, the fat-binary benefits from the advantages offered by both AOT-compiled and JIT-compiled approaches.

Unlike Sulong-OpenMP, the fat-binary-based approach does not support the execution of parallel programs. Further, creating fat-binaries requires a custom tool-chain that bundles LLVM IR with the binary executable, while Sulong-OpenMP uses unmodified LLVM IR.

3.4.3 Native OpenMP implementations

Similar to clang from the LLVM project, there are multiple OpenMP implementations available from different software vendors such as Intel, PGI, GCC and IBM with various degrees of completeness for the latest OpenMP specifications 5.0. Such implementations take an OpenMP program written in C/C++/Fortran as input and generate an AOT-compiled executable binary. This is a de facto approach for executing OpenMP applications.

3.5 Summary

In this chapter, we discussed the extension to the Sulong project that enables the execution of OpenMP programs on a JVM. First, we discussed the implementation problem for Sulong-OpenMP that covered the generation of LLVM IR for an OpenMP program. Then we discussed three implementation approaches and their trade-offs for extending Sulong: Pthreads-based, Function Morphing and the Hybrid approach. We chose the Hybrid approach to implement Sulong-OpenMP; because it simplified incremental adding of the OpenMP features compared to Pthreads-based approach. The Hybrid approach is the extended version of the Function Morphing approach. It uses the C implementation (converted into LLVM IR), for some of the runtime functions that are relatively cumbersome to implement in Java using the Function Morphing approach.

We then discussed the implementation of some of the commonly used features of OpenMP using the Hybrid approach. This covered the implementation of the fork-join model, work-sharing and synchronisation constructs. The discussion on the work-sharing constructs covered the implementation of OpenMP-For loops, master and single directives, and the synchronisation constructs covered the barrier, critical and flush directives.

Finally, the related work section provided two main categories of the OpenMP implementation. First, the implementations that provided OpenMP-like features to Java using the Java Threading API. Second, the implementations from other compiler vendors, that use the Ahead-Of-Time (AOT) compiled binaries for executing OpenMP programs. The AOT-based approach is conventional and the most common way of executing OpenMP programs.

Chapter 4

Experimental Methodology

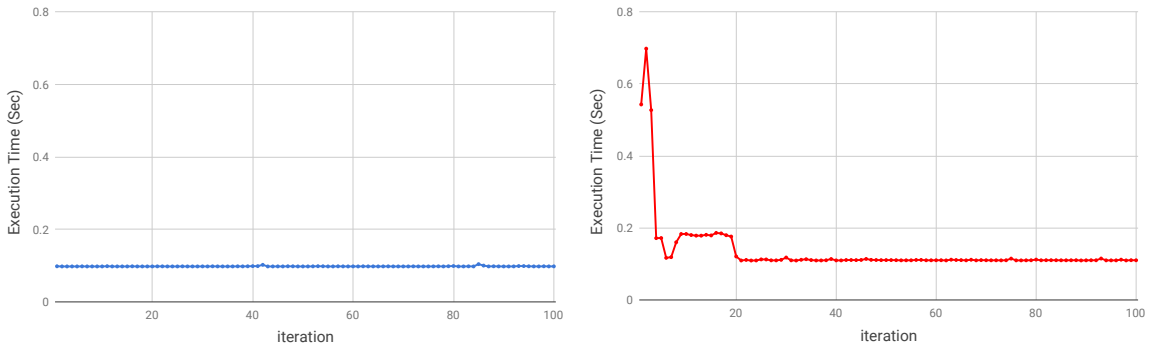
This chapter discusses the experimental methodology that we followed to evaluate Sulong-OpenMP. As we use a JVM for executing OpenMP programs, we will begin our discussion with the key metrics used for performance evaluation of the JVM hosted execution (Section 4.1). Further, we will discuss the techniques used for the evaluation of JVM hosted languages and the experimental methodology used for the evaluation of Sulong-OpenMP (Section 4.2). We will conclude this chapter by describing the experimental setup that covers the selected benchmarks, hardware and the software stack used to evaluate Sulong-OpenMP (Section 4.3).

4.1 Evaluation Metrics

In this section, we discuss the metrics used to evaluate JVM hosted executions. This covers both the metrics that we use (*peak performance*) and the ones that we do not consider while measuring the execution time of the benchmarks.

4.1.1 Peak Performance

Execution of programs begins in a slow interpreter mode on JVMs. During the execution, when a method of the program is invoked more than the pre-defined threshold, the JVM considers the method *hot* and JIT-compiles it. Subsequent invocations of that method use its JIT-compiled version. The compiled version is faster than the interpreted version because i) it avoids calls to the method's bytecode handlers and ii)



(a) C (Native) (b) Java

Figure 4.1: A comparison of execution behaviour for the nbody benchmark from the Shootout benchmark suite executed 100 times in a single invocation of native/JVM. Figure 4.1a shows execution of the benchmark executed natively while Figure 4.1b shows the execution of the Java version of the benchmark executed on a JVM.

it is optimised using the profiling information collected during interpretation. Therefore, as the number of JIT-compiled methods increases, the execution of the program becomes faster accordingly. This process continues until all the frequently used parts of the program get compiled and the performance improvement plateaus. At this point during the execution, the application is said to have reached its *peak performance* and no further performance improvements are expected [Sea15]. The execution phase from the start of the application until it reaches its peak performance is referred to as the *warm**ing**-up phase*, while the execution phase at the peak performance is referred to as the *warm**ed**-up phase*.

Figure 4.1 shows the execution time of 100 iterations for C and Java versions of the `nbody` benchmark from the Shootout benchmark suite [Guo18]. The C version is executed using a native binary executable generated using clang, and the Java version is executed on a JVM. Here, the benchmark is executed repeatedly in a loop within a single native/JVM invocation, and the execution time of each iteration is recorded. Figure 4.1a shows that the execution time of 100 iterations for the C version is nearly constant. On the other hand, the corresponding initial iterations for the JVM hosted execution in Figure 4.1b are much slower (∼7 times) compared to the later iterations of the benchmark. After about 25 iterations of the benchmark, the execution time for the JVM hosted execution stabilises, and the benchmark is said to be executing at its peak performance. In the performance evaluation of Sulong-OpenMP, we use the execution time of the iterations running at the peak performance. Section 4.2 provides

more detail about detecting the warmed-up state of a benchmark.

As discussed in Section 2.2, JVMs use a tiered compilation approach by employing multiple JIT compilers. The highest tier/top-tier of JIT-compilation trades-off compilation time for the performance. Thus, a method compiled using the top-tier JIT compiler is the fastest version of the method that the JVM can generate. However, there is a rare possibility that a method might get slower on compilation using a top-tier JIT-compiler [BBTK+17]. To avoid such JIT-compilation cases, it is necessary to identify and choose the compilation level of the corresponding methods to the tier that generates their fastest version. Depending on the JVM, tuning the compilation level of methods may be achieved by using the JVM-specific flags. Thus, such compilation tuning can be cumbersome and fragile. However, this is not a task expected to be performed by users. Therefore, we decided to ignore such compilation anomalies (if present), and use the top-tier JIT compiler (i.e., Graal compiler) for all the eligible hot methods during the execution using Sulong-OpenMP.

The performance evaluation metrics for different executions are listed below.

- **JVM hosted executions** use peak performance as a metric for evaluating performance [Tol19, SDM+13]. For long-running applications on JVMs, the warming-up phase of an application is negligible compared to the total execution time. Thus, the warm-up time can be ignored.

- **Native executions** of OpenMP programs use the wall-clock time as a metric for evaluating performance. The purpose of using OpenMP parallelism is to reduce the total execution time of the programs.

- **Sulong-OpenMP** uses peak performance as a metric for evaluating performance. We do not aim to provide Sulong-OpenMP as a replacement for native executions. We provide Sulong-OpenMP as a system that enables users to execute the OpenMP programs along with the benefits offered by the GraalVM, such as support for the execution of polyglot OpenMP programs. Therefore, our objective is to achieve reasonable execution performance, which will make the execution on Sulong-OpenMP practical. We aim to achieve our objective by minimising the performance gap between the native and Sulong-OpenMP-based execution. Peak performance enables us to measure a lower bound of this gap.

Barrett et al. demonstrated that, although the peak performance is a metric for benchmarking the executions on virtual machines, it can be challenging to use [BBTK+17]. The primary reason is that often the benchmarks fail to reach the steady-state of peak performance (i.e., the peak performance may keep fluctuating). Some benchmarks may achieve a steady-state at a performance lower than the peak performance while others may never achieve the expected steady-state performance. In both scenarios, it is difficult to pick the exact iteration when a benchmark reaches its peak performance. Also, the number of iterations and the warm-up time are specific to the VM and the benchmark. We address this problem by using the approach suggested by Kalibera et al. This uses manual inspection of the execution to determine when the benchmark is warmed up [KJ13]. Along with the manual inspection, we ensure that the computationally intensive functions of the benchmarks are compiled by the top-tier JIT compiler. In the case of OpenMP programs, these functions have the parallel regions, and they are compiled by the Graal compiler.

4.1.2 Inapplicable Metrics

In this section, we discuss the aspects of JVM hosted execution that we do not consider for the performance evaluation of Sulong-OpenMP.

Memory consumption

A JVM hosted execution takes a significantly large amount of memory compared to the native execution of its equivalent program. A JVM needs many components such as the interpreter, JIT compilers, garbage collectors, code cache to store the compiled versions of the program available in the memory during the execution. The memory overhead is further increased while executing a Truffle hosted language (e.g., Sulong) on a JVM. A Truffle hosted implementation needs the AST representation of an input program and the AST interpreter itself in the memory. Further, as an AST is self-optimising, it needs to store the information about the current specialisation state in the memory. This makes the execution of Truffle hosted languages memory-intensive compared to the execution of the AOT compiled binary of the program. Smaller memory footprint may improve performance by effective utilisation of cache hierarchy. Sulong implements techniques, such as variable liveness analysis, to reduce memory

footprint. Although these techniques can increase performance, they are not discussed thoroughly as they complementary to the functionality of Sulong-OpenMP.

A precise measurement of memory consumed by an application, running on a JVM at a given point, can be difficult because the application does not manage its own memory. When an application creates an object, the JVM allocates memory for it on the heap that is reclaimed when the object is no longer used (i.e., not referenced by any other object); thus, it is unreachable from the application. JVMs employ the Garbage Collector (GC) to identify the unused objects and claim the memory allocated to them. This cleaning operation or a *collection cycle* involves computation. A certain part of GC computations may need to pause the execution of an application. This makes triggering a GC cycle important for both memory consumption and performance. JVMs avoid frequent GC cycles and perform them when they are necessary using the pre-defined heuristics. Therefore, when we measure the memory allocated at any point during the execution, it may not be the exact representation of the total amount of memory used by the application at that point. Further, the discussion on memory consumption is valid in the context of a specific amount of memory because the application behaviour may change for a different amount of available memory. For example, smaller heap size may cause frequent GC cycles that pause the execution of an application more often which leads to increased execution time compared to the execution with a larger heap size. Therefore, we do not explicitly attempt to record memory consumed by an application. We only look at the time spent while performing GC activities in the execution profile of an application, recorded using our performance analysis technique (discussed in Chapter 6). This performance analysis aims to determine the percentage of overhead incurred from the implementation inefficiencies of Sulong-OpenMP.

Start-up time

The start-up time for an application is the time taken from launching the JVM until it executes the first line of the `main` method of the application. This involves time taken to set up the runtime environment, create the specified number of compiler and garbage collection threads, load necessary classes and libraries, etc. This may be crucial for applications running for a short period. However, for the use-cases of Sulong-OpenMP,

we expect this time to be much smaller compared to the overall execution time of the application. Additionally, GraalVM offers an option to create a binary executable for a Java application, the Sulong-OpenMP interpreter in our case, that can be used for the applications in which the start-up time needs to be shorter [WSH+19]. The AOT compiled executables avoid spending time to set up the execution environment. Consequently, the start-up time of an application is reduced significantly.

Warming-up Phase

Sulong-OpenMP is built on top of the system, that trades-off the warm-up time for higher performance. Therefore, we do not measure the warming-up phase of application (i.e., the time required from the beginning until the application is warmed-up and reach the peak performance). As mentioned previously, JVMs collect profiling information that is used to perform speculative optimisations and the Truffle-based interpreters specialise the program ASTs that makes the warming-up phase much slower.

4.2 Benchmarking Methodology

In this section, we discuss the important aspects of benchmarking for the JVM hosted execution. Then we describe the benchmarking technique that we used for the evaluation of Sulong-OpenMP.

4.2.1 Key Aspects

Calculating the execution time

One of the basic techniques to measure the execution time for a benchmark is to use the command-line utility, *time*. It measures the wall-clock execution time for an application, that includes start-up time, warming-up and warmed-up phase of the benchmark. As the utility measures execution time externally, it is not possible to extract the time spent in different execution phases of the benchmark.

We want to measure the peak performance of the application therefore we need to ensure that the benchmark is warmed-up before we start measuring the execution time.

One of the approaches used by the JVM hosted executions is to run a benchmark in a loop, allow the benchmark to warm-up during the first few iterations, and then measure the execution time for the warmed-up iterations. The built-in library functions are used to measure time before starting and after finishing the iteration to compute the execution time for every iteration. We can use the time taken for multiple iterations as a representative of the execution time of the benchmark. Using multiple iterations is useful, especially, to minimise the noise introduced by non-deterministic events, such as garbage collection. The input parameters of a benchmark can further be tuned to ensure that every iteration of the benchmark executes for a sufficiently long time, thereby minimising the noise. For example, the implementation of Ruby using Truffle and Graal, TruffleRuby, has configured the benchmarks' parameters so that the warmed-up iterations would execute for about ∼2 seconds [Sea15].

Consistent Execution environment

There exist several confounding variables that can reduce the reproducibility of the benchmark results, such as the percentage of available memory at the time of execution, types of daemon processes running, and temperature of the processor to name a few. Influence of the confounding variables could be minimised using the sophisticated techniques provided by tools such as the *krun* [BBTK+17] or *Collective Knowledge Framework (CK)* [Fou09]. These techniques provide mechanisms to ensure that i) the CPU frequency did not change before and after executing iterations of the benchmark; ii) the temperature is constant at the time of launching the benchmark process; iii) every invocation of the benchmarking process is done after rebooting the machine. We do not use such a sophisticated benchmarking process because of the time constraints. Instead, we use the guiding principles behind these techniques described in Section 4.2.

In the case of multi-threaded execution, scheduling of threads on CPU cores can influence the performance of an application. The modern shared-memory systems are typically Non-Uniform Memory Access (NUMA) systems. On NUMA systems, location of the memory, relative to a CPU core, determines the time taken to access that location. Scheduling of threads and memory access pattern may change the proportion of slower memory accesses performed by an application. Further, the CPU cores typically share one or more levels of caches. The performance of an application

can be improved when the threads are scheduled to benefit from the cache locality.

The Linux utility `taskset` enables us to restrict the cores that the threads belonging to a process can run on. A JVM also starts the compiler and GC threads before beginning the execution of an application. JVMs provide flags to specify the number of threads for compilation and GC services. The HotSpot JVM provides the `-XX:CICompilerCount` flag to specify the number of compiler threads and the `-XX:ParallelGCThreads` flag to specify the number of parallel GC threads. We use `taskset` to ensure the same CPU cores are used for every benchmark invocation. While using a system with multiple NUMA nodes, we use the second NUMA node when the first one is fully occupied. This approach aims to maximise the proportion of accesses to the faster region of memory. We do not tune thread pinning further because it is out of the scope of this work.

Limiting the non-deterministic behaviour of the JVM

A JVM hosted execution involves multiple non-deterministic aspects, such as compilation of methods and garbage collection (GC) cycles. A different compilation order of methods may cause variations in performance. Therefore, the JIT-compilation order should be recorded; and the compilation decisions should be replayed while executing the benchmark, to reproduce the behaviour. We considered this option less feasible and our attempt to achieve similar behaviour made executions more non-deterministic. In our attempt, we disabled the background compilation of methods and used a single compilation thread to make the order of JIT compilation of the methods reproducible. However, occasionally we noticed that the computationally important methods could not get JIT-compiled. This unexpected behaviour is caused when the compilation request for a method got expired because the previous requests took longer to process. Consequently, those methods were executed using their slower versions; either the interpreted or the one from the lower tier of compilation. Therefore, we decided not to control the JIT compilation order for benchmarks.

On the other hand, to make the GC deterministic, one can disable it completely. However, this may cause execution to exceed the available heap memory or reduce the performance of the next iterations because the previous iterations may impact locality of allocated memory. Requesting a full GC at the beginning of the iteration

is an option but JVMs do not provide any guarantees about when the GC would be performed. Therefore, we decided not to take any steps to control the GC behaviour during the benchmark execution.

4.2.2 Benchmarks Execution Setup

Benchmark Harness

We have set up a benchmark harness using a combination of bash and python scripts to execute benchmarks. We use the harness to measure the execution time of the *warmed-up* iterations for the JVM hosted executions to compare with the equivalent iterations of the native execution. The harness executes a benchmark in a loop and measures the execution time for each iteration of the benchmark. We choose the execution time of the warmed-up iterations after the benchmark reaches its *peak performance*. As discussed in Section 4.1, it is difficult to identify when the application is warmed-up precisely. Therefore, we use the methodology used to evaluate Sulong previously as presented in [RGW+16], where we execute a benchmark for `N=100` times using a harness. We measure the execution time for every iteration and use the geometric mean of the execution times of the last 50 iterations as the execution time for the benchmark. We use geometric mean to reduce the impact of outliers that may get introduced as a result of non-deterministic events such as garbage collection. All the iterations are performed in a single JVM invocation to ensure that the benchmark is warmed-up. We manually ensure that all the computationally intensive methods of the benchmarks are JIT-compiled. We noticed that most of the benchmarks reached a near-steady-state of the peak performance in less than 25 iterations.

All the benchmarks from the selected NPB suite have a custom mechanism to measure the execution time using the C standard library functions declared in `time.h` [BBJ+91]. Every benchmark chooses certain parts of the benchmark based on its functionality, typically the computational kernels, to represent its execution time. They ignore the time spent in setting up the benchmark, such as reading input parameters from a file and initialising the arrays with input data. We modified the benchmarks to run in a loop for the specified number of times. Execution time for every iteration is reported using the existing mechanism. The number of iterations can be specified

as an input to a program. While modifying benchmarks we took additional care to ensure that every iteration executes in the same manner. This typically required re-executing the initialisation step so that the benchmarks and their inputs are set up in the same manner before executing every iteration. All the benchmarks in NPB suite provide a verification function to ensure the parallel execution is correct. The same functions are used to verify the correctness of every iteration. Importantly, we use the same source code to generate a native executable and LLVM IR for the benchmarks to avoid bias, if any present, towards a specific type of execution.

The On-Stack-Replacement (OSR) mechanism allows the JVM to reduce the warm-up time by switching to the compiled version of the method during the execution of the method itself [FF03]. On the contrary, the typical JIT compilations use the compiled version of a method for its *next invocation*. OSR is useful when a method has a long running loop that causes JVM to consider it hot, then JIT-compile, and switch to use the compiled version from the *next iteration*. OSR avoids the wait for switching to the faster mode of execution until the long-running loop completes, thereby reducing the warm-up time. This approach replaces the current version of a method being executed (i.e., the present version of a method on the stack), with the new version. Therefore, the approach is referred to as on-stack-replacement. Sulong-OpenMP extends the version of Sulong that does not support OSR. The OSR support for Sulong is added in the later versions [MLR+19].

The lack of OSR support required additional care while setting up the benchmark harness for Sulong. For executing a benchmark, Sulong executes its `main()` function only once. Thus, in the absence of OSR, Sulong could not switch to the JIT-compiled version of the `main()` function even when it is considered as hot. Subsequently, the computation performed in the `main()` is executed in the slow interpreted mode. Therefore, to ensure that the entire benchmark is JIT compiled, we wrap the entire benchmark in a separate function, and that function is invoked by the harness for the desired number of iterations. Although the Java implementation supports OSR, we use the same approach for the harness executing the Java implementation of the benchmarks to match the execution behaviour of the Sulong's harness.

Summarising the benchmark results

Here, we discuss the statistical methods used to summarise the benchmark results. The benchmark harness contains scripts to process the benchmark output and perform the desired calculations. As discussed previously, to calculate the execution time for a benchmark (T_{bench}) we use the *geometric mean* of the last 50 out of 100 iterations as shown below.

$$T_{bench} = \sqrt[50]{T_{51}T_{52}\cdots T_{100}}$$

The executions using Sulong-OpenMP are slower than their native executions. Therefore, we calculate the slowdown while executing with Sulong-OpenMP relative to their native execution as a metric of comparison. The slowdown for a benchmark (S_{bench}) is calculated by dividing its execution time using Sulong-OpenMP ($T_{Sulong-OpenMP}$) by the time required for executing natively (T_{native}) as shown below.

$$S_{bench} = \frac{T_{Sulong-OpenMP}}{T_{native}}$$

Therefore, the slowdown of 1 represents the execution using Sulong-OpenMP is as fast as its native execution. The slowdown value of 1.20 represents the 20% performance overhead against the native execution. To summarise the results for the entire suite, we calculate the *geometric mean* of the slowdown values for all benchmarks in the suite as shown below.

$$S_{suite} = \sqrt[n]{S_{bench1}S_{bench2}\cdots S_{benchn}}$$

Steadiness of the peak performance is measured by calculating the standard deviation for the last 50 iterations of the benchmark. Calculation of the variations in execution is important for analysis. If the warmed-up iterations have a high degree of variations then it indicates that the peak performance might not have been reached. Additionally, when variations are high, it is difficult to draw a meaningful conclusion about the performance of a benchmark or the underlying implementation.

4.3 Benchmarking Environment

4.3.1 Selected Benchmarks

We use the NAS Parallel Benchmarks (NPB) suite for evaluation of our OpenMP implementation [BBJ+91]. The NPB is a commonly used suite of benchmarks to evaluate the performance of supercomputers in the High-Performance Computing (HPC) community. The NPB suite offers the benchmark versions that are parallelised for both shared and distributed memory clusters. These versions are offered using OpenMP, Message Passing Interface (MPI) and hybrid (MPI + OpenMP) parallelisation approach. We used the 3.0.0 version of the NPB suite that also bundles the Java implementation of the benchmarks. We use the Java version of the benchmarks for comparing with Sulong-OpenMP. This comparison aims to identify potential performance bottlenecks and opportunities for the JVM hosted executions.

The NPB suite has 8 benchmarks — 5 computational kernels and 3 pseudo applications, all derived from the Computation Fluid Dynamics (CFD) applications. The following list provides a brief overview of the benchmarks based on the description provided in [SB96].

- **EP**: kernel benchmark is an embarrassingly parallel problem that represents typical *Monte Carlo* applications. The benchmark performs minimal communication and thus the benchmark can provide an upper limit for floating-point performance that a system and execution environment can achieve.

- **MG**: kernel benchmark is a simplified version of the geometric multigrid kernel that solves a 3-D Poisson Partial Differential Equation (PDE). The benchmark represents a realistic CFD application and focuses on highly structured communication.

- **CG**: kernel benchmark uses the conjugate gradient method to find an approximate solution of the smallest eigenvalue of a large sparse matrix. The benchmark performs sparse matrix-vector multiplication and focuses on unstructured communication.

- **FT**: kernel benchmark uses the Fast Fourier Transform (FFT) to solve a 3-D PDE that includes computations such as matrix multiplication, 1-D, 2-D and

3-D FFTs. Consequently, the FT benchmark can use optimised external library implementations to perform these computations.

- **IS**: kernel benchmark performs a sorting operation that is important in *particle method* codes. The benchmark does not perform floating-point arithmetic. Thus, the benchmark can be used to evaluate the integer performance of the execution environment.

- **LU**: pseudo-application is a lower-upper Gauss-Seidel solver that represents commonly used CFD algorithms. The LU benchmark exhibits a lower amount of parallelism compared to SP and BT benchmarks. This behaviour is reflected in its execution using Sulong-OpenMP (discussed in Chapter 5).

- **SP and BT**: pseudo-applications are the scalar penta-diagonal and block tri-diagonal solvers, respectively. They have similar structures.

Each benchmark has a set of predefined inputs providing 5 different sizes that are categorised into different *classes*. The newer version (v3.4) of the benchmark suite offers three additional bigger classes of the problem sizes. The NPB suite specifies the meaning of classes as listed below:

- Class S: Small input size suited for the test purpose.

- Class W: A problem size suitable for 90's workstation.

- Class A, B, C: Standard input sizes where each class is ∼4x larger than the previous class.

The larger input is useful while scaling the number of OpenMP threads because it creates more work for every thread and avoids thread starving for work. On the other hand, small benchmark sizes more clearly show up inefficiencies in the runtime. We use the class 'W' for kernels and class 'S' for the pseudo-application. These are the smallest two classes, and their choice is made based on two criteria: i) the ability to execute successfully on Sulong; ii) the overall execution time of a benchmark. Executions with the larger inputs crashed the JVM for both Sulong and Sulong-OpenMP. We have not successfully identified the root cause of the crashes. Regarding the execution time,

we noticed the larger inputs for the pseudo applications caused benchmarks to take a significantly long time, over a few minutes for each warming-up iteration, consequently making the execution time for the benchmark suite much longer.

4.3.2 Hardware

The experiments run on a system with 4 physical (8 hyper-threaded) cores Intel Core i7-6700 with 16GB of memory running Ubuntu 18.04 (4.15.0-48-generic). To generate scaling results, we used a larger system with 16 physical cores consisting of 2 Non-Unified Memory Access (NUMA) sockets of Intel Xeon E5-2690 CPUs. The system has 378 GB of memory and Ubuntu 16.04 (4.15.0-36-generic). For both systems, we disable the processor frequency scaling and set it to the maximum of 3.4 GHz (for the 4 core system) and 2.4 GHz (for the 16 core system), by using the *performance* governor.

