RIJKSUNIVERSITEIT GRONINGEN

Java in Java using Truffle

by

B.J.Grooteman

A thesis submitted in partial fulfillment for the
degree of BSc in Computing Science

August 2017

RIJKSUNIVERSITEIT GRONINGEN

Abstract

Science and Engineering Faculty
Computing Science

by B.J.Grooteman

In this paper the usefulness of an interpreter for the Java language as a replacement for an Object Algebra interpreter is researched. This Object Algebra interpreter is currently used in the Recaf project. The Truffle framework and Graal Virtual Machine are used to eliminate the low level concerns that usually come with building an interpreter. The language that is supported on this interpreter is a subset of Java. The comparison is done by running several benchmarks on both platforms and comparing the runtimes. From the results can be seen that running Recaf on the GraalVM already increases the performance. When we run the same benchmarks on the Truffle language we achieve better performance on all the benchmarks, which might indicate that this interpreter would be a good substitute for the current Recaf interpreter.

Contents

Chapter 1

Introduction

Nowadays there is an abundance of programming languages, each with it's own special features. But with all these programming languages also comes a new syntax and a new environment which can be a hastle. In 2016 A. Biboudis, P. Inostroza and T. van der Storm published a paper on Recaf [1], a lightweight tool that makes it possible to extend Java with new language constructs. The general approach of this tool is to translate the code into an Object Algebra and ran on an interpreter, which makes it easier to manipulate. A downside to this approach is the loss of performance. This bachelor project aims to test if a replacement for this interpreter written in Truffle would be able to achieve better performance.

1.1 Goals

The main goal of this project was to create a Java-like language that runs on the GraalVM using Truffle. The language had to be a subset of Java that was able to do the basic operations and have controlflow but without the class-support. The main questions that had to be answered were *Can we replace the backend of Recaf with an interpreter written in Recaf* and *What is the effect on performance?*.

1.2 Related work

As for Java interpreters I could not find any papers / projects on that. There are papers on implementing other languages in Truffle, those are listed below:

- Truffle C [2]
- TruffleRuby [3]
- FastR [4]

TruffleRuby, FastR and TruffleJs are implementations that are written by the creator of Truffle, these all seem to achieve better performance in some benchmarks than their native implementation. TruffleC is an academic paper on running C on Truffle without significant speedloss. I also found some hobby projects that implemented Lisp (Mumbler) and C++ (Cover) on Truffle, it is intresting to note that both of those implementations report that they did not get the performance they hoped for.

Chapter 2

Design

2.1 Truffle

Truffle is a framework for implementing languages as simple interpreters. It was introduced in 2012 by Thomas Ürthinger et al. in their paper on Self-Optimizing AST Interpreters [5]. In this paper they present Truffle as a framework that makes it easy to implement languages as interpreters, without losing a lot of performance and with the added benefit of ready-to-use language tools, such as debuggers and profilers, and better language interoperability. Truffle achieves this by running on it's own custom VM, the GraalVM, and by modifying the Abstract Syntax Tree at runtime to incorporate type feedback. Truffle also abstracts a lot of the low level concerns away, which making implementing a language a task of defining the different AST nodes, building the AST from source code, and feeding it to Truffle.

2.2 JeSSEL

JeSSEL stands for Java SubSet Language, and is the name of the programming language that was written on Truffle for this project. It has the same basic syntax as Java but lacks some of it's features. JeSSEL runs completely on the GraalVM. JeSSEL is a statically typed language.

2.2.1 Features

Each of the features is accompanied by a small code snippet that shows how this feature can be used in JeSSEL. Each piece of code is enclosed by a class, this is mandatory because of the use of JavaParser, which refuses to parse the file when no class is specified.

- Java-Syntax: JavaParser is used as the parser, normal Java syntax can be used.

```
1  public class Hello {
2    void main() {
3      println("Hello, world!");
4    }
5  }
```

LISTING 2.1: Hello world in JeSSEL

- The primitive types (int, float, long, double, char), including String as a primitive type.

```
1  public class Hello {
2    void main() {
3      boolean bool = true;
4      int intt = 1;
5      float floot = 2.0;
6      long loong = 3;
7      double dooble = 4.0;
8      char c = '5';
9      String str = "Six";
10   }
11 }
```

LISTING 2.2: All supported types in JeSSEL

- If and While statements, together with continue and break statements.

```
1  public class Hello {
2    void main() {
3      boolean b = true;
4      int five = 5;
5      if(b) {
6        five = 1;
7      }
8      while (five != 5) {
9        five++;
10     }
11   }
12 }
```

LISTING 2.3: Controlflow in JeSSEL

- Arrays, of all primitive types.

```
1  public class Hello {
2    void main() {
3      int[] firstArr = new int[10];
4      firstArr[0] = 4;
5      long[] secondArr = {firstArr[0],
6        firstArr[0],firstArr[0]};
7      int result = secondArr[2];
8    }
9  }
```

Listing 2.4: Arrays in JeSSEL

- All binary and unary operations, including the bitwise operations.

- Functions, that can accept parameters, and return values.

```
1  public class Hello {
2    int add(int a, int b) {
3      return a+b;
4    }
5
6    void main() {
7      int a = 4;
8      int b = 5;
9      int result = add(a,b);
10     println(result);
11   }
12 }
```

Listing 2.5: Functions in JeSSEL

2.2.2 Limitations

JeSSEL is currently not a full substitute for Java. Due to time constraints and Truffle limitations, some features of Java can not be used in JeSSEL.

- **No for-loop, no do-while, no conditional expressions.** Since these are each constructs that do not really add extra features to the language. Their functionality can be mimicked with while loops and if statements.
- **No Classes/Objects.** Classes were outside of the scope of the project, and since those were not supported the decision to drop Objects as well was made.
- **Arrays can not be passed to, or returned from functions.** This limitation was imposed by myself due to the fact that I could not get it work properly.
- **No modules/imports.** This limitation is because of Truffle. No good way of using imports (yet).

The use of JavaParser also makes it so that a class has to be defined around the code, otherwise the file is not recognized as a proper Java file and JavaParser declines to parse it.

2.2.3 Grammar

Below is my best attempt at a grammar of JeSSEL.

⟨*class_decl*⟩ ::= `class` ⟨*identifier*⟩ ⟨*set_innerdecl*⟩.

⟨*set_innerdecl*⟩ ::= `{` [⟨*methoddecl*⟩]* `}`.

⟨*methoddecl*⟩ ::= ⟨*returnedtype*⟩ ⟨*identifier*⟩ `(` [⟨*paramlist*⟩] `)` ⟨*block*⟩.

⟨*returntype*⟩ ::= `void` | ⟨*type*⟩

⟨*field_decl*⟩ ::= ⟨*type*⟩ ⟨*identifier*⟩ [`=` ⟨*field_init*⟩] `;`.

⟨*field_init*⟩ ::= ⟨*expression*⟩ | `{` [⟨*field_init*⟩* [,]] `}`.

⟨*block*⟩ ::= `{` ⟨*statement-list*⟩ `}`.

⟨*stat-list*⟩ ::= ⟨*statement*⟩ `;` ⟨*stat-list*⟩ | ⟨*statement*⟩ .

⟨*statement*⟩ ::= ⟨*expression*⟩ `;`
| `if` `(` ⟨*expression*⟩ `)` `{` ⟨*statement*⟩ `}` [`else` `{` ⟨*statement*⟩ `}`]
| `while` `(` ⟨*expression*⟩ `)` `{` ⟨*statement*⟩ `}`
| ⟨*field_decl*⟩
| `{` ⟨*stat-list*⟩ `}`
| ⟨*empty*⟩.

⟨*expression*⟩ ::= ⟨*literal*⟩ | ⟨*expression*⟩ ⟨*infix*⟩ ⟨*expression*⟩ | ⟨*prefix*⟩ ⟨*expression*⟩.

⟨*infix*⟩ ::= `+` | `-` | `*` | `/` | `%` | `^` | `&` | `|` | `&&` | `||` | `<<` | `>>` | `=` | `<` | `>` | `<=` | `>=` | `==` | `!=`.

⟨*prefix*⟩ ::= `++` | `--` | `-` | `~` | `!` .

⟨*literal*⟩ ::= ⟨*boolean*⟩ | ⟨*identifier*⟩ | ⟨*number*⟩ | ⟨*string*⟩.

⟨*type*⟩ ::= `int` | `long` | `float` | `double` | `boolean` | `String` | `char`.

Boolean, Identifier, Number and String are just the Java implementations of each. Number being a long or a float. Arrays are also not in this grammar, since they are not fully supported.

Chapter 3

Implementation

To implement a language in Truffle the nodes that will make up the Abstract Syntax Tree (AST) have to be specified. An AST built of these nodes then needs to be constructed from inputted source code. This AST then can be passed to the Polyglot engine, this will use the nodes to execute the code. This Polyglot engine is totally abstracted away and provided by Truffle, and handles all the low level implementation.

3.1 The Pipeline

Before source code is ran in JeSSEL it goes through a couple of steps. In the figure below can this pipeline be seen.

