Performance Analysis for Languages Hosted on the Truffle Framework

Swapnil Gaikwad
University of Manchester
Manchester, UK
swapnil.gaikwad@manchester.ac.uk

Andy Nisbet
University of Manchester
Manchester, UK
andy.nisbet@manchester.ac.uk

Mikel Luján
University of Manchester
Manchester, UK
mikel.lujan@manchester.ac.uk

ABSTRACT

It is attractive to host new or existing language implementations on top of, or reusing components of, existing managed language runtimes such as the Java Virtual Machine (JVM) or the Microsoft Common Language Infrastructure (CLI). A benefit is that software development effort may be reduced, as only one managed language runtime needs to be optimised and maintained, instead of a separate compiler/runtime for each language implementation.

For example, the Truffle framework combined with a JVM offers support for executing Javascript, Ruby, R, LLVM IR-compiled languages, etc., as well as supporting the execution of applications combining multiple programming languages within a *Polyglot application*. In trying to understand the runtime performance of *Sulong* (i.e. the Truffle project which enables LLVM IR execution), we found a lack of tools and guidance. A similar situation is found for benchmarks written in Ruby and R when executed as Truffle hosted languages. Further, it is non-trivial to relate performance back to the hosted language source code, and to determine whether JVM service overheads, such as garbage collection or JIT compilation, are significant.

We describe how to visually analyse the performance of Truffle-hosted languages based on *Flamegraphs*, allowing time to be related to sampled call-stacks. We use the Linux tool *perf* and the JVM agent *perf-map-agent*, along with enhancements to the Graal JIT compiler that map sampled call-stacks onto JVM-hosted *guest* language source code. This paper demonstrates the ease and flexibility of using these modified tools, with low overhead during execution time. We also illustrate applicability of the techniques to understand the performance of Polyglot applications.

CCS CONCEPTS

• **Software and its engineering** → Virtual machines; Runtime environments; Just-in-time compilers;

KEYWORDS

Profiling, Java virtual machine, Ruby, R

We acknowledge the support of the UK EPSRC grants PAMELA EP/K008730/1, and AnyScale Applications EP/L000725/1. Swapnil Gaikwad is supported by Ph.D. studentship funded by Oracle Labs.

ManLang'18, September 12–14, 2018, Linz, Austria

ACM Reference Format:
Swapnil Gaikwad, Andy Nisbet, and Mikel Luján. 2018. Performance Analysis for Languages Hosted on the Truffle Framework. In *15th International Conference on Managed Languages & Runtimes (ManLang'18), September 12–14, 2018, Linz, Austria.* ManLang'18, Linz, Austria, 12 pages. https://doi.org/10.1145/3237009.3237019

1 INTRODUCTION

The main goal of profiling analysis is to aid the process of optimisation by highlighting where time is spent during execution, especially to help identify performance bottlenecks. Traditional approaches are based either on profiling or tracing. Sometimes, such analysis involves monitoring the behaviour of a service such as Just In Time (JIT) compilation or Garbage Collection (GC) in managed runtime environments. The output (or *profile*) is stored in log files, or sent to a remote tool for further analysis. The profile can be subsequently viewed on a dashboard *GUI* to visualise and monitor the health-status of an application or system.

A recent trend is to host new or existing language implementations on top of, or reusing components of, existing Java Virtual Machines (JVMs): such as, Truffle [26] working together with GraalVM [7], and Eclipse OMR with Eclipse OpenJ9. A benefit is that software development effort is reduced, as only one managed language runtime needs to be optimised and maintained, instead of a separate compiler/runtime for each language implementation. For example, the Truffle framework combined with a JVM, especially with GraalVM, offers support for executing JavaScript (known as Graal.js), Ruby (known as TruffleRuby), R (known as FastR), LLVM IR-compiled languages (known as Sulong), as well as supporting the execution of applications combining multiple programming languages within a *Polyglot application* [12].

In the context of guest languages hosted on managed runtimes, profiling can help to identify, isolate and measure the performance of specific guest language features under different use-case scenarios, making it easier to identify where optimisation is likely to yield significant performance improvements. In this paper, we focus on optimisation opportunities that identify inefficient usage of Truffle APIs by guest language implementations as described in Section 6.1. The identification of such issues may help guide optimisations for language implementation and/or improvements to the Truffle API.

This paper focuses on sampling based profiling (see Section 2) for guest languages hosted on the JVM using the Truffle framework. A language implementation can be hosted on a JVM by creating an *Abstract Syntax Tree* (AST) interpreter using the *Truffle* language implementation framework. Multiple Truffle language implementations can interact within a single process address space at relatively low overhead by avoiding data marshalling at language boundaries [13]. Such *Polyglot* programming allows different portions of an

application to be implemented using whichever language is most appropriate. However, analysing execution time behaviour and performance for such applications is inherently more complex than when dealing with a single language application.

In this paper, we propose modifications to existing tools that have low overhead, and are easy to use. Our approach is novel as it combines the best of existing tools for JVMs as well as using Graal JIT compilation to extract information that relates source code methods in Truffle hosted languages, to sampled call-stacks produced using sampling based performance analysis for Java applications, using `perf`, a standard Linux tool, and the JVM agent `perf-map-agent`. Thus, we extend the reach of performance analysis using sampling profilers to include Truffle hosted languages. The contributions of this paper can be summarised as follows:

- We evaluate the relative performance of 3 Truffle framework language implementations; Sulong, TruffleRuby and FastR are compared to Java and native C performance using the *Computer Language Benchmark Game (CLBG)* shootout benchmarks [14].
- We provide extensions to (see Section 4) standard Linux tools for profiling Truffle framework language applications, using well-known *flamegraph* visualisations (see Section 3). The approach is generally applicable to managed runtime hosted languages, on platforms other than Java. We generate additional profiling that maps guest language function names to their Java method implementations. The overall approach has a low sampling overhead of less than 1.25% on our benchmarks.
- We demonstrate that standard tools can be used to profile Polyglot applications. Our approach can easily disambiguate the profiles associated with different Truffle hosted languages by colouring them separately in the flamegraphs.

2 BACKGROUND ON PROFILING

The two main approaches to profiling are tracing and sampling. Tracing instruments code to measure microarchitecture or system performance metrics. Sampling profilers repeatedly collect snapshots of the stack trace of called functions/methods. The call-stack traces capture the code running on-processor at the sampled moments. The execution time of specific methods can be statistically related to the number of times a specific method appears in all stack traces — as long as the samples are collected fairly and that all points in a program run are selected with equal probability, i.e. randomly to remain statistically independent.

A single snapshot on its own is not meaningful, but if many samples are collected over time, and code stack traces are highly correlated, then we can safely assume that the threads of a program are spending most of their time executing code in the most heavily correlated call-stack. Unfortunately, Java based profilers that directly rely on the JVM Tool Interface (JVMTI) to sample call-stacks suffer from the problem of *safepoint bias* [17]. Call-stacks are inherently skewed towards safepoints where threads may voluntarily yield the processor. All threads must typically yield the processor, or be blocked within a JVM, in order for the JVMTI `GetStackTrace` method to be used to obtain call-stacks for all threads. Mytkowicz *et al.* [17] demonstrated that four different commercial and open source profilers often produced demonstrably different results both for the sampling overhead (varying between 1.1-1.5x) and for identification of the actual hottest method in each DaCapo benchmark.