Modern CPUs change the CPU frequency to optimise power consumption while executing the application. It is referred to as Dynamic Frequency and Voltage Scaling (DVFS) [Wik13c]. At lower frequency, a thread executes slowly, causing longer execution time; vice-versa happens when the frequency increases. In the case of parallel execution, if one of the application threads slows down because of the DVFS then the entire application takes longer to execute as a result of the increased time spent at the synchronisation points. We ensured the constant CPU frequency, by selecting the performance governor that sets the frequency to the maximum value for all cores of the processor.

4.3.3 Software

Our OpenMP extension, Sulong-OpenMP, is built on top of the commit *b0ab114* [1] of Sulong that is part of the GraalVM release candidate 1.0.0-rc6. The underlying version of Sulong supports LLVM 6.0. Thus, we use *clang* from the pre-built bundle of binaries for the LLVM 6.0.0. We use Clang with `-O2` optimisation flag to generate LLVM IR for OpenMP programs. Native executions use the same compiler settings which we use for generating LLVM IR for Sulong-OpenMP. To use Graal as a top-tier JIT compiler, we need to use a JVM that has JVMCI support enabled. Oracle Labs

[1] https://github.com/graalvm/sulong/commit/b0ab114

provide pre-built packages of the JVMCI enabled JDKs built on top of OpenJDK. We use the JDK version 1.8.0_172 that supports JVMCI version 0.46.

4.3.4 Visualisation Techniques

Benchmarking

We use the bar charts for visualising the benchmark comparisons using various modes of executions. Each benchmark has a set of bars representing each type of executions such as native, Java, Sulong and Sulong-OpenMP. We put an additional group of columns to summarise the benchmarks results. We use violin plots [Was12] to show the steadiness of peak performance of the benchmarks. We use the same warmed-up iterations that are used to calculate the execution time for the violin plots of the corresponding benchmarks.

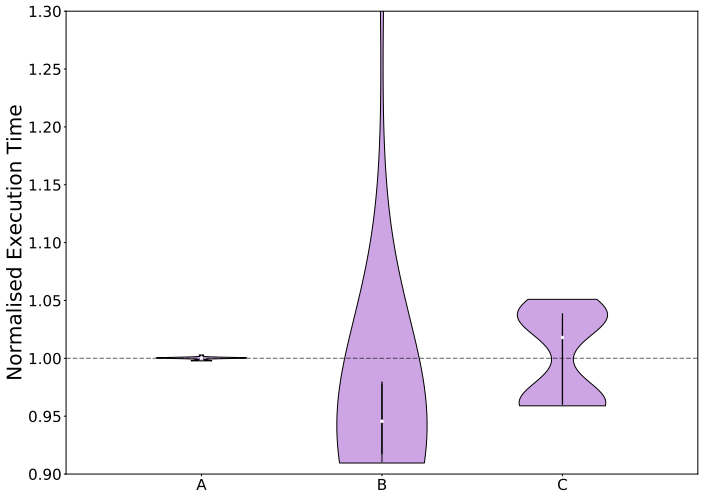


Figure 4.2: Figure shows an example of a violin plot that are used to show steadiness of peak performance of the selected benchmarks.

Figure 4.2 shows an example of a violin plot used in Chapter 5 to show steadiness of the peak performance. The violin plot shows execution time for the last 50 iterations which are used to measure the peak performance of that benchmark. The execution time of each iteration is normalised to the representative execution time of that benchmark, i.e., the geometric mean of the last 50 iterations. In Figure 4.2,

execution time for three benchmarks namely A, B and C are shown. The black line in the middle of each violin marks iteration at 25^{th} and 75^{th} percentile. A white dot at the centre represents the median value. Here, the length of a violin corresponds to the variations in execution time. Thus, a longer violin represents higher variation. In Figure 4.2, benchmark A shows the least variations amongst the three benchmarks. The benchmark C has the majority of the iterations either above or below its representative execution time. The benchmark B has a majority of the iterations below its representative execution time but, a few of them took much longer (up to 30% more time) to finish which increased the representative execution time.

Performance Analysis

We use Flamegraphs [Gre16] to visualise the execution profile of a benchmark. We use our sampling profiler (discussed in Chapter 6) to record the call-stack samples during the execution of a benchmark. We use flamegraphs to identify performance bottlenecks by visualising the time spent while executing a benchmark. Now, we explain the generation of a flamegraph using the recorded call-stack samples with an example.

```
1  void evaluate() { /* Expensive Computation*/ }
2  void initialise() { /* Initialising Data */ }
3  void compute() { evaluate(); }
4  void output() { /* Output Results */ }
5  int main()
6  {
7     initialise(); //20% of the time
8     compute();    //60% of the time
9     output();     //20% of the time
10    return 0;
11 }
```

Listing 4.1: A pseudo-C code with three functions where different percentage of the time is spent during execution.

The Listing 4.1 shows a pseudo-C code comprising three functions where the code spends time: 20% in `initialise()`, 60% in `compute()` and 20% in `output()`. The `compute()` function calls the `evaluate()` function where it spends all of its time. If we stop the execution of this a program after a specific interval, and record the execution call-stack at that time; we would find approximately 60% of the recorded samples reporting that the program is executing the `compute()`, 20% of the time `initialise()`, and so on. Note that the accuracy of the recorded profile is proportional to the *sampling frequency*: the frequency at which the call-stacks are recorded. When we use flamegraphs to visualise the recorded samples, the resulting flamegraph would be illustrated in Figure 4.3.

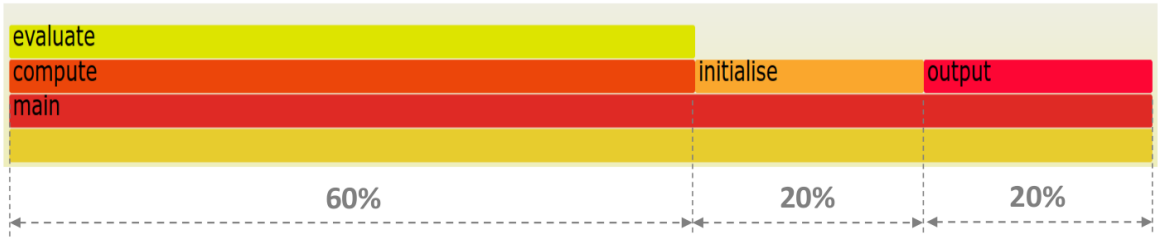


Figure 4.3: An example of a flamegraph that visualises the execution profile of the code in Listing 4.1.

In Figure 4.3, the percentage of width on X-axis is proportional to the percentage of time spent in the respective function. Thus, the `main()` function has the width corresponding to 100%. The Y-axis represents the call-stack depth. Thus, the `evaluate()` function is on top of `compute()` function representing that the 60% of the time spent in the `compute()` function is in turn spent in the `evaluate()` function.

Figure 4.4 shows an example of the flamegraph generated for the execution of a real-world application. The flamegraph visualises the call-stacks recorded on executing the `regexdna` benchmark from the Shootout benchmark suite [Guo18]. The Shootout benchmark suite is implemented in multiple programming languages. The profile in Figure 4.4 uses the Java version of the benchmark. The colour of a stack frame in the flamegraph represents the type of the frame. Yellow represents C++ code, green is Java JIT-compiled code, teal represents inlined Java methods, red is native/library code. Occassionally, `perf` may fail to walk the call-stack, resulting in broken call-stack samples. In the flamegraph, broken call-stack samples are reported as `unknown` (more details in Chapter 6). Flamegraphs can highlight the call-stacks matching the specific

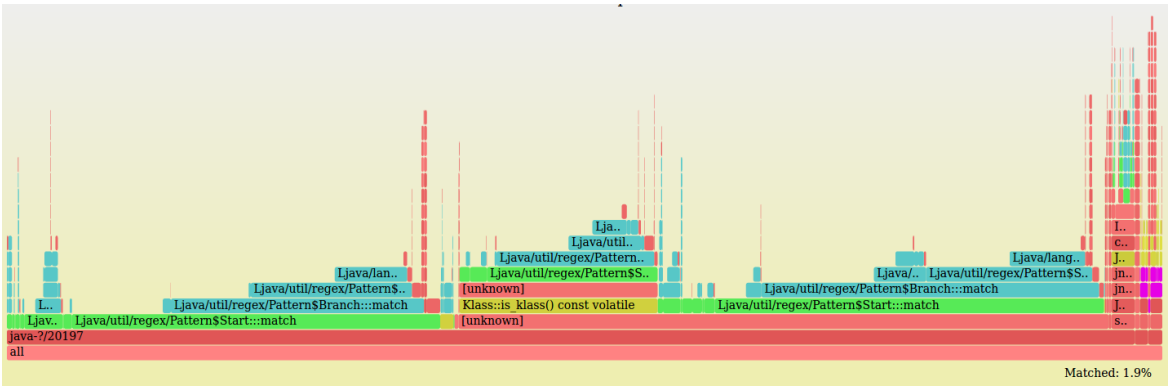


Figure 4.4: Flamegraph of a profile for executing Java version of the `regexdna` benchmark from the Shootout benchmark suite. Yellow represents C++ code, green is Java JIT-compiled code, red is native/library code or it is marked as Interpreter. The magenta colour highlights the call-stacks matching the searched keyword, 'GC'.

keyword. Figure 4.4 shows the highlighted call-stacks in the magenta colour that match the keyword 'GC'. At the right bottom corner of the flamegraph, the percentage of the matched call-stack samples is reported. In this example, 1.9% of the call-stack samples matched the keyword 'GC'. We can use this value to approximate the time spent by the benchmark while performing GC activities. Note, the GC is performed on separate threads that need to be accounted while estimating the time spent in GC.

4.4 Summary

In this chapter, we discussed the metrics which are widely used for evaluating JVM hosted executions, that include the warming-up and warmed-up execution phases. We will use the peak performance as a metric for evaluation of the benchmarks on Sulong-OpenMP. We also discussed the aspects that are important for the JVM hosted execution, but are not measured explicitly. They include the overhead of garbage collection and start-up time.

The execution of a program on a JVM involves several components interacting in a non-deterministic manner, such as the JIT-compilation order of methods or triggering of the garbage collection. This makes the JVM hosted executions difficult to reproduce. We discussed the steps that are commonly used to reduce the non-determinism and make the executions more consistent while benchmarking the JVM hosted executions.

We have created the benchmark harness to execute the selected NAS Parallel

Benchmark suite. The harness implements the evaluation methodology that has been used for the evaluation of Sulong in the past. The harness ensures that the benchmarks are warmed-up by executing them in a loop, and reports the execution time of each iteration. Every iteration of the benchmark initialises its input identically, and the results after execution are verified for correctness.

Modern CPUs use techniques such as frequency scaling to optimise power consumption that may vary the execution speed unexpectedly. We discussed the configuration of the hardware platform that we use for benchmarking. We configure constant CPU frequency to reduce the variations in the execution time. Finally, we discussed the techniques to visualise the benchmarking results using bar charts and execution profiles using the flamegraphs.

Chapter 5

Evaluation

This chapter presents the evaluation of our OpenMP extension to Sulong, Sulong-OpenMP, using the NAS Parallel Benchmark (NPB) Suite. We begin with a high-level overview of the performance comparisons provided in this chapter and the rationale behind them. It is followed by the comparison of the single and multi-threaded performance of Sulong-OpenMP with the native and Java implementations of NPB suite. We conclude our discussion with a deep dive into the various optimisations done to reduce the overhead for the single-thread performance of Sulong-OpenMP. This chapter contains a superset of the results that we presented at the MPLR 2019 conference [GNL19].

5.1 Overview

The objective of the evaluation of Sulong-OpenMP is to identify the performance bottlenecks and reduce the performance gap relative to the native execution. The native executions use a binary executable generated using clang with optimisation level `-O2`. As Sulong-OpenMP extends Sulong to execute OpenMP programs on a JVM, this chapter aims to identify the overheads incurred from different layers used during the execution. We discuss the overheads incurred by using the JVM, Sulong and Sulong-OpenMP. Both Sulong-OpenMP and the native execution use the same benchmark source code to generate LLVM IR and binary executable respectively using the same version of clang. Therefore, we use the native executions as a reference implementation for Sulong-OpenMP and use the same to compare with.

As we discussed in Chapter 3, Sulong-OpenMP is built on top of Sulong. Consequently, Sulong-OpenMP inherits both the pros and cons of the underlying system. To highlight the overhead of the underlying system, we compare the sequential execution using Sulong (Sulong-sequential) to the sequential native execution (Native-sequential). We use Sulong-sequential as a baseline to evaluate the overhead of our implementation. Thus, we compare Sulong-sequential with the OpenMP version of the benchmark executed on Sulong-OpenMP with 1 OpenMP thread (Sulong-OpenMP(1 thread)). Here, the rationale is that the OpenMP extension added to Sulong should not impact the single-thread performance of Sulong. Therefore, theoretically, our objective should be to achieve execution of Sulong-OpenMP (1 thread) with zero overhead compared to the Sulong-sequential. However, this would not be a fair target because of the differences in generated LLVM IR that we discussed in Section Section 3.1. The LLVM IR generated with OpenMP support has calls to the OpenMP runtime library function that are not present for the sequential version, generated using clang but without the `-fopenmp` compiler flag. Hence we expect an additional overhead for the runtime library calls while executing Sulong-OpenMP (1 thread) compared to Sulong-sequential. To determine whether the observed overhead is acceptable, we do a similar comparison of Native-sequential with Native-OpenMP (1 thread).

Execution on a JVM uses an entirely different approach than the AOT compiled approach used by the native execution. Therefore, to understand the implications of executing on a JVM, we use the Java implementation of the benchmarks from the NPB suite. We compare the sequential and the multi-threaded executions of the NPB suite benchmarks in Java to their equivalent executions with native and Sulong-OpenMP. It is important to note here that comparing the different language implementation is an area of research in itself [MDM16]. Our objective behind such a comparison is to gauge the optimisation potential and find possible opportunities for the execution of Sulong-OpenMP on JVMs. If the Java implementation of a benchmark is faster than the Sulong-OpenMP, we consider this as an opportunity for improving Sulong-OpenMP or Sulong itself, because both are implemented in Java and use a JVM for execution. However, if the Sulong-OpenMP is faster than the Java implementation, it is highly likely as a result of the sub-optimal Java implementation of the benchmark that we do not discuss at length.

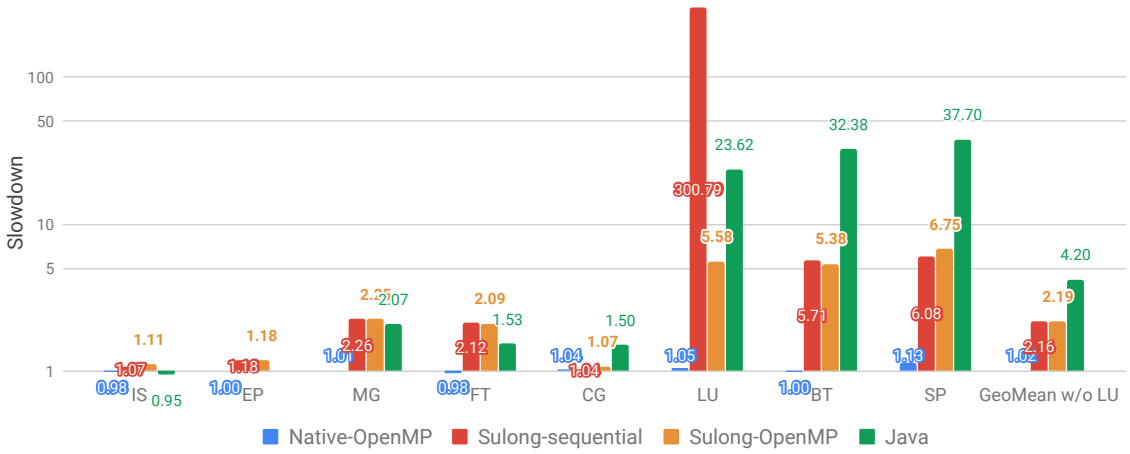


Figure 5.1: Comparison of overhead for executing on Native-OpenMP (1 thread), Sulong-sequential, Sulong-OpenMP (1 thread) and Java (1 thread). The execution times are normalised to execution time of the Native-sequential. Logarithmic scale on Y-axis depicts the slowdown compared to the native thus, 1 represents zero overhead while lower is better.

In the case of multi-threaded executions, we mainly compare the scaling behaviour to identify any parallelisation overhead incurred by the implementation in addition to the ability of an application to scale. Therefore, the objective of the Sulong-OpenMP is to match the scaling behaviour of the execution of Native-OpenMP. Also, it is important to note here that so far our primary objective had been to minimise the overhead for executing Sulong-OpenMP (1 thread) compared to Sulong-sequential. Thus, we have not much pushed for improving the multi-threaded performance yet. The objective of Sulong-OpenMP is not to replace the native execution approach, rather complement it by offering additional features, such as detecting concurrency issues and memory leaks. Therefore, the objective behind improving multi-threaded performance is to increase the usability of Sulong-OpenMP.

5.2 Single Thread Performance

Figure 5.1 shows a comparison of a single-thread performance that includes Native-OpenMP (1 thread), Sulong-sequential, Sulong-OpenMP (1 thread), Java (1 thread). The figure shows execution times are normalised to the execution time of Native-sequential. The Y-axis represents slowdown using a logarithmic scale thus, lower values are better. Here, 1 on the Y-axis represents the benchmark executed as fast

as its Native-sequential equivalent. The values above 1 represent the execution time longer than the corresponding Native-sequential execution.

Native executions

The geometric mean of the overhead of the Native-OpenMP executions is about 2% compared to Native-sequential executions. This overhead is primarily caused because of the additional OpenMP runtime library calls in the LLVM IR of the OpenMP versions of the benchmarks. The runtime calls impact performance in two ways. First, the runtime calls have the typical function calling overhead that involves creating a new stack-frame for making the call and storing the current execution state to resume when the runtime call finishes. Second, the runtime calls perform the unnecessary computation for distributing iterations of the loops amongst the available OpenMP threads. For the sequential executions, the computation is unnecessary as there is only one thread available and the runtime assigns the whole iteration space to it. Therefore, the slowdown is proportional to the time spent in the runtime function calls made by the function. On the contrary, some of the kernels executed little faster with Native-OpenMP compared to the native execution, such as `IS` and `FT`. We hypothesise that such a difference arises from the difference in the generated code that impacted the other optimisations done by the clang. For example, we discussed in Section 3.1 that the OpenMP regions get outlined into a different functions. This may lead to inlining of some functions in the outlined OpenMP region that could not be inlined in the sequential version because of the compilation heuristics. We do not investigate this further, as the purpose of the evaluation is to understand the impact of enabling OpenMP support on the single-thread performance of Native-OpenMP so that we can compare the Sulong-sequential and Sulong-OpenMP fairly.

Sulong-based executions

In the case of Sulong-sequential executions, the geometric mean of the slowdown compared to the Native-sequential execution is about 2.16x (116% overhead). This is the overhead of the underlying system Sulong-OpenMP inherits. Therefore, to measure overhead incurred by adding Sulong-OpenMP support, we use Sulong-sequential executions as a baseline. The geometric mean of the overhead for Sulong-OpenMP relative

to Sulong-sequential is about 3% but not zero. To determine whether Sulong-OpenMP has an acceptable single-thread overhead, we perform a similar comparison for the native executions, i.e., Native-OpenMP (1 thread) relative to Native-sequential. The overhead of OpenMP support for the native execution is about 2%. Thus, although the overhead for Sulong-OpenMP is not zero, we consider this is an acceptable overhead. Section 5.4 presents a detailed discussion on the optimisations implemented in Sulong-OpenMP that bring the overhead for supporting OpenMP down in an acceptable range.

We do not include the execution time for the `LU` benchmark while calculating the geometric mean as highlighted in Figure 5.1. The Sulong-sequential execution of the `LU` benchmark takes significantly longer, a slowdown of about 300x, compared to the Native-sequential. Incorporating the execution time for `LU` in geomean calculation incorrectly shows Sulong-OpenMP faster than the Sulong-sequential thus, geomean does not include the `LU` benchmark. During the Sulong-sequential execution of the `LU` benchmark, one of the computationally intensive function cannot get JIT compiled because it is too big. The Graal compiler cannot JIT compile the big function because the number bytecodes for its Truffle-AST representation exceeded the limit on the maximum number of bytecodes that Graal can compile. Consequently, the method gets executed in the slow interpreted mode. On the other hand, Sulong-OpenMP execution of the `LU` benchmark does not have a similar slowdown. The big function contains multiple OpenMP pragmas that chop the large body into small functions while generating LLVM IR. As discussed previously in Section 3.1, the OpenMP regions get outlined into separate functions when the LLVM IR is generated. The outlined functions are small enough so all of them get JIT-compiled. This results in faster execution of the whole function compared to its interpreted mode of execution which happens in the case of Sulong-sequential. Therefore, the large difference between the executions with Sulong-sequential and Sulong-OpenMP for the `LU` benchmark is the difference between JIT-compiled and interpreted mode of execution.

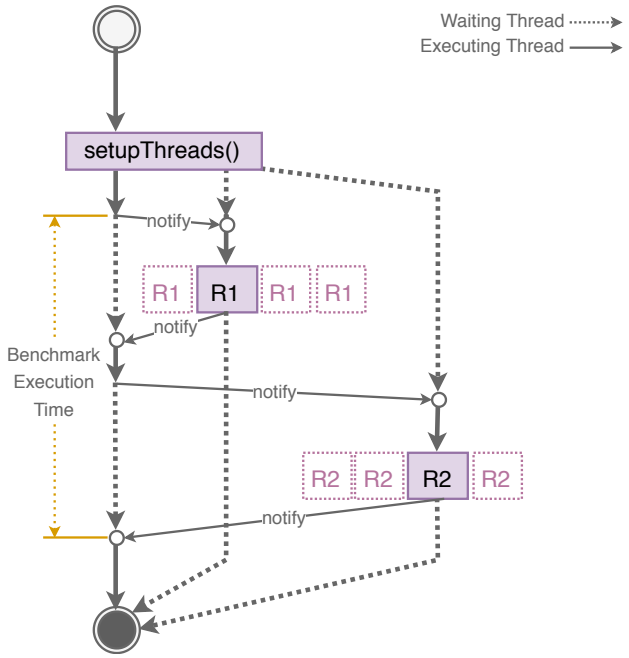


Figure 5.2: A block diagram representing execution of the Java implementations of the benchmarks from NPB suite. The diagram highlights the single-threaded execution. The parallel execution is similar to the single-threaded execution which is shown using the purple dotted blocks.

Java-based executions

The execution of Java implementation of the NPB suite is slower than the Native-sequential execution except for the `IS` benchmark. The Java version of `EP` benchmark is not reported because the NPB suite does contain it. Compared to both Sulong-sequential and Sulong-OpenMP, the Java executions are faster on kernels and slower on the pseudo-applications of the benchmark suite. The primary reason behind the faster/slower execution behaviour of the Java version of the benchmarks is their implementation choice.

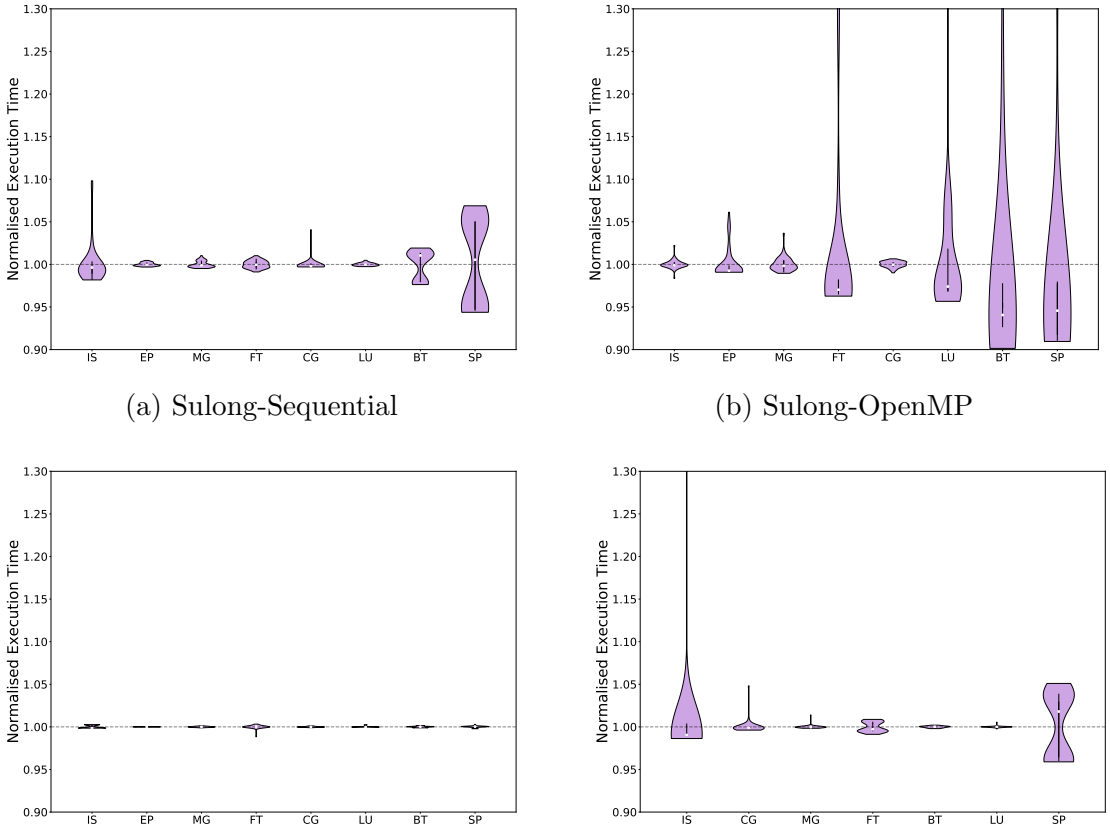
The execution model for of Java implementation of the benchmarks from the NPB suite is shown in Figure 5.2. The execution resembles the master-slave model than the fork-join model used by the C implementations. The Java implementations move parallel parts of the benchmark into pre-defined methods. R1 and R2 in Figure 5.2 represent the parallel regions of the benchmarks. The pre-defined methods are equivalent to the OpenMP regions in the C versions. Thus, a benchmark forks Java threads that then execute predefined methods and synchronise afterwards to achieve behaviour equivalent to the OpenMP C programs. To achieve the behaviour of `OpenMP For`, the

Java implementations use a method called `setupThreads()`. The `setupThreads()` method configures a local iteration range for a thread based on specified threads count. Once the threads are configured, the master thread notifies slave threads to execute the iteration range assigned to them. Importantly, the computation performed in the `setupThreads()` method is not accounted towards the computation time of the benchmark. On the other hand, the C implementation performs the calculation of local ranges using the runtime calls, such as `@__kmpc_for_static_init_4()`, to the OpenMP runtime library. The runtime calls performed for the native implementation are accounted towards their execution time.