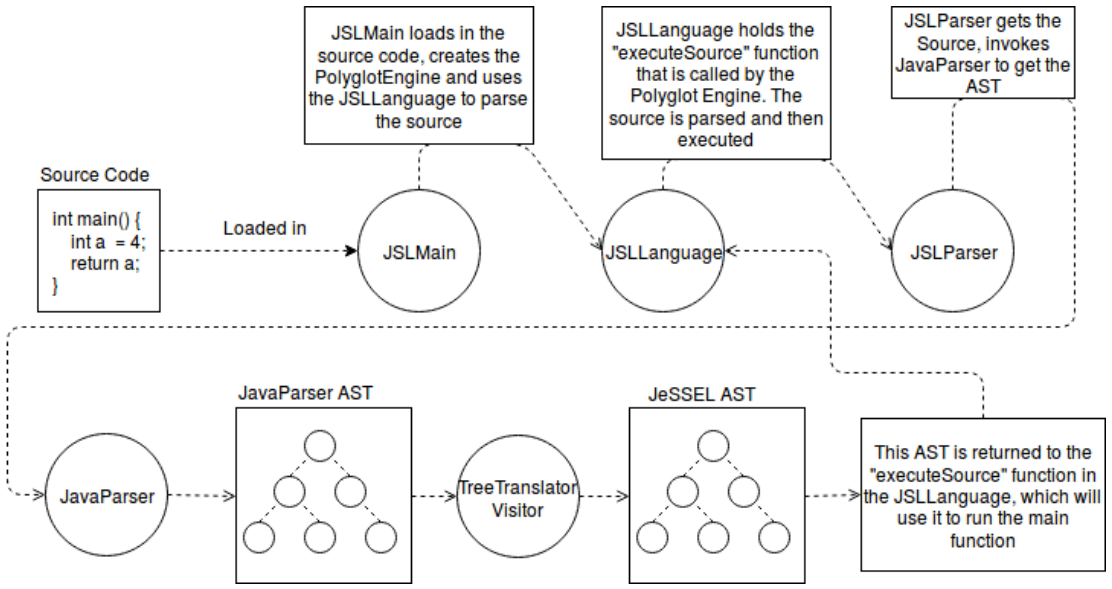

FIGURE 3.1: Pipeline

When a JeSSEL program is executed, the source code is loaded into JeSSEL by the main class (`JSLMain`). `JSLMain` also creates the PolyglotEngine and registers the `JSLLanguage` in it. JSLLanguage is a singleton class that is used to execute the source as a JeSSEL source. This is designed in such a way that multiple languages could be registered to the PolyglotEngine and could all be ran from the same command (by loading a source into the JSLMain class). The PolyglotEngine decides based on the MIMEType of the source which language needs to be executed and invokes the `executeSource` function of that language. This function needs to create a `CallTarget` and return that to the Engine. A CallTarget is the root of the AST from which execution should begin. In `JSLLanguage` this is done by redirecting the Source to the JSLParser. Here the source gets parsed by the JavaParser and turned into a AST (JPAST). This AST is buildup from JavaParser Nodes while we need them to be buildup from JeSSEL nodes. The translation is done by the `TreeTranslatorVisitor` which uses the Visitor pattern to walk over the JSAST. Since we want to generate expressions as well as statements, therefore keeping track of this in the visitor is not desirable. The generation of the new JeSSEL AST is thus done by the `NodeFactory`, who has two stacks to keep track of the statements and expressions. When the visitor has visited the whole tree, we have a Map that contains all functions in the source, each with their own AST's. The `JSLLanguage` uses this map to extract the main function from this map, and uses it to build a CallTarget, which is then passed to the Engine and executed.

3.2 Types

Truffle is a framework built for building interpreters, mostly focused on dynamically typed languages. Since we want to use it for a statically typed (Java-like) language we need to work around this.
To keep track of the types of expressions, we introduced an extra `class` which all the expressionNodes implement. This `class`, called `JSLTypedExpressionNode`, imposes the constraint that all children should implement a `getType()` function. This means that we know each Expression in our language has a type, which can then be used do typechecks during the translation process. By doing so we lose some of the advertised property of Truffle (the self-specializing AST that uses type-feedback to do so) but make sure the language is typed (as Java would be). Another solution would be to only do type checking during parsing and after that letting Truffle do the optimizing. A downside to this is that it would result in having to write guards that show which nodes should be specialized, which would take a lot of time, and clutter up the code. This approach was discarded due to the fact that every node would need code to differentiate between the different types at runtime, which would lead to a lot of duplicate code.

3.3 JeSSEL in Truffle

JeSSEL makes use of Truffle in multiple ways, here I will show some examples and explain why they are useful.

3.3.1 Annotations

Truffle has multiple annotations that are used in JeSSEL. These are pointers for the underlaying framework that something needs to be done with the function or class. Some of the most used are explained here:
@Specialization: This is an annotation that can be put above a function. It tells Truffle that the function is a specialization of an operation. These specializations can have guards which will then be used at runtime to determine which of the specializations should be used. In Dynamic languages this is quite a useful feature since it makes it so you can define multiple functions for the same operation with different inputs. In JeSSEL we have types which make this annotation less useful, and while it is possible to still write guards that distinguish between different types of input, this will get messy quick. On a mailing list I found a conversation about this problem, and the people from Truffle advised to split the operations in different files and just differentiate between them during parsing. Link to that mailing list: http://mail.openjdk.java.net/pipermail/graal-dev/2016-July/004472.html So that is how it is implemented in JeSSEL.
@ExplodeLoop: This is an annotation that can be matched with a function that contains a loop with a fixed length. When this is the case the `ExplodeLoop` can be used to tell the compiler to roll the loop out at compilation. One of the examples is in the BlockNodes, which consists of a list of bodyNodes that needs to be ran. We can tell the compiler to get rid of the loop and unroll the elements.

```
1   @ExplodeLoop
2   @Override
3   public void executeVoid(VirtualFrame frame) {
4     for(JSLStatementNode statement : bodyNodes) {
5       statement.executeVoid(frame);
6     }
7   }
```

Listing 3.1: ExplodeLoop annotation in JSLBlockNode

@TruffleBoundary: This is an annotation that can be matched with a function. It tells Truffle that the function should never be compiled. This can be put at functions that are a lot of code but are not used very often. You do not want these functions cluttering the compiled code (which would make is slow), we always want it to be handled by the interpreter at runtime.

Usage of the Annotations in JeSSEL

In Table 3.1 can every usage of these Annotations be found. This gives an overview of

where the Truffle compiler will (try to) optimize the code.

In addition to the @Specialization annotations mentioned in Table 3.1, this annotation

Table 3.1: Input Values of each benchmark

Annotation	**Where**
@Specialization	5x in JSLPrintlnBuiltin 1x in JSLTimeBuiltin 3x in DispatchNode
@ExplodeLoop	1x in InvokeNode 1x in BlockNode
@TruffleBoundary	5x in JSLPrintlnBuiltin 1x in JSLTimeBuiltin

is also used in *every* ExpressionNode in the `expression` package (34x) and in *every* read and write node in the `local` package (25x). These, however, do not really add any functionality since we differentiate between those nodes during parsing but are just there because Truffle requires them to be there.

3.3.2 Profiling

Truffle provides Profiles which are basically counters. These are especially useful in controlflow statements. The idea is that Truffle keeps track of how often a branch is taken, and optimizes itself based on the assumption that if we took the if-branch nine times in a row, the tenth time will have a large chance to be the if-branch again. Behind the scenes Truffle rearranges it's code to be able to execute the expected branch faster.

```
1  public final class JSLIfNode extends JSLStatementNode {
2    @Child private JSLExpressionNode conditionNode;
3    private final ConditionProfile condition =
4      ConditionProfile.createCountingProfile();
5
6    @Override
7    public void executeVoid(VirtualFrame frame) {
8      if(condition.profile(evaluateCondition(frame))) {
9        doIf();
10     } else {
11       doElse();
12     }
13 }
```

Listing 3.2: Minimalized example of a profile

This is also a nice example of a Truffle feature that directly helps to optimize the interpreter, would probably not be the first thing that is implemented when a interpreter is written from scratch, and is very easy to use.

3.3.3 Implementation of JeSSEL

By using Truffle it should be easier to write an interpreter. For JeSSEL, the final implementation totaled around 5100 lines of code. Table 3.2 show how these SLOC is distributed over the project. The parser package is the most SLOC, which is due to the

TABLE 3.2: SLOC per package

Package name	SLOC	Package Description
main	250	Top level package, holds the main classes
runtime	425	Holds the runtime Exceptions and the JSLContext, which is used to lookup values at runtime
parser	1541	Translates the JavaParser AST into a usable JeSSEL AST
builtin	98	Holds the implementation of the builtin nodes
node	554	The top level package for all AST nodes, holds the abstract JSLStatement and JSLTypedExpression classes that all the nodes use.
expression	863	This package holds all the nodes that represent expressions. Nodes for all binary expressions
local	881	Holds all the nodes that are used to create, write and read variables in the local scope.
controlflow	337	All the controlflow statement nodes can be found in this package, the
interop	74	Code for interoperability between languages that run on the GraalVM.
call	140	Holds the invoke and dispatch nodes that handle the function calls
Total	5163	

fact that we transform the JavaParser way in a peculiar way. If a proper parser would be written, or the visitor pattern would be used to translate the JavaParser AST to the proper AST, this could be reduced a lot.
The full source code of JeSSEL can be found on GitLab
https://gitlab.com/BGrooteman/BachelorProject/tree/master/JeSSEL.

Chapter 4

Benchmark design

The cause for this project was, as stated before, to see if Truffle could provide a better performing backend to the Recaf project. Since the project time limited me to implement just the basics of the JeSSEL language, really comparing the performance of Recaf against Recaf on Truffle is not (yet) possible. We can however, see how the JeSSEL language performs in comparison with the Recaf interpreter. To do so, six benchmarks are ran both on Recaf and JeSSEL. In this chapter is explained how these benchmarks were set up.

4.1 Requirements

Getting good unbiased results from benchmarks has proven to be a task that requires some consideration. To make sure the performance is properly measured some requirements for the benchmarks were put in place.

- Reproducibility: The benchmarks should be set up in such a way that the results can be easily reproduced/verified.
- Uniformity: The benchmarks should be ran in the same environment, under the same conditions. The way of measuring time should be the same and independent of the language implementation.
- Accuracy: The time that is measured should be sufficiently accurate to distinguish the languages. Nanoseconds should be used.