The main cause of such variable results is the significantly different perturbation to the placement of safepoints in the compiled methods by different profilers. Safepoint placement is an implementation decision of the JVM, therefore different JVMs are likely to report hot-methods with different results and accuracy if a profiler suffers from the problem of Safepoint bias. Further problems concerning sampling based profilers, and also for bytecode based instrumentation using JVMTI, is that any injected bytecode introduces overhead to perform stack trace collection. The injected bytecode also affects safepoint placement decisions, the accuracy of JIT compiler analysis and also the application of dynamic JIT optimisations, leading to, e.g., changes to inlining decisions, and compiled code layout. Consequently, GC and memory behaviour can also be affected. Current popular and widely used profilers such as VisualVM, JProfile and YourKit suffer from the problem of safepoint bias when sampling processor call-stacks.

The *honest-profiler* [3] and *async-profiler* [2] are sampling profilers that have extended the proof-of-concept ideas presented in [17] to profile applications more accurately. The `async-profiler` employs a hybrid approach for its stack trace collection based on i) `AsyncGetCallTrace` to obtain information on a thread's Java frames, without the requirement to be at a safepoint; and ii) `perf` to provide information on native code and kernel frames. The `perf_event` API of the Linux kernel is used to configure stack samples to be placed into a memory buffer, and a signal is delivered when a sample has been taken. Its signal handler then calls `AsyncGetCallTrace` to capture the Java stack that it then merges with the `perf` stack trace of native and kernel information.

The main benefits of this approach over a purely `perf` approach, such as that used by `perf-map-agent` is that it can be used to profile applications running on older Java releases that do not support the `-XX:+PreserveFramepointer`[1] flag that is necessary for the stack walking of `perf` for Java thread stacks. Java Mission Control, by Oracle, uses a similar approach to `async-profiler`.

Our approach relies on `perf` to record kernel, native and Java call-stacks and `perf-map-agent` to record the maps of code addresses to Java methods. Both `perf` and the `async-profiler` can be configured to record call-stacks based on hardware and software performance counters supported by `perf_events` such as: last-level cache/data TLB load misses, L1-dcache misses, branch misses, page faults, and instructions retired. The main benefit of the `perf/perf-map-agent` approach over `async-profiler` are:-

- `-XX:+DebugNonSafePoints` enables the code addresses of inlined methods to be mapped, in a way that makes it possible to identify inlining decisions in call-stacks.
- We use the symbol table dumped by `perf-map-agent` in conjunction with additional information from Graal JIT compilation of Truffle APIs to map call-stack frames to their guest language function names as explained in Section 4 and outlined in Figure 5.

[1] The overhead of running with `-XX:+PreserveFramePointer` was less than 4% for all DaCapo benchmarks during preliminary experiments.

- As our approach does not use JVMTI and we do not inject any bytecode for the purposes of profiling, then any effects on performance and behaviour are minimised to the effects of dumping additional symbol table information, sampling and frame pointer preservation overhead.

Note that non-root processes can capture call-stacks using `perf_events` if `/proc/sys/kernel/perf_event_paranoid` is set to 1 and `/proc/sys/kernel/kptr_restrict` is set to 0.

2.1 Challenges for Sampling Profilers

We focus our discussion on Intel processors, but the approaches outlined in this paper are applicable to any processor that supports Linux `perf` and `perf_events`, including ARMv7 and ARMv8 processor implementations. The main challenges are i), determining an appropriate sample rate frequency that leads to call-stack sampling that is sufficiently accurate, and ii), sampling skid, although this is highly dependent on the processor implementation and the specific event counter used to initiate sampling.

Accuracy — The accuracy of the collected profile is proportional to the sampling frequency. If the sampling frequency is too low then samples may be relatively sparse, and potentially could fail to accurately represent the proportion of time, or the hardware events associated with critical methods. If the sampling frequency is too high, then the level of overhead due to collecting and recording call-stacks will become significant and the execution time behaviour of the application will be significantly perturbed. The overhead of sampling using Intel *Precise Events Based Sampling* (PEBS) is evaluated in [1].

Sampling Skid — This occurs, when a reported instruction address of a sample does not precisely represent the actual instruction address causing a call-stack sampling interrupt. This is because, determining the exact instruction that causes a sampling interrupt to occur is complicated by instruction level parallelism and out of order execution models where many instructions are in flight at any time instant. Skid may lead to an incorrect method, or source code line to be identified as causing a performance bottleneck. The tool `perf` performs low-latency sampling on an Intel processor if sampling occurs on a PEBS event. Note that the subset of events supporting precise reporting varies by processor model. Typically samples are collected until a buffer becomes full, then userspace is notified to process the buffer. In our experiments we use `perf` with default settings for PEBS on an Intel processor.

3 FLAMEGRAPH VISUALISATIONS

Traditional flat profiles generated from the `prof` utility present how much time a program spends in each function, and the number of times that function was called. It indicates which functions burn most of the cycles, but it does not give the calling context, and it does not, for example, lead developers towards the causes of high GC pause latencies and poor application performance. Flat profiles make it difficult to disambiguate an expensive time-consuming call-stack from non-expensive ones when the same function is invoked from multiple locations. This is true even if call graph view of `prof` is used. Usage of `prof` is undesirable because a program and its libraries need to be compiled with profiling and debugging support which may significantly affect execution time.

Influential work by Brendan Gregg [11] has demonstrated that flamegraphs, (see examples in Figures 2, 3), can help to visually highlight the significant stack traces in a profile. Stack traces having identical callers are merged, then any non-identical child callee nodes in the collected stack traces appear as a new control flow path. The benefits of this approach are that call-stack contexts leading to the execution of *hot* methods become clearly visible. Control flow paths are visualised by presenting the unique names of methods inside rectangular blocks that are typically organised in lexicographical order from left to right. Divergence in a control flow path is indicated by more than one rectangular block being stacked at one level.

For example, consider the simple C-like pseudo-code fragment present in Figure 1 where a `main` method is entered that calls `initialise` and returns, then `compute` is called that also calls `evaluate` and returns to `main` before finally calling the `output` method that returns and the program exits. For this simple example, we would expect that sampling the call-stack of the program during its execution would result in the generation of a flamegraph similar to that in Figure 2. Different colours can be easily assigned to the rectangular blocks in the flamegraph in order to convey meaning about the type of code that a name inside a block refers to — such as JIT compiled Java, native, or kernel code. Figure 1 also includes the folded call-stack used to generate the flamegraph of Figure 2. The simple text format makes it easy to manipulate, and to identify function names and class hierarchies that can appear in the symbol names that appear in call-stack samples. Tools such as `c++filt` can be used to demangle C++ symbol names.

It is important to note that in a flamegraph, width denotes relative importance, as it is proportional to how many samples match. For Figure 2 we interpret this as the vast majority of the execution time being spent in the `evaluate` method, and an approximately

```
void evaluate() {   /* do something expensive */ }
void initialise() { /* initialise data */     }
void compute()  {    evaluate();              }
void output()   {   /* output results */      }
int main()  {
        initialise();
        compute();
        output();
        return 0;
}

// Folded call-stack
main; initialise 20
main; compute; evaluate 60
main; output 20
```

Figure 1: Simple C-like Pseudo-code and example folded call-stack used to generate the flamegraph in Figure 2. The folded call-stack indicates the sampled code-paths and the aggregated total of matching call-stack samples. This indicates that approximately 60%, 20% and 20% of execution time is spent in `evaluate`, `initialise` and `output`, respectively.