In the case of `IS` benchmark, the C implementation performs all the computation by calling the `rank()` function in a loop. Thus for both the native and Sulong based executions, allocation of the local variables in `rank()` function is done for every loop iteration within the benchmark. However, the Java implementation populates them as the member variables of the object and avoids the allocation during every loop iteration. Such implementation differences helped the Java implementation to perform competitively compared to the native executions and better than the Sulong based executions most of the kernels.

The Java implementation is slower on all the pseudo-applications than both native and Sulong based executions. This difference can also be attributed to the suboptimal implementation of the benchmark. Java implementations create dedicated classes that bundle one or more parallel regions of a benchmark. The `setupThreads()` method creates instances of all the thread classes that are necessary for executing the benchmark. In Figure 5.2, for a single-threaded execution, the `setupThreads()` method creates two additional threads for executing R1 and R2. These threads start execution and wait for the master thread to notify them to do the assigned computation. Threads notify back the master when finished. Therefore, the parallel execution with N threads requires the Java implementation to create N * R Java threads where R is the number of the groups of parallel regions. The pseudo-application have higher value of R: LU (5), BT (5), SP (6).

The purpose of evaluating Java implementation of the NPB suite is to identify optimisation opportunities and limitations for JVM hosted executions. Therefore, we



(a) Sulong-Sequential

(b) Sulong-OpenMP

(c) Native

(d) Java

Figure 5.3: Violin plot of the warmed-up execution time distribution for 50 iterations of each benchmark. Y-axis: values closer to 1 represent low variations (better).

do not investigate reasons for slowdown for the Java implementation of the pseudo-applications further. This excludes investigation of the influence of garbage collection done in the background on separate threads or overhead incurred by using region-specific threads.

Steadiness of the peak performance

In the previous chapter, we discussed the steadiness of the peak performance for JVM hosted executions of the benchmarks. If the benchmarks have a high degree of variations then it is difficult to comment about the application behaviour. Figure 5.3 shows steadiness of 4 types of executions using the violin plots. In the violin plots, the execution times are normalised to the representative execution time of the benchmark, geomean of the last 50 benchmark iterations. Thus, values farther than 1 represent

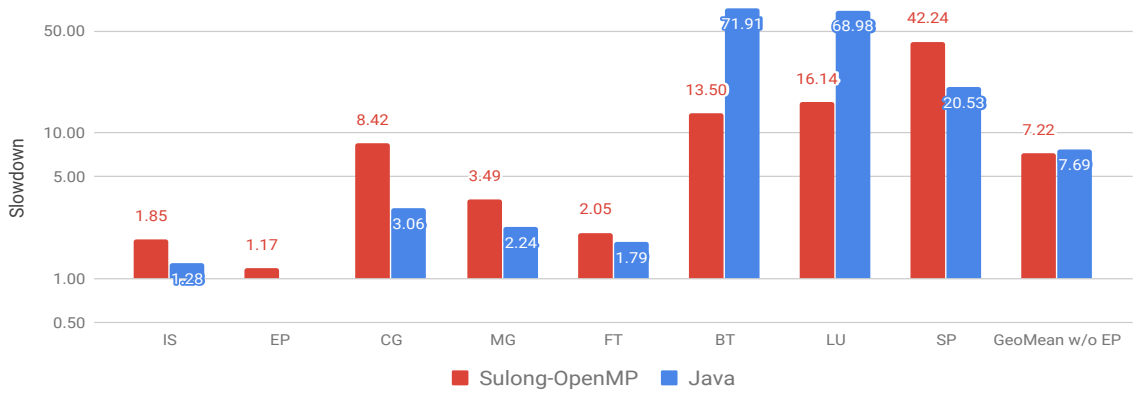


Figure 5.4: The results show slowdown (lower is better) for executing on Sulong-OpenMP (4 threads) and Java (4 threads) relative to the Native-OpenMP (4 threads). Execution times are noramised to the time of the Native-OpenMP executions.

the executions with high variations. Figure 5.3(a) shows that the native-OpenMP execution is the one with the least variations amongst the benchmark iterations. For the Sulong-sequential executions, the kernels have a small variation of about 1% while the pseudo-applications have relatively large variation up to 6%. The Sulong-OpenMP executions have higher degree of variations, kernels (up to 4%) and pseudo-applications (up to 10%). We expect higher variations for the parallel execution compared to their sequential equivalent and do not investigate them further.

5.3 Multi-thread Performance

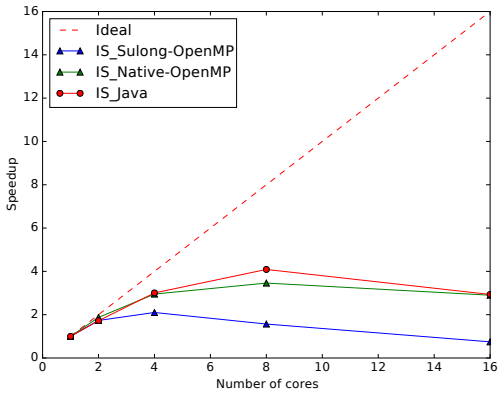
In this section, we discuss the performance of Sulong-OpenMP on multiple threads using two steps. First, we present the results to demonstrate the current implementation status of Sulong-OpenMP compared to the native implementation using clang and Java. Second, we discuss the scaling of Sulong-OpenMP and Java compared to the Native-OpenMP execution.

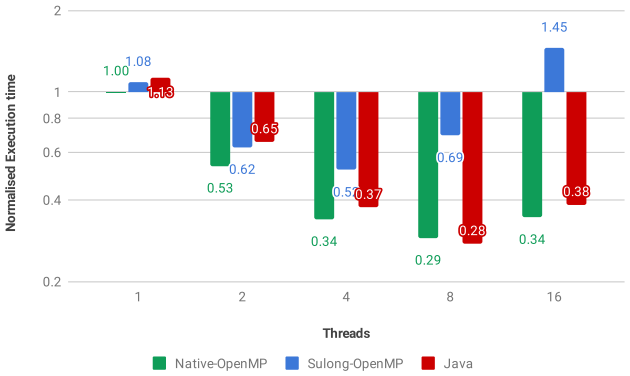
Figure 5.4 shows the slowdown for NPB suite benchmarks using Sulong-OpenMP (4 threads) and Java (4 threads). The execution times in the figure are normalised to the execution time of Native-OpenMP (4 threads). Figure shows the slowdown varies from about 1.17 (∼20% overhead) for the `EP` to 42.2x for the `SP` benchmark (∼4120% overhead). When calculated separately, the geometric mean of slowdown for kernels is 2.66x (∼166% overhead) while for pseudo-applications it is much larger, 20.93x. The

primary reason for the higher overheads across all the benchmarks is the current way Sulong-OpenMP executes OpenMP parallel regions. Execution of the parallel regions on Sulong-OpenMP mainly involves the execution of the `outlined` OpenMP function generated in LLVM IR. Sulong-OpenMP uses a thread-pool to execute the `outlined` functions that impact the execution of pseudo-applications more than kernels.

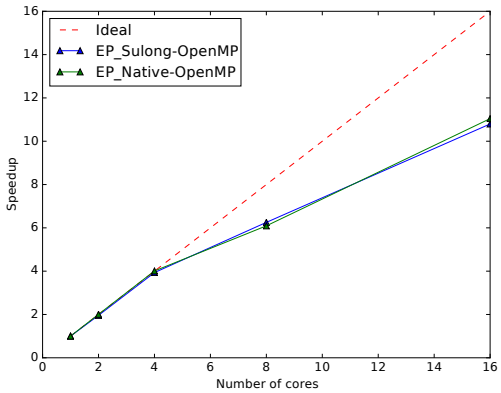
Sulong-OpenMP implements a thread-pool that creates Java threads equal to the number specified by environment variable `OMP_NUM_THREADS`. The thread-pool executes the top-level outlined OpenMP function by taking its state in the form of `Frame`, the representation for the stack-frame of a guest language function, and its arguments. Such assignment of the `Frame` of a function requires it to be *materialized*. The materialization of `frame` limits the Graal compiler to aggressively optimise the parent function that calls the `outlined` OpenMP function. JIT-compiled code for the method with materialized frame relies on Truffle's representation for guest language frames. It requires maintaining additional data structures that limit the compiler optimisations, such as intrinsification of accessor methods [Cor19b]. Pseudo-applications in the NPB suite have more OpenMP parallel regions than the kernels. Thus, the pseudo-applications have relatively higher overhead compared to kernels. We addressed the frame materialization issue for single-threaded executions using a work-around, referred to as *1 thread specialisation*. Here, we avoid the `frame` assignment and execute the `outlined` function on the master thread. Therefore, we hope that this issue can be addressed with additional engineering efforts, which would help to improve the performance of the multi-threaded executions on Sulong-OpenMP.

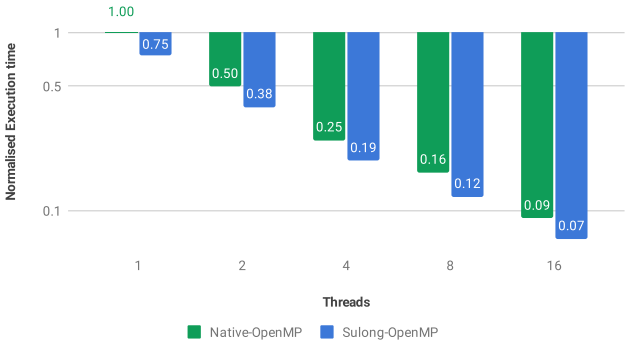
Figure 5.5 shows the scaling results for the benchmarks from the NPB suite. The execution of the EP benchmark on Sulong-OpenMP scales similar to the Native-OpenMP execution up to 16 cores. However, the MG benchmark showed the worst scaling amongst the NPB suite kernels on Sulong-OpenMP compared to the Native-OpenMP execution. In the case of pseudo-applications, the scaling is poor because the small problem size caused many threads to starve for work. As discussed in Section 4.2, Sulong-sequential executions crash the JVM for larger problem sizes which also limited our ability to perform scaling tests. For most of the benchmarks, scaling of Java implementation of NPB suite appears better than Native-OpenMP and Sulong-OpenMP. As described in the previous section, Java implementation may use


Ideal
IS_Sulong-OpenMP
IS_Native-OpenMP
IS_Java
Speedup
Number of cores

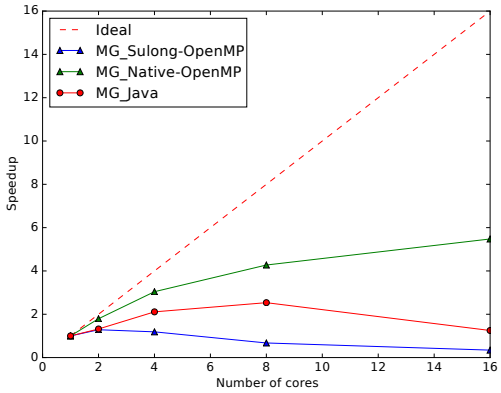

Normalised Execution time
1.00
1.08
1.13
0.53
0.62
0.65
0.34
0.50
0.37
0.29
0.69
0.28
0.34
1.45
0.38
Threads
Native-OpenMP
Sulong-OpenMP
Java

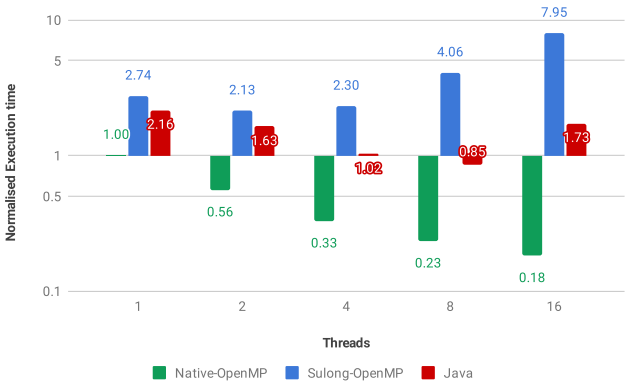
(a) IS


Ideal
EP_Sulong-OpenMP
EP_Native-OpenMP
Speedup
Number of cores

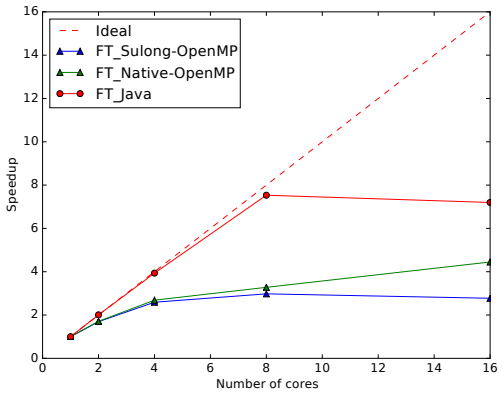

Normalised Execution time
1.00
0.75
0.50
0.38
0.25
0.19
0.16
0.12
0.09
0.07
Threads
Native-OpenMP
Sulong-OpenMP

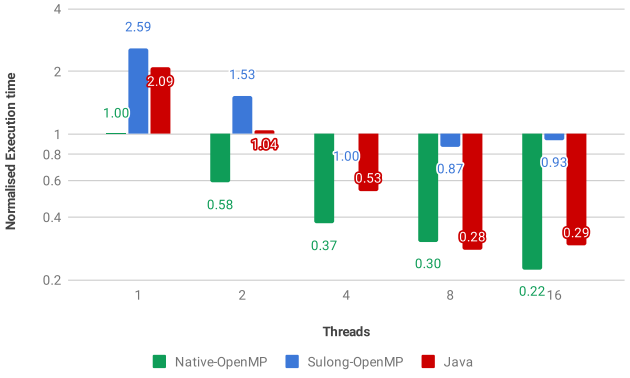
(b) EP


Ideal
MG_Sulong-OpenMP
MG_Native-OpenMP
MG_Java
Speedup
Number of cores

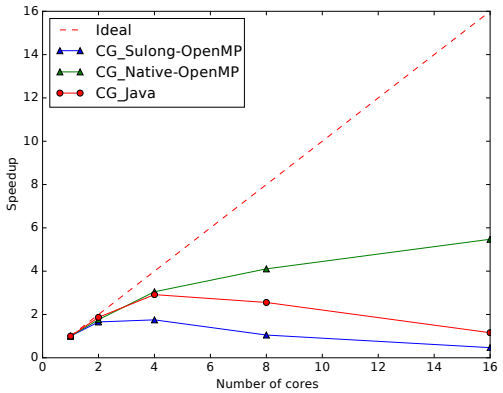

Normalised Execution time
1.00
2.74
2.16
0.56
2.13
1.63
0.33
2.30
1.02
0.23
4.06
0.85
0.18
7.95
1.73
Threads
Native-OpenMP
Sulong-OpenMP
Java

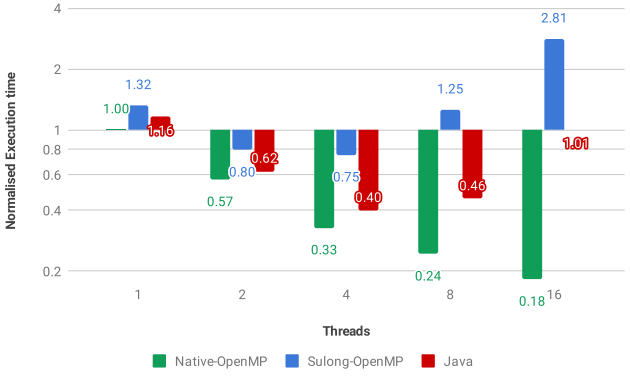
(c) MG


Ideal
FT_Sulong-OpenMP
FT_Native-OpenMP
FT_Java
Speedup
Number of cores

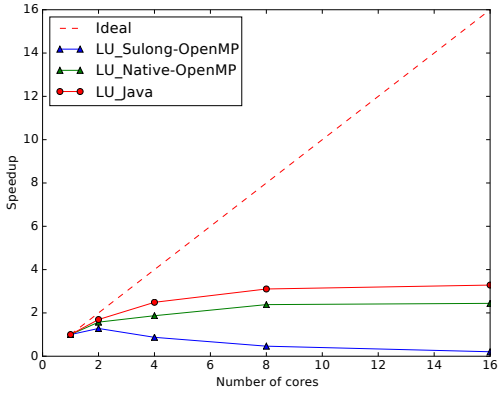

Normalised Execution time
Threads
Native-OpenMP
Sulong-OpenMP
Java

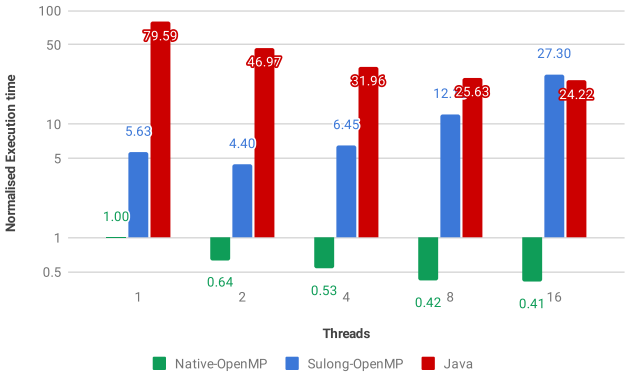
(d) FT


Ideal
CG_Sulong-OpenMP
CG_Native-OpenMP
CG_Java
Speedup
Number of cores

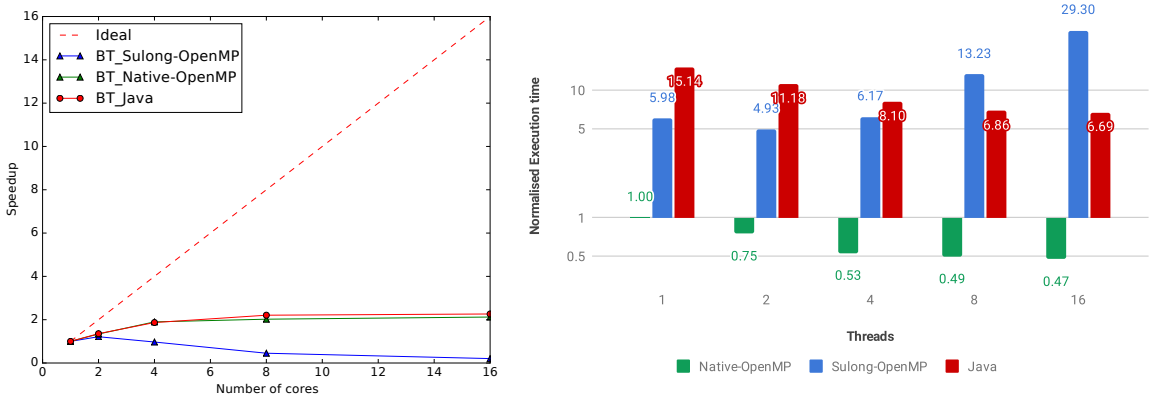

Normalised Execution time
Threads
Native-OpenMP
Sulong-OpenMP
Java

(e) CG

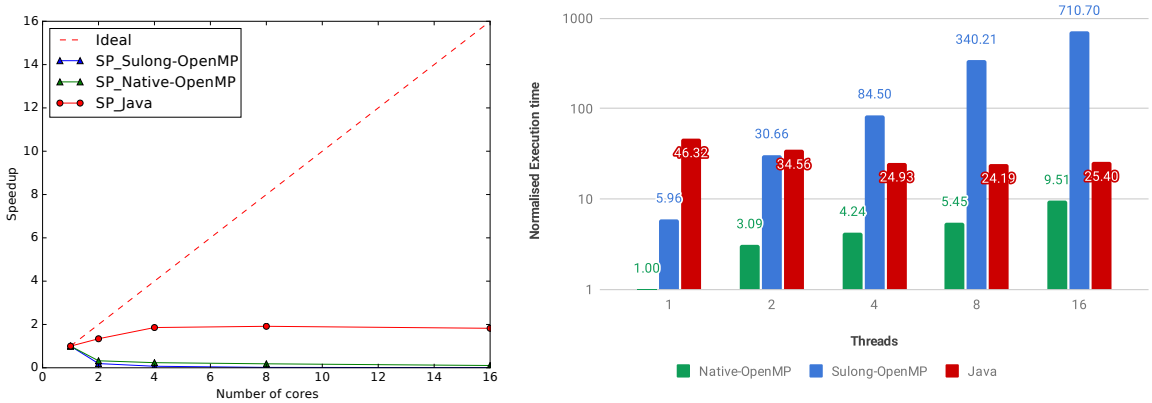

Ideal
LU_Sulong-OpenMP
LU_Native-OpenMP
LU_Java
Speedup
Number of cores


Normalised Execution time
Threads
Native-OpenMP
Sulong-OpenMP
Java

(f) LU



(g) BT



(h) SP

Figure 5.5: The figure shows two plots for each benchmark from NPB suite. i) On the left, scaling plot of the benchmarks are shown where speedup is calculated relative to the time taken using 1 OpenMP thread on that execution environment. These plots help to visualise the behaviour of a benchmark on the given execution environment when number of OpenMP threads are increased. ii) On the right, execution times normalised to Native-OpenMP (1-thread) for that benchmark are shown. These plots help to compare time taken by different execution environments on a uniform scale. Here, value 1 mean execution took same time as Native-OpenMP (1-thread), values less than 1 represent executions faster than Native-OpenMP (1-thread).

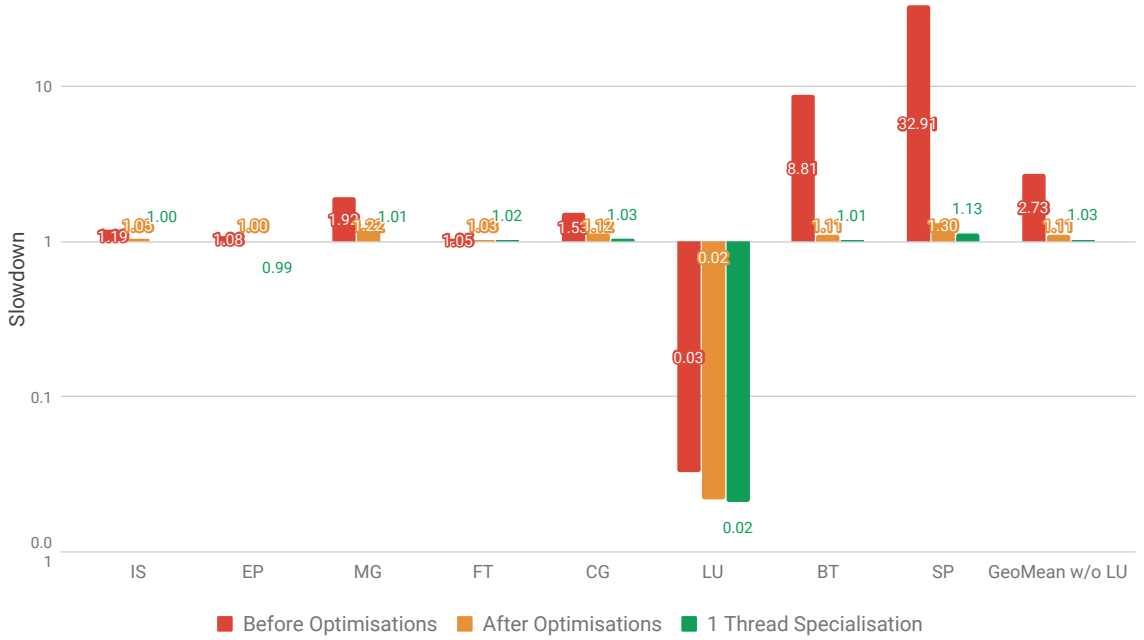


Figure 5.6: A comparision of before and after optimisation of the single thread performance. The execution times are normalised to the execution times of the Sulong-sequential execution. Logarithmic scale on Y-axis depicts the slowdown relative to the Sulong-sequential. Thus, 1 represents zero overhead while lower is better.

multiple times more threads than the specified number of OpenMP-like threads that may benefit the multi-threaded execution. We do not discuss the effects of implementation choice of the Java benchmarks on scaling as it may not help the thread-pool based implementation approach of Sulong-OpenMP.