As shown in the 'Virtual Machine Warmup Blows Hot and Cold' paper by Edd Barrett et al. [6] on benchmarking and the importance of properly warming up the VM before running the benchmarks is of significant importance. In the setup section is explained how these requirements are implemented in the actual benchmarks.

To test the performance of JeSSEL the benchmarks need to test several aspects of the language. We especially focus on micro benchmarks due to the fact that JeSSEL does not have classes/objects, nor any way of making modules of the code. This would make implementing a larger program/macro benchmarks challenging. The main aspects the benchmarks should focus on are:

- Basic operations
- Controlflow such as If and While statements

Based on these focus points six benchmarks were selected.

4.2 Benchmarks

The benchmarks that were ran are inspired by 'the Benchmark game'[7] and 'Are we fast yet'[8]. The other benchmarks I found on several places. All the code for the benchmarks (in .jsl files) can be found in Appendix A.

- **Greatest Common Divisor** Calculating the GCD between every number in a range(*r1*, *r2*) and a fixed number *k*.
  Inputs: r1, r2, k
- **Factorial** Calculating the factorial of *f*, *size* times. Implemented in an iterative fashion.
  Inputs: f, size.
- **PrimeFinder** This benchmark counts the number of prime numbers within a certain range up to *size*. It does this by for every number in the range performing a loop over [0, num/2] and testing if we can find a factor.
  Inputs: size
- **Fibonnaci** Calculation of the fibonacci sequence up to *f*, *n* times. Implemented in an iterative fashion.
  Inputs: f, n1, n2, size
- **FactorCounter** This benchmark counts the number of proper factors for every number in the range [0, size]. Does this by having a double nested loop with an if statement that checks if we found a proper factor.
  Inputs: size
- **While** A simple doubly nested while loop that does not really calculate something. To test how well the platforms handle big nested loops.
  Inputs: size

4.3 Setup

All the benchmarks are run on an ASUS laptop running Ubuntu 16.04, with 8 Cores at 2.00GHz and with 6GB of RAM.
I built a small environment that makes it easy to run the benchmarks. This environment is needed because the results need to be reproducible. As for the benchmarks, we want a structure where every new input is run from an empty cold VM, to ensure the uniformity requirement. Another reason for buildin the program is that doing this manually would become a lot of work/ a big inconvenience. The figure below shows how that is setup.

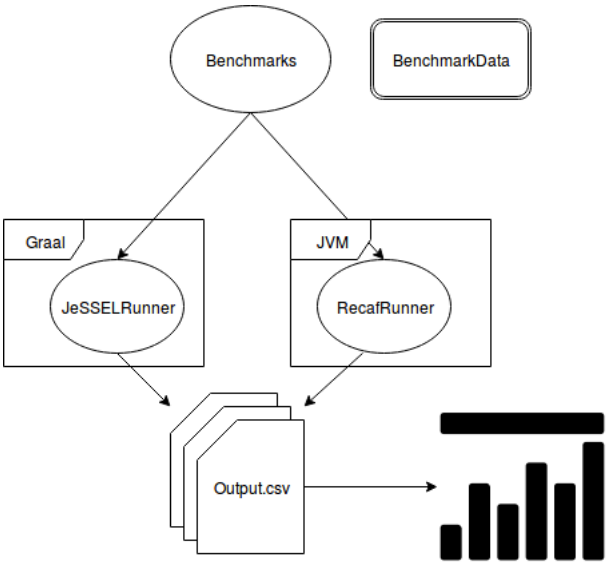

FIGURE 4.1: How the benchmark are setup

BenchmarkData This is a static class that holds all the benchmark data. It contains a list of Benchmark's which can then be used by the runners to get the run details for the benchmarks. Every benchmark extends the Benchmark class, which makes it easy to extract all the data in the runners.
Benchmarks This is the main class, from here the VM's are booted up for running. This happens for every different input of every benchmark. The Benchmark itself runs on the JVM and uses the ProcessBuilder to start new (cold) instances of the GraalVM and the JVM. Both the VM's are ran without any extra parameters:

```
java RecafBenchmarkRunner
```

```
./graal/java JesselBenchmarkRunner
```

Runners The runners have the language-specific code that runs on the VM. Firstly the runners parse the benchmark into a runnable AST, this parsing process should not be part of the benchmark. The runner then runs this parsed source 50 times to warm up the VM, these runs are also not part of the measurements. When the VM is warmed up we run the parsed source again 75 times, but now each run is timed with System.nanoTime(). The time for each run is recorded and after the measurements

written to a .csv file for later analysis.
The upside to this setup is that all the benchmark specific information, like the input values and number of repetitions are not intertwined with the code that actual runs the benchmarks. This makes it so that it is easy to change these values, add other benchmarks or verify results.
After all the benchmarks are run, the .csv files were to create tables and graphs from the results. Python with Matlibplot was used for creating the different plots and LateX table generator was used to create the tables. Three different plots are included, a scatter plot showing the time each run took, a barplot to compare the total runtime of both VM's and a graph that showed the relative speedup of JeSSel to Recaf.

4.3.1 Recaf

The whole goal of this project was to test if JeSSEL would be a better performing back-end than the current interpreter of Recaf. As shown above we designed benchmarks to test the performance. To make sure we run the same code on Recaf as we run on Truffle, a parser was written to parse the JeSSEL files into an Object Algebra that can run on the Recaf interpreter. By doing this it was ensured that the front-end of Recaf did not introduce shortcuts or workarounds, and we really test the performance of the two interpreters. This parser for Recaf was also written using JavaParser. For the benchmarks a version of Recaf was used that did not contain higher order abstract syntax (HOAS), but instead used an Environment that contained the variables. The result of the parsing process is an IExecEnv that can be called and has to be supplied a label and an empty Environment.

Limitations of Recaf
The benchmarks are written so that they can both be ran on JeSSEL as well as Recaf. Recaf did pose some extra limitations on the - No arrays
- Typesafe, but due to the fact that JeSSEL has (automatic) casting this caused problems. This means we can write 'double d = 1 + 2.4' in JeSSEL but not in Recaf.
- No real functions, and therefore also no way of passing arguments to the main function.
This last limitation posed some problems for the benchmarks, since there needed to be runvalues inserted into the program. I solved this by adding the initial values in the Environment before starting the actual execution. This way the variables could be accessed during execution.

Chapter 5

Results

5.1 Ran values

Table 5.1 shows the input values that correspond to the ran benchmarks. The picked values are not based on anything but were picked to most properly show different magnitudes of inputs.

TABLE 5.1: Input Values of each benchmark

	Input 1				**Input 2**				**Input 3**				**Input 4**			
FactorCounter	**size**				**size**				**size**				**size**			
	2500				5000				7500				10000			
Factorial	**size**	**f**			**size**	**f**			**size**	**f**			**size**	**f**		
	10000	20			10000	25			10000	30			10000	35		
Fibonacci	**size**	**f**	**n1**	**n2**	**size**	**f**	**n1**	**n2**	**size**	**f**	**n1**	**n2**	**size**	**f**	**n1**	**n2**
	10000	40	0	1	10000	80	0	1	10000	120	0	1	10000	160	0	1
GCD	**r1**	**r2**	**k**		**r1**	**r2**	**k**		**r1**	**r2**	**k**		**r1**	**r2**	**k**	
	1	1000	82		1	10000	683		1	50000	217		1	100000	6917	
PrimeFinder	**size**				**size**				**size**				**size**			
	1000				5000				10000				50000			
While	**size**				**size**				**size**				**size**			
	500				1000				2500				5000			

5.2 Benchmark results

The results that were written to the .csv are used to make plots and tables. Since they would break up the report too much if they would be shown in this section, they can be found in Appendix B, C and D. In Appendix B the tables containing the sum of each

benchmark, with their corresponding inputs, when run 75 times.
In Appendix C the corresponding plots can be found. Firstly the scatterplots containing the runtimes for each separate run of the benchmarks. This can show whether there is a lot of variance, can give insight in whether the VM was warmed up properly and give clues about whether the benchmark process was interrupted/skewed. Appendix D contains plots that put the total runtimes (sum of each benchmark) of JeSSEL and Recaf next to each other, to give a faster overview of the performance of both. I chose to compare the total runtimes instead of the average runtime for one run since it would make for very small numbers, and the numbers would be still in the same relation to eachother. The differences show better when magnified a bit, readers can see the variances in runtime also in the dedicated plots.

Table 5.2 shows a plot with relative speedups of JeSSEL over Recaf. These times are calculated by $\frac{t^{JeSSEL}}{t^{Recaf}}$. Where the times are the sums of the timed 75 runs for each benchmark (The fraction wouldn't change if we where to do it with the average runtimes since both times would be divided by the same factor).

TABLE 5.2: Relative Speedup of JeSSEL over Recaf. 1x is the Recaf runtime

	Input 1	Input 2	Input 3	Input 4
While	6789.6	11052.3	49245.9	105856.3
PrimeFinder	19.6	12.3	15.8	15.1
GCD	0.4	20.8	17.8	18.3
Fibonacci	121.4	130.2	130.3	121.3
Factorial	31.2	37.4	41.5	31.7
FactorCounter	18.6	19.0	18.5	19.0

Chapter 6

Conclusion

In this chapter I discuss the results that are presented in the last chapter, have a small discussion and talk about future work.

6.1 Performance

If we look at Figure 6.1, the relative speedup of JeSSEL with respect to Recaf, we see that JeSSEL outperforms Recaf on each benchmark quite clearly. In the case of the *While* and *Fibonacci*, a speedup of more than 100x was measured. The rest of the graph is not shown since it would make the other results not visible. The real speedups of each benchmark can be found in Table 5.2.