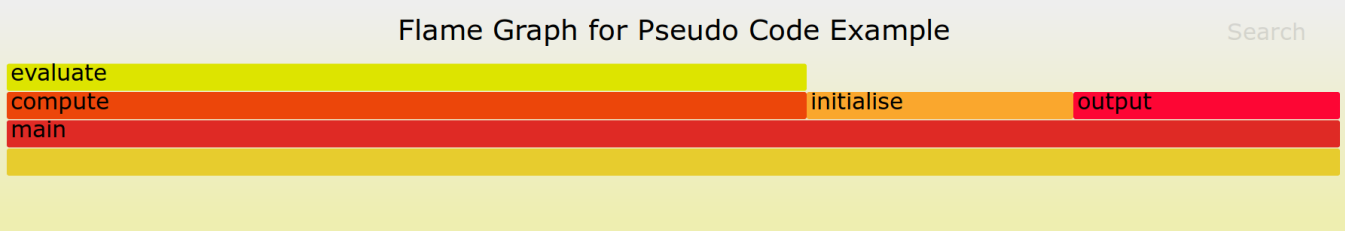

Figure 2: A simple artificial flamegraph for the pseudocode in Figure 1

Figure 3: Flamegraph of the DaCapo-9.12-bach avrora benchmark processor stack traces using *perf-map-agent* with `perf` 100Hz sampling. Yellow is C++ code, green is JIT compiled Java code, orange is kernel code, red is native/library, or it is marked as Interpreter. Purple indicates a searched for function that accounts for 28.9% of total samples (see bottom right hand side). Note that the selected function only accounts for 9.02% in one stack. Broken stacks appear as *unknown*. Inlined methods appear as teal/blue. Requires the use of `-XX:+PreserveFramePointer` and `-XX:+DebugNonSafepoints`.

equal (but lesser) amount of time being spent in `initialise` and `output`. Height in a flamegraph merely indicates the depth of the call-stack, that can be limited or controlled by the tool used to collect the stack traces or by system wide limits. For example, on Linux this limit can be increased beyond 127 for `perf` using `/proc/sys/kernel/perf_event_max_stack`.

Call-stack traces can be related to time, or to changes in a microarchitecture counter such as last level cache misses that are accessed by `perf_events`. In this way, it is possible to produce flamegraphs to characterise the microarchitectural behaviour of a running application. Scalable Vector Graphic (SVG) visualisations of flamegraphs can be generated that include JavaScript support for analysis within a web-browser. Searching for a method in a web-browser rendered SVG will highlight each matching method instance in the flamegraph call-stacks, and the bottom right-hand corner of the web-browser window indicates the total and percentage appearances of the method in all call-stack samples. Therefore, the search feature makes it easy to evaluate, and to highlight the relative performance costs of methods having multiple call sites (see Figure 3).

The interpretation of flamegraphs when multiple threads are present needs some consideration. Remember that for sequential single threaded execution of a native application, the width of a specific call-stack column is proportional to the number of matching call-stack samples, that in turn is proportional to the wall-clock time, as long as Dynamic Voltage and Frequency Scaling (DVFS),

e.g., Intel Turboboost, is disabled, and a userspace governor is used to hold the voltage and frequency to a constant value. However, when multiple threads execute the same code in parallel, then typically the number of matching call-stacks will increase, and then the width is proportional to the total execution time of all threads, rather than the wall-clock time because some threads will have overlapped execution on separate processors. Exposing the thread-id can be useful in a flamegraph, as load imbalance in different threads is visualised as different call-stack widths. Note that for the Truffle framework and JVM based execution, even if an application is sequential, multiple threads are created to execute JVM services, such as JIT compilation, GC, and VM operations.

Figure 3 shows a processor sampled flamegraph for the DaCapo-9.12-bach avrora benchmark with 100Hz sampling. This type of flamegraph enables inlining decisions to be examined visually. Teal/blue indicates JIT inlined methods, the methods are identified correctly as long as any sampling skid [5] does not cause an incorrect method to be attributed. Sampling skid (see Section 2.1) occurs when the *program counter (PC)* value reported by sampling is different to the actual PC value in execution when the sampling interrupt was raised. Note the flamegraph columns where a blue rectangle/method is directly below the green, which indicates a call site where the JIT compiled and inlined code has called a method that it has chosen not to inline. It is a fairly trivial matter to generate flamegraphs where thread PIDs are preserved, this can sometimes be beneficial as mentioned previously, but can quickly become confusing if many threads exist in a thread pool of an application. However, as the call-stacks reported by `perf` are textual then it is relatively trivial to filter out uninteresting information using standard text processing tools. This is clearly evident as many of the scripts for the production of SVG flamegraphs are written in `Perl`. The generated flamegraphs displayed in this paper are on-processor graphs, because they only demonstrate stacks for threads in execution when call-stack sampling occurred. They do not represent the importance of off-processor time when threads are blocked.

3.1 The Truffle Framework

Truffle is a language implementation framework for creating AST interpreters that execute on top of the JVM [25]. Graal is a JVMCI hosted compiler that enables Truffle AST interpreters to be JIT compiled and self-optimised by specialising nodes in the AST based on the observed datatypes of the variables of an application. By using this approach, Truffle framework languages referred to as *guest languages*, hereafter, benefit from the JVM infrastructure that has evolved over many years. Truffle also provides optimisations such as polymorphic inline caching that especially helps to improve the performance of dynamically typed languages. Interpreted execution generates profiling information, that is consumed by Graal to aid JIT compilation and further aggressive optimisations [9] such as *partial evaluation* [24], and inlining of AST nodes in order to improve performance.

3.2 Challenges for Profiling Guest Languages

Unfortunately, profiling the execution of a guest language application results in call-stacks that contain the Truffle framework method code addresses relating to `OptimizedCallTarget::callRoot`. A separate object instance of `OptimizedCallTarget` is created to bind each guest language function to a Java method for execution, and an instance variable of type `RootNode` is used to represent the actual AST of the guest language method that contains the textual name of the guest source code method. Therefore, the names of the source language functions are not available in the recorded call-stacks, as each JIT compiled guest language method appear as an instance of `OptimizedCallTarget::callRoot` having a unique code cache address. We dump the guest language function name (recorded in `RootNode` instance variable) and address of the JIT compiled `OptimizedCallTarget::callRoot` method for it, in the code cache, using the `-Dgraal.TraceTruffleCompilation=true` flag. This mapping is then used to associate the guest language function names to the stack frames of the associated `OptimizedCallTarget::callRoot` methods in the flamegraph.

4 EXPERIMENTAL METHODOLOGY

This section describes the methodology used to execute benchmarks on selected language implementations, and how we use sampling based profiling.

Warmed-up Application Execution of Java and guest language application implementations is used, where the computationally intensive and frequently executed methods of an application have already been compiled, and applications achieve stable performance. Note, that in general, it is difficult to determine when an application reaches a warmed-up state [4], so we use the methodology previously presented in [19] for the shootout benchmarks and Sulong. A harness is used to execute a benchmark `N=100` times, that wraps execution of the main method of the original code inside a loop, without exiting the Java process. The execution time of each of the last 50 invocations of the benchmark is measured. The threshold for Graal JIT compilation method invocation count is set to 1000. This is sufficient to ensure that no JIT compilation activity occurs during the last 50 invocations, we additionally calculate the geometric mean of execution time across the 10 benchmarks.