5.4 Reducing overhead of Sulong-OpenMP

This section describes the optimisation journey of Sulong-OpenMP to reduce the single-thread overhead. Figure 5.6 shows that the overhead of Sulong-OpenMP (1 thread) relative to Sulong-sequential is reduced from about 173% (slowdown 2.73x) to just about 3%. We explain how the performance bottlenecks for the Sulong-OpenMP were identified, which subsequently shaped its implementation.

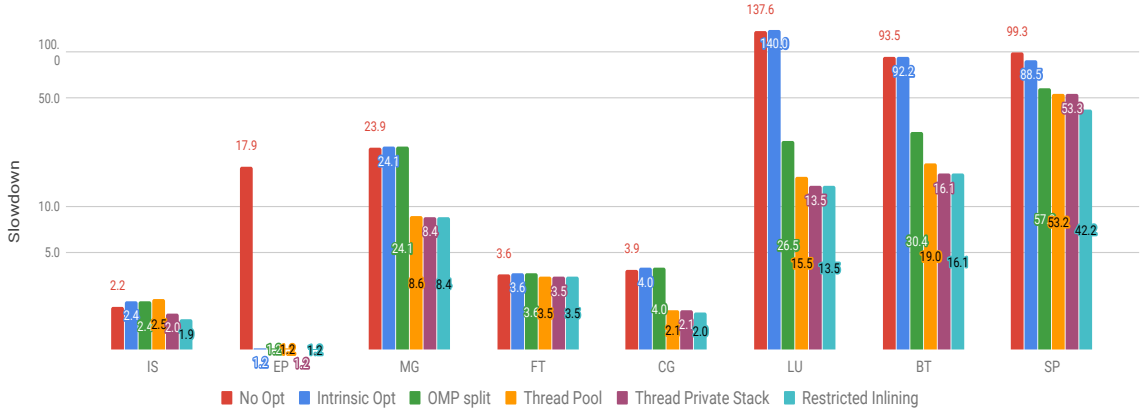


Figure 5.7: The results show slowdown for executing on Sulong-OpenMP (4 threads) compared to the Native-OpenMP (4 threads). Different bars for each benchmark show changed slowdown to represent the impact after applying an optimisation. Logarithmic scale on Y-axis depicts the slowdown compared to the native execution thus, 1 represents zero overhead while lower is better.

Figure 5.6 summarises optimisations applied to Sulong-OpenMP in three categories: i) 'Before optimisations' shows the performance of an initial version of Sulong-OpenMP, which successfully executes the NPB suite using the hybrid approach (described in Section 3.2). ii) 'After optimisations' shows the performance of Sulong-OpenMP on applying the optimisations (described later in this section). iii) '1 Thread Specialisation' shows the performance of Sulong-OpenMP after adding specialisation for the single-threaded execution. This significantly improves the single-thread performance by avoiding the `frame` materialization issue (described in Section 5.3).

5.4.1 Optimising calls to Sulong intrincsics

Figure 5.7 shows the effect of different optimisations on the execution time for executing the NPB suite benchmarks using 4 OpenMP threads. Execution times are normalised to the execution time of the Native-OpenMP (4 threads). Because of its embarrassingly parallel nature, we expected that the `EP` benchmark would have minimal overhead compared to Native-OpenMP (4 threads). On the contrary, we observed a significant slowdown (about 18x) for the `EP` benchmark. Hence, we chose `EP` as the first benchmark to analyse and address performance issues. We use the methodology described in the Chapter 6 for performance analysis. The Sulong-OpenMP (4 threads) executions are profiled, which are visualised using flamegraphs.

The Sulong provides its own implementation for some of the C library functions, such as `sqrt` and `log` using Java `Math` class. These implementations are referred to as *Sulong intrinsics*, and are used instead of their native versions. Intrinsics avoid calling the native library implementations using relatively slow mechanism that involves using the Native Function Interface (NFI) [GRS+13]. Figure 5.8 shows a flamegraph for warmed-up executions of the `EP` benchmark using Sulong-OpenMP (4 threads). The enlarged part of the flamegraph highlights methods, where the execution spends ∼54% of the total execution time. In the enlarged part, almost all the time is spent in two methods: `StackPointer.close()` and `StackPointer.newFrame()`. These methods are called from the `doDirectIntrinsic()` method, which can be seen from the stack frame below the highlighted methods. The `doDirectIntrinsic()` method in Sulong handles the calls to its intrinsic functions. Sulong calls the `newFrame()` and `close()` methods to create and destroy a stack frame for the intrinsic calls respectively.

For the NPB suite benchmarks, we noticed that the creation of a new stack frame is not necessary while calling Sulong intrinsics, and can be avoided. In our initial OpenMP implementation, we used a naïve approach that synchronised the creation of a stack frame. The synchronisation exacerbated the overhead for calling Sulong intrinsics. To reduce overhead for calling Sulong intrinsics, we disabled the creation of call-stack frames while calling them. This approach is analogous to the tail call optimisation technique that is commonly used by compilers [Wik09]. Avoiding the creation of stack frames, significantly reduced the overhead of executing `EP` benchmark. Figure 5.7 shows, on Sulong-OpenMP (4 threads) the overhead from about 1690% (slowdown of ∼18x) to just 20% relative to the Native-OpenMP (4 threads).

5.4.2 Splitting OpenMP regions

After optimising the Sulong intrinsics that benefited only the `EP` benchmark, we chose the pseudo-applications as our next optimisation target. The pseudo-applications were chosen because they were significantly slower (about 100x) compared to the Native-OpenMP (4 threads) as shown in Figure 5.7. The primary reason for the slowdown was that the majority of the code failed to JIT compile and executed in slow interpreted mode. The large size of the compilation unit caused the failure of JIT-compilation. JVM specification limits the maximum size of the code generated by a JVM for a Java

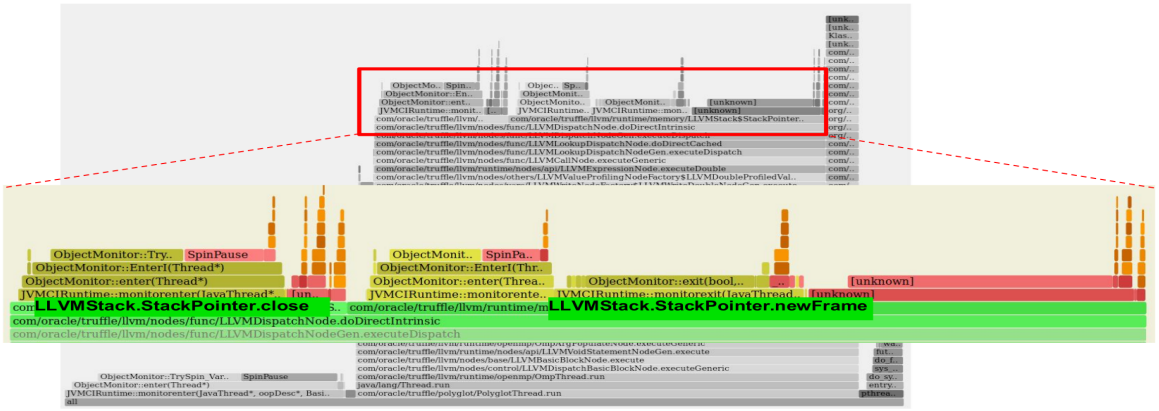


Figure 5.8: Figure highlights a part of the flamegraph that contributes about 54% of the total time spent in executing `doDirectIntrinsics()` method. The primary operations performed by this method involves creating (using `StackPointer.newFrame()`) and destroying (using `StackPointer.close()`) the `LLVMStackFrame`.

method to 64 kB [Cor19a]. When the limit on the method size is exceeded, the Java compiler throws `code too large` exception. Although an individual method may not be too large, the inlining decisions made by the compiler may cause the generated method size to exceed the threshold. In the case of Truffle-based ASTs, inlining the AST node methods of a guest language function may result in a large method. These JIT-compilation failures may severely impact the performance of an application.

In the case of pseudo-applications, computationally intensive parts of the code, covering one or more OpenMP parallel regions, failed to JIT compile. This caused a significant portion of applications to execute in slow interpreted mode. Unlike Sulong-OpenMP, Sulong-sequential executions did not face failures for JIT-compilation. The sequential executions use the LLVM IR, generated without enabling OpenMP support. In the absence of OpenMP support, the Clang made different inlining decisions. The execution of this sequential version of LLVM IR did not breach the limit on generated code. For example, the `@adi` function of the `BT` benchmark is not inlined in its parent function (`@main`) while generating the sequential version of LLVM IR. Instead, the Clang inlined some of the functions called from `@adi` into it. On the other hand, while generating LLVM IR with OpenMP support, the `@adi` function is inlined in `@main`. The difference between these inlining decisions occurs because the way LLVM IR is generated for OpenMP programs. The `@adi` function has five OpenMP parallel regions that outlines them into five separate functions. The outlining chopped the `@adi`

function in five parts in the OpenMP version that does not happen for the sequential version. These initial decisions influenced the inlining of the subsequent functions in the call tree. Thus, the LLVM IR for the same function may look significantly different with and without OpenMP support. The sequential version of the `LU` benchmark faced the large generated code size issue that we discussed earlier, in Section 5.2.

We address the large code size issue using the clang's approach of generating LLVM IR for OpenMP parallel regions. We noticed that the functions generating the large code size contain a large OpenMP parallel region that encapsulates multiple For-loops annotated with `OpenMP-For` pragmas. We manually split those large OpenMP parallel regions into multiple, typically two or three, smaller regions as shown in Figure 5.9. In the case of pseudo-applications, the large OpenMP blocks were typically comprised of multiple separate OpenMP for-loops. For example, the `compute_rhs()` function of `BT` benchmark contains an OpenMP parallel block with 21 OpenMP for-loops. Such large blocks were split into separate smaller OpenMP parallel regions comprising a subset of the for-loops as shown in Figure 5.9. The for-loops typically operated over global arrays and hence splitting the parallel region require fewer code modifications to ensure correctness. The OpenMP parallel block in `compute_rhs()` function is split into three parallel blocks (as 8 + 6 + 7 for-loops).

The splitting of large OpenMP parallel regions, resulted in multiple small `outlined` functions. Therefore, the large functions that could not be JIT-compiled previously, are broken down into functions smaller enough to get JIT-compiled. This code modification did not have any noticeable impact on the performance of Native-OpenMP execution. To ensure consistency, we used the modified version of the benchmarks for further comparisons of Sulong-OpenMP (4 threads) relative to Native-OpenMP (4 threads). This optimisation is only applied to the pseudo-applications that helped to reduce their slowdowns significantly: LU (from 140x to 26.5x), BT (from 92.2x to 30.4x) and SP (from 88.5x to 57.3x). In the case of SP benchmark, the improvement in execution time was not as significant because a larger portion of the benchmark still executed in the slow interpreted mode.

```
1  void foo() {
2  #pragma omp parallel
3  {
4      #pragma omp for
5      for( ) {
6      // For-loop 1
7      }
8      /* computation
9      outside the loop */
10     #pragma omp for
11     for( ) {
12     // For-loop 2
13     }
14 }
15 }
```

```
1  void foo() {
2  #pragma omp parallel
3  {
4      #pragma omp for
5      for( ) {
6      // For-loop 1
7      }
8      /* computation
9      outside the loop */
10 }
11 #pragma omp parallel
12 {
13     #pragma omp for
14     for( ) {
15     // For-loop 2
16     }
17 }
18 }
```

Figure 5.9: Simplified illustration of splitting a large OpenMP parallel region into the small regions. Listing on the left shows a large OpenMP parallel region with two for-loop. Listing on the right shows the larger parallel region split into two small OpenMP parallel regions.

5.4.3 Thread-private stack implementation

The Sulong handles native memory allocations on the stack using the `LLVMStack` class. Implementation of `LLVMStack` uses the Unsafe API of Sun that allows to allocate memory off the Java heap and retrieve its pointer as a long value [MPM+15]. The stack memory is allocated for variables in LLVM IR using the `alloca` instruction. Sulong services these stack allocations using methods of `LLVMStack`. Sulong-OpenMP is built on top of Sulong that supports only the single-threaded applications. Therefore, access to memory allocation methods does not require synchronisation. Initially, to avoid the race between OpenMP threads, Sulong-OpenMP synchronised access to the memory allocation methods. Here, all the threads shared the same instance of `LLVMStack`. In addition to the synchronisation overhead, the stack sharing approach also impacted the CPU caching when different OpenMP threads allocated and accessed memory on a single cache line.

To highlight the importance of the thread-private instance of `LLVMStack`, we explain how Sulong supports the stack allocations using the `LLVMStack`. Listing 5.1 shows a simple C function (`incr()`) that increments and returns its only the integer argument.

```
1  int incr( int a ) {
2      return a + 1;
3  }
```

Listing 5.1: An `incr()` function in C that increments and returns the argument.

```
1  define i32 @incr(i32) {
2        %2 = alloca i32
3        store i32 %0, i32* %2
4        %3 = load i32, i32* %2
5        %4 = add i32 %3, 1
6        ret i32 %4
7  }
```

Listing 5.2: Simplified LLVM IR for the `incr` function from Listing 5.1.

Listing 5.2 shows LLVM IR for the `incr()` function. The line 2 of Listing 5.2 shows an `alloca` instruction that is used to allocate space for a 32-bit integer pointed by the variable %2 on stack. The %2 variable stores the argument `a` passed to the `incr()` function. The `store` and `load` instructions on line 2 and 3 respectively, first copy and then load the first argument of the function from the address pointed by %2. The value argument stored in %3 is incremented using the `add` instruction, and the resulting value is returned using the subsequent instruction. The LLVM IR from Listing 5.2 is generated without any optimisations for the purpose of simplicity. The LLVM IR for an application or a benchmark contain many `alloca` instructions.

Now we explain the execution of LLVM IR shown in Listing 5.2 on Sulong using `LLVMStack`. At the beginning of the execution, Sulong allocates a sufficiently large portion of memory (20 MB in the default case) `LLVMStack`. Sulong allocates memory using the `UNSAFE` API from Sun that returns a raw pointer to a memory location as a long value. `LLVMStack` uses that raw pointer to perform the pointer arithmetic while serving memory allocation requests. `LLVMStack` creates a new stack-frame for every function in LLVM IR where the memory for local variables is allocated. This stack-frame is destroyed when the function returns. `StackPointer` class manages the local memory allocation on stack-frame. The `StackPointer` is incremented to allocate memory requested by the `alloca` instruction. Effectively, the `StackPointer` class emulates the behaviour of native stack pointer for a C function. Each LLVM IR function uses a `StackPointer` instance to store the starting address of the current stack-frame. When a function returns, its `StackPointer` is reset to the starting address of stack-frame.

To support multi-threaded execution using the `LLVMStack` and the `StackPointer`, we shared the same instance of `StackPointer` amongst all OpenMP threads. To avoid races during the allocation of local variables, access to the stack-pointer manipulating methods were synchronised. Figure 5.10 shows the allocation of local variable %2 from Listing 5.2 on the `LLVMStack`, using the stack-pointer sharing mechanism. Unnecessary synchronisation involved in this approach made it slow and inefficient. Further, when the size of allocations is sufficiently small, it may cause cache thrashing.

We address the issue of the shared stack by allocating a separate `LLVMStack` instance for each OpenMP thread. Sulong is modified to use the private instance of

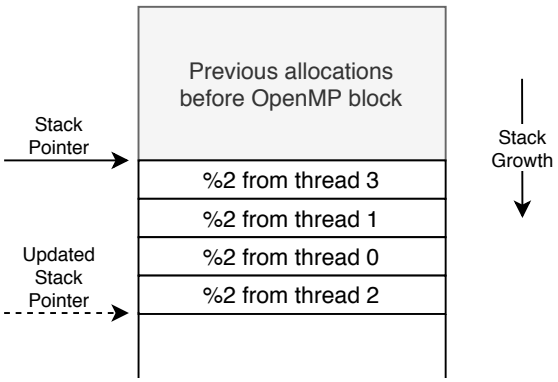


Figure 5.10: The Figure shows the allocation of the local variable named `%2` on the `LLVMStack` by four OpenMP threads while executing the alloca instruction on the Line 2 of Listing 5.2. In the absence of the thread private stack mechanism, the stack pointer is shared by the OpenMP threads that result in the allocations to be performed at the subsequent memory locations. Such subsequent allocations performed by multiple threads may suffer from cache trashing when they lie on the same cache line.

`LLVMStack` to perform thread-local allocations. In this way, Sulong-OpenMP effectively creates a private stack per thread.

Use of thread-private stacks avoids the synchronisation overhead during the allocation of local variables. Thus, the benefits of this change are proportional to the number of stack allocations performed by a benchmark. The thread-private stacks mainly benefited the `IS` benchmark from the NPB suite. Slowdown for the `IS` benchmark is reduced from about 2.5x to 2.0x on Sulong-OpenMP (4 threads) compared to Native-OpenMP (4 threads). The other benchmarks from the NPB suite did not benefit as much as the `IS` benchmark. These benchmarks performed a majority of their memory accesses on global data structures, such as arrays, that are not allocated on thread-private stacks.

5.4.4 Thread pool implementation

Initially, we used a naïve approach where new Java threads were created at the beginning of every OpenMP block, and they are merged/destroyed at the end of the block. This approach simplified the implementation of an implicit barrier at the end of the OpenMP blocks. Undoubtedly, this approach was suboptimal because it creates/destroys threads repeatedly while executing a program with multiple OpenMP blocks. In the case of the `MG` benchmark, the approach created/destroyed threads 1820 times

Figure 5.11: Flamegraph of CPU profiled execution of MG benchmark from NPB suite on Sulong-OpenMP (4-threads). An enlarged section of the flamegraph at the bottom demonstrates that circa 69% of the collected samples are related to garbage collection. The enlarged top portion of the flamegraph shows the samples from OpenMP application threads where the highlighted stack frames in magenta colour are for the functions that create/initialise Java threads.

within a single execution of the benchmark. The benchmark executes an OpenMP block in a loop, that required repeated thread creations. Such a large number of thread creation also increased pressure on the garbage collector (GC), which can be observed in Figure 5.11. Therefore, we planned to implement a thread-pool to use the fixed set of Java threads throughout the execution.

Figure 5.11 shows a flamegraph of the warmed-up execution of `MG` benchmark before the implementation of thread pool. As discussed in Section 4.3, the width on the X-axis of a flamegraph is proportional to the time spent in a particular method whilst the Y-axis corresponds to call-stack depth. The enlarged section at the bottom of the flamegraph shows that approximately 69% of the call-stack samples match the methods associated with garbage collection. This indicates significantly high GC pressure during the execution of the benchmark. Although certain aspects of garbage collection can be performed concurrently on the background GC threads, it is still necessary to fully halt application threads for portions of garbage collection. The enlarged section at the top of the flamegraph shows the approximately 11% of the overall samples (highlighted in magenta colour) match the methods performing the thread life-cycle management activities. It is important to note that, the flamegraph includes the call-stack samples for the threads running in parallel. Thus, the application may have spent more than 11% of its wall-clock execution time in initialising/destroying threads.

Sulong-OpenMP implements a thread-pool where it creates the required number of OpenMP threads at the beginning of the execution. Each thread in the thread-pool waits for `work` to be assigned. Whenever an OpenMP block is encountered, the Truffle-based AST associated with the block is given to all the OpenMP threads for execution. The thread-pool implementation reduced slowdown for the `MG` benchmark from about 24.1x to 8.6x (speedup of about 2.8x). The other benchmarks that considerably benefited from the thread-pool are CG (from 4.0x to 2.1x), LU (from 26.5x to 15.5x) and BT (from 30.5x to 19.0x).

5.4.5 Restricted Inling

After applying the optimisations discussed earlier, the `SP` benchmark had maximum overhead amongst all the NPB suite benchmarks. `SP` had overhead of 430% (slowdown

of 5.3x) while executing on Sulong-OpenMP (1-thread) compared to that on Sulong-sequential execution. `MG` benchmark had the second-highest overhead of 22%, which was much lower compared to the `SP` benchmark[1]. These overheads were proportional to the benchmark computations present in the *outer* function — the function that calls the top-level `outlined` functions. Such outer functions were not highly optimised because of frame materialization issue (discussed in Section 5.3). The `SP` benchmark inlined other computationally intensive functions in the outer function; as a result its performance further worsened. As a workaround, only for the SP benchmark, we forced clang not to inline those computationally intensive functions in the outer function. Clang provides the `__attribute__((noinline))` attribute to restrict inlining. This restricted inlining decreased overhead of `SP` benchmark on Sulong-OpenMP (1 thread) compared to that on Sulong-sequential from 430% (slowdown of 5.3x) to 30%. For the execution using Sulong-OpenMP (4 threads) compared to Native-OpenMP (4 threads) overhead of the `SP` benchmark reduced from about 53x to 42x.

5.5 Summary

In this chapter, we evaluated the implementation of Sulong-OpenMP using the NAS Parallel Benchmark (NPB) Suite. We used the C version of the benchmarks to compare Sulong-OpenMP with its reference implementation using clang. We compared Sulong-OpenMP with the Java versions of the NPB suite benchmarks to identify the benefits and/or bottlenecks for the JVM hosted executions. We calculated the single-threaded performance to measure the overhead incurred for adding OpenMP support by comparing Sulong-OpenMP to Sulong-sequential. As clang generates different LLVM IR for sequential and OpenMP version, we expected their performance to reflect the same. LLVM IR for the OpenMP version has additional calls to OpenMP runtime functions. To determine the value acceptable overhead, we measured the native executions with and without OpenMP support.

Comparison of the single-threaded executions showed that the Sulong-OpenMP executes the NPB suite with an acceptable overhead of 3% compared to Sulong-sequential. The implementation choice of Java versions of the benchmarks made them

[1]The overheads for single-thread performance are not shown separately in Figure 5.6 as they are aggregated into the 'After optimisations' category.

faster on kernels but slower on pseudo-applications compared to Sulong-OpenMP (1-Thread). The Java versions precomputed certain values that were not accounted for their measured execution time. However, on the pseudo-applications, the Java versions created threads, multiple times than the specified number. These threads were then assigned a specific parallel region for execution.

Comparison of the multi-threaded executions showed that the Sulong-OpenMP still has a significant performance gap to cover. One of the major sources of the overhead arises from the current thread-pool implementation, which requires the `Frame` of the top-level `outlined` OpenMP function to be materialized. The frame materialization restricts the Graal compiler to aggressively optimise the function that wraps the `outlined` function. This limitation also impacts the scaling performance of the benchmarks on Sulong-OpenMP. The `EP` benchmark on Sulong-OpenMP showed the scaling behaviour similar to the Native-OpenMP executions, where the frame materialization issue has a negligible impact. Another reason for relatively poor scaling of Sulong-OpenMP is lack of parallelism. Sulong-OpenMP uses much smaller problem sizes from class S and W because the underlying Sulong system could not support execution of the larger problem sizes. Consequently, threads starve for the work when number of threads increase. There may be additional sources of overhead that might have impacted multi-threaded executions on Sulong-OpenMP but are yet to be explored.

In the final section of the chapter, we discussed various optimisations to reduce the overhead of Sulong-OpenMP. The first technique improved calling Sulong-intrinsics using the approach similar to tail call optimisation. Then we discussed the implementation of the thread-private stack and the thread-pool for OpenMP threads that are important for the multi-threaded execution of Sulong-OpenMP. The optimisations to split OpenMP regions and restricted inlining benefited from the way clang generates LLVM IR. These optimisations aimed to increase the proportion of benchmark computations that are executed in JIT-compiled mode.

The current implementation of Sulong-OpenMP mainly focused on single-threaded performance. After applying all the optimisations, the overhead for Sulong-OpenMP implementation reduced significantly (from about 273% to an acceptable value of 3%), compared to Sulong-sequential execution. In the next chapter, we discuss the performance analysis approach that we used to analyse the slowdown for MG benchmark.

Chapter 6

Performance Analysis of Truffle Hosted Languages

In this chapter, we discuss the work that aims to improve the ability to do performance analysis of the Truffle hosted languages. The performance analysis technique, discussed in this chapter, aided the development of Sulong-OpenMP. This work is discussed separately because it also covers performance evaluation of other JVM hosted executions. Section 6.1 begins our discussion by highlighting the need for effective performance analysis techniques that are used for JVM hosted language implementations. Section 6.2 describes the approaches and their trade-offs for profiling JVM hosted executions. This section also highlights the challenges involved in using the existing approaches, for profiling JVM hosted executions of Truffle languages. To overcome these challenges, we devised a performance analysis technique by extending the existing JVM profiler: `perf-map-agent`. Section 6.3 focuses on describing our performance analysis technique. We then evaluate and compare the performance of three Truffle hosted language implementations viz. TruffleRuby (for Ruby), Sulong (for LLVM IR) and FastR (for R), with Java and native (C) execution. Section 6.4 and Section 6.5 describe the experimental methodology and results of the evaluation respectively. We use our technique to analyse the performance anomalies observed while evaluating Truffle hosted languages. Section 6.6 presents two case studies to demonstrate the usefulness of our technique. Section 6.7 concludes this chapter by outlining the current limitations of the technique. We published this work in ManLang'18 [GNL18].

6.1 Introduction

Profiling helps to identify the performance bottlenecks by measuring where an application spends time during the execution. In the case of JVM hosted executions, the time is also spent on additional tasks, such as JIT compilation and Garbage Collection (GC). Measuring the time spent on these activities is crucial for performance analysis. As discussed in Section 2.2, during execution, some parts of the application may execute in interpreted mode, and the rest executes in JIT-compiled mode. The JIT-compiled mode of execution is relatively faster than the interpreted mode. This makes it important to execute computationally intensive and frequently executing parts of the application, in the JIT-compiled mode. An ideal profiler helps to identify the mode of execution for performance-critical parts of an application.

