A GraalVM-hosted bytecode interpreter for MonNom

Xinjie Xu
Australian National University

INTRODUCTION

Virtual Machines (VMs) run user's codes using the Just-In-Time (JIT) compilation technique that emits platform code in runtime to execute the program. However, VMs are black boxes – the user usually doesn't know how the code will be emitted through the VMs' JIT-ting pipeline. Moreover, different VMs usually emit different platform codes that perform differently.

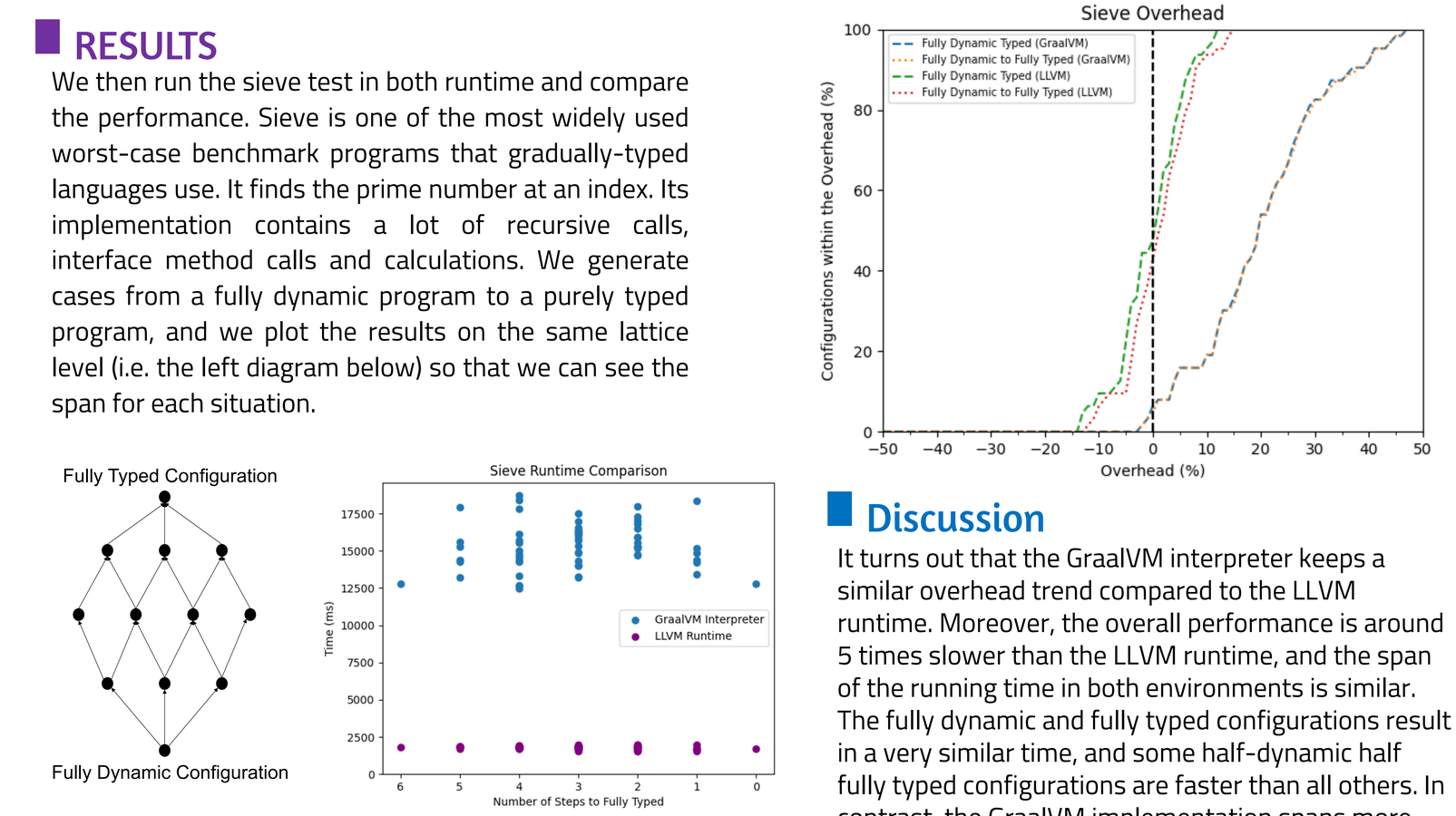
This project is the implementation of a subset of the current MonNom (a gradually typed programming language written by Dr Fabian Muehlboeck) runtime in GraalVM. It is designed to be capable of processing most of the non-generic versions of the MonNom features (i.e. objects, control flow, dynamic dispatch, etc.). Moreover, this project aims to explore the difference in the performance of the runtime for the programming language MonNom implemented in LLVM and GraalVM. Moreover, MonNom is a gradually-typed language, the performance of the program from fully dynamic type annotated to fully type annotated would be an interesting perspective to discover as well.

METHODOLOGY

The project reads the MonNom bytecode generated by its compiler and then converts the bytecode into AST nodes. Furthermore, the project applies Tail Call Elimination and some Back Copy Propagation to optimise the performance just like the LLVM runtime.

RESULTS

We then run the sieve test in both runtime and compare the performance. Sieve is one of the most widely used worst-case benchmark programs that gradually-typed languages use. It finds the prime number at an index. Its implementation contains a lot of recursive calls, interface method calls and calculations. We generate cases from a fully dynamic program to a purely typed program, and we plot the results on the same lattice level (i.e. the left diagram below) so that we can see the span for each situation.



Discussion

It turns out that the GraalVM interpreter keeps a similar overhead trend compared to the LLVM runtime. Moreover, the overall performance is around 5 times slower than the LLVM runtime, and the span of the running time in both environments is similar. The fully dynamic and fully typed configurations result in a very similar time, and some half-dynamic half fully typed configurations are faster than all others. In contrast, the GraalVM implementation spans more than the LLVM implementation, which causes the overhead of the program to be up to 50% from the baseline (whereas the LLVM one is just up to 15%). However, this is still in the range of -50% to 50% overhead, rather than multiple times of overhead compared to other gradually-typed languages.

The reason the GraalVM implementation is slower is currently identified as a huge amount of allocation (boxing/unboxing) in Java. Another assumption is the current project did not utilise GraalVM's features (i.e. NativeImage, StaticShape, etc.).