Dissertation

Marcel Taubert (mt652)
20962335

School of Computing
MSc Advanced Computer Science
University of Kent

Supervisor: Dr. Michael Vollmer

August 24, 2023

1 Introduction

In the modern world we live in today, everything is controlled by code. Everyone of us is using technology controlled by code, if they want it or not. But not many people know how the code that programmers around the world write gets executed on their device.

1.1 What is an interpreter?

In general there are two ways to execute code. Both approaches include the process of translating the human readable code into something that the computer can understand.

During the first technique the code is compiled (translated) into machine code. The end product is a stand alone program that can be executed at any time on the architecture it was compiled for.

The second approach is called 'interpreting'. There are two primary methods to how an interpreter works. One approach is called a tree-walk interpreter.

In this case the interpreter just walks the generated abstract syntax tree and executes it.

The second approach is a virtual machine interpreter which involves an extra step between the generation of the syntax tree and the execution. A compiler walks the tree and generates byte code. This bytecode will then be fed into the virtual machine which executes it.

We will use a virtual machine interpreter in this paper.

Interpreters are generally easier to implement and have some other advantages that makes them more approachable than compilers like portability or a fast edit-compile-run cycle as stated by the authors of vmgen [3].

1.2 What is a virtual machine interpreter?

A famous technique to implement interpreters is to build a virtual machine interpreter. A vm interpreter is generally divided into two systems. A frontend and a backend. [3]

The frontend consists of a compiler that takes the written code and produces a sequence of bytecode instructions. The backend is a virtual machine that gets the stream of bytecode instructions as input and executes them. [3]

The bytecode or intermediate representation used in a vm interpreter is usually designed to be very similar to a real machine. [3]

Some real world examples of virtual machine interpreters are for example the JVM (Java Virtual Machine) or the PVM (Python Virtual Machine). (TODO: link to them???)

1.3 Virtual machine

When we are talking about virtual machines we distinguish between stack based vm's and register based vm's.

Each of the version has it's advantages and disadvantages.

Stack based virtual machines are generally easier to implement than the register based approach. That shows for example in the complexity of the intermediate representation used in the virtual machine. The stack based bytecode has a tendency of being smaller and by that more efficient in comparison to the rather complex bytecode instructions of the register based approach.

```
typical stack based instruction:
Push 1    -- push value to the stack

typical register based instruction:
LD R1, 42 -- load value '42' into register R1
```

This simplicity is the reason we have to decided to rely on a stack based virtual machine in this paper.

1.4 Optimizations

When you are running your Java program the compiler does not just generate bytecode for the JVM from your written code. The compiler will try it's best to optimize as much as possible to ensure that the execution of the produced bytecode will be as fast as possible. This optimization part of the compiler is by far one of the most crucial tasks of a compiler.

During the development of this project we have looked at a number of different optimizations. Some of the optimizations that we have implemented for the project are threaded code and peephole optimizations (superinstructions) (TODO: SHOULD I EXPLAIN THEM HERE)

1.5 Automation

Writing an interpreter for a programming language can be a tedious and challenging task on it's own. Thinking about how to keep the code efficient and additionally implement optimizations makes it even harder.

One solution to this is automating the process of writing a virutal machine interpreter by providing a configuration file.

1.6 Objectives

This paper will use the Rust programming language to build each part of a virtual machine interpreter with the optimal goal of automating the generation of it self depending on a given configuration of a user.

We will explore techniques and approaches on how to build a vm interpreter in Rust and use benchmarking to visualize results depending on applied optimizations.

2 Description of the problem

When it comes to the problem that we are trying to solve there are two main points. Efficiency and Automation.

2.1 Efficiency

In terms of efficiency at run time, nativ compiled machine code will always outperform the interpreted version. So why would we even care about writing efficient interpreters and not just use native code compilers.

One of the main reasons interpreters are preferred over compilers is that native code compilers are more complex to develop and difficult to maintain. [1]

Another big advantage of the interpreting approach is that compilers can only generate native code for one target system while the virtual machine interpreter stays the same on every system. By that the interpreter is portable and the generated code does not depend on the underlaying machine.

Summed up the first problem is that interpreters will never be as fast as native compiled machine code but by contributing to increase of efficiency we get all the benefits of using interpreters while minimizing the disadvantages to compilers.