It is a popular approach to use JVM for executing programming languages other than Java. Examples of this approach are: Truffle framework (from the GraalVM project) [Cor15] and Eclipse OMR (from the Eclipse J9 project) [Fou16]. This approach benefits from reusing the components of JVM that are evolved over many years, such as JIT compilers and garbage collectors. Additionally, this approach requires only one managed language runtime to be maintained and optimised. Consequently, this avoids the need for a separate runtime for each language. For example, the Truffle framework offers support for R (using FastR), Ruby (using TruffleRuby) and JavaScript (using GraalJS).

Performance analysis of guest languages, hosted on a JVM, poses a new set of challenges for conventional Java profilers. The Java profilers do not expect any language other than Java to be executed on a JVM. Further, it becomes more challenging for the execution of *polyglot* applications (i.e., the applications are written in two or more languages). GraalVM offers support to execute programs written using multiple Truffle hosted languages [GSS+18]. Conventional Java profilers cannot recognise the functions of a guest language program, during its execution. Instead, the profilers record executions of the guest language program, as that of a Java program. Consequently, these profiles show the Java methods, that implement guest language functions instead of functions themselves. To analyse the performance of guest language execution, it is necessary to map Java profiles to the corresponding guest language functions.

Such mapping of profiling information can help to identify, isolate and measure the performance of specific guest language features. This can further help to guide the optimisation efforts, where language implementation can be benefited more.

In this chapter, we discuss our sampling-profiler-based approach for Truffle hosted language executions on JVM. We extend the existing Java profiler: `perf-map-agent`. The profiling approach is also aware of the polyglot executions and can highlight different guest profiles uniquely. In the next section, we discuss the existing profiling approaches, their trade-offs, and challenges to profile Truffle hosted languages.

6.2 Profiling Approaches & Challenges

This section begins with a discussion of two common profiling approaches viz. Tracing and Sampling. We then highlight the challenges for the existing sampling-based profilers, to accurately profile JVM hosted executions. We then discuss the profilers, that mitigate these challenges and improve the accuracy of recorded profiles.

Tracing vs Sampling

In the case of Tracing approach, typically, an application is instrumented by adding instructions to record the events of interest (e.g., entering and returning from a function). During the execution, these instrumenting functions record the events that are then processed to calculate the information of interest (e.g., time spent in the specific function). As the tracing profilers inject the profiling code in an application, they can capture every event of interest, whenever it occurs. This makes the tracing approach to record a profile accurately. However, the injected code is executed each time the event occurs. Consequently, this induces an overhead, proportional to the time spent in executing the profiling code.

On the other hand, the sampling profilers inspect a specific metric at a certain frequency, such as call-stack of an application during its execution. These *samples* of information are then aggregated to calculate the information of interest (e.g., time spent in the specific function). Typically, sampling is performed using an external tool (e.g., the Linux utility *perf*), that records the call-stack samples. Therefore, it is easier to tune the overhead of sampling profilers by changing their sampling frequency. This

flexibility makes the sampling-based profiling approach suitable for use in production environments. Thus, we chose the sampling-based profiling approach for this work.

Challenges for sampling profilers

The two main challenges for sampling profilers are as follows: i) sampling rate, that influences the accuracy of the profile; ii) sampling skid, that may impact the precision of profile.

1. **Accuracy**: The accuracy of the profile depends on the selected sampling frequency. Sampling profilers pause the application to record a sample. Thus, when the sampling frequency is higher, the profiler incurs higher overhead during the execution. Higher overheads may impact the behaviour of an application. In [AH17], Akiyama et at. discussed the issues arising from higher sampling frequencies, using Intel Precise Events Based Sampling (PEBS). However, when the sampling frequency is too low, the recorded samples may miss certain events. For example, call to a function would not be reflected in the recorded samples, when it is invoked and returned in the interval between two samples. When the profiler misses such events, the application methods report disproportionate time spent in executing them.

2. **Precision**: Sampling skid may impact the precision of a recorded profile. The skid occurs when the sampling event is recorded on the line of code, where it does not exactly occur. This may cause reporting of an incorrect method in the call-stack sample. The skid may occur as a result of the architecture-level processor implementation mechanisms, that are used for sampling; for example, delay in propagating the sampling event to the processor.

   It is complicated to determine the exact line of code when a sampling interrupt occurs. A line of code is converted to multiple instructions, that may not necessarily execute in the same order, because of the out-of-order execution [Wik13b]. This further complicates with instruction-level parallelism, which enables multiple instructions in flight at a given point of time [Wik13a].

6.2.1 JVM Profilers

One of the popular approaches used by the sampling Java profilers is to use JVM Tools Interface (JVMTI). The JVMTI provides a native interface (in C or C++), which allows tools to inspect and control the state of the application that is running on the JVM [Ora04]. Tools can achieve this using the software agents, referred to as JVMTI agents. A JVMTI agent can request the current execution call-stack sample from the JVM. A call-stack sample can be retrieved using the `GetCallStackTrace` method. An individual call-stack sample does not provide much information; however, when call-stack samples are collected over a period of time, they can represent the execution behaviour of an application. A proportion of the call-stack samples, matching a particular part of a program, indicate the corresponding proportion of time in that part of the program e.g., if 20 out of 100 samples match `foo()`, indicates that approximately 20% of the execution time is spent in `foo()`.

Safepoint Bias

The profilers that rely on the JVMTI agents to collect call-stack samples, face an issue of *Safepoint Bias* that may lead them to record an inaccurate profile. Mytkowicz et al., studied four popular Java profilers: *hprof*, *xprof*, *YourKit*, and *jprofile*; and found that the profilers often produce incorrect profiles [MDHS10]. When used on the same benchmarks, these profilers disagree on time spent in the hot methods and/or may identify incorrect methods as hot. One of the primary reasons for these inaccuracies is the Safepoint Bias.

An application is said to be at SafePoint when the state of the JVM is well described. Safepoints are used by the JVM to perform maintenance tasks such as GC and servicing the JVMTI requests. Figure 6.1 shows how the Java application threads reach the safepoint. JVM sets a flag to request a safepoint when it needs to perform a task that requires an application at a safepoint. During the execution, each application thread periodically checks whether a safepoint is requested, referred to as *safepoint poll*. Whenever a thread notices a safepoint request, it suspends itself. As seen in Figure 6.1, threads T1, T2 and T3 wait at the safepoint when they are executing the safepoint poll. The thread T4 and T5 are already at the safepoint, as they are executing the native calls. The native threads wait on their return until the

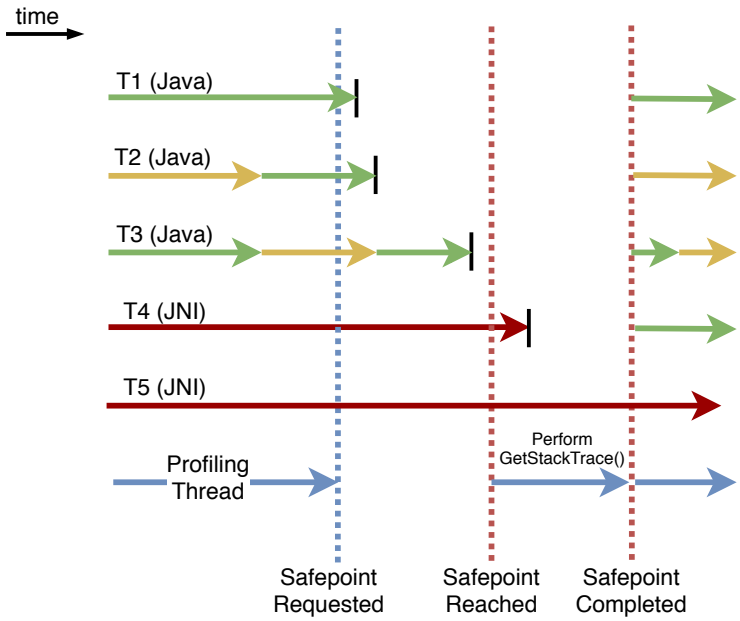


Figure 6.1: The figure shows how the Java threads reach the safepoint. The execution states of the Java threads are coloured as: executing(green), descheduled (yellow), and native (red). Image reproduced from [Wak15].

requested operation at the safepoint is completed; in this case, collecting the call-stack sample using the `GetStackTrace` JVMTI function. It is important to note that the application threads are not immediately stopped when the safepoint flag is raised. The number of instructions that are executed until the thread is at the safepoint, is unknown. Therefore, the call-stack samples collected using the `GetStackTrace()` function, are skewed towards the safepoints when they are collected. This behaviour is called the *safepoint bias*.

Putting a safepoint poll is JVM implementation-dependent. The HotSpot JVM polls for a safepoint in both interpreted and JIT-compiled mode of execution. The interpreted mode polls after executing every bytecode. The JIT-compiled code polls before taking the back-edge in a non-counted loop, and returning from a function [Wak15]. When a method is inlined then it would not have a safepoint at the entry/return of the inlined method. This may incorrectly reflect the time spent in the inlined method as the spent in the callee of that method. Furthermore, if the method is called at multiple call-sites, then time spent in that method may get distributed amongst call-sites. This causes the method to be fully or partially hidden in the profile.

Linux Perf

The safepoint issue can be avoided by using the external sampling profiler such as the Linux utility *perf*. Perf is a tool, based on `perf_events` API, which can be used to read the performance counters in the processor by abstracting the hardware-specific details [per15]. Perf helps to find the performance bottlenecks of an application by measuring the metrics, such as cache-misses and branch-mispredictions. Perf can be used to record the call-stack samples at the granularity of a thread, process or a CPU. To record the call-stack samples, perf uses the frame pointer register to walk the application call-stack. Therefore, perf needs the application to populate the frame pointer register correctly. By default, many compilers perform optimisation to reuse the frame pointer register as a general-purpose register. This can be avoided using a compiler flag, e.g. `--fno-omit-frame-pointer` for *gcc*. This issue of reusing frame pointer register is present for the HotSpot JVM; consequently, making perf to record the incomplete Java call-stacks. The JDK versions 8u60 onwards provide the `-XX:+PreserveFramePointer` flag that enforces correct value to be present in the frame pointer register.

AsyncGetCallTrace

The internal API of OpenJDK has a method AsyncGetCallTrace (AGCT). This method can be used to sample the call-stack of an application running on the OpenJDK JVM. The AGCT call does not require JVM to be at a safepoint to get serviced. Thus, the collected call-stack samples are not impacted by the safepoint bias. However, some of the call-stack samples may be broken because the JVM may not be in a well-defined state when those samples are collected [Pro19]. Some of the reasons why JVM may not be in a well-defined state are: i) the interrupted thread for sampling is in the middle of deoptimisation, ii) the thread is created but not yet started executing any Java code. Therefore, the correctness of the collected profile is proportional to the correctly sampled call-stacks.

The profilers, such as the honest profiler, use the AGCT-based approach for profiling [War19]. The *Honest Profiler* attaches a JVMTI agent that registers a signal handler for the `SIGPROF` signal. This handler delivers the signal periodically at a specific sampling frequency. The signal handler records the call-stack sample using the

AGCT call. When the signal is delivered, a random thread of the application is selected to execute the signal handler routine. The selected thread is not necessarily an application thread or it may not be at the safepoint. Therefore, the call-stack sample may not be retrieved. To deal with this issue, the Honest profiler also reports the failed call-stack samples that can be used to determine the accuracy of the profile. The higher percentage of failed samples indicate that the JVM spent a large amount of time in performing activities, such as GC and deoptimisation.

The `Async profiler` extends the approach used by Honest Profiler. The Async-profiler uses ACGT calls to capture JVM call-stacks, and it uses perf to record the corresponding native call-stacks. Honest Profiler samples only the JVM call-stacks, but not the native ones. When the bottleneck lies in the underlying native operations, such as networking or file I/O, the Honest Profiler associates time spent in the corresponding Java method, which performs the slow native operation. The Async profiler addresses this issue by collecting the native call-stacks using the `perf_events` API. This API also enables Async profiler to record performance counters, including cache misses, branch mispredictions and page faults [Pan19]. The Async profiler has options for better usability, such as the built-in support to generate flamegraphs of the recorded profile.

6.2.2 Challenges for Truffle hosted executions

In the case of executing Truffle hosted language implementation on the JVM, profilers are unaware of the guest language. Hence they profile the execution as any other Java application. This makes it difficult to map the profiling information in the context of executing the guest language program.

Figure 6.2 shows profile recorded for the warmed-up execution of the *NBody* benchmark from the CLBG suite of benchmarks using Sulong-sequential. This profile is recorded using the perf-based sampling profiler and visualised using the flamegraph. Flamegraph shows the native call-stacks in red, Java in green and the inlined Java methods in teal colour. The profile shows more than 99% of execution time is the Java method: `OptimizedCallTarget.callRoot()`. Truffle uses an instance of the `OptimizedCallTarget` class to create a *root* node of AST, which represents the guest language function. To execute this guest language function, the `callRoot()`

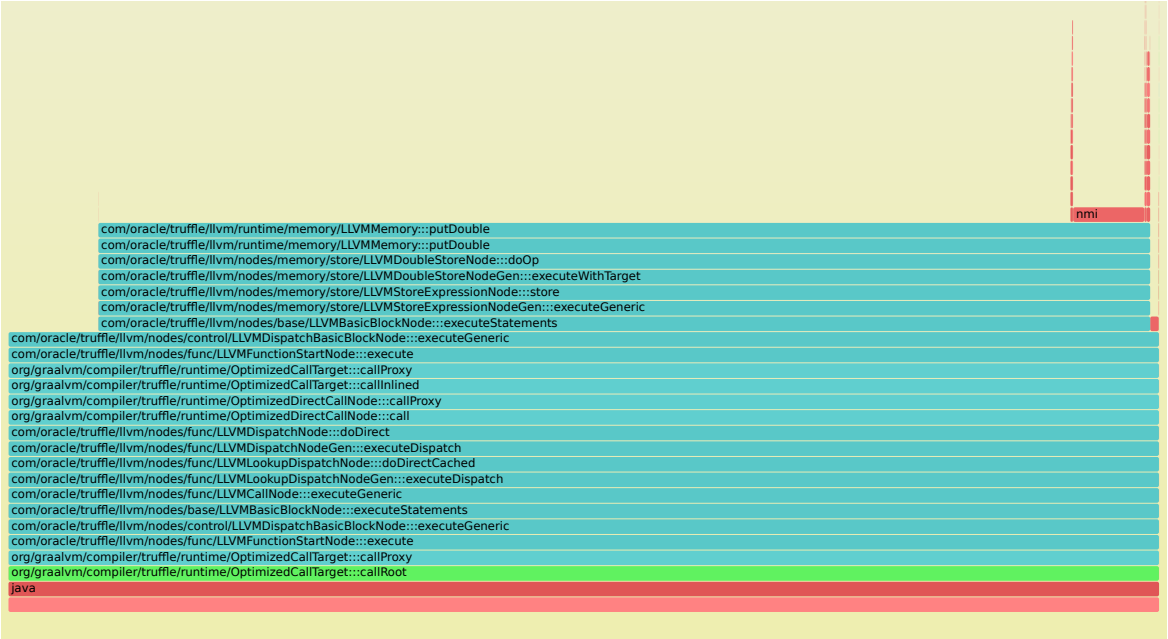


Figure 6.2: The flamegraph visualises execution profile of the `NBody` benchmark from the Shootout benchmark suite on Sulong-sequential. Warmed-up execution of the benchmark is profiled to generate this flamegraph.

method of its corresponding AST is invoked. However, we cannot determine the exact guest language function by only inspecting the flamegraph. This scenario is illustrated in Figure 6.2. In this flamegraph, we cannot determine the name of LLVM IR function that corresponds to the widest Java stackframe (i.e., the green rectangle with name `OptimizedCallTarget.callRoot()`). The profile becomes confusing when there are multiple guest language functions present. All these functions are represented as separate stackframes in the flamegraph, but with the same method name: `OptimizedCallTarget.callRoot()`. This happens because the root nodes for all guest language functions are instances of the `OptimizedCallTarget` class, and executed using their `callRoot()` method. The profile becomes further confusing when the guest language functions are inlined by Graal. These inlined functions are represented using the `OptimizedCallTarget.callInlined()` name. It is difficult to determine which guest language functions are inlined, by simply inspecting the flamegraph.

Polyglot programs are written using multiple Truffle hosted languages. The difficulty of determining guest language names in the profiles further exacerbates for the polyglot executions. All the guest language functions are represented using the `callRoot()` method name. As all implementations are Java-based, it is difficult to

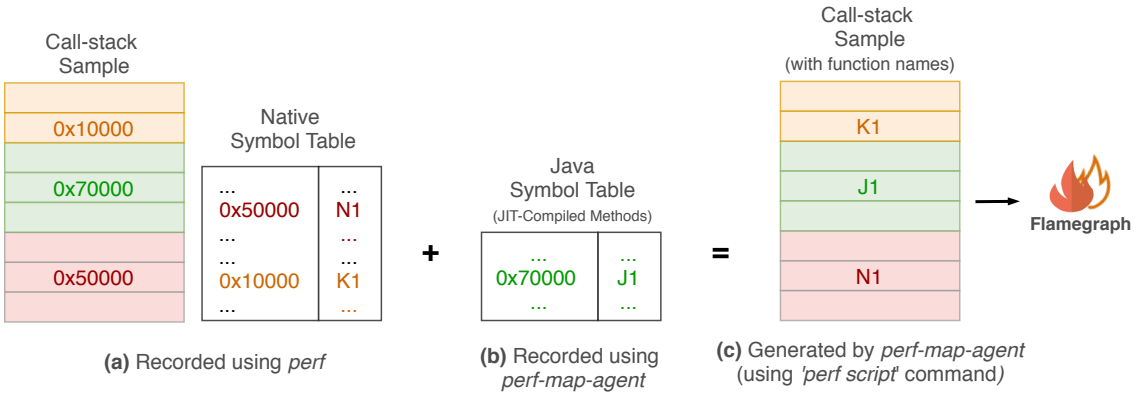


Figure 6.3: High-level working of perf-map-agent. (a)`perf` records call-stack samples with stackframe addresses, and the native symbol table to map those addresses to corresponding function names. (b) The missing mapping for the JIT-compiled Java method is dumped by perf-map-agent. (c) Combines the information from (a) and (b) to obtain call-stack samples with names of the JIT-compiled Java methods.

distinguish between the call-stacks associated with different guest languages.

6.3 Performance Analysis Technique

In this section, we discuss our approach, based on the *perf-map-agent* profiler, to address the challenges for profiling the Truffle hosted language executions on JVM. This approach uses the existing perf-map-agent profiler, which uses `perf` to record the call-stack samples. We extend perf-map-agent to map the guest language function names to their corresponding Java methods in the recorded profile. Firstly, we will discuss the working of perf-map-agent. We then proceed our discussion on extending perf-map-agent to recognise the guest language functions in recorded profiles.

6.3.1 Perf-map-agent profiler

Figure 6.3 depicts high-level working of `perf-map-agent` profiler. `perf` records the call-stack samples that contain stackframe addresses and native symbol table, as shown in Figure 6.3(a). This information is enough to visualise the profile of native executions, which contains human-readable function names instead of hexadecimal stack

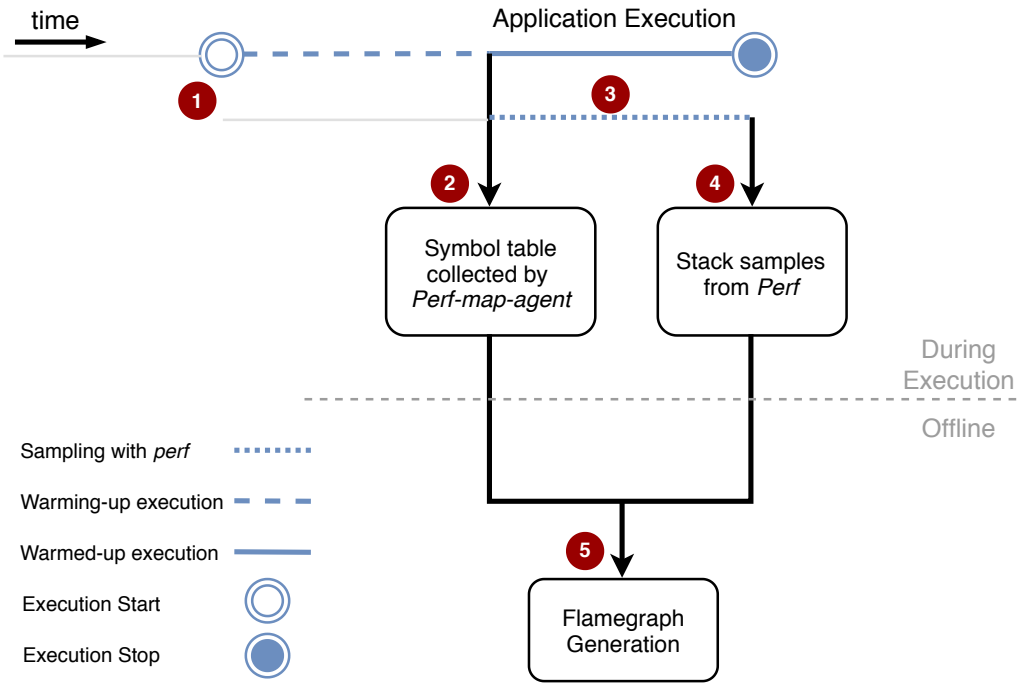


Figure 6.4: The workflow for profiling Java executing using perf-map-agent. The recorded profile is visualised using the flamegraph. Numbers in red circles represent the sequence of profiling steps.

addresses. However, `perf` cannot map the method names for the interpreted and JIT-compiled methods in JVM. Thus, perf-map-agent provides the missing piece of information for mapping JIT-compiled methods in JVM. The perf-map-agent is a JVMTI agent that can be attached to JVM, which is executing the program of interest. When attached, the JVMTI agent dumps the symbol table for JIT-compiled methods in that JVM, as shown in Figure 6.3(b). This symbol table consists of method names and their stackframe addresses. In the call-stack samples recorded using `perf`, stackframe addresses for the JIT-compiled code can be mapped to their respective method names, using the dumped Java symbol table. This is illustrated in Figure 6.3 where the missing JIT-compiled Java method name *J1* (in native symbol table) is mapped to its stackframe address. Thus, the recorded profile can use the Java method names correctly in the flamegraph instead of their stackframe addresses.

Figure 6.4 outlines the workflow of profiling a Java execution using the perf-map-agent. As the figure depicts, the perf-map-agent is attached to the JVM when the application is warming-up (represented by the dashed line). The agent retrieves mappings of *method name* $\rightarrow$ *stackframe address* for all the JIT-compiled methods (illustrated in

Figure 6.3(b)), that are *entrant* [1] at the time of attachment. The call-stack sampling is initiated immediately using `perf`, which then records the warmed-up phase of the application. The collected profile is processed offline to map stackframe addresses to their respective Java method names (illustrated in Figure 6.3(c)). Such a profile can then be used to generate a flamegraph.

Note that, the perf-map-agent dumps the symbol table only once — at the time when the agent is attached to the target JVM process. Thus the JIT compilations that occurred later on during the sampling phase, cannot be mapped to their Java method names. If continuous JIT-compilations are expected, then the JVMTI agent can be extended to dump the symbol table at a certain time interval. Periodic dumping of the symbol table ensures that mappings for all the JIT-compiled methods are available. Unlike the `AsyncGetCallTrace`-based approach, the perf-map-agent dumps the names of only JIT-compiled methods, but not that of the interpreted methods. Thus, the interpreted method names are invisible in the profile, and the corresponding call-stack frames are named as `Interpreter`. However, when the program spends a significant portion of time during interpretation, visualising the interpreter call-stacks becomes crucial. We do not target such applications with our profiling approach. In such cases, even the profilers with safepoint bias may provide a better approximation of time spent during the interpretation, because the safepoint poll is performed after executing every bytecode. Additionally, time spent during the interpretation phase is significant for short-running applications, because methods are not executed enough number of times to become hot. However, for short-running applications, the performance engineering approach may not remain the same. One may want to consider the ahead-of-time compilation (AOT) based approach that avoids the interpretation overhead by design.

6.3.2 Extention to the perf-map-agent profiler

As discussed previously, the perf-map-agent profiler cannot map Java methods to corresponding guest language functions, making them invisible in the profile, e.g. the `@nbody` function in Figure 6.2. All JIT-compiled guest language functions are shown

[1] A JIT-compiled method is `entrant` when it can be invoked during the execution. When an assumption associated with the JIT-compiled method fails, it is deoptimised. Before deoptimisation begins, the method is marked as `not entrant` to indicate that it is not valid anymore and should not be invoked.