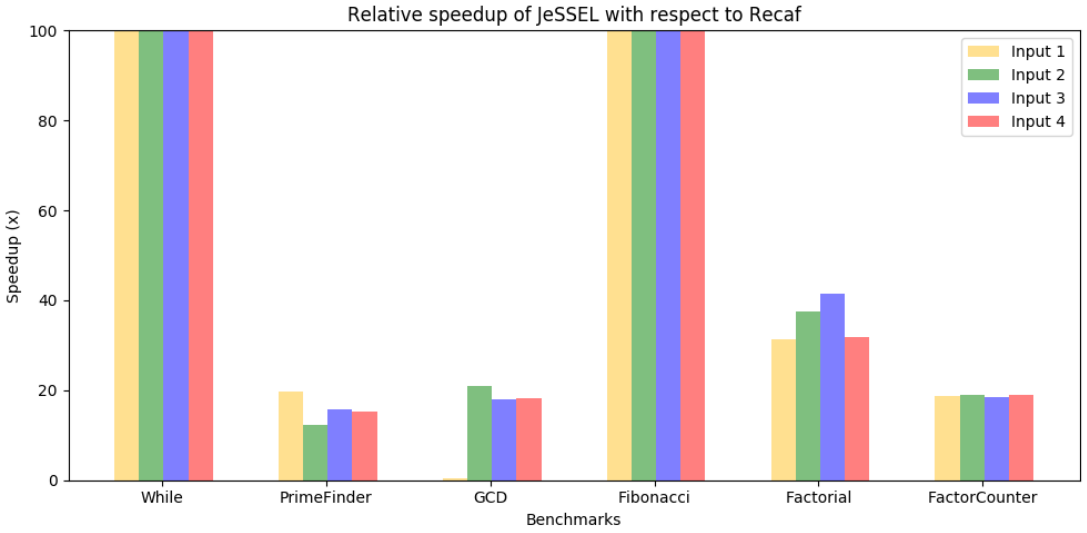

FIGURE 6.1: Relative speedup of JeSSEL with respect to Recaf (1x)

Hypothetical explanation:
If we look at the benchmarks, we see that these are the two benchmarks that have the most amounts of iterations to go through while they do not differ much from the other benchmarks in other aspects.
Three possible explanations for the performance would be

1. JeSSEL is fast due to profiles/optimizations
2. Recaf is slow due to costliness of loops
3. GraalVM is faster than JVM

JeSSEL is fast due to profiles/optimizations
To test whether the profiles caused the speedup, all benchmarks were run again, but this time with the *Profiles* removed from the code of the while loop. The effects were not significant, basically having no influence on the performance. Therefore we can eliminate the profiles as reason for the huge speed difference.
During the process it was suggested that the speedup was caused by the types that were used at runtime. By keeping track of the types, Truffle does not have to do tests during runtime, to find out which specialization to take (which would also take time). While this might play a role in a small part of the speedup, it can not explain why Truffle is so much faster on benchmarks with lots of loop iterations.
Another explanation for the speedup could be the annotations, which might cause Truffle to optimize certain parts of the code away, turn them into compiled code or do something else that might speedup the process. When looking at annotations and where we used them, as can be seen in Table 3.2, we seen that most annotations are used on classes that probably do not have a big influence on the performance. The only annotation that will probably have some real influence is the `@ExplodeLoop` in the JSLBlockNode class. It can be seen that the benchmarks that perform the best have a lot of iterations in their while loop, which contains no annotations. However Truffle has a dedicated `LoopNode` that is used to implement the while loop, but those optimizations are done behind the scenes.

Recaf is slow due to costliness of loops
The speedup we see could also be explained the fact that Recaf currently uses a slow implementation, in that case, it is not that any annotations, or smart optimizations were the cause of the difference in speed. JeSSEL performed the tasks reasonably, it was Recaf that ran them exceptionally slow. If this is the case, it is hard to pin down one implementation detail that led to JeSSEL being 20 times faster, since we can merely conclude that the Object Algebra implementation is inefficient. Since the performance of Recaf was the initial reason for this project, this is a plausible explain

GraalVM is faster than JVM
To test this the benchmarks were run Recaf on the GraalVM. In Table 6.1 can be seen that for almost all benchmarks and input, an speedup is achieved (1.0 would be equal performance). Only the first input of GCD seems to be a lot slower (but we also saw that that was the input on which JeSSEL was not faster. Strangely enough Recaf performs

very well on that input).

TABLE 6.1: Relative speedup of Recaf on Graal with respect to Recaf of JVM

RecafOnGraal	Input 1	Input 2	Input 3	Input 4
While	0.9502	1.2055	1.2369	1.2041
PrimeFinder	0.6325	1.8380	1.8352	1.7613
GCD	0.1988	1.4535	1.4115	1.4303
Fibonacci	1.3357	1.4744	1.3373	1.4081
Factorial	1.4015	1.3411	1.3891	1.3002
FactorCounter	2.5245	2.4126	2.1970	2.2140

In the end, we can think of several reasons for the speedup. To get real evidence of which one has a bigger influence on the runtime the VM's should be profiled. Currently there is lacking documentation on profiling the GraalVM which is why it is not included. The reasons being: *Behind the scenes optimizations by Truffle*, *GraalVM* and *Object Algebra is not very efficient*. What we can say for sure is that, by replacing the JVM with the GraalVM for Recaf, we can get almost twice the performance in most cases. To really speed it up, however, replacing the Object Algebra with JeSSEL can in most cases lead to a speedup of almost 20x.

6.2 Discussion

The results between the Recaf and JeSSEL benchmarks on While are more than a 1000 times faster, across all inputs. Such a huge speedup is not realistic, and might be due to some kind of caching, or hidden optimization. Disabling inline caching, removing the profiles and making sure the outcome is used (in the runner, so that it is not optimized away) all did not significantly change the outcome. Therefore it is hard to properly explain these results.

The scatter plots do not really accurately show the difference between the runs on the platforms. Because the time per run differed so much between Recaf and JeSSEL, putting them together in one plot also did not show accurately how the runtimes behaved. The problem with the current plots is that the y-axis is not the same between plots, therefore some variance might seem smaller/bigger than it actually is. The alternative would be to fix the axis for both plots, but this would mean we would have the same problem as having the runtimes in one plot together (The variance of the JeSSEL runs would be so small compared to the difference in between JeSSEL and Recaf that it would be neglected).

6.3 Future work

In the future, JeSSEL could be improved; to support more features, as well as improve the performance:

- **Add Arrays as function arguments and returns:** Implementing this would make it easier to run different benchmarks (like the 8-queens problem) and give the language more flexibility. However, Recaf currently does not really support arrays.
- **Run Recaf on JeSSEL:** Due to time constraints I did not have to get Recaf running on JeSSEL but the results of the benchmarks look promising in terms of performance. The main problem that needs to be solved would be how add new functionality to the JeSSEL language (which is the main goal of Recaf, to make Java extendable with new constructs). I should be possible to just add new nodes, but this will be harder than just adding an Object algebra due to the constraints of Truffle (Nodes need to have some prerequisites before Truffle recognizes them properly).
- **Remove the types at runtime** Rewrite the type system to be dynamic at runtime. Then it could be tested whether this is a big part of the observed speedup.

Appendix A

JeSSEL benchmark code

```
1  class FactorCounter {
2   int main(int size) {
3     int i = 2;
4     long factors = 0;
5     while (i < size) {
6       factors = 0;
7       int j = 1;
8       while (j < i/2) {
9         if (i % j == 0) {
10          factors++;
11        }
12        j++;
13      }
14      i++;
15    }
16    return factors;
17  }
18 }
```

Listing A.1: FactorCounter.jsl

```
1  class Factorial {
2    int main(int size, int factorial) {
3      int i = 0;
4      long sum = 1;
5      while(i < size) {
6        int count = 1;
7        sum = 1;
8        while(count <= factorial) {
9          sum = sum * count;
10         count++;
11       }
12       i++;
13     }
14     return sum;
15   }
16 }
```

Listing A.2: Factorial.jsl

```
1  class Fibonacci {
2    int main(int size, int fib, int n1, int n2) {
```

```
3       long sum = 0;
4       int i = 0;
5
6       while(i < size) {
7         int firstVal = n1;
8         int secondVal = n2;
9         int j = fib;
10
11        while (j > 0) {
12          sum = firstVal + secondVal;
13          firstVal = secondVal;
14          secondVal = sum;
15          j--;
16        }
17        i++;
18      }
19      return sum;
20    }
21  }
```

Listing A.3: Fibonacci.jsl

```
1   class GCD {
2     int main(int r1, int r2, int k) {
3       int a = 0;
4       int b = 0;
5       while (r1 < r2) {
6         a = r1;
7         b = k;
8         int r = 0;
9         int flag = 1;
10
11        while(flag == 1) {
12          r = a % b;
13          if (r == 0) {
14            flag = 0;
15          }
16          a = b;
17          b = r;
18        }
19
20        r1++;
21        k++;
22      }
23      return b;
24    }
25  }
```

Listing A.4: GCD.jsl

```
1   class PrimefFinder {
2     int main(int size) {
3       int start = 3;
4       int primeCounter = 1;
5       while (start < size) {
6         int primeCheck = 2;
7         int halfOfStart = start/2;
8         while(primeCheck < halfOfStart ) {
9           if(start % primeCheck == 0) {
10            break;
11          }
12          primeCheck++;
```

```
13        }
14
15        if (primeCheck == halfOfStart ){
16          primeCounter++;
17        }
18
19        start++;
20      }
21      return primeCounter;
22    }
23  }
```

Listing A.5: PrimeFinder.jsl

```
1   class PrimeFinder {
2     int main(int size) {
3       int outer = 0;
4       int middle = 0;
5       while (outer < size) {
6         middle = 0;
7         while(middle < outer){
8           middle++;
9         }
10        outer++;
11      }
12      return middle;
13    }
14  }
```

Listing A.6: While.jsl

Appendix B

Total runtime tables

All the values are in seconds and truncated, without rounding, to 4 decimals.
In these tables the total time can be found, this total time is the sum of the 75 runs that were measured of each benchmark. The time in the tables is in seconds.