2.2 Automation

Many programmers will have the idea of implementing their own programming language at some point in their career. Most of them will have noticed that buidling an interpreter is a challenging task and requires a lot of work and a clear structure. In addition to that it shows that many parts of an vm interpreter are similar and repetitive. For example the code for executing VM instructions will be similar for most of the instructions. [3]

But what happens when the interpreter does not give the expected outcome in terms of efficiency. It results in manual rewriting of a codebase just to change the implementation of some part of the vm interpreter to see if the performance increases. Rewriting the whole interpreter to test if the performance is better using a different development approach is not only time consuming but also error prone.

One solution to this problem is automation. The user should be able to provide a configuration file and based on that we will generate an efficient vm interpreter. It will already use efficient implementation techniques and come with built in optimizations. It also provides easy extensibility for the user without needing to change any source code.

3 Literature review

When talking about virual machines we consider only the 2 main paradigms in this paper, namely stack based and register based virtual machines.

While virtual machines are great to use in combination with interpreters each architecture has it's advantages and disandvantages.

Stack based virtual machines use a stack to manage state. The stack pointer, often abbreviated with sp, is used to keep track of the top of the stack. Instructions like PUSH and POP can manipulate the stack by adding or removing values from it while incrementing or decrementing the stack pointer.

The main advantage of a stack-based virtual machine lays in it simplicity. The implementation is straight forward, the instructions are kept simple and optimiztions are easy to implement.

Register based virtual machines work on a different principle. They use a set of virtual registers which are designed to mimic hardware registers as a place to store data. Instructions such as LOAD or STORE can modify the registers.

The instructions used in register based virtual machines are larger since they need to encode the operands. As an example the STORE instrucion needs to know the value to store and the register where it should store the value.

The execution cycle involve 3 steps. Fetching, decoding and executing. First the machine needs to get the instruction from memory. Then it needs to decode the instruction to know which operation to perform and which registers to use. This decoding step can add a small overhead to the machine but since it usually involves just simple logical operations it is still very fast. In the last step the operation is executed and the registers are modified accordingly.

Since all operations are performed on registers, the need to push and pop from the stack is eliminated which results in efficient data access.

Regarding the effort it takes to create an efficient register based virtual machine interpreter M. Anton Ertl says:

> From the viewof the compiler writer, many languages can be easily compiled for stack machine code. To achieve better performance with a register machine, the compiler must perform optimizations, e.g., global reg- ister allocation (which needs data flow analysis). This would eliminate one ofthe advantages ofusing an interpreter, namely simplicity. [2]

A good example to compare stack based virtual machines with register based machines on a real world project is Lua. Lua runs programs on a virtual machine interpreter. It compiles the code into instructions for a virtual machine.

When Lua was first created in 1993 it used a stack based virtual machine. 10 years later in 2003 with the release of Lua 5.0 they have changed their approach and now use a register based vm. [4]

To understand how Lua profited from the register based virtual machine we first need to understand how Lua works.

Lua creates a prototype for each function in the code. The prototype contains an array with all the instructions to execute this function and an array of all the Lua values and constants used. [4]

Upon entering a function, an activation record is preallocated on the stack. This record holds all funcion registers, where local variables are stored. Resulting in efficient variable access. [4]

In addition to that register based instructions do not need any PUSH or POP instruction which are expensive in Lua because of the way Lua represents values.

Looking at the disadvantages of register based vm's in terms of generated code size and single instruction size in Lua we can see that they are not too bad.

Lua's stack based virtual machine had an instruction set of 49 instructions where the register based approach only needed 35. The size of a single instruction increased from 2 bytes to 4 bytes with the change of the virutal machine but this relativises itself due to less generated instructions.

Here an example

```
local a,t,i
a = a+i
a = a+1
a = t[i]
```

Figure 1: *Example Lua code from [4]*

The generated instructions vary a lot from Lua 4.0 to Lua 5.0.

```
PUSHNIL      3
GETLOCAL     0
GETLOCAL     2
ADD
SETLOCAL     0
GETLOCAL     0
ADDI         1
SETLOCAL     0
GETLOCAL     1
GETINDEXED   2
SETLOCAL     0
```

Figure 2: *Generated instructions for Lua 4.0 (stack based vm) from [4]*

Comparing Figure 2 and Figure 3 we can see even though each instruction in Figure 3 has 3 operands, resulting in larger size, it just generated a total of 4 instructions compared to 11 in Figure 2