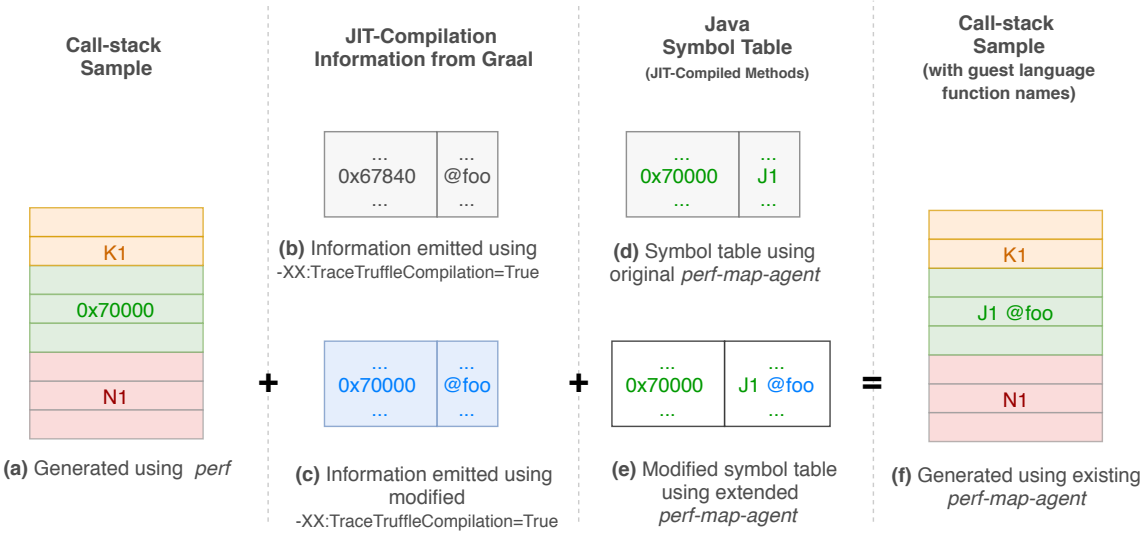


Figure 6.5: High-level working of the extension to perf-mag-agent profiler. (a) call-stacks recorded using `perf` cannot map names for the JIT-compiled Java methods. (b) existing `-XX:TraceTruffleCompilation` flag of Graal dumps *code address* → *guest language function name* when its AST is JIT-compiled. (c) we modified the working of -XX:TraceTruffleCompilation flag to dump the *method entry points* that are recorded by `perf`. (d) original perf-map-agent dumps Java symbol table with *method entry point* → *Java method name*. (e) We update the orignal table in (d) to contain new mapping *method entry point* → *Java method name + guest function name*. (f) Now the existing workflow of of perf-map-agent can show guest language function names in the recorded call-stacks, which was missing previously.

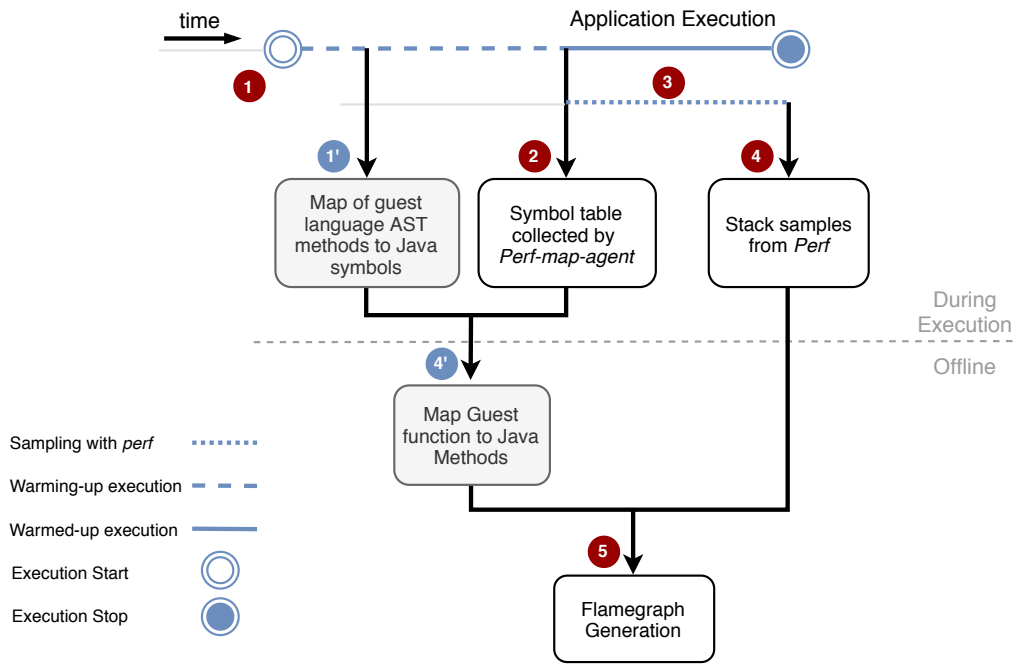


Figure 6.6: The extended workflow for profiling the JVM hosted executions using perf-map-agent. Numbers in red/blue circles represent the sequence of profiling steps (Step x' comes after x). Blue circles represent additional steps to the perf-map-agent based approach shown in Figure 6.4.

as `OptimizedCallTarget.callRoot()` in the flamegraph. The mappings dumped by existing perf-map-agent enables us to visualise `callRoot()` name (equivalent to *J1* in Figure 6.3) instead of its hexadecimal stackframe address. However, we cannot see the name of guest language function (`@nbody`). This makes it difficult to map the profiling information of the JVM hosted execution to the corresponding guest program for performance analysis. We address this issue by extending the perf-map-agent profiler that maps the guest language method names to their matching call-stack samples.

Figure 6.6 outlines the modifications done to the profiling approach, based on the perf-map-agent. During the warming-up phase, when a guest language method is JIT-compiled, we record its name and the stackframe address (illustrated in Figure 6.5(c)). The symbol table dumped by the original perf-map-agent contains mapping of stack-frame addresses → Java method names (illustrated in Figure 6.5(d)). However, this mapping shows `callRoot()` method name for all guest language functions instead of their actual names, as seen in Figure 6.2. We update the dumped symbol table with appropriate guest language function names (illustrated in Figure 6.5(e)). We then use the modified table to create call-stacks with guest language function name (illustrated

Figure 6.7: The flamegraph visualises execution profile of the `NBody` benchmark from the Shootout benchmark suite on Sulong. Flamegraph also shows the guest language function name '`@nbody`' (in the green rectangle, appended after `callRoot`).

in Figure 6.5(f)). The profile visualisation techniques, such as flamegraphs, can use these call-stacks. Such a flamegraph then shows the guest language function name along with the Java method name, as `@nbody` in Figure 6.7. Now, multiple guest language functions in the profile can easily be distinguished and are mapped correctly.

High-level working of extended perf-map-agent approach is shown in Figure 6.5. We modified Graal's implementation for the `-XX:TraceTruffleCompilation` flag, to retrieve the correct mapping of the guest language function name to its stack-frame address. When the `TraceTruffleCompilation` flag is set to `true`, Graal dumps the compilation information about the guest language functions. This compilation information consists of a function name and code address (illustrated in Figure 6.5(b)). On the other hand, the dumped symbol table contains Java method entry points (for the guest language functions) and their Java method names (illustrated in Figure 6.5(d)). Our objective is to match the guest language function name to their corresponding Java method names (illustrated in Figure 6.5(e)). The original implementation of `-XX:TraceTruffleCompilation` flag dumps code address, while `perf` records stack address/ method entry point for the method. Although, entry points and code addresses are for the same JIT-compiled method, we cannot map them deterministically because entry points are not at the fixed offset from their code addresses. Linear search that begins from the code address (e.g. `0x67840` → `0x70000` in Figure 6.5) may cause incorrect matching in certain scenarios. Incorrect matches result in showing wrong

guest language function names in the profile, which is worse than failing to show any function name. Therefore, we use a different approach to extract accurate mapping of method entry points → guest language function name.

Our approach extracts the necessary mapping from Graal when JIT-compilation of the guest language function occurs. The method entry point for the address is recorded by Graal when the guest language function is compiled. However, the entry point does not get dumped when the `TraceTruffleCompilation` flag is set to `true`. We fix this issue by modifying Graal, to dump the entry points when the tracing is enabled. This newly dumped information now contains the mapping of guest function names to their method entry points (illustrated in Figure 6.5(c)). As discussed previously, perf-map-agent dumps the symbol table that contains the mapping of method entry points to Java method names (illustrated in Figure 6.5 (d)). We use both these mappings to accurately match the guest functions to their stackframe addresses (illustrated in Figure 6.5(e)). This enables us to generate the flamegraph shown in Figure 6.7.

6.3.3 Profiling polyglot applications

This section describes the enhancements done to visualise call-stack samples recorded for the polyglot executions, using the flamegraphs. This work complements the visualisation of profile for a Truffle-hosted guest language.

Figure 6.8 shows the profile for executing a synthetic polyglot application, visualised using a flamegraph. The application calculates Fibonacci sequences, and it performs array manipulations using three languages: R, JavaScript and Python. We use the Truffle-hosted implementations of the languages FastR (R), GraalPython (Python) and TruffleJS (JavaScript), which are bundled with `graalvm-0.31`. The profile is recorded using an async-profiler for better insights into the interpreted methods. The flamegraph in Figure 6.8 uses green colour for all the call-stacks because they all are Java stack-frames. However, this makes it difficult to distinguish call-stacks corresponding to different guest languages and map the profiling information to a specific language.

We modified the `flamegraphs.pl` script that is used to plot flamegraphs. This script uses different colours to identify call-stacks for Truffle-hosted languages uniquely, as shown in Figure 6.9. Modifications are limited only to the script generating flamegraphs;

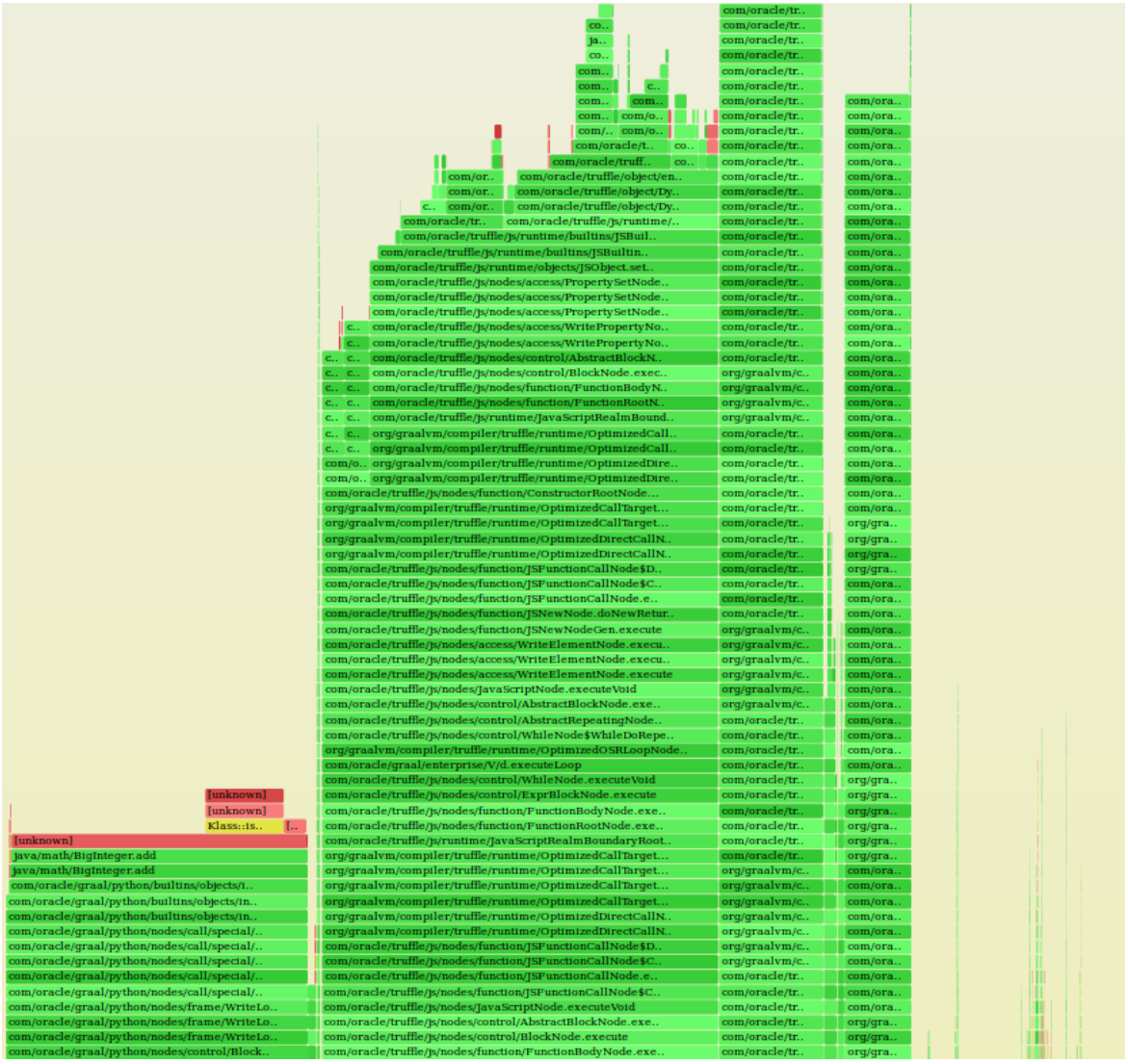


Figure 6.8: The profile for a synthetic polyglot benchmark recorded using the `async-profiler`.

Figure 6.9: The modified version of the profile shown in Figure 6.8 where the stack-frames for different languages are coloured using different colours as shown in Table 6.1.

Language	Color
GraalPython	Brown
TruffleJS	Blue
FastR	Cyan
Truffle Framework	Light Green
Graal	Lime Green

Table 6.1: Colour scheme used in the flamegraph in Figure 6.9.

making it useful to visualise call-stack samples that can be recorded by any other profiler. We demonstrate the use of modified flamegraph script with `async-profiler` in Figure 6.9, which now identifies the interpreted and JIT-compiled methods. Languages are identified using their API names, present in the recorded call-stacks. Flamegraph in Figure 6.9 uses colour scheme shown in Table 6.1 to highlight call-stacks. Integration of modifications from extended 'perf-map-agent' into `async-profiler`, would enable the later to identify the guest language functions names.

for GraalPython (in brown), TruffleJS (in blue), FastR (in cyan), Truffle framework API (in light green) and Graal (in lime green).

6.4 Experimental Methodology

6.4.1 Evaluation Objective

In this work, we aim to provide tools that can help to identify the performance bottlenecks easily for the Truffle-hosted language executions on a JVM. In this section, we evaluate the implementation of the Truffle-hosted languages on a selected suite of benchmarks. We then carry out performance analysis using the perf-map-agent based technique, discussed in the previous section.

The objective of this evaluation is to identify the source of overhead; whether it is arising from i) the application, ii) from JVM services such as JIT-compilation and GC, iii) from the language implementation using the Truffle framework. To achieve this objective:

- We execute the same set of benchmarks implemented, using different Truffle

hosted languages (TruffleRuby, Sulong, FastR) to compare the language implementations.

- We use the Java implementation of the benchmarks as our baseline, for separating overheads induced by JVM hosted execution.

- We use the native C execution to gauge lower bound on the performance gap, which could be achieved by a guest language implementation that is hosted on a JVM. Profiling these executions enable us to compare the implementation of specific guest language features to that of their native equivalent.

It is important to note that, comparing different language implementations can be tricky because of differences in their semantics and standard libraries [MDM16]. At times, we try to keep benchmark implementations as similar as possible, by preferring a naïve approach to ensure consistency across their implementations. More importantly, our objective is not to determine whether one language or its feature is better than the other; instead, it is to identify the performance improvement opportunities for language implementations.

6.4.2 Benchmarking Environment

Benchmarks

We use the Shootout benchmarks from the *Computer Language Benchmark Game* (CLBG) that are commonly used for comparing the performance of different language implementations[Guo18]. We use the benchmark implementations in Java, Ruby and C (for native and Sulong) languages. We deliberately use the sequential version of benchmarks to ensure consistent implementation across the languages. We explicitly avoid the use of parallel library functions at the benchmark implementation level, such as the parallel variants in the streaming API of the Java Development Kit (JDK). However, the JVM hosted executions use multiple threads created for the VM services, such as JIT compilation and GC. Further, the guest language implementations may use parallelism offered by the JDK while implementing their standard library functions. However, we consider this as an implementation choice, which itself may be suboptimal and needs to be identified as part of the performance analysis.

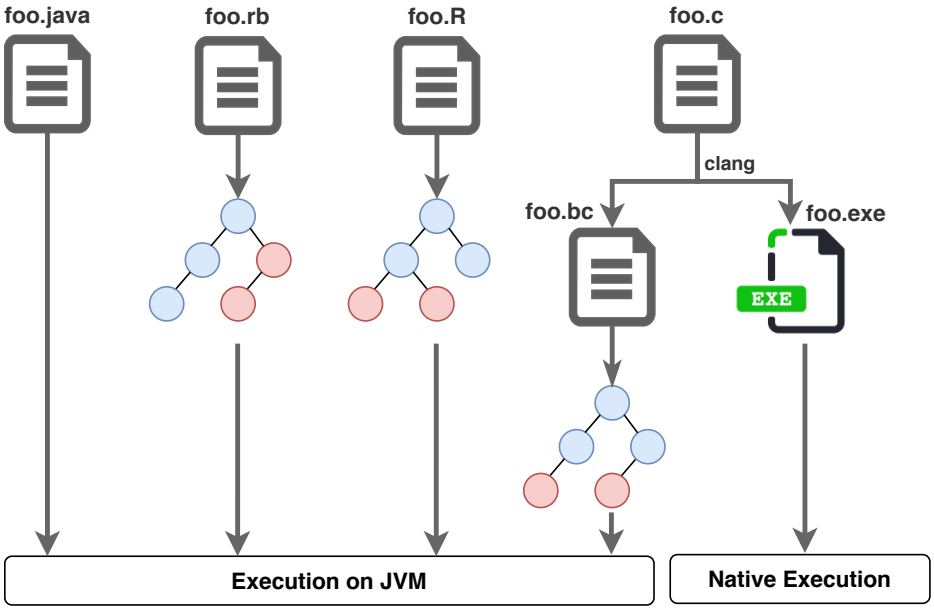


Figure 6.10: The modes of execution for the different language implementations.

We are not using `meteor` benchmark, because it failed to execute correctly on the version of Sulong that we use in the evaluation. Also, note that we do not present results for the `regexdna` benchmark using FastR, as we discovered a bug with the implementation (see [Gai15]). This bug arose from the implementation of `gsub()` library function in FastR, which differed from that of in GNU R, causing FastR to calculate incorrect results.

Experimental Setup

The system that we use here, is different from the one used in the evaluation of Sulong-OpenMP. We use a system with the following features: i) 2 physical (4 hyper-threaded) cores of Intel Core i7-3537U, ii) 8GB memory, iii) running Ubuntu 16.04 (4.13.0-38-generic). We disable the processor frequency scaling and set it to a fixed 1.0GHz using the *userspace* governor. We did not set the frequency to maximum because, in the case of overheating, we observed that the cooling system in the machine was not strong enough to avoid forced scaling-down of frequency.

Figure 6.10 shows setup for executing each of the selected implementations. Table 6.2 shows softwares and their corresponding versions that we use in our experiments. We use the latest version of Sulong available then that supports LLVM 5.0.

Software	Version
Sulong	commit b779587
TruffleRuby	commit c15ae86
FastR	commit c49d3b7
GraalVM	vm-enterprise-0.33
Clang	5.0

Table 6.2: Software versions used in for the experiments.

Note that, this version of Sulong is the older version than the one used by Sulong-OpenMP. The LLVM IR for the Sulong based execution is generated using *clang* with `-O3` optimisation flag.

Used Methodology

The experimental methodology used to execute benchmarks on selected language implementations is similar to the one used for evaluating native, Sulong and Sulong-OpenMP based executions in Chapter 4. We measure the *peak performance* of the JVM hosted execution, where all the computationally intensive methods are JIT-compiled and performance improvement plateaus.

We use a separate benchmark harness for each language implementation that typically wraps the `main` method of a benchmark in a loop. This harness executes 100 iterations of a benchmark and uses the geometric mean of the last 50 iterations as a representative of its execution time. JIT-compilation threshold of Graal is set to 1000 invocations of a method. We manually ensure that every benchmark is JIT-compiled and no JIT compilation activity occurs during the last 50 invocations. We also calculate the geometric mean across 10 benchmarks to represent the overall behaviour of the implementation. We analyse the wide variations in performance, using the profiling approach based on perf-map-agent (discussed in Section 6.3). The call-stack sampling frequency for `perf` is set to 1000Hz.

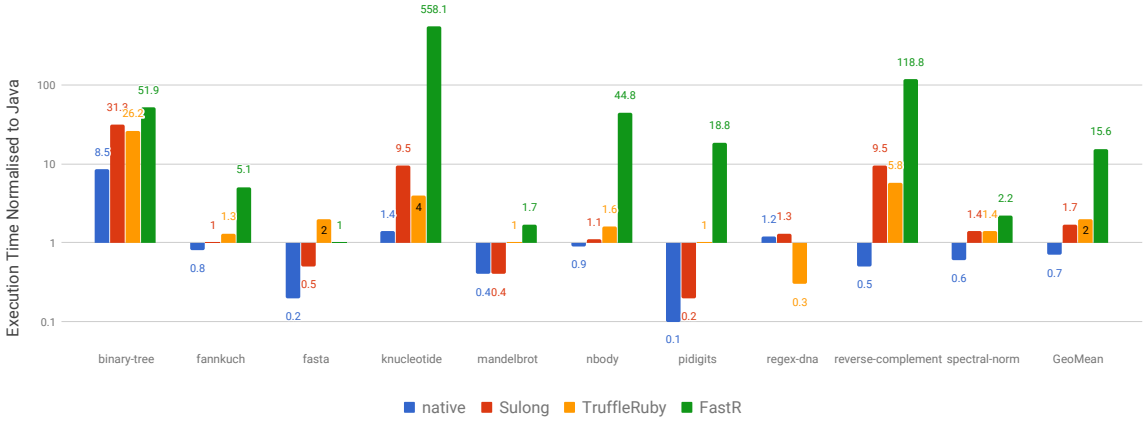


Figure 6.11: Logarithmic execution time of CLBG Shootout suite benchmarks normalised to Java execution time. 1 represents the execution time required for Java implementations (lower is better).

6.5 Results

Figure 6.11 shows the execution time of the benchmarks from the CLBG Shootout suite. The execution time shown on a Y-axis using a logarithmic scale is normalised to that of the Java version of the benchmark. It is interesting to note that, although different Truffle hosted language implementations use the same underlying framework, they demonstrate significant variations in their performance. Such performance variations raise questions i) are the language semantics making it difficult to convey information to a compiler? or ii) is it the guest language implementation that has scope for improvement?

Sampling overhead

The profiling activity involves performing additional operations to record information of interest, which incurs an inevitable overhead during the execution of an application. In the case of `perf` based sampling, the execution of an application is paused to record the current call-stack sample, which delays the execution. The extended period is an overhead for the execution, and hence proportional to the sampling frequency used by a profiler.

Figure 6.12 shows overhead incurred for the execution of Shootout benchmarks from the CLBG suite on Sulong, at different sampling frequencies. The overhead for

Figure 6.12: Sampling overhead for executing benchmarks from the Shootout benchmark suite on Sulong using different sampling frequencies. Blue bars represent the overhead for sampling using perf, green bar shows overhead when benchmarks are executed with the `-XX:+PreserveFramePointer` JVM flag only. Red bars show the overhead of using both, the perf utility and the `-XX:+PreserveFramePointer` JVM flag. Yellow bars represent the standard deviation in percentage of the geomean execution time measured without profiling enabled, and it is calculated using the 51..100 invocations of the benchmark in our test harness. Figure shows that the profiling overhead is low enough to be comparable with the variations in execution time (shown as Yellow bars) for multiple iterations of a benchmark .

using `perf` as a sampling profiler is less than 1%, compared to the original warmed-up execution time of the benchmarks. Perf-map-agent needs to start the JVM with `-XX:+PreserveFramePointer` flag, in addition to using `perf` for sampling. Overhead for using the perf-map-agent based approach is less than 1.25%, compared to the warmed-up executions.

Analysing BinaryTrees benchmark

In Figure 6.11, results for binarytree benchmark show that the execution on Java is much faster than even the native C execution. As mentioned earlier, to keep the benchmark implementation across the languages similar; we use the simple implementation, instead of the optimised one, for the binarytree benchmark. This simple implementation of the benchmark performs memory allocation/freeing operations for the tree-nodes, as a part of benchmark computation. These operations accounted for execution time of the benchmark. The native, and consequently the Sulong-based execution, performs dynamic memory management using `malloc` and `free` functions, on

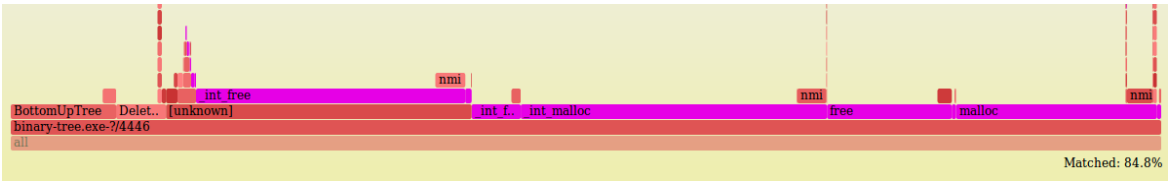


Figure 6.13: The flamegraph shows a profile for executing the binarytree program natively. The call-stacks for the functions that allocate and free memory are highlighted in magenta colour. The highlighted call-stacks show that the native execution spends 84.4% time in performing memory allocation/freeing.