Sampling Profiling workflow is outlined in Figure 5. A map of guest language AST names to Java symbols is generated, we enable `-XX:+PreserveFramePointer` during our experiments in order to minimise the number of stack samples having broken or unknown stacks. Broken stacks occur when it is not possible to unwind the call-stack due to inconsistent stack state, and the omission of a frame-pointer in native code, or in Java JIT compiled code. Unknown call-stack values occur when a symbol is not available/found for an address within a call-stack. For example, debug symbols need to be installed for OpenJDK using a command such as `apt-get install openjdk-8-dbg`, whereas these are shipped as standard with Oracle HotSpot. We use *perf-map-agent* to generate a map of Java JIT compiled method names to code-cache addresses. The map is placed in `/tmp/perf-map-PID.map` where PID is the process identifier of the application under evaluation. Perf is run as `perf -F frequency -g -p PID`, in order to sample the desired process PID at a specific `frequency`, with call-stack recording `-g` enabled. We process the perf collected call-stacks with our own script that relates guest language method names to `OptimizedCallTarget::callRoot` JIT compilation symbols produced by Graal, and then standard scripts [10, 20] are used

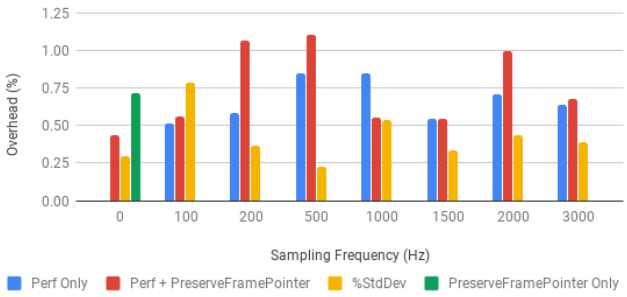

Figure 4: Sampling overhead for executing benchmarks from the shootout benchmark suite on Sulong using different sampling frequencies. The blue bars represent the overhead for sampling using perf, and the green bar shows overhead when benchmarks executed with the `-XX:+PreserveFramePointer` JVM flag only. The red bars show the overhead of using both the perf utility and `-XX:+PreserveFramePointer` JVM flag. The yellow bars represent the standard deviation as a percentage of the geomean execution time, and it is calculated using the 51..100 invocations of the benchmark in our test harness. Overhead is calculated by measuring percentage increase in geometric mean execution time during execution with a selected profiling configuration, relative to the geometric mean execution time without profiling. It can be seen that the overhead is less than 1.25%.

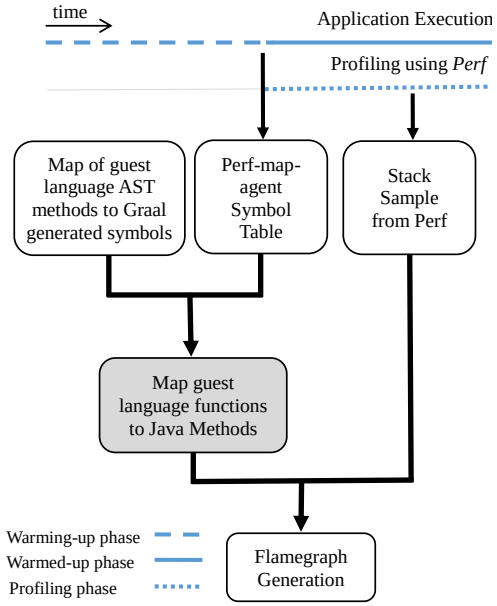

Figure 5: Modified workflow to generate flamegraphs using `perf` and *perf-map-agent*. We use the flag `-Dgraal.TraceTruffleCompilation=true` to output additional information to associate guest language function names with Java methods in combination with the JIT compilation log.

to produce flamegraphs. The current limitations of our approach to dumping symbol table information when deoptimisation and dynamic JIT recompilation occur are presented in Section 8.

Sampling overhead: profiling activity inevitably incurs an overhead during execution of the application. Overhead is calculated as the increase in execution time when using `perf` enabled profiling relative to the execution time without profiling. `Perf` pauses process execution whilst collecting a call-stack sample. Thus, at a higher sampling frequency, a process pauses more often and this results in increased execution time. Figure 4 shows the incurred overhead when the benchmarks from shootout benchmark suite are executed on Sulong at different sampling frequencies. It can be seen that `perf` incurs less than 1% sampling overhead relative to the original execution time of the warmed-up benchmarks while using `perf` along with `-XX:+PreserveFramePointer` JVM flag incurs overhead that is less than 1.25% on our benchmarks.

Profiling Polyglot Applications: it is possible to write Polyglot applications using different Truffle hosted languages. When such a program is visualised using a flamegraph, it is difficult to distinguish between the methods from different language implementation. We have modified the flamegraph generation script to use a new colour palette that distinguishes methods from different Truffle framework languages using separate colours. Figure 6 show a flamegraph for a Polyglot program execution where Python is coloured in brown, JavaScript in blue, R in white and the Graal compiler methods in dark green colour. We believe that this is a very useful feature for identifying the significant routines and call-stack contexts of Polyglot applications.

5 EVALUATION

5.1 Goal

Our objective is to provide tools that easily identify where time is spent during the execution of guest language applications and Polyglot applications. Identifying the significant sources of overhead, and whether this relates to JVM service overheads (JIT compilation and garbage collection) or specific guest language methods, will enable performance gaps between different language implementations to be better understood. In this evaluation, we execute the same set of benchmarks on different Truffle framework language implementations including Sulong for LLVM IR, FastR for R and TruffleRuby for Ruby [23, 8, 19], these languages were chosen based primarily on the availability of source code implementations for our benchmarks.

We also execute the Java and native C version of the benchmarks to be able to compare against the execution of Sulong. Thus, we can understand the amount of performance traded off by running on a JVM. Sulong enables LLVM IR, produced by compiling C with LLVM compilation tools, to be executed as a Truffle framework language. The direct comparison of Java with native execution gives an approximate lower bound on the performance gap that could be achieved by a guest language implementation. We also compare the