4 Description of work

The goal of this project is to build an virtual machine interpreter and explore the implementation of such in the Rust programming language.

```
LOADNIL     0 2 0
ADD         0 0 2
ADD         0 0 250 ;1
GETTABLE    0 1 2
```

Figure 3: *Generated instructions for Lua 5.0 (register based vm) from [4]*

4.1 Tools used

During the development of this project we used three programming languages. We build the virtual machine and all parts of the interpreter in Rust and used Python to visualize any benchmarking results. The input language of the interpreter is the Imp programming language (see section 4.5). [5]

But why did we use Rust and not the C programming language like most other systems do? One reason for that is that as stated before most of the already existing solutions are written in C and we wanted to explore something new. Another reason why we decided to use Rust is that we were looking for a low level systems programming language that allows us to work as efficient as possible while still offering high level features like match statements and iterators. An extension to that is Rust's expressive type system and strict memory management rules which helps avoiding many kinds of errors that appear in standard C programming.

4.2 Virtual Machine

The virtual machine is the part that exectues the byte code that was generated by the compiler.

4.2.1 Implementation

The virtual machine we have built is very basic. It contains a representation of the stack and a HashMap to hold the variables.

```
pub struct ByteCodeInterpreter {
    stack: Vec<usize>,
    pc: i32,
    variables: HashMap<String, usize>,
}
```

ByteCodeInterpreter struct definition

The optimized version of it implementing the 'computed goto' optimization (see section 4) is a bit more complicated. Here we also have the mapping of ByteCode to the function that executes this bytecode. This optimization is described in more detail at 4.

7

```
pub struct ByteCodeInterpreterThreaded {
    stack: Vec<usize>,
    pc: i32,
    variables: HashMap<String, usize>,
    ops: HashMap<Discriminant<ByteCode>, Instruction>,
    instructions: Vec<ByteCode>,
}
```

ByteCodeInterpreterThreaded struct definition

4.2.2   Stack

The virtual machine we built is stack based. This means there are no registers and everything happens on the stack.

When you add two numbers there will be a 'push' instruction for both values and an add instruction which will pop the last two values from the stack, adds them together and pushes the result back on the stack.

```
The code '1 + 2' would produce the following instructions:
```

```
inst:   stack:
        []
push 1  [1]
push 2  [1, 2]
add     [3]
```

4.2.3   Byte code

To define a byte code we have decided to use Rust's powerful enums. They are not only able to perfectly represent each instruction with it's parameters but also helps during development due to Rust's powerful type system.

```
#[derive(Debug, Clone, PartialEq)]
pub enum ByteCode {
    Push(usize),
    Pop,
    Add,
    Sub,
    Mul,
    Var(String),
    Eq,
    NEq,
    Lt,
    Gt,
    Lte,
```

8

```
    Jz {
        label: String,
        offset: i32,
    },
    ...
}
```

Rust representation of the Byte Code

One downside of using the convenient Rust way of using enums to represent byte code is that they are relatively large in memory.

One byte code instruction adds up to 32 bits in memory. Enums in rust behave like a tagged union. That means the size of each variant is equal to the size of the biggest field + an enum tag.

The largest field in our byte code representation is the Jz label. It contains a String and an i32.

```
A string in Rust contains:
    8 bits -> pointer to the chars
    8 bits -> len of the string
    4 bits -> the i32 value
    8 bits -> enum tag
    + padding

    adds up to 32 bits.
```

This decision might be good for developing since the Rust compiler will warn if you for example forget a variant in a match statement but it is not really efficient.

4.3   Interpreter

The interpreter consists of a scanner and a parser. We have decided to handwrite both parts instead of generating them. Even though the scanner and the parser are not the most performance critical parts of an interpreter we decided to handwrite them in Rust.

4.3.1   Scanner

The scanner was implemented very straight forward by using a match statement to iterate the source code and output a stream of tokens.

Tokens are defined as the token type and an optional literal.

```
pub struct Token {
    pub token_type: TokenType,
    pub literal: Option<Object>,
}
```

Token definition

```
pub enum Object {
    Num(f64),
    Bool(bool),
    Variable(String),
    DivByZeroError,
    ArithmeticError,
}
```

Object definition

```
pub enum TokenType {
    LeftParen,
    RightParen,

    Minus,
    Plus,
    ...
    Eof,
}
```

TokenType definition

4.3.2 Parser

The parser takes the stream of tokens as input and builds up an abstract syntax tree. In the Rust code it is represented as a 'Vec¡Rc¡Stmt¿¿'.

```
As an example the code 'x := 10;' will get tokenized into
'[Identifier,Assignment, NumberLiteral, Semicolon]'
and the parser will output a tree structure like:

ExpressionStmt {
    expr: AssignExpr {
        name: "x",
        value: LiteralExpr {
            value: Num(10)
        }
    }
}
```

4.3.3 Interpreter (testing)

To be able to test the scanner and parser before writing the bytecode generator we build a small interpreter that just runs the code.