FactorCounter	Input 1: 2500	Input 2: 5000	Input 3: 7500	Input 4: 10000
JeSSEL	1.3903s	5.3193s	11.6260s	20.0874s
Recaf	25.9618s	101.2630s	216.1363s	382.9358s

Table B.1: Total runtime of 75 runs of the FactorCounter benchmark

Factorial	Input 1: 20	Input 2: 25	Input 3: 30	Input 4: 40
JeSSEL	0.0636s	0.0642s	0.0728s	0.1072s
Recaf	1.9923s	2.4090s	3.0281s	3.4027s

Table B.2: Total runtime of 75 runs of the Factorial benchmark

Fibonacci	Input 1: 40	Input 2: 80	Input 3: 120	Input 4: 160
JeSSEL	0.0603s	0.1145s	0.1638s	0.2364s
Recaf	7.3310s	14.9202s	21.3552s	28.6824s

Table B.3: Total runtime of 75 runs of the Fibonacci benchmark

GCD	Input 1: 1.000	Input 2: 10.000	Input 3: 50.000	Input 4: 100.000
JeSSEL	0.3690s	0.1075s	0.5178s	1.3271s
Recaf	0.1695s	2.2465s	9.2687s	24.3126s

Table B.4: Total runtime of 75 runs of the GCD benchmark

PrimeFinder	Input 1: 1.000	Input 2: 5.000	Input 3: 10.000	Input 4: 50 000
JeSSEL	0.0356s	0.8926s	2.4515s	50.3259s
Recaf	0.7020s	11.0574s	38.8166s	761.9351s

Table B.5: Total runtime of 75 runs of the PrimeFinder benchmark

While	Input 1: 500	Input 2: 1000	Input 3: 2500	Input 4: 5000
JeSSEL	0.0001s	0.0001s	0.0002s	0.0004s
Recaf	0.5456s	1.9577s	12.1152s	47.1183s