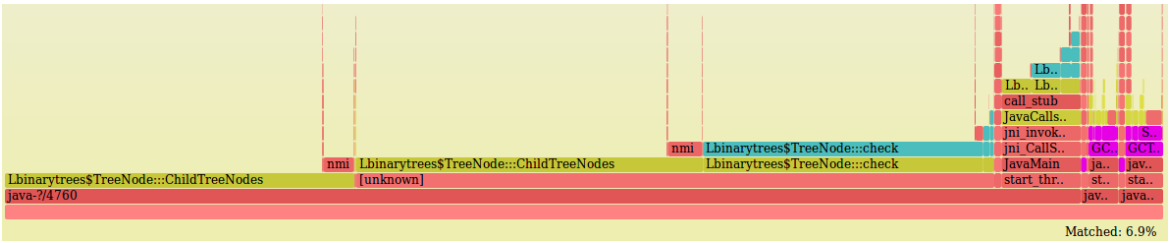


Figure 6.14: The flamegraph shows a profile for executing a Java program for the binarytree benchmark. In the flamegraph, call-stacks for GC-related functions are highlighted in magenta colour. Here, 6.9% of execute time is spent in GC. GC is performed using 2 threads that are represented by two separate towers (named by their thread IDs that are not visible).

the same thread that executes the benchmark. The profile for the native (Figure 6.13) shows that it spends about 84.4% of execution time in functions that allocate or free memory. On the other hand, Java-based execution spends only 6.9% of its total execution time in memory management, as shown in (Figure 6.14). Furthermore, JVM uses two separate threads for GC that execute in parallel without impacting the wall-clock execution time, as much as its native counterpart. Figure 6.14 aggregates the call-stack samples for different JVM threads, which do not highlight the background execution of GC threads. If we compare the raw execution times of native and Java, then we observe that the native implementation is 7.2 times slower than Java; but when we subtract their memory management overheads, the remaining execution times are comparable (i.e., time spent purely in the benchmark computation is nearly equal).

Analysing knucleotide & regexdna benchmarks

For both, knucleotide and regexdna benchmarks, the native and Sulong-based executions are slower than Java. The primary reason for this behaviour is the use of external

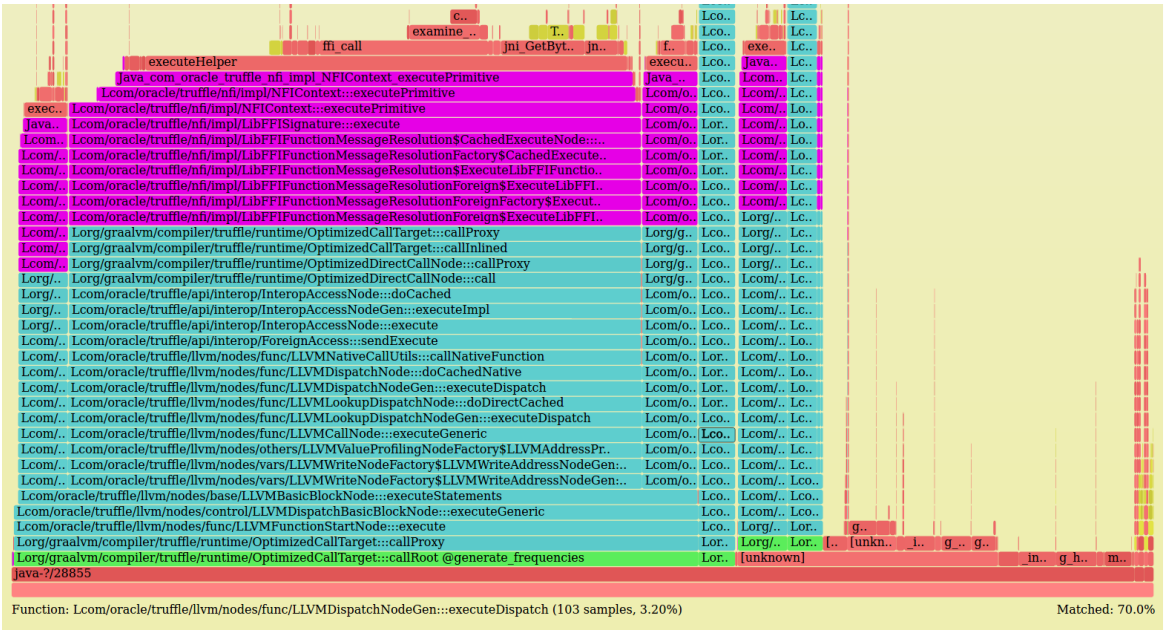


Figure 6.15: Flamegraph shows profile of executing knucleotide benchmark on Sulong. Flamegraph highlights the call-stack frames associated with calls made using Graal NFI in magenta colour and shows about 70% of the call-stack samples matching the keyword NFI.

library functions. The native implementation of knucleotide uses the external library `glib-2.0` for creating a hashmap; while the regexdna benchmark uses the Perl Compatible Regular Expressions (PCRE) library for matching regular expressions. On the other hand, Java uses built-in implementation for creating hashmaps and for matching regular expressions, using the classes from `java.util` and `java.util.regex` packages of JDK respectively. Further, the JIT-compiler can aggressively optimise the parts of these libraries for their usage in the corresponding benchmarks. The Java-based implementations could outperform native and Sulong implementations because they used the JIT-compiled versions of the built-in methods of JDK.

The regexdna implementation on TruffleRuby uses a Java port of the Oniguruma library. This library uses the same Java classes (i.e., from `java.util.regex` package) for matching regular expressions. Therefore, similar to Java, TruffleRuby also outperformed native and Sulong implementations.

Although both Sulong and native implementations called the same external library functions, the slowdown for Sulong was much higher compared to the native implementation. In the case of Sulong, the overhead for calling external library functions is much higher compared to the native execution. This is because Sulong needs to use

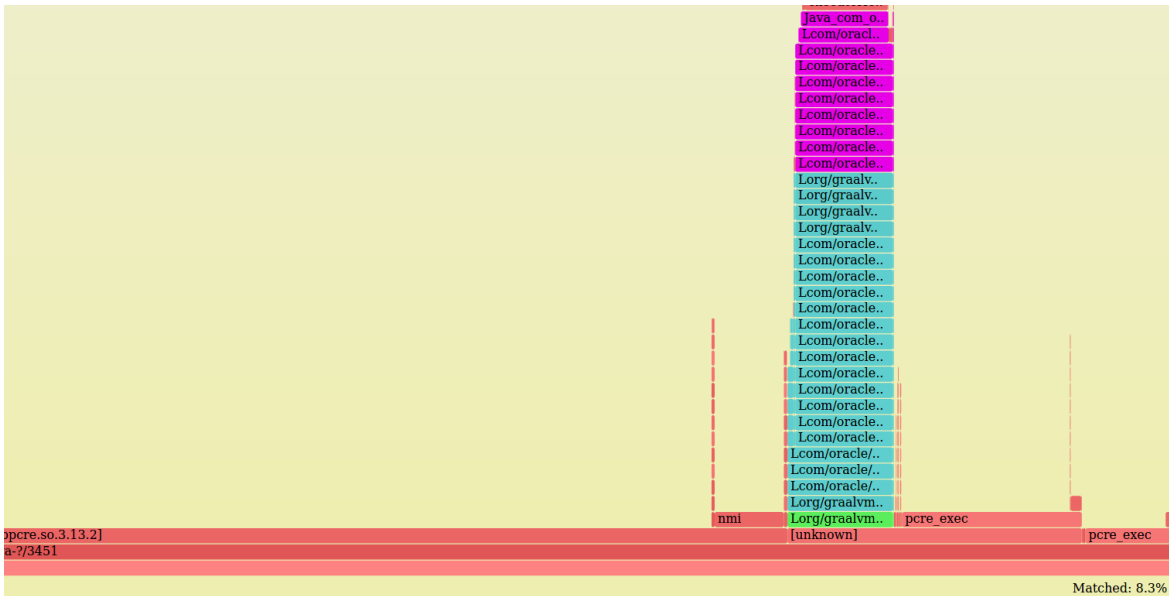


Figure 6.16: Flamegraph shows profile of executing regexdna benchmark on Sulong. The keyword search for "nfi" in the flamegraph is highlighted using magenta colour, and it matched 8.3% call-stack samples (shown at the right bottom corner).

Native Function Interface (NFI) mechanism of Graal to make external library calls. The slowdown for Sulong, compared to the native execution, is proportional to the time spent in calls using NFI. The overhead of calling external library function for Sulong is highlighted using the flamegraphs for knucleotide (in Figure 6.15) and regexdna (in Figure 6.16) benchmarks. Flamegraph for the knucleotide benchmark shows that it has about 70% of the call-stack samples matching the keyword "NFI"; while for the regexdna benchmark, the matching call-stacks are only 8.3%. The knucleotide benchmark on FastR performed extremely slow (about 500x). This is discussed later in Section 6.6.

6.6 Case Studies

6.6.1 Case Study 1: Slowdown of knucleotide on FastR

The knucleotide benchmark on FastR is more than 500x slower than its Java equivalent; and even after warm-up the FastR is 35% slower than the GNU R v3.2.3. Figure 6.17 shows part of the flamegraph that highlights the `REnvironment.put()` method, where the benchmark spends about half of its execution time (46.2%). `REnvironment.put()`

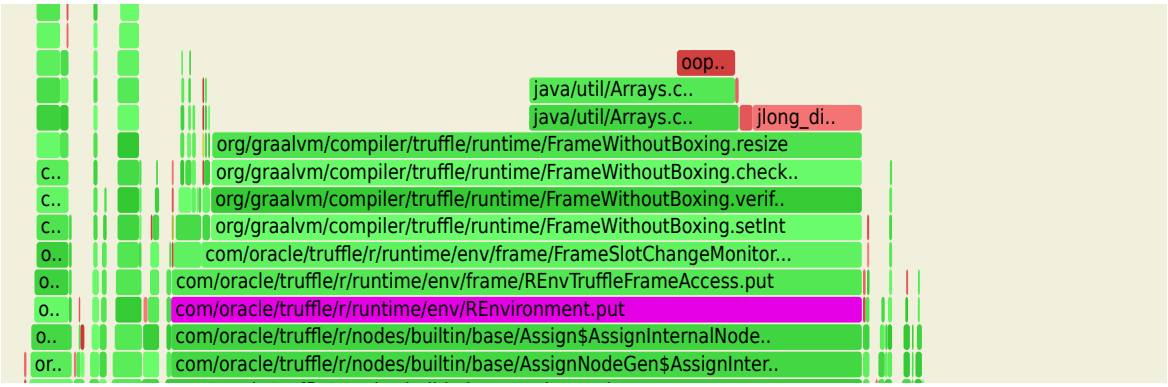


Figure 6.17: The flamegraph shows a part of a profile where about 46.2% time is spent when the knucleotide program executed on FastR. The *REnvironment.put* function, highlighted in magenta colour, implements `assign` function for *Environments* in R that is equivalent to the insert operation on a HashMap.

implements the `put()` function of the `Environment` package in R. This function provides a `HashMap` update functionality in the R benchmark source. Majority of the total execution time is spent in the `FrameWithoutBoxing.resize()` method while performing the `put` operation.

In the case of running knucleotide benchmark on Java, majority of the execution time is spent on updating `HashMap`, and on allocating arrays. However, in FastR, the `FrameWithoutBoxing`-based implementation of *Environments* is about 240x slower than Java. On examining the usage of the FastR `resize()` function in a debugger, we observed that: each time when a new *key* is inserted into the *Environment* variable, it calls the `resize` operation. The `resize` operation increments the size of all underlying data structures only by 1. Therefore, every `put` operation leads to resizing of the global map. Java-based implementation of knucleotide uses the HotSpot's implementation of the `HashMap`. In this case, size of the map is doubled when the `put` operation cannot find a slot to insert a value. This avoids frequent resizing of the map and is much efficient compared to the approach of FastR.

The overhead of resizing the map for FastR is high because the underlying class: `FrameWithoutBoxing` uses three member-arrays to provide map-like functionality. Thus, for every `resize` operation, all three member-arrays are copied. This copy operation involves allocating new arrays with an extra slot, and copying all the elements to new arrays; both of which are expensive operations. The benchmark inserts keys throughout its execution. Therefore, the size of the array that provides `HashMap`

functionality increments by 1 for every insert operation. This severely increases overhead of `put` operation in FastR, compared to that in Java. Additionally, the old arrays need to be garbage collected, which further stresses the GC. Note that, we do not claim that this is the only source of overhead in comparison to Java; but we do claim that our analysis techniques have enabled easy identification of this issue. It further backs our expectation that Truffle runtime library classes may be used in the ways that are inefficient for Truffle framework language implementations. Thus, the implementation of *Environments* in FastR can be improved significantly by i) potentially by mapping directly onto Java `HashMap`, or ii) by modifying the implementation of `FrameWithoutBoxing` class in Truffle. The key message is that the bottlenecks in the language implementation, such as the `FrameWithoutBoxing` based implementation of *Environments* in FastR, can be easily identified using the extended perf-map-agent based approach.

6.6.2 Case Study 2: Unexpected slowdown for a Ruby test

When we interacted with TruffleRuby community with our performance analysis work, they shared an interesting usecase with us that demonstrates an unexpected slowdown in the seemingly innocuous code. A code snippet from the usecase is shown in Listing 6.1.

```
1  TEST_A = [35745007559640128, 21462147108176960, 2333148165535564864,
2  487195639305610309, 81269024335471685, 469583564253826181]
3
4  TEST_B = [86312355559572544, 62173472877773888, 9351163663556674624,
5  691699238949762116, 89233839996970053, 969579166208363671]
6  ...
7  average(TEST_A)     # test a
8  average(TEST_B)     # test b
9  ...
```

Listing 6.1: A simple Ruby program that returns an arithmetic average of the array elements. The warmed-up execution of `test b` is about 5.9 times slower than the `test a`.

Figure 6.18: Flamegraph for warmed-up execution of the `test a` from Listing 6.1. Flamegraph shows that about 99% of the execution time is spent in calculating arithmetic average.

The code in Listing 6.1 contains two tests: `test a` and `test b`, both of them calculate arithmetic average of the elements in arrays `TEST_A` and `TEST_B` respectively. Interestingly, the warmed-up execution of the tests show that the `test b` is about 5.9x slower than that of `test a`.

Figure 6.18 and Figure 6.19 show profiles for the warmed-up executions of `test a` and `test b` respectively. Here, the profiles look quite different for seemingly equivalent code. When the profile for `test b` is observed, it can be seen that about 59% of total execution time is spent while adding the numbers and 23% of the time is spent while performing division (for computing average). Importantly, it used the Java `BigInteger` class for the arithmetic operations. On the other hand, `test a` spends about 99% of its execution time in calculating the arithmetic average, and the call-stacks shows that the primitive division is being used. Note that although the percentages for the tests are comparable (99% for `test a` and 82% for `test b`), their values are for different wall-clock times (i.e., if `test a` takes X seconds, then `test b`

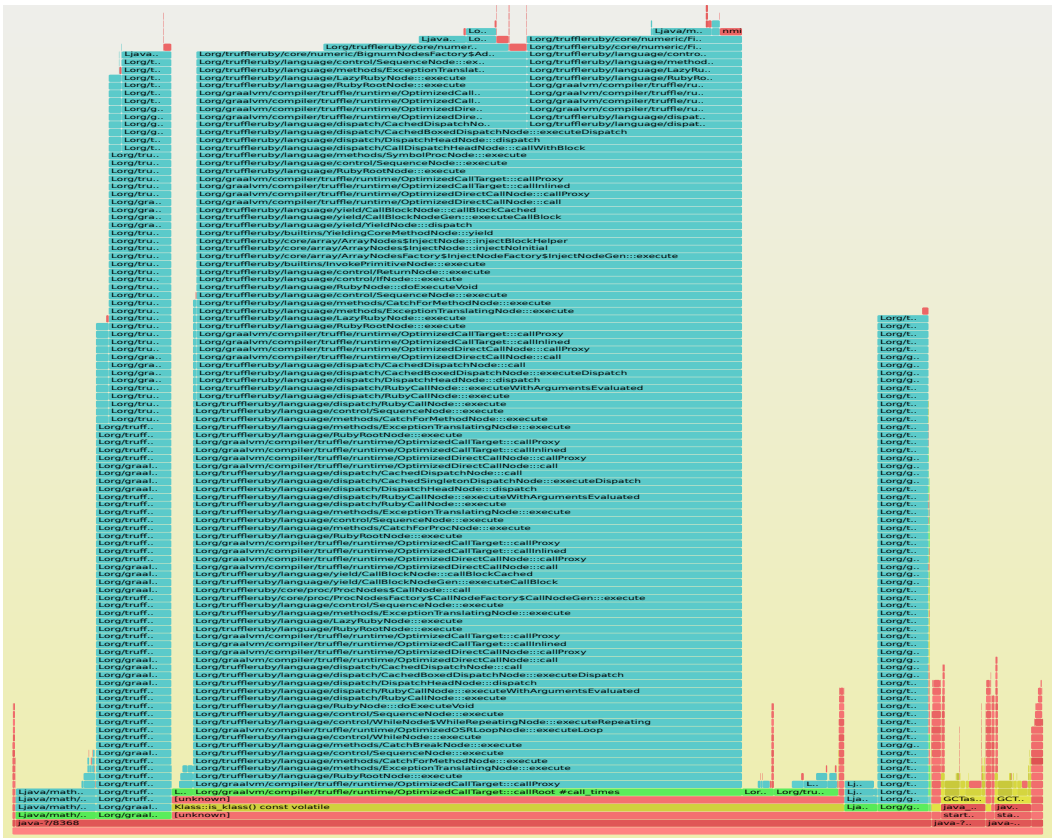


Figure 6.19: Flamegraph for warmed-up execution of the `test b` from Listing 6.1. Flamegraph shows that about 82% of the execution time is spent in calculating the arithmetic average compared to 99% for `test a` in Figure 6.18. Although percentage time in the case of `test b` is smaller compared to `test a`, the wall-clock time for `test b` is about 5.9 times the time taken for `test a`.

takes 5.9X seconds. Thus, 80% of X is smaller than 80% of 5.9X). The reason for such high overhead is that in the case of `test b`, execution switched to use `BigInteger` class from Java for arithmetic operations which has much higher overhead compared to its primitive equivalent used in `test a`. Call-stacks for the `BigInteger` class are not clearly visible in the formatted image of the flamegraph (in Figure 6.19), but they can be searched in the svg format of the image. Furthermore, use of `BigIntegers` also increased GC pressure that is visible in the form of wider yellow towers (in Figure 6.19 comapared to `test a` in Figure 6.18) at bottom right corner of the flamegraph. The profile for `test b` shows that about 10% of the time is spent in performing GC operations. The GC occurs in the background, which indicates that the reported percentages of time spent in performing arithmetic operations is higher than they appear. The percentages also indicate that the operations using `BigInteger` increase GC pressure, compared to its equivalent primitive operations. This is highlighted by the time spent in GC for `test a`, which is relatively negligible (about 0.2%).

The usecase showed in Listing 6.1 is a simplistic version of the behaviour that can be found in different parts of the application or library functions. Ruby is a dynamically typed language. This allows the underlying implementation of Ruby to switch from using the smaller and faster representation of the numbers to that on the larger and significantly slower representation. The ability of Ruby to switch transparently to a different representation makes it difficult for the end-user to know why the same code performed significantly slower with different data. We have seen in Section 2.3 that the AST specialisation moves towards more generic operations to avoid continuous re-specialisation. Thus, once the operation is specialised to handle the slower generic behaviour of the input, it gets optimised for the generic version and stays slower until the application terminates. For long-running applications (common usecase for Ruby web applications), such behaviour can lead to a less obvious slowdown. Using our approach, a short profile sample of a slow application can help to identify unexpected slowdowns, even in production environments. A short profile sample can be collected using a few slower HTTP requests for a web application.

6.7 Limitations

6.7.1 Limitations of perf

As our perf-map-agent based approach uses the `perf` utility to record the call-stack samples, we also inherit its limitations. First, `perf` uses the FramePointer register to walk the application call-stacks. Therefore, we need to start the JVM with the `-XX:+PreserveFramePointer` flag, so that we can record call-stacks for the JIT-compiled methods of Java applications. Second, the call-stacks for the methods being interpreted are shown as *Interpreter* in the corresponding flamegraph. The detailed split of time spent in the interpreted methods is not available, which might be of interest to some applications. Third, occasionally the recorded call-stacks are broken when the `perf` cannot walk the call-stack. The broken stacks are shown as a stackframe named *unknown* in the flamegraph. When some of the call-stack samples for a method contain a broken stackframe, such broken samples cannot be matched under the same tower in the flamegraph. Consequently, the flamegraph now reports two separate towers (with and without broken stackframe) for the method. This may impact the readability because the collective percentage of the samples for the method are now split into two groups. To overcome this issue, the search functionality of the flamegraphs can be used. When a common method name is searched, it matches the call-stack samples across the towers, then highlights the matching stack-frames and reports the percentage of the matched call-stack samples.

6.7.2 Profiling Interpreted and Inlined methods

Our current approach maps the guest language method names only for the JIT-compiled methods of an application, but not for the interpreted or the guest methods that are inlined in other JIT-compiled methods.

Our main objective is to identify the performance bottlenecks during execution. Therefore, until now we focused on the JIT-compiled methods because they are executed frequently, and deemed as `hot` by the JVM. We are not showing the names of methods that are being interpreted. However, this may be important for some applications. These applications can use the `AsyncGetCallTrace` based profilers (discussed

in Section 6.2), such as `async-profiler`, to record the names of interpreted methods. In the future, we plan to integrate our work with the `async-profiler` to enable tracing of the interpreted methods.

Currently, our approach is unable to recognise the guest language function names for interpreted methods. To address this limitation, two approaches are considered. i) by using the JVMTI calls to retrieve the value of a guest language function name from a frame local variable. Initial experiments showed that this approach had a significant overhead, which may impact the application behaviour. ii) by using the instrumentation framework of Truffle. Truffle offers a built-in sampling profiler: `CPUSampler`. This profiler creates a shadow call-stack for an application and reports the time spent in the respective guest language functions [Cor17].

Regarding the inlined methods, the information would add valuable information to the recorded profile. One of the ways to address this is to retrieve the guest language method name and dump the inlining information at the end of the JIT compilation. We have not yet evaluated the feasibility of this approach or the overhead incurred by it.

6.7.3 Handling deoptimisation and JIT recompilation

As we discussed in Section 6.3, the `perf-map-agent` dumps a snapshot of the symbol table of the *enterant* JIT-compiled methods only once, when the agent is attached. The methods that are JIT-compiled after that point could not be mapped to their Java method names (and the guest language function names) because their name mapping would not be present in the symbol table. This issue can be addressed by dumping a new snapshot whenever the JIT-compilation or deoptimisation occurs. The JIT-compilation/deoptimisations can be detected using a JVMTI agent that is listening to the events named as `Compiled Method Load` and `Unload` [Ora04].

6.7.4 AOT compiled methods

Currently, we cannot match the guest language function names for the Ahead-of-Time (AOT) compiled applications. As discussed previously, the Substrate VM allows us to create an AOT compiled binary using Graal. These binary files can then be executed as

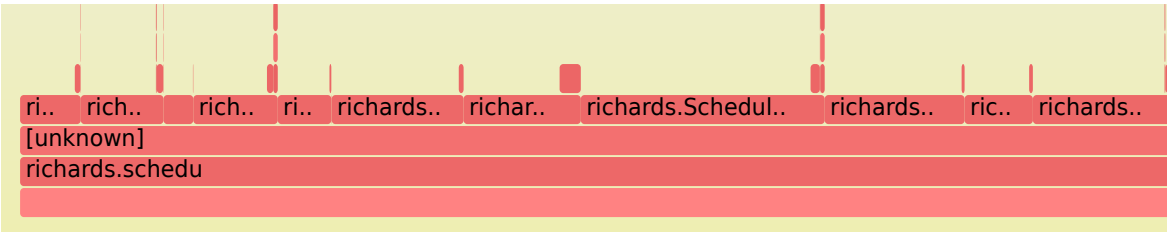


Figure 6.20: A profile for executing an AOT-compiled binary, generated using the Substrate VM, for the Richards benchmark written in Java.

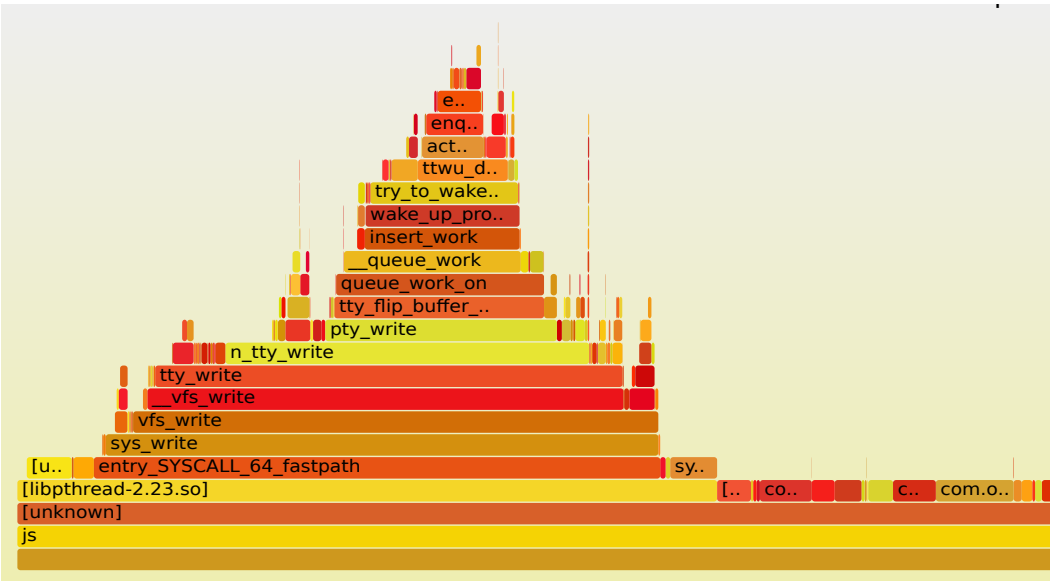


Figure 6.21: Native application using `libpolyglot` to access C and Truffle JavaScript together. The C application executes a JavaScript code that prints the square root of the first million integers, starting from 1. This profile shows that the application spent 37.41% of the total execution time while printing the results on the standard output.

a standalone executable (i.e., without using a JVM). Figure 6.20 shows flamegraph of a profile for executing the AOT-compiled binary for the Richards Benchmark written in Java [Ric99]. The flamegraph demonstrates that our approach can be used with AOT-compiled binaries, as it is based on `perf`. The names on the stackframes correspond to the Java method names in the benchmark. However, when an AOT compiled Truffle hosted interpreter for a guest language is used, the profile would contain the `callRoot` method name.