It uses the visitor pattern to traverse the abstract syntax tree generated by the parser and executes the code immediatly.

To use the visitor pattern approach we implemented a statement visitor and an expression visitor.

```
pub trait StmtVisitor<T> {
    fn visit_block_stmt(&self, stmt: &BlockStmt) -> Result<T, ()>;
    fn visit_if_stmt(&self, stmt: &IfStmt) -> Result<T, ()>;
    fn visit_expression_stmt(&self, stmt: &ExpressionStmt) -> Result<T, ()>;
    fn visit_print_stmt(&self, stmt: &PrintStmt) -> Result<T, ()>;
    fn visit_while_stmt(&self, stmt: &WhileStmt) -> Result<T, ()>;
}
```

Statement visitor example

For each statement we have setup a structure holding the necessary data.

```
#[derive(Debug)]
pub struct IfStmt {
    pub condition: Rc<Expr>,
    pub then_branch: Rc<Stmt>,
    pub else_branch: Option<Rc<Stmt>>,
}
```

Structure holding data for an if statement

4.4   Optimizations

Optimizaions play a critical role in the field of compiler design. There are many ways to optimize code.

Some common techniques are:

1. **Dead Code Elimination**:
   Remove parts of the code that are never to be used.

   Consider the follwng C code. In this case the variable 'c' is never used so the statement 'int c = 3;' does not need to be compiled

   ```
   int main() {
       int a = 1;
       int b = 2;
       int c = 3; // never used

       return a + b;
   }
   ```

2. **Constant folding**:
   When calcualtions are only using constant values and by that can be evaluated during compile time it will directly replace it with the computed value.

Consider the follwing C code. In this example 'int a = 10 + 20;' only uses values known at compile time so the compiler can internally replace the statement with 'int a = 30;'.

```
int main() {
    int a = 10 + 20;
    printf("%d", a);

    return 0;
}
```

3. **Superinstructions**:
   Superinstructions is a term used for the case when combining two or more instructions into a single big instruction.

   Consider the following C code.

```
int main() {
    int i = 0;
    while (i < 10) {
        i += 1;
    }
}
```

If we naively 'compile' this code we would get instructions like this:

```
// in the loop
load i -- get the variable at i and push it to the stack
push 1 -- push the value '1' to the stack
add    -- pop the first 2 values from the stack, add them, and push the result
```

We always need 2 instructions to push the constant value to the stack and then add the top 2 values on the stack together.

But we can create a superinstruction called 'push_add' which would change the generated code to this.

```
// in the loop
load i     -- get the variable at i and push it to the stack
push_add 1 -- superinstruction that does the push and then the add
```

That change might seem very insignificant but we can extend this idea of superinstructions to combine already built superinstructions with each other and by that save many instructions.

4. **Computed goto**:
   When executing the bytecode inside of a virtual machine interpreter for example you will usually find a big switch statement inspecting the current instruction and executing it. In our case, since we used Rust we have a big match statement like this.

```rust
while self.pc < instructions.len() as i32 {
    let inst = &instructions[self.pc as usize];
    match inst {
        ByteCode::Push(value) => {
            self.stack.push(*value);
        }
        ByteCode::Pop => {
            self.stack.pop().unwrap();
        }
        ByteCode::Add => {
            let a = self.stack.pop().unwrap();
            let b = self.stack.pop().unwrap();
            self.stack.push(b + a);
        }
        ...
    }
}
```

This 'while' loop will iterate over every single instruction in the program. That means for every instruction it has to go through all the possible values inside of the 'match' statement until it find the matching result.

The 'Computed goto' optimization tries to make this process more efficient.