Table B.6: Total runtime of 75 runs of the While benchmark

Appendix C

Runtime Graphs

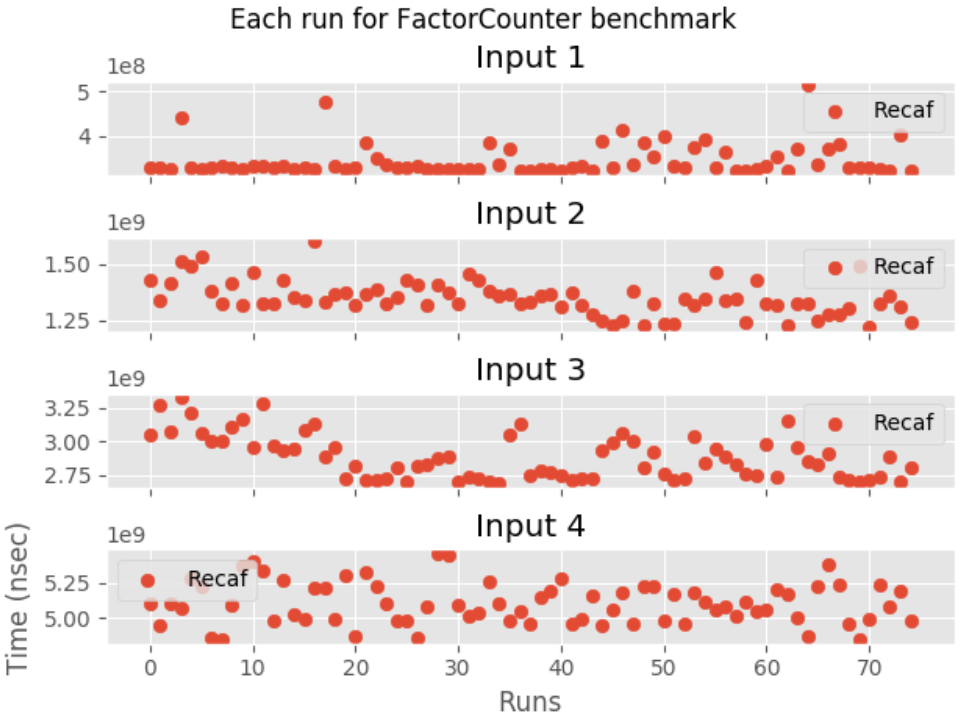

FIGURE C.1: Scatterplot of the FactorCounter runs on Recaf

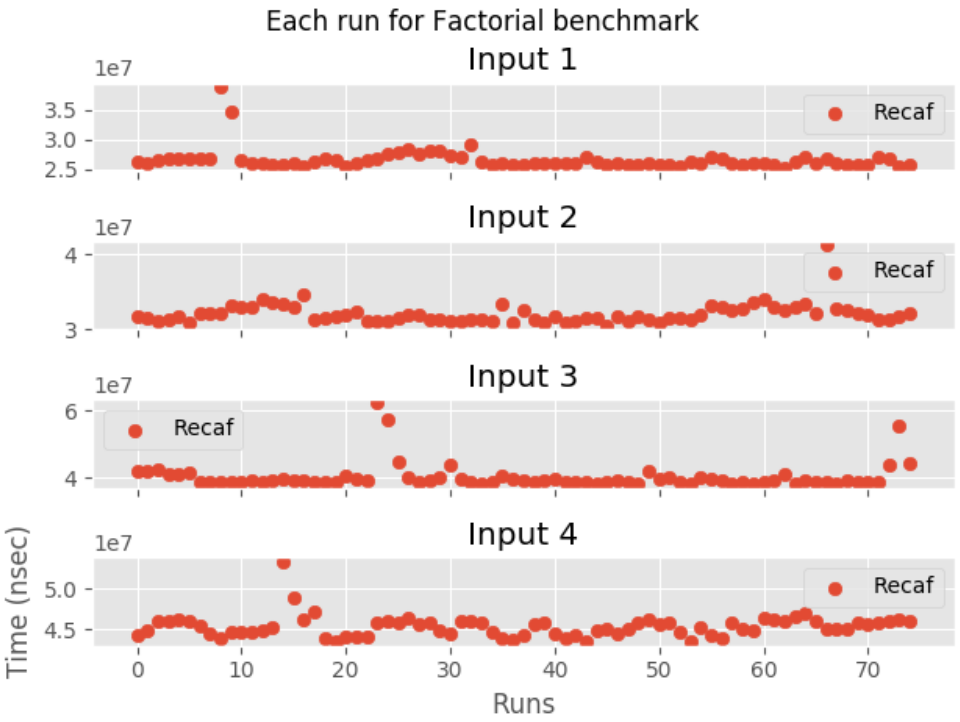

FIGURE C.2: Scatterplot of the Factorial runs on Recaf

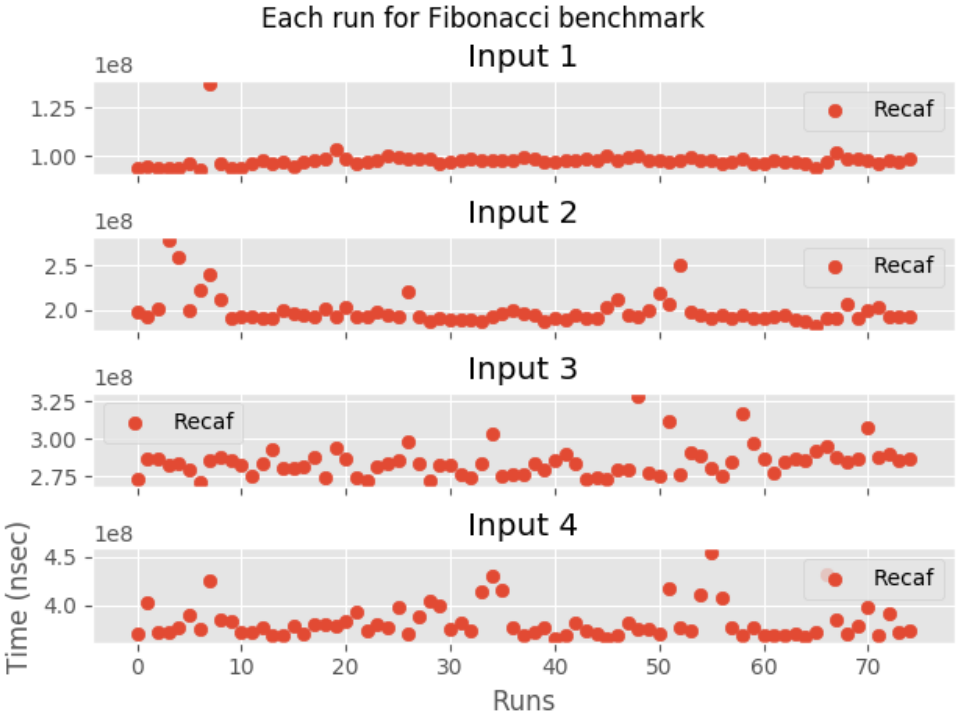

FIGURE C.3: Run scatterplot for the Fibonacci runs on Recaf

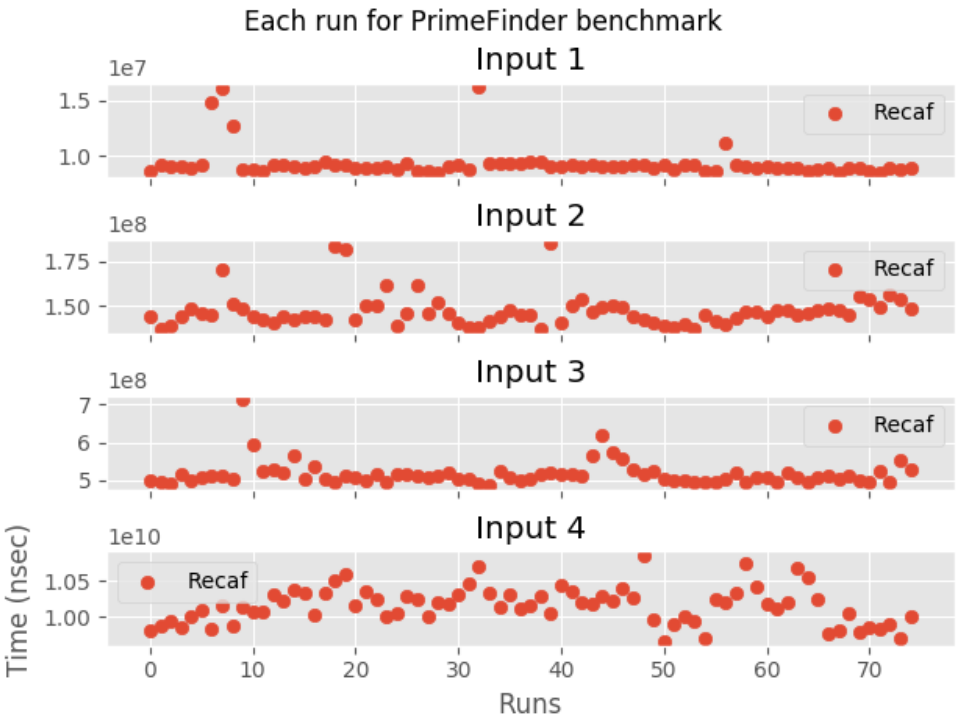

FIGURE C.4: Scatterplot of the PrimeFinder run on Recaf

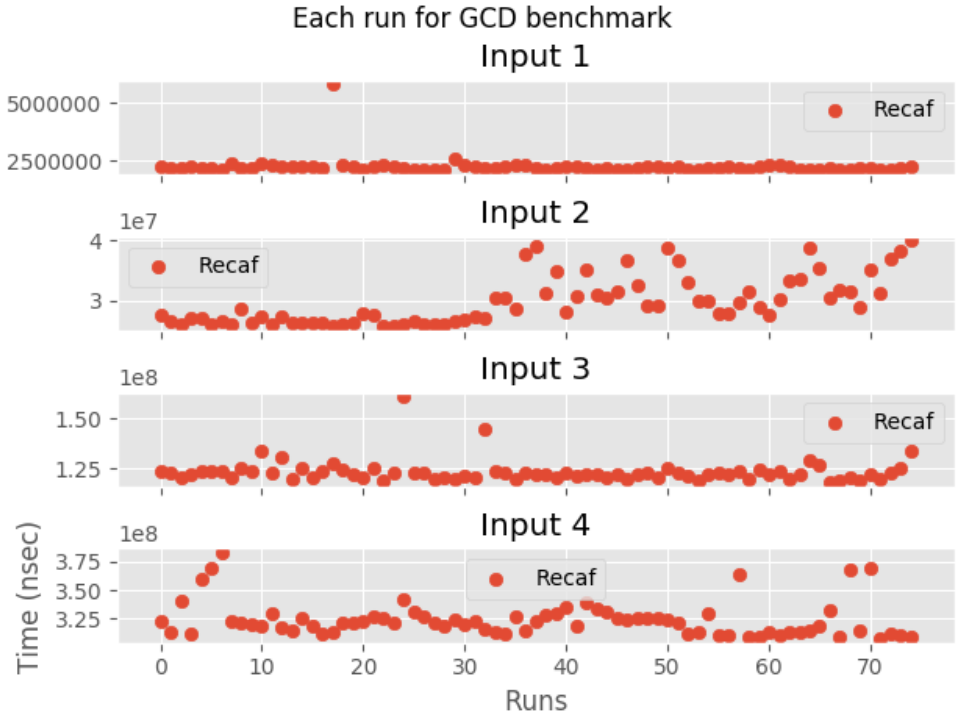

FIGURE C.5: Run scatterplot for the GCD runs on Recaf

FIGURE C.6: Run scatterplot for the While benchmarks run on Recaf

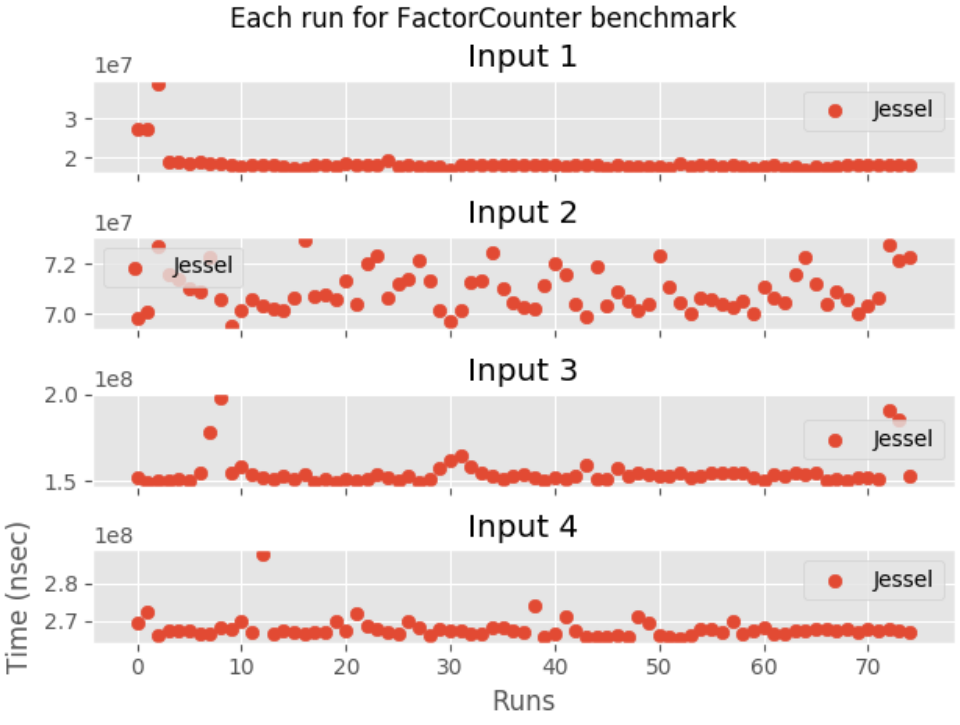