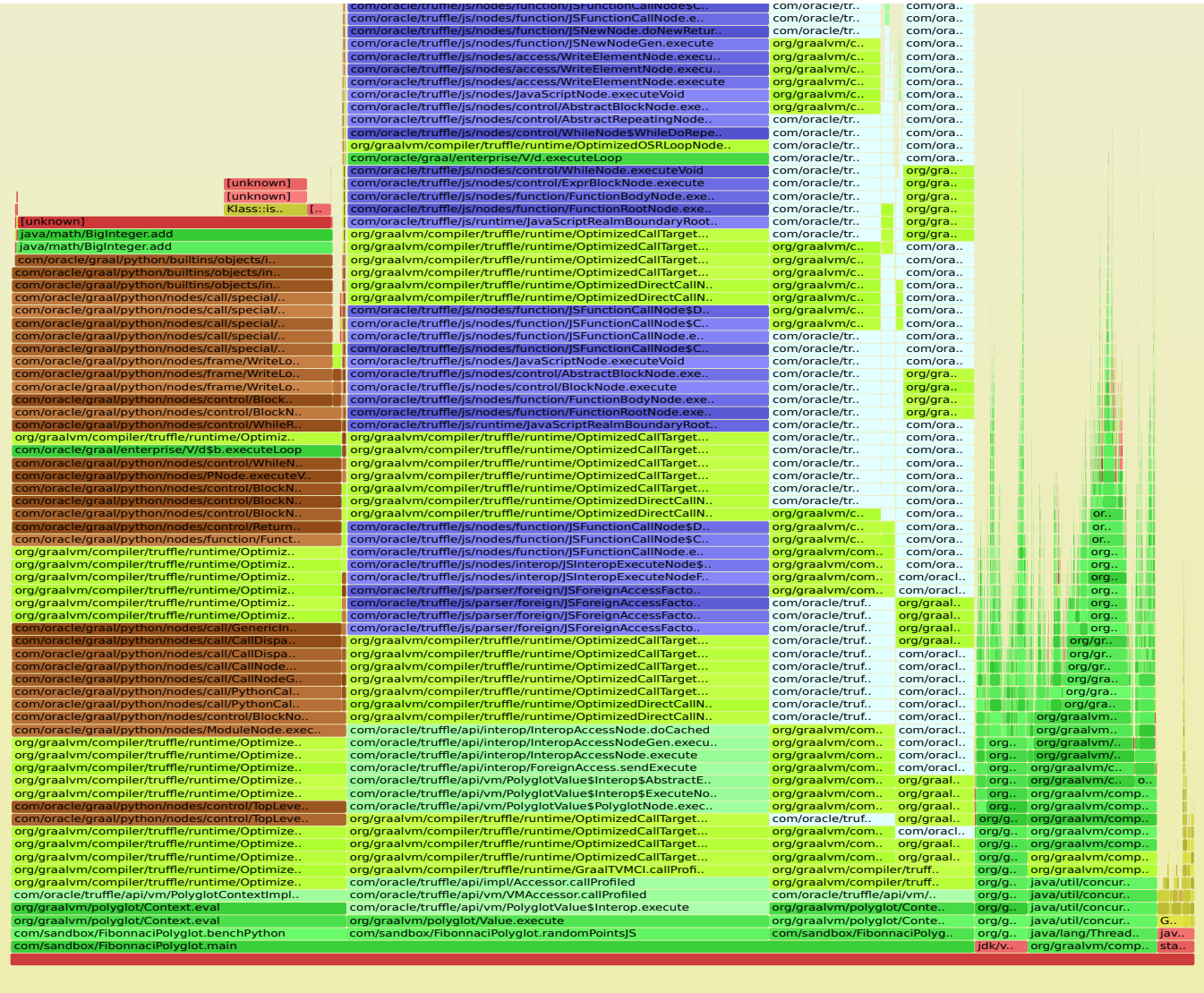

Figure 6: A flamegraph visualisation of a profile for a Polyglot program executing small synthetic computations, where fibonacci, and array manipulations are computed using different languages bundled with `graalvm-0.31`. Here, we demonstrate that with `async-profiler`, we can identify interpreted and JIT-compiled methods. We can still identify the different languages (due to different Truffle API names in call-stacks), coloured as GraalPython (brown), TruffleJS (blue), FastR (white), Truffle framework API (dark green) and Graal (lime green). Integration of our techniques to map Truffle AST source functions to Java methods with `async-profiler` (currently ongoing) will enable us to identify the specific guest language functions in a Polyglot profile.

performance of different guest language implementations against each other.

5.2 Benchmarks

Our experiments use the shootout benchmarks[14] from the *Computer Language Benchmark Game* (CLBG). The meteor benchmark is not included as it failed to execute correctly on the version of Sulong that is evaluated. We deliberately chose sequential benchmarks to control better the experiments. However, the underlying JVM used to execute the guest languages will create multiple threads for GC, JIT compilation, and various other VM services. Further, additional application threads may be created by objects as a result of using the standard JDK class library.

The shootout suite is selected because it is commonly used to compare the performance of different language implementations. Note, that in order to keep the benchmark implementation similar across the languages, we use the simple and not the optimised implementation for the CLBG binarytree benchmark, consequently

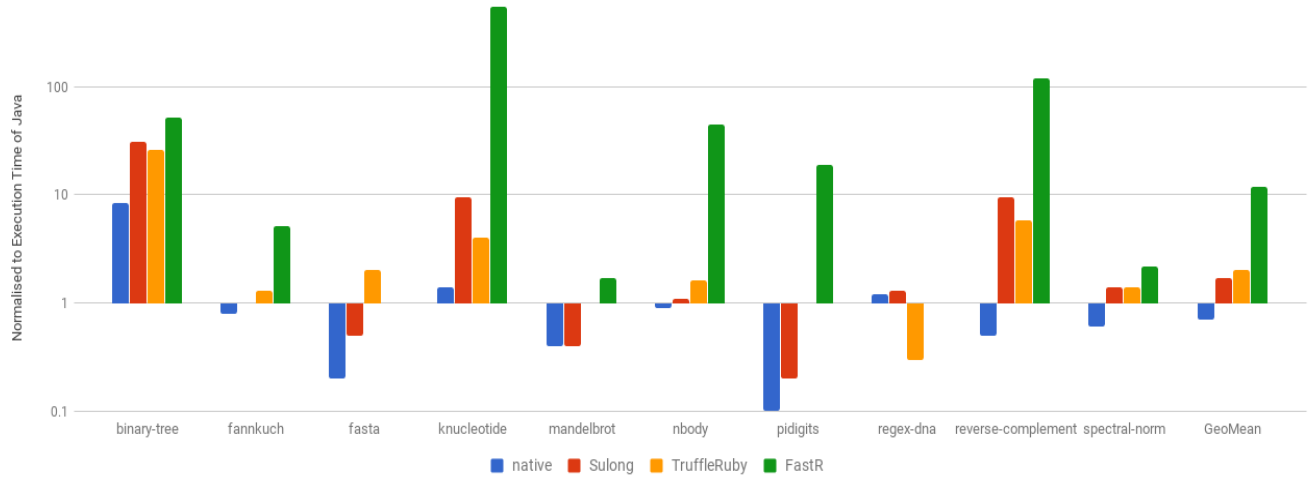

Figure 7: Logarithmic execution time of CLBG shootout suite benchmarks normalised to Java execution time. Results for native, Sulong, Java, TruffleRuby and FastR implementations are presented. 1 represents the execution time required for Java implementations, values less than 1 represents faster execution than Java (this means lower is better).

the malloc and free overhead is significant and the native version is slower than Java. Note, also we do not present results for FastR for the regexdna benchmark as we discovered a bug with the implementation of the FastR library function `gsub()` (see [6]) that means incorrect results are calculated.

5.3 Experimental Setup

We use a system with 2 physical (4 hyper-threaded) cores Intel Core i7-3537U with 8GB of memory running Ubuntu 16.04 (4.13.0-38-generic). The processor frequency scaling is disabled and set it to a fixed 1.0GHz frequency using the *userspace* governor. We use Sulong commit *b779587* that supports LLVM 5.0, TruffleRuby commit *c15ae86*, FastR commit *c49d3b7* and GraalVM version vm-enterprise-0.33. The LLVM IR for the Sulong based execution is generated using *clang*, an LLVM frontend for C programs, with `-O3` optimisation flag. Figure 8 shows setup for execution each of the selected implementations.

6 RESULTS

Figure 7 shows the execution time of the selected benchmarks normalised to the Java execution time and plotted on a Y-axis logarithmic scale. It can be seen that the different guest language implementations demonstrate significant variations in performance compared to the execution of Java even though they all use the same underlying Truffle framework executing on the same JVM. The variations in performance lead us to question whether the language semantics make it difficult to convey information to the compiler or if there are performance weaknesses in aspects of some guest language implementations.