To implement that, we have to create an array of function pointers where the index of the array corresponds to an instruction or use a HashMap with the byte code instruction as the key and the function as a value like we did in Rust.

```rust
pub type Instruction = fn(interp: &mut ByteCodeInterpreterThreaded);
... {
    ops: HashMap<Discriminant<ByteCode>, Instruction>
}
```

```
ops.insert(std::mem::discriminant(&ByteCode::Push(0)), Self::op_push);
ops.insert(std::mem::discriminant(&ByteCode::Pop), Self::op_pop);
ops.insert(std::mem::discriminant(&ByteCode::Add), Self::op_add);
```

Then each of the functions (Self::op_push, Self::op_add, ...) will finish with a call to a 'next()' function.

This next() function will increment the program counter (the index of where we are in the byte code) and call the function which was mapped to the next instruction in the HashMap.

In our case in Rust the next function looks like the following.

```
fn next(&mut self) {
    // incrementing the program counter;
    self.pc += 1

    // check for end of stream
    if self.pc >= self.instructions.len() as i32 {
        return;
    }

    // call the next function
    self.ops[&self.instructions[self.pc as usize]](self);
}
```

This threaded code approach results in a short and fast instruction dispatch sequence and can lead to better branch prediction accuracy on machines with branch target buffers as stated by the authors of vmgen [3].

4.4.1   Replacing superinstructions in the byte code

To create a byte code that involves superinstructions we first need to generate the normal byte code.

In a second iteration we go through the sequence of instructions and check if we can combine any of the instructions into a superinstruction.

This step will need to iterate through all the instruction once and by that grow linear with the size of the byte code.

4.5   Input language

When you want to build a virtual machine interpreter you need to have an input language. Instead of creating our own programming language for this project we chose the programming language Imp. [5]

14

"Imp is a simple imperative programming language which embodies a tiny core fragment of conventional mainstream languages such as C or Java". [5]

```
Z := X;
Y := 1;
while Z != 0 do
    Y := Y * Z;
    Z := Z - 1;
end
```

Example code in the Imp programming language

Not only is Imp simple but it is already fully defined and has many examples we can test agains.

The book 'IMP Simple Imperative Programs' by Benjamin C. Pierce [5] describes the language in detail and provides for example BNF grammar definitions of the syntax which makes it easy understandable and provides a guideline when developing the scanner and parser of the interpreter.

The language has no scope so all variables are global. This fact makes it easier for us since we do not need to keep track of scopes or environments and the variable lookup in our interpreter is just one single HashMap that holds all global variables.

Variables can be assigned without the need to give an explicit type since all the variables in Imp are numbers. Again this makes it very convinient for us to implement.

```
a := 1;
b := a;
```

Variable assignment in the Imp programming language

Imp defines while loops which are delimited by the 'do' and 'end' keyword and do not need any parentheses around the condition. Those particular circumstances makes it easy to parse.

```
a := 0;
while a < 10 do
    a := a + 1;
end
```

While statement in the Imp programming language

Similar to while loops, if statements are also defined without any unnecessary parentheses and it is delimited with describing keywords like 'then', 'else' and 'end'.

```
a := 0;
if a < 10 then
    a := a + 1;
end
```

If statement in the Imp programming language

For convinience and testing purposes we added our own keyword 'print' to the language. It helps during development to get a better view into the execution of the interpreter.

```
a := 0;
if a < 10 then
    print a + 1;
else
    print a - 1;
end
```

Print statement in the Imp programming language

Furthermore we have implemented the modulo operator and a 'continue' statement, to manually continue to the next iteration of a loop. By doing that have more possibilities to create benchmark programs.

Imp does not include function calls or any of the commonly known features of todays high level languages such as classes, lambdas, structs or even strings and arrays.

5 Results

5.1 Benchmarks

5.1.1 Setup

To create the benchmarks we have used a system with the following specs.

1. **CPU**:
   11th Gen Intel i7-11370H (8) @ 4.800GHz

2. **Operating System**:
   Fedora Linux 37 (Workstation Edition) x86_64

The script 'benchmark.py' executes the benchmarks and generates the visualization using matplotlib.

To collect the benchmark timings and reduce the chance of outliers that can occure because of external reasons such as system load or memory availability the script executes each command 10 times and calculates the average.

To toggle different superinstructions inside of the Rust code from the outside we have used Cargo's 'features' option.

This shows how to write conditional code that will be turned on or off during compilation in Rust.

```rust
enum ByteCode {
    ...
    #[cfg(feature = "PushAdd")]
    PushAdd(usize),
    #[cfg(feature = "AssignPushAdd")]
    AssignPushAdd {
        name: String,
        value: usize,
    },
}
```

5.1.2 Test Programs

To get the benchmark data we used multiple small programs that generate different bytecode. By that we can see which optimization give the most benefit for what kind of program.