FIGURE C.7: Run scatterplot for the FactorCounter runs on JeSSEL

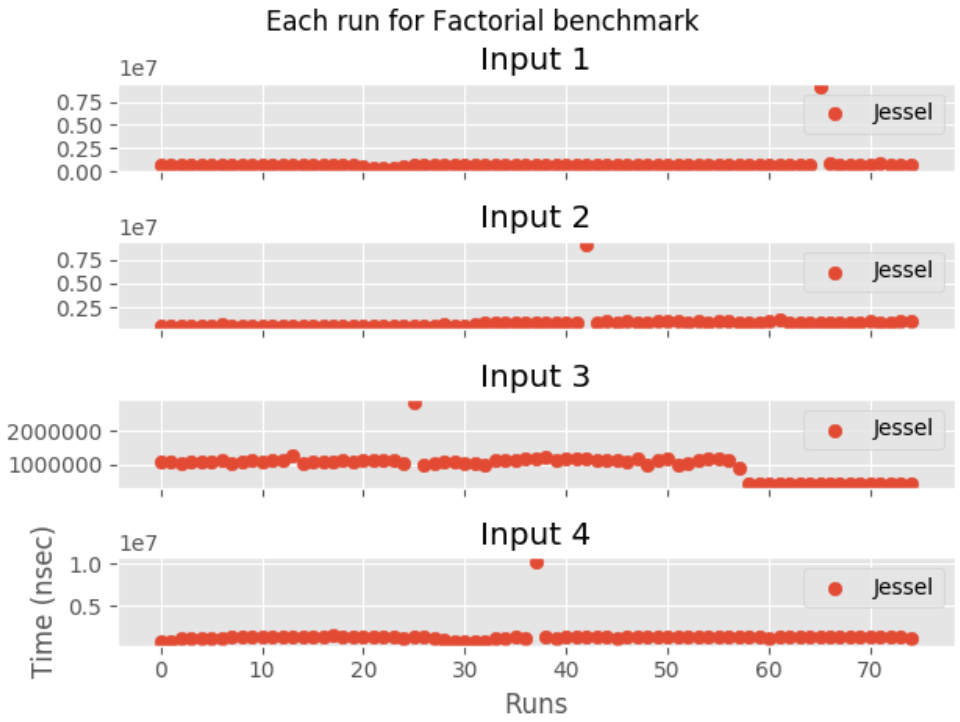

FIGURE C.8: Run scatterplot for the Factorial runs on JeSSEL

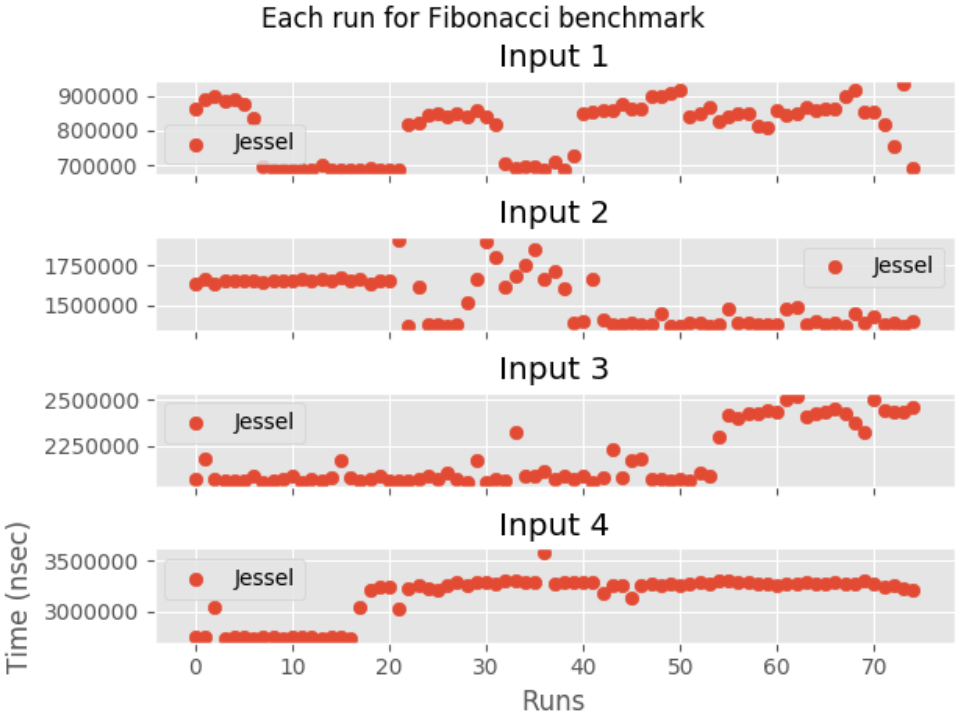

FIGURE C.9: Run scatterplot for the Fibonacci runs on JeSSEL

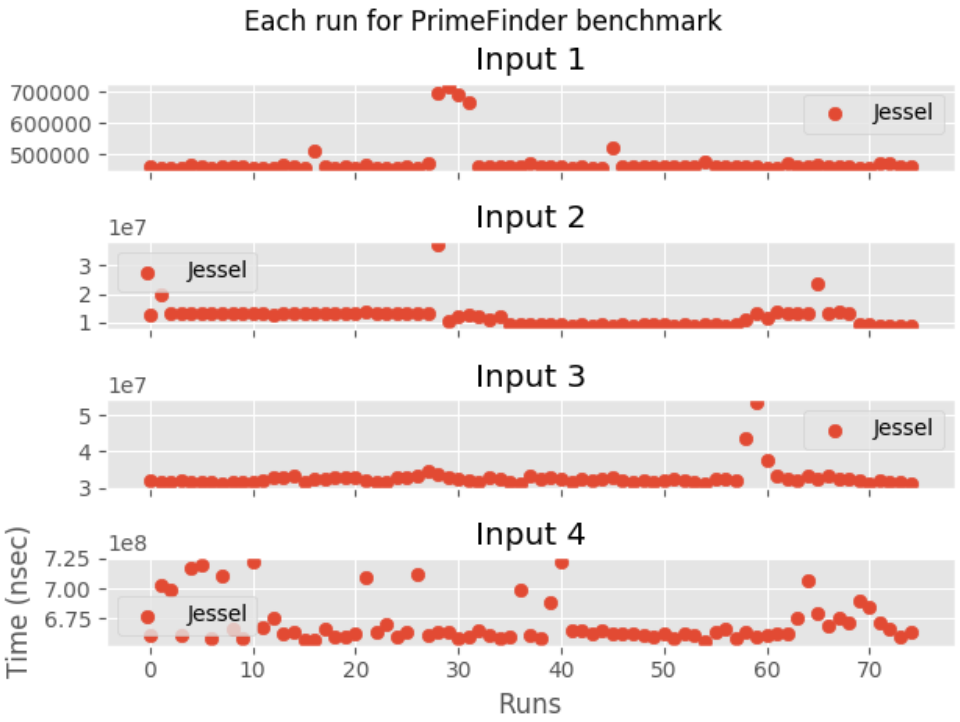

FIGURE C.10: Run scatterplot for the PrimeFinder runs on JeSSEL

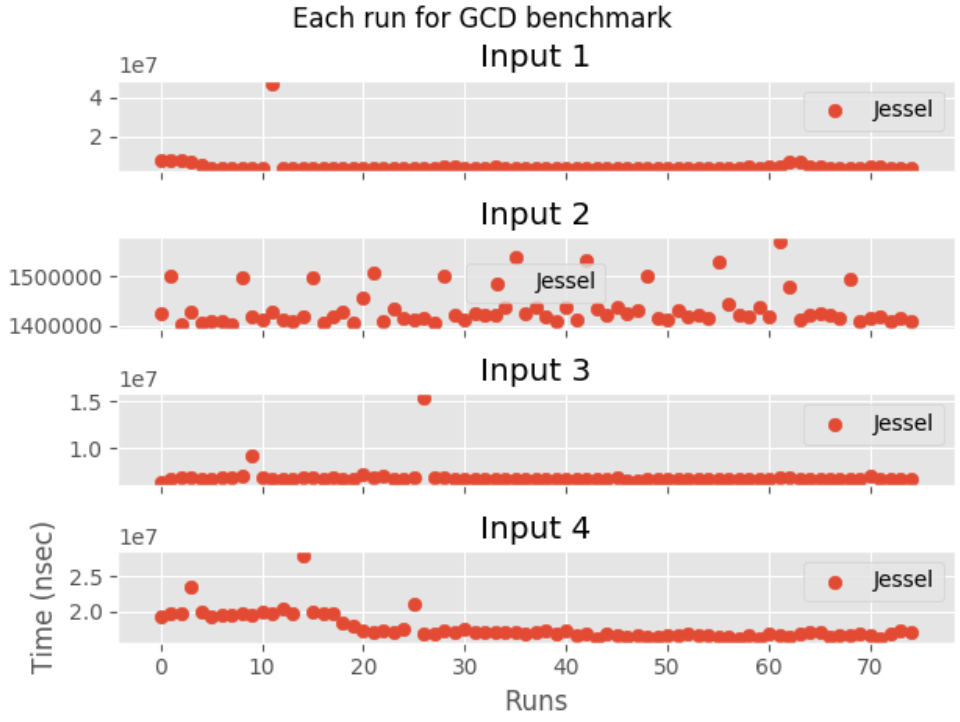

FIGURE C.11: Run scatterplot for the GCD runs on JeSSEL

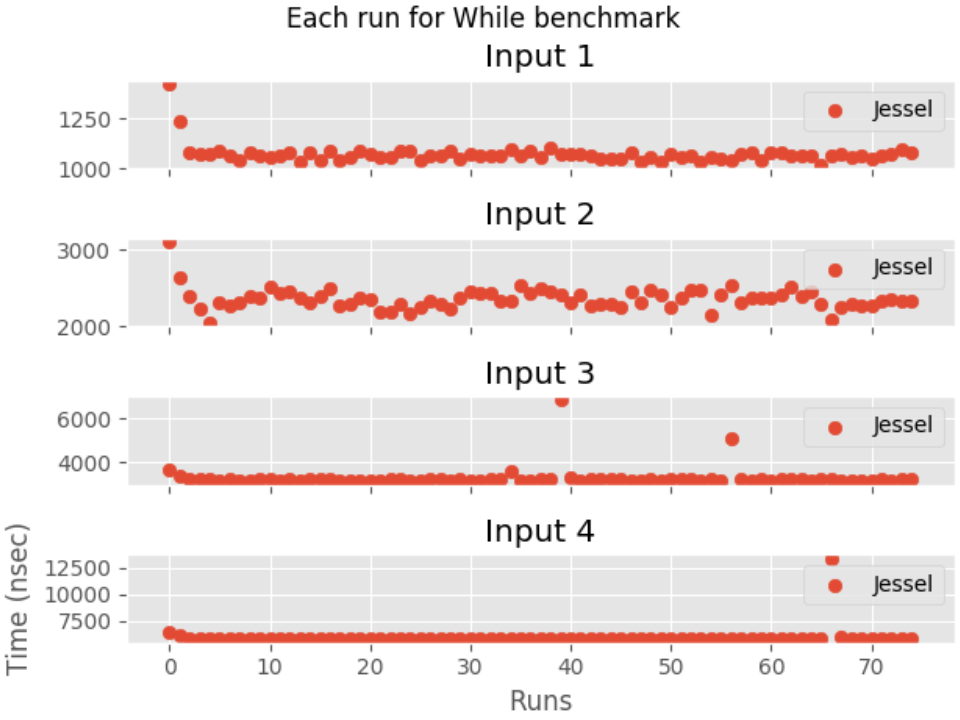

FIGURE C.12: Run scatterplot for the While runs on JeSSEL

Appendix D

Comparison total runtime benchmarks

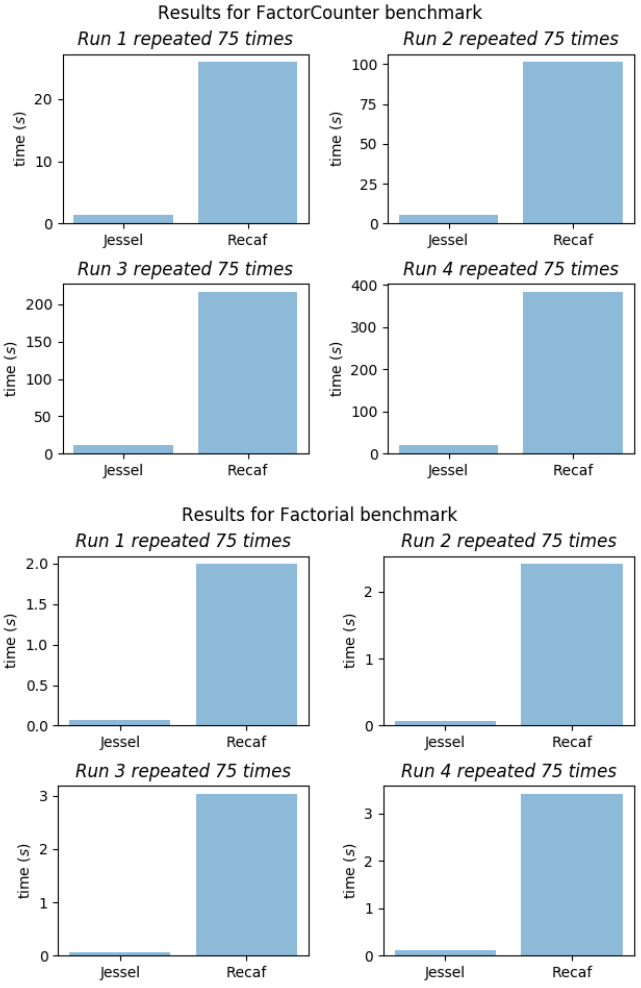

FIGURE D.1: Barplots comparing the total runtimes of Recaf and JeSSEL for each benchmark