In Figure 7, results for binarytree benchmark show that the execution on Java is much faster even than native C execution. Note, we use the naïve implementation of the benchmarks to make the

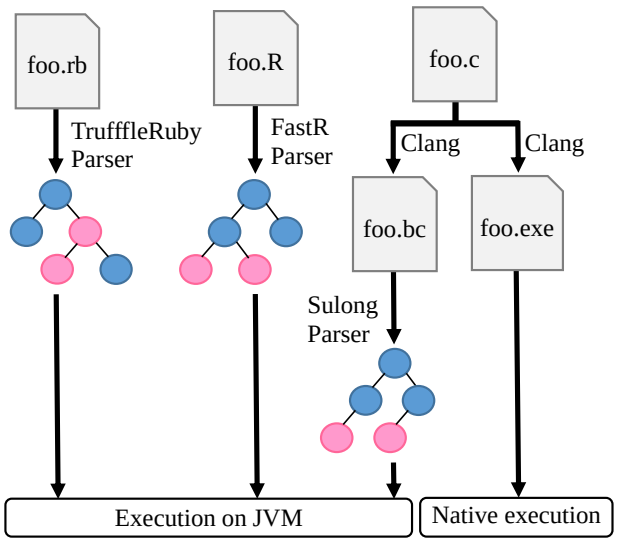

Figure 8: Different modes of execution where the Truffle hosted language implementations such as TruffleRuby, FastR, Sulong converts input program to Truffle AST that is then executed on a JVM; for the native execution a source program is converted to a binary using `clang`.

different language implementations as similar as possible. In the case of the binarytrees benchmark, the native and hence, the Sulong version (derived from the native LLVM IR) performed dynamic memory management as part of the benchmark. Figure 9 and 10 shows that the native implementation spends about 84.4% of its execution time in functions doing the memory allocation or free

while the Java implementation of binarytrees spends only 6.9%. Additionally, garbage collection is performed by 2 separate threads in parallel and thus, it contributed much less relative wall-clock execution time than what is seen in the profile. If we compare the raw execution time then the native implementation is 7.2 times slower than Java but when we subtract memory management overhead then the execution times are comparable.

Both k-nucleotide and regexdna use the external library glib-2.0 for creating a hash table and PCRE for matching regular expressions. In both cases, the execution of native and therefore the Sulong implementation is slower than the Java implementation. In the case of regexdna benchmark, the Java implementation uses `java.util.regex` while TruffleRuby uses the Java port of Oniguruma library to match regular expressions which dominates the computation in the benchmark that is also reflected in their performance. In the case of Sulong, the slowdown is much higher because it must use the Native Function Interface (NFI) to call native library functions from Java. This slowdown is proportional to the time spent in calls made using NFI. A search in the corresponding SVG flamegraphs for keyword *NFI* shows that for k-nucleotide 89.1% samples match the keyword while for regexdna it is just 8.3%. We do not include results for executing the regexdna benchmark for FastR due to the bug we identified[6]. Results also show that the FastR implementation is consistently slower, having a geomean of about 10.1, compared to equivalent Java programs.

6.1 Case Study: Analysis of the k-nucleotide slowdown on FastR

The k-nucleotide benchmark on FastR is more than 500x slower than Java. Also, the warmed-up execution of FastR is 35% slower than the GNU R v3.2.3. After examining the flamegraph we observe that FastR spends about half of its execution time 46.2% in the `FrameWithoutBoxing.resize` method as shown in Figure 11. The `resize` function is called from the `REnvironment.put` method that provides a `HashMap` update functionality in the R benchmark source.

In the case of running k-nucleotide on Java, the majority of the execution time was spent on `HashMap` update and array allocation operations. 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 a new *key* is inserted into the *Environment* variable, the `resize` operation is called yet the size of the data structure providing `HashMap` like functionality is only incremented by 1. The `HashMap` implementation of Oracle doubles the size of the underlying data structure when it resizes the HashMap, this is more efficient than FastR, as explained below. Currently, the hash map functionality in FastR is provided using the `FrameWithoutBoxing` class and its `resize` operation that requires allocating and copying 3 member arrays of the data structure that results in expensive operations. The benchmark inserts keys throughout its execution and increments the size of the array providing `HashMap` functionality by 1 each time, rather than doubling its size, severely increasing overhead in comparison to Java. Note, 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. Further, it backs up our expectation that Truffle runtime library classes, may be used in ways that are inefficient for Truffle framework language implementations. Thus, the implementation of *Environments* in FastR can be improved significantly, potentially by mapping directly onto Java `HashMap`, or 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 our approach.

In this section, we showed that the techniques and tools presented in this paper could be used for performance analysis of the Truffle hosted guest languages.

7 RELATED WORK

Marr *et al.* provided a benchmark suite that contains 14 benchmarks implemented in six programming languages including Java, JavaScript, Ruby, Crystal, Newspeak, Smalltalk [16]. Their main objective was to identify performance improvement opportunities by comparing different language implementations to optimise their compilers and runtime systems. Clearly, our work is similar, except that we present modified tools that enable the performance of different Truffle guest language implementations to be compared against native C and Java. Sarimbekov *et al.* [21] used CLBG benchmarks to characterise the workloads of JVM hosted languages that included Java, Scala, Clojure, Jython, JRuby, and JavaScript. In a complementing study, Li *et al.* [15] performed an exploratory study characterising workloads for JVM hosted languages that included Java, Scala, Clojure, JRuby, and Jython, using both CLBG and real-world application benchmarks. Both papers use characterisations to identify differences between Java and non-Java workloads that can be used to guide development efforts for improved performance. In comparison to these studies, our objective is more specific. We aim to understand the execution behaviour, and the performance of different Truffle hosted language implementations. It was expected that a smaller performance gap would be present than what the experimental results found.

Our approach to sampling based profiling uses a language agnostic tool based on `perf` that is external to the JVM and immune to the potential inaccuracies in measurements incurred by JVM based profiling approaches due to safepoint bias, and/or instrumentation overheads. For example, Savrun-Yeniçeri *et al.* [22] implemented an instrumentation based profiling framework for Truffle languages. The language implementation is modified to generate a Truffle AST of a program that inserts profiling nodes before events of interests such as function enter-exit, condition node. As the profiling code is part of the AST, it is also a candidate for JIT compilation that reduces the profiling overhead. Additionally, when profiling is not enabled the profiling code gets eliminated as dead-code during JIT compilation. This approach provides zero overhead when profiling is disabled. The overhead of this technique ranges from 4% to 45% for control flow and type distribution profiling. Note, that as mentioned previously, inserting additional code for profiling can change optimisation decisions made by the JIT compiler, therefore the profiled code will not be the same as the original code. Also, this profiling approach requires modification of the language implementation, to ensure that profiling nodes are inserted into the AST.

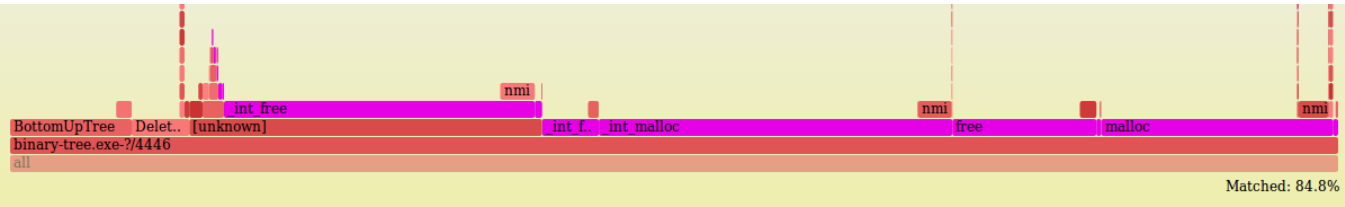

Figure 9: The flamegraph shows a profile of executing the binarytree program natively. In the flamegraph, call-stacks for the functions that allocate and free memory are highlighted in magenta colour and it shows that 84.4% time of the native execution is spent to perform memory allocation and free.