As test programs we used:

1. **Factorial.imp**:
   A simple program to calculate 10!.

   ```
   result := 1;
   n := 10;
   while n > 1 do
       result := result * n;
       n := n - 1;
   end
   print result;
   ```

2. **Fib_70.imp**:
   A program to calculate and print the first 70 fibonacci numbers.

   ```
   a := 0;
   b := 1;
   n := 70;

   while n > 1 do
       print a;
       temp := a;
       a := b;
       b := temp + b;
       n := n - 1;
   end
   ```

17

3. **Gcd.imp**:
   A program to calculate the greatest common divisor of two numbers.

   ```
   a := 234234;
   b := 234;

   while a != b do
       if a > b then
           a := a - b;
       else
           b := b - a;
       end
   end

   print a;
   ```

4. **If.imp**:
   A program that uses many if statements and assignments inside of a loop.

   ```
   i := 1;
   a := 0;

   while i < 1000 do
       i := i + 1;
       a := a + 1;

       if a < i then
           a := 0;
       end
       if a < i then
           a := 0;
       end
       if a < i then
           a := 0;
       end
       if a < i then
           a := 0;
       end
   end
   ```

5. **Increment_loop.imp**:
   A program that uses many assignments and addtions.

   ```
   a := 0;
   b := 0;
   c := 0;
   ```

```
        d := 0;
        e := 0;
        f := 0;
        g := 0;
        i := 0;

        while i < 1000 do
            i := i + 1;
            a := a + 1;
            b := b + 1;
            c := c + 1;
            d := d + 1;
            e := e + 1;
            f := f + 1;
            g := g + 1;
        end

        print i;
```

6. **Increment_loop.imp**:
   A program that calculates the sum of the first 1000 natural numbers.

```
        N := 1000;
        Sum := 0;
        i := 1;

        while i <= N do
            Sum := Sum + i;
            i := i + 1;
        end

        print Sum;
```

7. **Fizz_buzz.imp**:
   A program that uses the modulo operator to calculate the famous fizzbuzz problem.

```
i := 0;

while i < 1000 do
    i := i + 1;
    if i % 3 == 0 && i % 5 == 0 then
        print 0;
        continue;
    end
    if i % 3 == 0 then
```

```
        print 1;
        continue;
    end
    if i % 5 == 0 then
        print 2;
        continue;
    end
    print i;
end
```

Each of the programs generates a byte code that uses some instructions intensivly. By that we can see which superinstruction has the most influence on run time and also see the time the generation for the byte code using this superinstruction combination took.

5.1.3   Results

The following images show the performance analysis charts.

The y-axis, calibrated in milliseconds, shows the time.

On the x-axis we show the different files. Each enty has 3 values attached to it.

1. **Generation time (blue)**:
   The time it took to generate the byte code for the application.

2. **Interpreting time (orange)**:
   The time it took to interpret the byte code.

3. **Interpreting time (threaded) (green)**:
   The time it took to interpret the byte code with the computed goto (or threaded code) optimization enabled.

Looking at the first diagram with no optimizations enabled we get the base line. Now we can compare it with some of the superinstruction optimizations.

First of all, we can see that the threaded code optimization causes a slow down of approximatly 300% compared to the non optimized version. But why is this? We have already explained how we have implemented the computed goto optimization for that see Section 4.

There you can see that we used a hash map to store the mapping of byte code instruction to the function that implements it. The key is a usize. This usize is the result of a call to the std::mem::discriminant function. It returns the index of a given variant in an enum.

Since Rust does not by default allow an enum with fields that hold data to be the key of a hash map we had to use this approach.