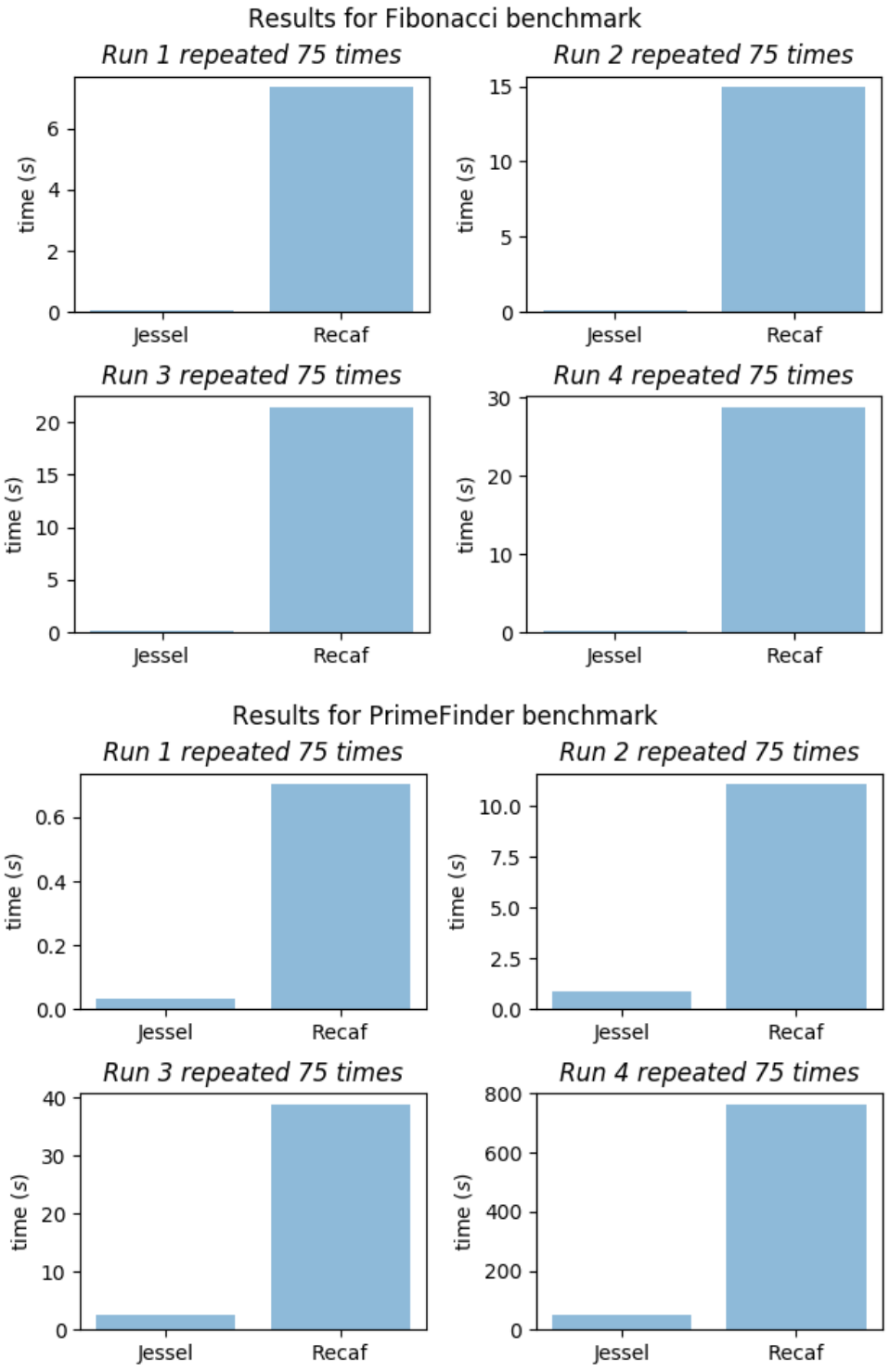

FIGURE D.2: Barplots comparing the total runtimes of Recaf and JeSSEL for each benchmark

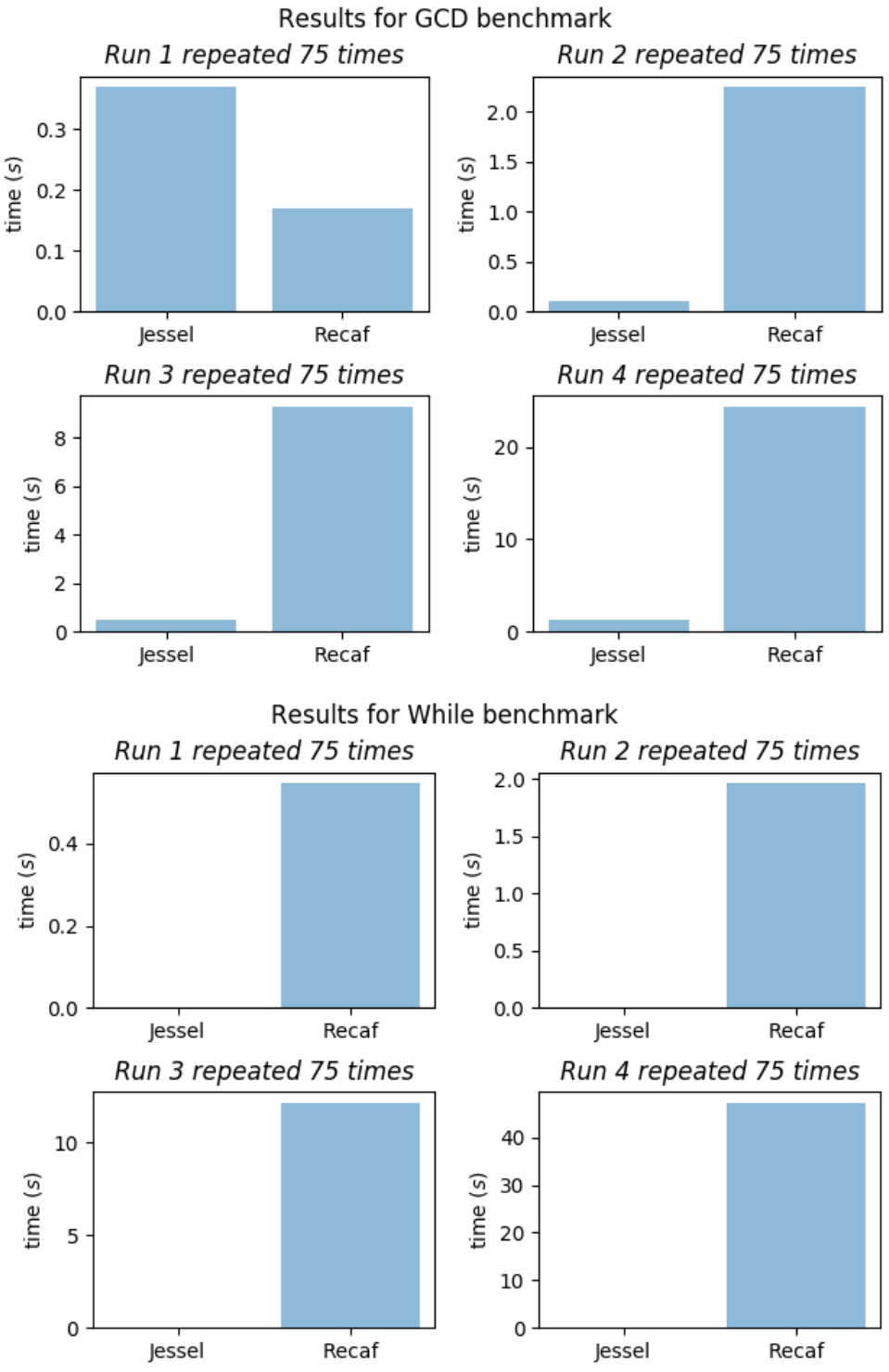

FIGURE D.3: Barplots comparing the total runtimes of Recaf and JeSSEL for each benchmark

Bibliography

[1] A. Biboudis, P. Inostroza, and T. van der Storm. Java dialects as libraries. *GPCE*, October 2016.

[2] M.Grimmer, M.Rigger, R. Schatz, L. Stadler, and H. Mossenbock. Trufflec: Dynamic execution of c on java virtual machine. *ACM*, September 2014.

[3] C.Seaton, B. Deloze, K. Menard, P. Chalupa, B. Fish, and D. MacGregor. Truffleruby, a high performance implementation of the ruby language. https://github.com/graalvm/truffleruby, 2017. Online; accessed 7 June 2017.

[4] ? Fastr, an alternative implementation of the r language. https://github.com/graalvm/fastr, 2017. Online; accessed 8 June 2017.

[5] T. Wurthinger, A. Woss, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing ast interpreters. *DSL*, October 2012.

[6] E.Barrett, C. Friedrich Bolz-Tereik, R.Killick, Sarah Mount, and Laurence Tratt. Virtual machine warmup blows hot and cold. *ACM*, April 2017.

[7] I. Gouy. The computer language benchmarks game. http://benchmarksgame.alioth.debian.org, 2016. Online; accessed 9 June 2017.

[8] S. Marr, B. Daloze, and H. Mssenbck. Cross-language compiler benchmarking: Are we fast yet? https://github.com/smarr/are-we-fast-yet, 2016. Online; accessed 8 June 2017.