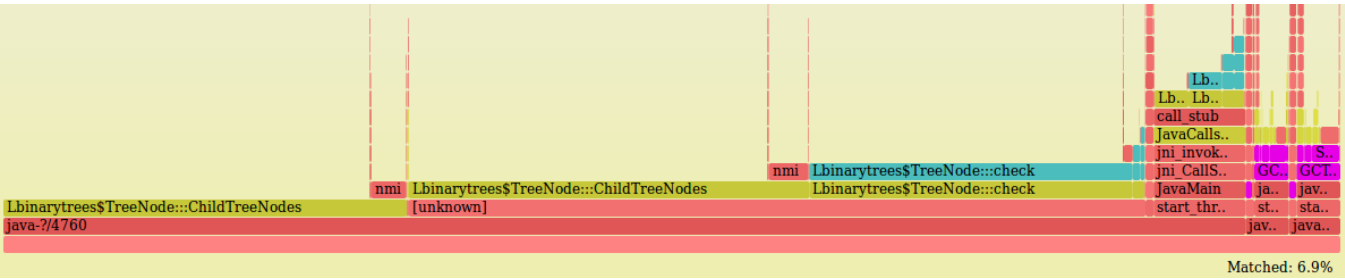

Figure 10: The flamegraph shows a profile of executing a Java program for the binarytree benchmark. In the flamegraph, call-stacks for GC related functions are highlighted in magenta colour. Here, 6.9% time is spent in GC and that is executed by 2 GC threads represented by two separate towers (named by their thread ids that are not visiable).

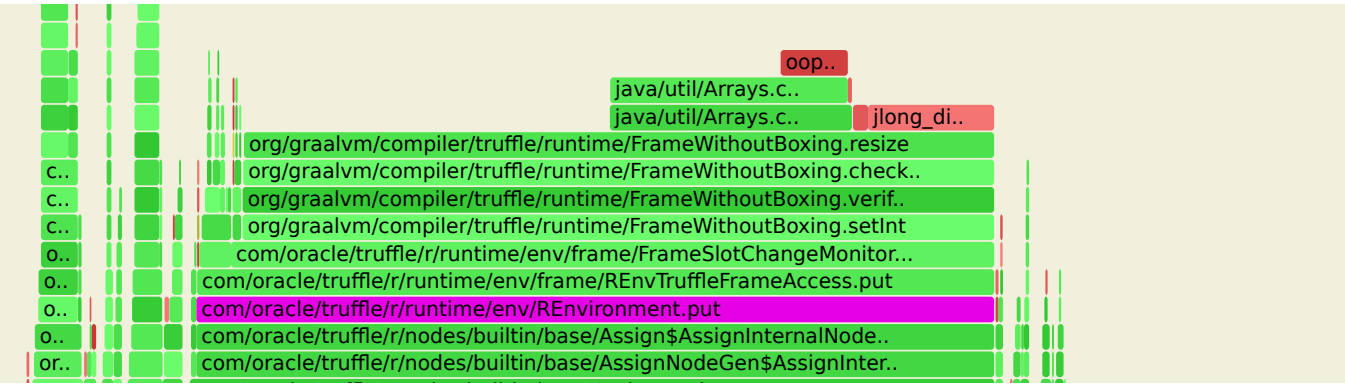

Figure 11: The flamegraph shows a part of a profile where about 46.2% time is spent when the k-nucleotide program executed on FastR. The *REnvironment.put* function, highlighted in magenta colour, implements `assign` function for *Environments* in R that is equivalent to the insert operation on a HashMap.

Our proposed analysis tools require no modification to the language implementation, although we incur the overhead of running with `-XX:+PreserveFramePointer`, we found this to be less than 4% on DaCapo-9.12-bach. The `perf` sampled call-stacks of our approach can be easily processed to visually highlight call-stacks matching specific JDK classes and methods using a user-defined colour palette, which we can use to disambiguate portions of call-stacks related to different language implementations. Further, if the SVG itself is searched then it is possible to determine the percentage of call-stack samples matching a specific method, or JDK class.

8 LIMITATIONS

We have presented tools and techniques that aim to identify bottlenecks via graphical analysis of application performance using flamegraphs and standard Unix tools where possible. The current implementation is preliminary, at a proof-of-concept level, with the following limitations that need to be addressed.

Limitations of perf: first, it requires that the application start with the `-XX:+PreserveFramePointer` JVM flag so that the `perf` utility can walk the call-stacks of JIT compiled methods. Second, Java interpreter call-stacks are only traced as *Interpreter*, thus the

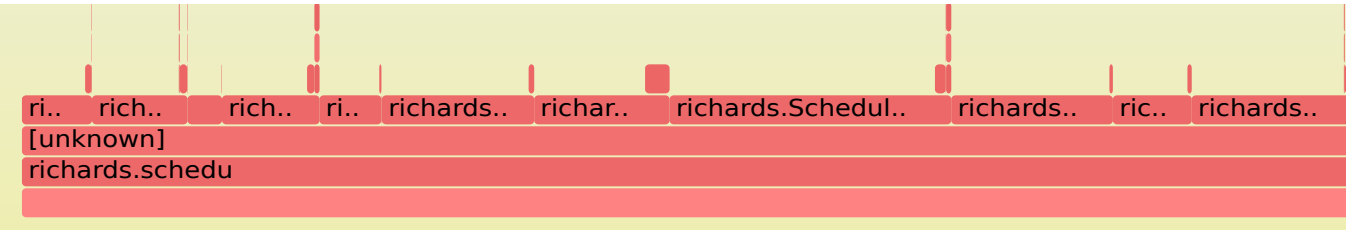

Figure 12: A profile for executing an AOT compiled binary generated using the Substrate VM for the Richards benchmark written in Java.

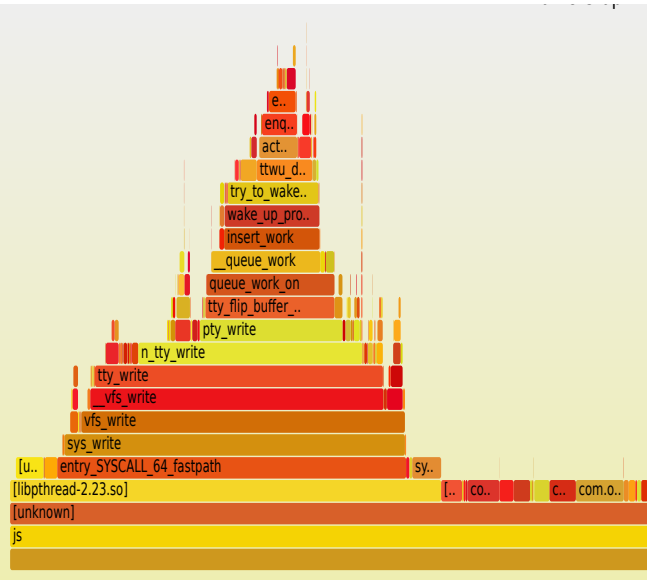

Figure 13: Native application using `libpolyglot` to access C and Truffle JavaScript together. The C application executes a JavaScript code that prints the square root of the first million integers from 1. This profile shows that the application spent 37.41% of the total execution time while printing the output on the standard output.

same is reflected in the corresponding call-stacks in a flamegraph. Occasionally the stacks are broken and are represented as *unknown* stack frames, this is even true for `async-profiler`. Presence of such unknown stack frames in some call-stack samples that are actually identical will lead to multiple call-stack towers in the flamegraph.