The native applications can call Truffle hosted language implementations using the Polyglot library. Profile for such an application in C is shown Figure 6.21. This C application calls a JavaScript function that computes and prints square roots of

integers ranging from 1 to 1 million. Using `perf` for sampling allows us to record call-stack samples for such execution, but we cannot map the guest language function names.

6.8 Summary

In this chapter, we presented the perf-map-agent based profiling approach used for performance analysis of the Truffle hosted guest language implementations. We discussed the main categories of approaches used for profiling: i) tracing, and ii) sampling; along with their trade-offs. We use the sampling-based profiling approach in this work because we found it flexible (i.e., its overhead can be tuned by changing the sampling frequency at the cost of accuracy).

We discussed the commonly used approaches for profiling, such as using the Linux utility `perf`. JVM does not maintain the frame pointer register with the values that `perf` uses to walk the call-stacks. The `-XX:PreserveFramePointer` flag of JVM allows `perf` to record the call-stacks of the JIT-compiled method. One of the profiling approaches we discussed is using the perf-map-agent. This approach records the call-stack samples using `perf`. The stack-frame addresses in recorded call-stack samples are then mapped to their Java method names by using the information retrieved from perf-map-agent. We then discussed alternative JVM profiling approaches using i) the `GetCallStackTrace()` function from JVMTI API, and ii) the `AsyncGetCallTrace()` function from API of OpenJDK. We avoided the approach that uses `GetCallStackTrace()` JVMTI function. Although this approach is commonly used by many profilers, it suffers from safepoint bias. The `GetCallStackTrace()` function is serviced only when the application is at a safepoint so that the call-stack sample of a thread can be safely collected. This results in the samples not getting collected when requested, instead, they get skewed towards safepoint. The safepoint bias often results in an inaccurate profile. We discussed *honest profile* and *async-profiler* who use `AsyncGetCallTrace()` to avoid the safepoint bias. We chose the perf-map-agent based approach because it avoids the safepoint bias and is easier to extend.

For the executions using Truffle hosted languages, instead of showing guest language function names, the existing profilers showed the names of the Java methods that

implemented the interpreter for a language. As a result, it becomes difficult to map the recorded profiling information to the execution of a guest program. We extracted the guest language function names by updating the Graal compiler and combined it with the post-processing step of the perf-map-agent profiler. This enabled us to show the correct guest language function names in the profiles that can then be visualised using flamegraphs.

We used our approach for performance analysis of three Truffle hosted language: FastR, TruffleRuby and Sulong. We demonstrated that our approach can identify performance bottlenecks for applications executing on a Truffle hosted language, or for the language implementation itself. We discussed the limitations of the current approach for identifying guest language functions for the interpreted and inlined methods (during JIT-compilation). Using the `perf` as a sampling profiler causes the call-stack samples to break occasionally. However, `perf` enables us to profile a wide range of use-cases offered by the GraalVM, such as i) the Ahead-of-Time (AOT) compilation, and ii) using the language implementation as an external library in a native application.

Chapter 7

Conclusions & Future work

7.1 Conclusions

In this thesis we present Sulong-OpenMP: an extension to the Sulong project for execution OpenMP programs on JVM. This work demonstrates that the OpenMP memory model could be implemented using the Java memory model, and it could execute OpenMP parallel programs correctly. We also present the performance analysis approach for Truffle hosted guest language implementations.

In this thesis, we have described our journey of implementing OpenMP support to Sulong and the implementation challenges that we faced. This work mainly addressed the overhead for executing OpenMP programs with 1 thread, compared to their sequential execution on Sulong. Evaluation of the NPB benchmark suite shows that the overhead of executing Sulong-OpenMP (1 thread), compared to that of Sulong-sequential, is about 3%. This overhead matches the 2% overhead of its native equivalent (i.e., Native-OpenMP (1 thread) compared to Native-sequential execution). When LLVM IR is generated for a program with OpenMP support enabled (by compiling with `-fopenmp` flag), it contains additional calls to the OpenMP runtime; which otherwise are absent when OpenMP support is disabled. Thus, we expect some overhead while executing with OpenMP support enabled. Although we focused on single-thread performance so far, the applied optimisations highlight the diminishing performance gap for multi-threaded execution. We are hopeful that a significant portion of this performance gap could be covered by addressing the existing limitations, described in Section 7.2.

In the final part of this thesis, we presented our perf-map-agent based profiling approach. This approach demonstrated its usability for performance analysis of Truffle hosted guest language executions, as well as language implementations themselves. The approach has low-performance overhead (less than 1.25%) and requires no modifications to the language implementations, which makes it suitable for use in production. GraalVM offers multiple ways of execution such as i) in a polyglot application, ii) bundled into an external library, and iii) an Ahead-of-Time compiled binary executable. For call-stack sampling, our approach uses a standard Linux tool: `perf`. Using `perf` makes our approach useful for the aforementioned ways of using language implementations. We used our approach for performance analysis of three Truffle hosted language implementations: TruffleRuby, FastR and Sulong. The approach presented in this thesis is expected to aid the identification of further potential directions for performance improvements.

7.2 Future Work

The current implementation of Sulong-OpenMP is in its primitive stage. However, JVM hosted execution has the potential to provide improved tools for OpenMP programs. Rigger et al., have demonstrated that Sulong subsystem can be used to detect memory bugs such as use-after-free and double deletion of memory for single-threaded programs [RSM+18]. For Sulong-OpenMP, one possibility is to build a low overhead data-race detection tool for OpenMP parallel programs. ThreadSanitizer (also known as TSan) is one of the popular data-race detection tools that uses compile-time instrumentation to track memory accesses done by multiple threads and processes them to detect races [SPIV12]. As Sulong-OpenMP manages memory allocations for the programs, it can also be extended to perform necessary instrumentation. Unlike AOT compilers, JVM hosted executions benefit from runtime information. This information can be used to identify parts of the program that are used by multiple threads and focus instrumentation efforts only on those parts to reduce execution overhead.

In this section, we discuss the existing limitations and future work for the implementation of Sulong-OpenMP. We also highlight the directions to improve the perf-map-agent based performance analysis approach.

7.2.1 Sulong-OpenMP

Performance of multi-thread execution

For multi-threaded execution using Sulong-OpenMP, optimisations that are presented, considerably improved the performance of executing benchmarks from the NPB suite on Sulong-OpenMP. However, there is still a substantial performance gap that Sulong-OpenMP needs to fill. One of the major sources of overhead comes from the existing limitation that requires to materialize the frame of top-level outlined OpenMP function. We observed that execution of the outlined function was as slow as its execution in the slow interpreted mode. In the future, we plan to focus primarily on improving the performance of Sulong-OpenMP.

OpenMP Features

Supporting all the OpenMP features require a significant amount of work. Therefore, we chose a subset of OpenMP features, that could demonstrate the ability of Sulong-OpenMP to execute OpenMP parallel programs on a JVM. Currently, we support only the features required to execute NPB suite; and excludes features such as OpenMP tasks and offload directives. We have done a preliminary evaluation of changes required for supporting OpenMP tasks on Sulong-OpenMP. We plan to use `ExecutorService` class from the `java.concurrent` package of JDK, that provides functionalities to create and manage a pool of Java threads. These threads can represent OpenMP threads and can then be assigned OpenMP tasks for execution.

Support for the offload directives can be added by integrating Sulong-OpenMP with the existing systems, such as TornadoVM. The TornadoVM project allows execution of the Java methods using the heterogeneous architectures [KCR+17]. This project currently targets the OpenCL compatible devices such as multi-core CPUs, GPUs and FPGAs. It takes the annotated Java code using the TornadoVM specific annotations. These annotations highlight parts of the code that are to be executed on the heterogeneous hardware. TornadoVM generates OpenCL equivalent code for the annotated parts of code. To support OpenMP offload directives using TornadoVM, it would first require to enable support for the Truffle framework. However, supporting both needs a significant amount of development efforts.

Sulong-OpenMP needs additional engineering efforts to improve its compatibility with the latest version of Sulong and GraalVM. This would benefit from the enhancements and bug fixes incorporated in the newer versions. Efforts are also required to address the JVM crashes for using larger input sizes for the NPB suite benchmarks, which has limited us from performing further scalability experiments using Sulong-OpenMP. Sulong-OpenMP could not execute the larger benchmark suites, such as PARSEC and SPEC-OMP suite, because of following reasons: i) limitations of underlying system: Sulong (previously mentioned in [RSM+18]), and/or ii) we could not generate and execute LLVM IR for the benchmark dependencies. Recently, Sulong has added support for the pthreads. It would be interesting to compare the execution of OpenMP programs on Sulong-OpenMP, to that on pthreads-enabled Sulong. This comparison would require pthreads-enabled Sulong to use the OpenMP runtime of clang (compiled to the LLVM IR).

To achieve completeness, Sulong-OpenMP needs to implement support for additional OpenMP runtime functions associated with the missing functionality. In order to prioritise the features to be implemented, one approach would be to choose additional benchmark suites and incorporate all the necessary functionalities for their execution. This approach may face a challenge where the underlying Sulong system might not be able to execute a single-threaded version of the benchmarks. This issue of inability to execute a single-threaded version could be avoided by selecting a microbenchmark suite focused on a specific set of OpenMP features. Examples of such microbenchmark suites are: i) EPCC OpenMP microbenchmark suite [Bul02] which focuses on synchronisation and loop constructs in OpenMP, ii) EPCC microbenchmark suite for OpenMP tasks [BRM12] which focuses on OpenMP tasks, iii) Barcelona OpenMP Tasks Suite (BOTS) [DTF+09] is another suite that focuses on OpenMP tasks. Furthermore, implementation of OpenMP features such as `sections` and `teams` may require extending Sulong-OpenMP's data structures.

7.2.2 Extended Perf-map-agent

AOT Profiling

Currently, our approach does not provide support for recognising AOT-compiled execution (of Truffle hosted languages) and are represented as any other native application. GraalVM offers AOT-compiled interpreters for the Truffle hosted languages such as JavaScript, R and Ruby, using the SubstrateVM. As future work, we can modify the AOT compilation infrastructure (in a similar manner to what we have done for Graal) for relating guest language names to their respective symbol addresses, and also for identifying inlining decisions. This guest language information can be very useful in the optimisation process.

Performance evaluation of Truffle hosted languages

We have seen huge variations in performance for the same benchmark computations when written using different Truffle hosted language implementations. Precise reasons behind these performance variations can be explored in the future. Performance differences may arise as a result of expressing the same computation in different ways; or because of language implementation inefficiencies. We have seen an example of language implementation inefficiency in the case study of FastR. However, we need to investigate further to determine if the CBLG Shootout benchmarks are suitable for making cross-language performance comparisons.

Bibliography

[AH17] Soramichi Akiyama and Takahiro Hirofuchi. Quantitative Evaluation of Intel PEBS Overhead for Online System-Noise Analysis. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017*, ROSS '17, pages 3:1–3:8, New York, NY, USA, 2017. ACM.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools (2nd Edition). chapter 9, pages 588–592. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[ARB19] OpenMP Architecture Review Board (OpenMP ARB). OpenMP Application Programming Interface. `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`, 2019.

[ARB20] OpenMP Architecture Review Board (OpenMP ARB). OpenMP Home Page. `https://www.openmp.org`, 2020.

[BBJ+91] David Bailey, E. Barszcz, Barton J.T, Browning D.S, Carter R.L, Dagum D, Fatoohi R.A, Paul Frederickson, Lasinski T.A, Robert Schreiber, Horst Simon, Venkat Venkatakrishnan, and Weeratunga K. The Nas Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5:63–73, 09 1991.

[BBTK+17] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.*, 1(OOPSLA):52:1–52:27, October 2017.

[BK00] J. M. Bull and M. E. Kambites. JOMP—an OpenMP-like Interface for Java. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 44–53, New York, NY, USA, 2000. ACM.

[Bla03] Bruno Blanchet. Escape Analysis for JavaTM: Theory and Practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, November 2003.

[BRM12] J. Mark Bull, Fiona Reid, and Nicola McDonnell. A Microbenchmark Suite for OpenMP Tasks. In Barbara M. Chapman, Federico Massaioli, Matthias S. Müller, and Marco Rorro, editors, *OpenMP in a Heterogeneous World*, pages 271–274, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[BS19] Petr Bělohlávek and Antonín Steinhauser. omp4j Project website. `http://www.omp4j.org`, 2019.

[Bul02] Mark Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. *Proceedings of First European Workshop on OpenMP*, 02 2002.

[CFE97] Brad Calder, Peter Feller, and Alan Eustace. Value Profiling. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages 259–269, Washington, DC, USA, 1997. IEEE Computer Society.

[Cor15] Oracle Corporation. Graal VM. `https://github.com/graalvm`, 2015.

[Cor16a] Oracle Corporation. GitHub Repository for Graal.js. `https://github.com/graalvm/graaljs`, 2016.

[Cor16b] Oracle Corporation. TruffleRuby. `https://github.com/oracle/truffleruby`, 2016.

[Cor17] Oracle Corporation. GraalVM Sampling Profiler. `https://www.graalvm.org/docs/reference-manual/tools/#profiler`, 2017.

[Cor19a] Oracle Corporation. JVM Sepcification: The code attribute. https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.3, 2019.

[Cor19b] Oracle Corporation. Source for NewFrameNode.java. https://github.com/oracle/graal/blob/release/graal-vm/20.2/compiler/src/org.graalvm.compiler.truffle.compiler/src/org/graalvm/compiler/truffle/compiler/nodes/frame/NewFrameNode.java#L135, 2019.

[Cor20] Oracle Corporation. GraalVM demos: Performance Examples for Java. https://www.graalvm.org/docs/examples/java-performance-examples/, 2020.

[CP95] Cliff Click and Michael Paleczny. A Simple Graph-based Intermediate Representation. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, pages 35–49, New York, NY, USA, 1995. ACM.

[DTF+09] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *2009 International Conference on Parallel Processing*, pages 124–131, 2009.

[EPFL04] Switzerland École Polytechnique Fédérale Lausanne. The Scala Programming Language. https://www.scala-lang.org/, 2004.

[FF03] S. J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 241–252, March 2003.

[Fou03] Apache Software Foundation. Apache Groovy. https://groovy-lang.org/, 2003.

[Fou09] CTuning Foundation. Collective Knowledge Framework (CK). https://github.com/ctuning/ck, 2009.

[Fou16] Eclipse Foundation. Eclipse OMR. `https://github.com/eclipse/omr`, 2016.

[Gai15] Swapnil Gaikwad. Bug reported on GitHub of FastR. `https://github.com/oracle/fastr/issues/15`, 2015.

[GNL18] Swapnil Gaikwad, Andy Nisbet, and Mikel Luján. Performance Analysis for Languages Hosted on the Truffle Framework. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ManLang '18, pages 5:1–5:12, 2018.

[GNL19] Swapnil Gaikwad, Andy Nisbet, and Mikel Luján. Hosting OpenMP Programs on Java Virtual Machines. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2019, page 63â€“71, New York, NY, USA, 2019. Association for Computing Machinery.

[Gre16] Brendan Gregg. The Flame Graph. *Queue*, 14(2):10:91–10:110, March 2016.

[GRS+13] Matthias Grimmer, Manuel Rigger, Lukas Stadler, Roland Schatz, and Hanspeter Mössenböck. An efficient native function interface for Java. pages 35–44, 09 2013.

[GSS+18] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Mikel Luján. Cross-Language Interoperability in a Multi-Language Runtime. *ACM Transactions on Programming Languages and Systems*, 40(2):8:1–8:43, 2018.

[Guo18] Isaac Guoy. Gouy, Isaac. The Computer Language Benchmarks Game. `https://benchmarksgame-team.pages.debian.net/benchmarksgame/`, 2018.

[HDS05] Jay P. Hoeflinger and Bronis R. De Supinski. The OpenMP

Memory Model. In *Proceedings of the 2005 and 2006 International Conference on OpenMP Shared Memory Parallel Programming*, IWOMP'05/IWOMP'06, page 167–177, Berlin, Heidelberg, 2005. Springer-Verlag.

[HWW+14] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A Domain-specific Language for Building Self-optimizing AST Interpreters. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 123–132, New York, NY, USA, 2014. ACM.

[IEE95] IEEE. POSIX Threads. `https://en.wikipedia.org/wiki/POSIX_Threads`, 1995.

[KBVP07] Michael Klemm, Matthias Bezold, Ronald Veldema, and Michael Philippsen. JaMP: An Implementation of OpenMP for a Java DSM: Research Articles. *Concurr. Comput. : Pract. Exper.*, 19(18):2333–2352, December 2007.

[KCR+17] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. Heterogeneous Managed Runtime Systems: A Computer Vision Case Study. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, page 74–82, New York, NY, USA, 2017. Association for Computing Machinery.

[KJ13] Tomas Kalibera and Richard Jones. Rigorous Benchmarking in Reasonable Time. *SIGPLAN Not.*, 48(11):63–74, June 2013.

[LLV03] LLVM. opt - LLVM Optimizer. `https://llvm.org/docs/CommandGuide/opt.html`, 2003.

[LLV18] LLVM. OpenMP: Support for the OpenMP language. `https://openmp.llvm.org/`, 2018.

[MDHS10] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Evaluating the Accuracy of Java Profilers. In *Proceedings of*

the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, pages 187–197, New York, NY, USA, 2010. ACM.

[MDM16] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-language Compiler Benchmarking: Are We Fast Yet? In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, pages 120–131, New York, NY, USA, 2016. ACM.

[MLR+19] Raphael Mosaner, David Leopoldseder, Manuel Rigger, Roland Schatz, and Hanspeter Mössenböck. Supporting On-stack Replacement in Unstructured Languages by Loop Reconstruction and Extraction. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2019, pages 1–13, New York, NY, USA, 2019. ACM.

[MPM+15] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at Your Own Risk: The Java Unsafe API in the Wild. *SIGPLAN Not.*, 50(10):695–710, October 2015.

[NED+13] Dorit Nuzman, Revital Eres, Sergei Dyshel, Marcel Zalmanovici, and Jose Castanos. JIT Technology with C/C++: Feedback-Directed Dynamic Recompilation for Statically Compiled Languages. *ACM Trans. Archit. Code Optim.*, 10(4), December 2013.

[Ora04] Oracle. Java Virtual Machine Tool Interface (JVM TI). `https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/index.html`, 2004.

[Ora09] Oracle. The invokedynamic instruction. `https://docs.oracle.com/javase/8/docs/technotes/guides/vm/multiple-language-support.html#invokedynamic`, 2009.

[Pan19] Andrei Pangin. Async Profiler GitHub Repository. `https://github.com/jvm-profiling-tools/async-profiler`, 2019.

[Par13] Terence Parr. ANTLR. `https://github.com/antlr/antlr4`, 2013.

[per15] Perf Tool Man Page. `https://man7.org/linux/man-pages/man1/perf.1.html`, 2015.

[Pro19] Honest Profiler. AsyncGetCallTrace errors and what they mean. `https://github.com/jvm-profiling-tools/honest-profiler/wiki/AsyncGetCallTrace-errors-and-what-they-mean`, 2019.

[Pug19] Willam Pugh. JSR 133, Java Memory Model and Thread Specification. `http://www.cs.umd.edu/~pugh/java/memoryModel/CommunityReview.pdf`, 2019.

[RGW+16] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*, VMIL, pages 6–15, New York, NY, USA, 2016. ACM.

[Ric99] Martin Richards. Richards Benchmark. `http://www.cl.cam.ac.uk/~mr10/Bench.html`, 1999.

[Ros09] John R. Rose. Bytecodes Meet Combinators: Invokedynamic on the JVM. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, VMIL '09, pages 2:1–2:11, New York, NY, USA, 2009. ACM.

[Ros19] John Rose. JEP 243: Java-Level JVM Compiler Interface. `https://openjdk.java.net/jeps/243`, 2019.

[RSM+18] Manuel Rigger, Roland Schatz, René Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. Sulong, and Thanks for All the Bugs: Finding Errors in C Programs by Abstracting from the Native Execution Model. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 377–391, 2018.

[SB96] Subhash Saini and David Bailey. NAS Parallel Benchmark (Version 1.0) Results 11-96. `https://www.nas.nasa.gov/assets/pdf/techreports/1996/nas-96-018.pdf`, 1996.

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

[Sea15] Chris Seaton. *Specialising Dynamic Techniques for Implementing the Ruby Programming Language*. PhD thesis, University of Manchester, 2015.

[SPIV12] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic Race Detection with LLVM Compiler. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification*, pages 110–114, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[SWHJ16] Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. Optimizing R Language Execution via Aggressive Speculation. In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS, pages 84–95, 2016.

[SWM14] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 165:165–165:174, New York, NY, USA, 2014. ACM.

[TC07] Linda Torczon and Keith Cooper. Engineering A Compiler. chapter 5, page 246. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2007.

[Tol19] Robert Tolksdorf. Programming languages for the Java Virtual Machine. https://vmlanguages.is-research.de/category/jvm-language/, 2019.

[VBB01] R. Veldema, R. A. F. Bhoedjang, and H. E. Bal. Jackal, A Compiler Based Implementation of Java for Clusters Of Workstations. In *IN PROC. OF PPOPP*, 2001.

[Wak15] Nitsan Wakart. Safepoints: Meaning, Side Effects and Overheads. http://psy-lob-saw.blogspot.com/2015/12/safepoints.html, 2015.

[War19] Richard Warburton. Honest Profiler GitHub Repository. https://github.com/jvm-profiling-tools/honest-profiler, 2019.

[Was12] Michael Waskom. Violin plot documentation. https://seaborn.pydata.org/generated/seaborn.violinplot.html, 2012.

[WHVDV+13] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.*, 9(4):30:1–30:24, January 2013.

[Wik09] Wikipedia. Tail Call. https://en.wikipedia.org/wiki/Tail_call, 2009.

[Wik13a] Wikipedia. Instruction-level parallelism. https://en.wikipedia.org/wiki/Instruction-level_parallelism, 2013.

[Wik13b] Wikipedia. Out-of-order Execution. https://en.wikipedia.org/wiki/Out-of-order_execution, 2013.

[Wik13c] Wikipedia. Power Management. https://en.wikipedia.org/wiki/Power_management#DVFS, 2013.

[Wik16] Wikipedia. HotSpot JVM. https://en.wikipedia.org/wiki/HotSpot, 2016.

[WSH+19] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. Initialize Once, Start Fast: Application Initialization at Build Time. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[WWH+17] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 662–676, New York, NY, USA, 2017. ACM.

[WWW+13] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward!, pages 187–204, New York, NY, USA, 2013. ACM.

Appendix A

OpenMP features supported on Sulong-OpenMP

Table A.1 shows OpenMP constructs used in the NAS Parallel Benchmark suite 3.0.

OpenMP Construct	**Benchmark Name**
`parallel` construct Parallel Worksharing-Loop	HelloWorld (with OpenMP for-loop)
`master` clause `critical` clause `shared` clause `private` clause `nowait` clause `barrier` clause	IS
`single` construct	FT
Reduction clause	EP, CG, MG
flush construct	BT, SP, LU

Table A.1: shows the sequence (from top to bottom) of implementing support for the specific OpenMP features (in the left column) on Sulong-OpenMP and its target benchmark (in the right column).

In Table A.1, left column contains a sequence of OpenMP features in the order of their implementation in Sulong-OpenMP. The right column contains a benchmark from the NAS Parallel Benchmarks (NPB) suite that uses the OpenMP feature in its corresponding left column. Note, the benchmarks in the right column may also use one or more previously implemented features.

While generating LLVM IR, Clang replaces OpenMP pragmas by the function calls to its OpenMP runtime library. Sulong-OpenMP provides its own implementation for the runtime function calls as described in Chapter 3. Table A.2 lists the OpenMP pragmas and corresponding runtime functions that Sulong-OpenMP implements for them.

OpenMP Construct	**OpenMP Runtime Function**
`parallel` construct Parallel Worksharing-Loop	omp_get_thread_num omp_get_num_threads @__kmpc_fork_call @__kmpc_for_static_init_4 @__kmpc_for_static_finish
`master` clause	@__kmpc_master @__kmpc_end_master @__kmpc_global_thread_num
`critical` clause	@__kmpc_critical @__kmpc_end_critical
`shared` clause	None
`private` clause	None
`nowait` clause	None
`barrier` clause	@__kmpc_barrier
`single` construct	@__kmpc_single @__kmpc_end_single
Reduction clause	@__kmpc_reduce_nowait @__kmpc_end_reduce_nowait
flush construct	@__kmpc_flush

Table A.2: shows the OpenMP constructs and corresponding clang-generated runtime library functions which are implemented by Sulong-OpenMP.

In Table A.2, the `shared`, `private` and `nowait` OpenMP pragmas do not have any runtime function calls associated with them because the generated LLVM IR achieves the necessary functionality.