Interpreted methods: our current approach does not map names for guest language methods that are interpreted. Since our primary objective is to identify the performance bottlenecks, we focus only on methods that are identified as hot, and thus JIT compiled by a JVM. However, ongoing work on integration of our techniques with `async-profiler` could provide a mechanism to walk interpreter call-stacks using `AsyncGetCallTrace`, and then to relate interpreted Truffle AST methods to guest language methods using our mappings.

Limitations of perf-map-agent: handling deoptimisation and JIT recompilation: `perf-map-agent` dumps a snapshot of the symbol table for the *entrant* JIT compiled methods in a JVM when the agent is attached. It does not consider methods that were JIT compiled earlier but are no longer entrant and the methods that later get JIT compiled or deoptimised. Note, that perf already provides a framework for managing dynamically changing symbol tables, as used by the JavaScript V8 engine. In future work, we will solve this problem by offline processing of timestamped perf stack-samples with JIT recompilation and deoptimisation information that can be logged using i), JVMTI callbacks within a modified perf-map-agent, and ii), extending JVM functionality under the control of flags.

AOT compiled methods: Substrate VM makes it possible to create an ahead-of-time (AOT) compiled binary using Graal. The binary then runs as a standalone executable without a JVM. Figure 12 demonstrates that our approach can generate a flamegraph of a profile for executing an AOT compiled binary of the Richards benchmark [18] that was originally written in Java.

The Truffle hosted language implementations can be called by native applications using the Polyglot library. Figure 13 shows a flamegraph of a profile generated using our techniques for such an example where a native C application executes a JavaScript program that computes and prints a square root for the first million integers starting from 1.

Since we use `perf` to sample the application, our approach should work for AOT compiled binaries and libraries as well. To map the guest language function names to their call-stacks in the flamegraph, we need a mapping between the function names and their code locations. An AOT compiled binary has fixed locations for the guest language functions. Therefore, it should be feasible to modify the AOT generation sequence of steps to dump the necessary mapping of guest language methods to symbol table addresses.

9 CONCLUSIONS AND FUTURE WORK

We have presented low overhead (less than 1.25%) techniques using standard Linux tools for profiling GraalVM and Truffle-hosted language implementations. The approach requires no modification to the Truffle language implementation, unlike previous profiling approaches. Note, that Truffle framework language implementations such as TruffleRuby and FastR have demonstrated that they are faster than existing alternative implementations whilst the JavaScript implementation with Truffle has shown that it is on a

reasonably level performance setting with the V8 JavaScript implementation by Google. The approach presented in this paper is expected to aid identification of further potential directions for performance improvements, as our results suggest that FastR is consistently slower than Java.

The precise reasons behind the performance differences across different language implementations of the same benchmark, perhaps due to different ways of expressing the same computation, or due to specific aspects in the Truffle framework language implementations are the subject of future work. For example, the FastR implementation has the poorest performance of all the language implementations studied. We need to study more realistic R use-cases in order to better understand any issues, and to determine if the CBLG shootout benchmarks are suitable for making cross-language performance comparisons.

Currently, we do not provide support for recognising AOT compiled execution of Truffle hosted languages and they are represented as any other native application. However, it is expected that future work could add to the AOT compilation infrastructure (in a similar manner to what we have done for Graal) to relate guest language names to symbol addresses, and even to identify inlining decisions. Such information could prove to be very useful for optimisation.

ACKNOWLEDGMENTS

We would like to thank Dr. Chris Seaton at Oracle Labs for his help and comments on early drafts of this paper, and to the Sulong project team in general, especially to Manuel Rigger.

REFERENCES

[1] Soramichi Akiyama and Takahiro Hirofuchi. 2017. Quantitative evaluation of Intel PEBS overhead for online system-noise analysis. *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017* (ROSS '17), 3:1–3:8. DOI: 10.1145/3095770.3095773.

[2] Andre Pangin et al. Async-profiler git repository. (2016). https://github.com/jvm-profiling-tools/async-profiler.

[3] Richard Warburton et al. Honest Profiler. https://github.com/jvm-profiling-tools/honest-profiler. (2014).

[4] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang.*, 1, OOPSLA, (Oct. 2017), 52:1–52:27. DOI: 10.1145/3133876.

[5] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. 2010. Taming hardware event samples for FDO compilation. *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (CGO), 42–52. DOI: 10.1145/1772954.1772963.

[6] Oracle Corporation. Bug reported on GitHub of FastR. https://github.com/oracle/fastr/issues/15. (2015).

[7] Oracle Corporation. Graal VM. (2015). https://github.com/graalvm.

[8] Oracle Corporation. TruffleRuby. https://github.com/oracle/truffleruby. (2016).

[9] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An intermediate representation for speculative optimizations in a dynamic compiler. *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages* (VMIL), 1–10. DOI: 10.1145/2542142.2542143.

[10] Brendan Gregg. Flamegraph git repository. (2011). https://github.com/brendangregg/FlameGraph.

[11] Brendan Gregg. 2016. The flame graph. *Queue*, 14, 2, 10:91–10:110. DOI: 10.1145/2927299.2927301.

[12] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Mikel Luján. 2018. Cross-language interoperability in a multi-language runtime. *ACM Transactions on Programming Languages and Systems*, 40, 2, 8:1–8:43. DOI: 10.1145/3201898.

[13] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-performance cross-language interoperability in a multi-language runtime. *Proceedings of the 11th Symposium on Dynamic Languages* (DLS), 78–90. DOI: 10.1145/2816707.2816714.

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

[15] Wing Hang Li, David R. White, and Jeremy Singer. 2013. JVM-hosted languages: they talk the talk, but do they walk the walk? *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (PPPJ), 101–112. DOI: 10.1145/2500828.2500838.

[16] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-language compiler benchmarking: are we fast yet? *Proceedings of the 12th Symposium on Dynamic Languages* (DLS 2016), 120–131. DOI: 10.1145/2989225.2989232.

[17] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2010. Evaluating the accuracy of Java profilers. *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), 187–197. DOI: 10.1145/1806596.1806618.

[18] Martin Richards. Richards benchmark. (1999). http://www.cl.cam.ac.uk/~mr10/Bench.html.

[19] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing low-level languages to the JVM: efficient execution of LLVM IR on Truffle. *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages* (VMIL), 6–15. DOI: 10.1145/2998415.2998416.

[20] Johannes Rudolf and et al. Perf-map-agent git repository. (2012). https://github.com/jvm-profiling-tools/perf-map-agent.

[21] Aibek Sarimbekov, Andrej Podzimek, Lubomir Bulej, Yudi Zheng, Nathan Ricci, and Walter Binder. 2013. Characteristics of dynamic JVM languages. *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages* (VMIL), 11–20. DOI: 10.1145/2542142.2542144.

[22] Gülfem Savrun-Yeniçeri, Michael L. Van de Vanter, Per Larsen, Stefan Brunthaler, and Michael Franz. 2015. An efficient and generic event-based profiler framework for dynamic languages. *Proceedings of the Principles and Practices of Programming on The Java Platform* (PPPJ), 102–112. DOI: 10.1145/2807426.2807435.

[23] Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. 2016. Optimizing r language execution via aggressive speculation. *Proceedings of the 12th Symposium on Dynamic Languages* (DLS). Amsterdam, Netherlands, 84–95. DOI: 10.1145/2989225.2989236.

[24] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical partial evaluation for high-performance dynamic language runtimes. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), 662–676. DOI: 10.1145/3062341.3062381.

[25] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Onward!), 187–204. DOI: 10.1145/2509578.2509581.

[26] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. *Proceedings of the 8th Symposium on Dynamic Languages* (DLS), 73–82. DOI: 10.1145/2384577.2384587.