After every instruction we need to call the discriminant function to get the index of the next byte code instruction. This overhead is most likely the main reason for the slow down.
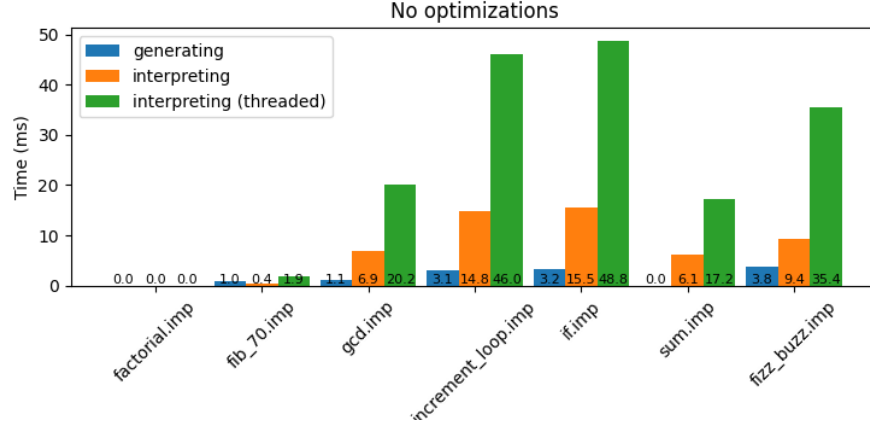
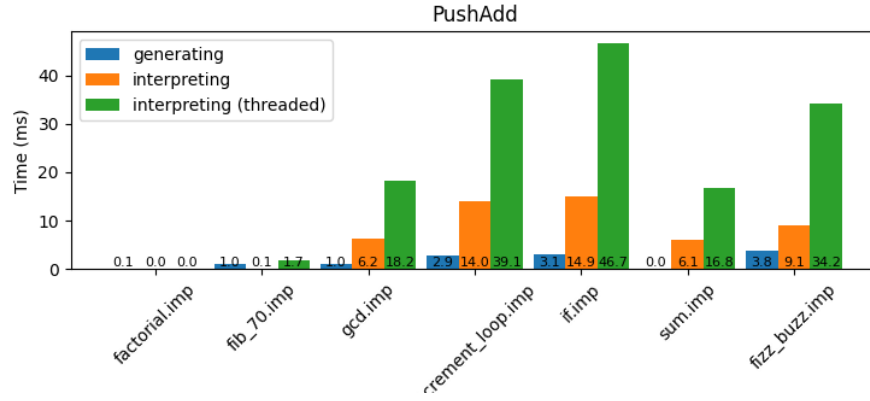Figure 4: *Benchmark results without any optimizations enabled*

Figure 5: Benchmark results with the superinstruction PushAdd enabled

When we are comparing Figure 2 (PushAdd) and Figure 1 (No optimizations) we can see that the implementation of the superinstruction PushAdd which combines the PUSH and the ADD instruction resulted in slight improvements in the run time.

The biggest performance benefit was gained in the increment_loop.imp program 6. The generated byte code for this program uses the combination of pushing the value 1 to the stack and adding it to a variable in a loop. Combining those two instructions we can gain almost a 20% speed improvement in the threaded code and a 6% speed improvement in non-threaded code.

The following shows the generated bytecode for the increment_loop.imp program without using any superinstructions.
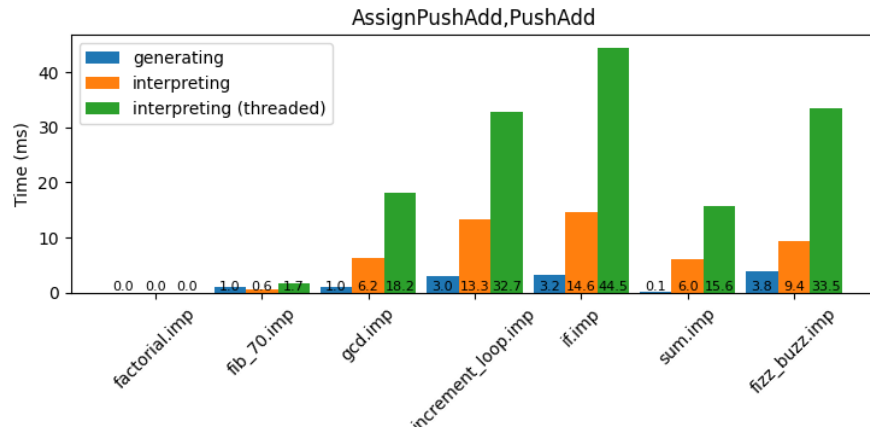
```
inside the loop...
```

Figure 6: Benchmark results with the superinstructions PushAdd and Assign-PushAdd enabled

```
Var("a")
Push(1)
Add
Assign("a")
Var("b")
Push(1)
Add
Assign("b")
...
```

This shows the generated bytecode using the PushAdd instruction.

```
inside the loop...
```

```
Var("a")
PushAdd(1)
Assign("a")
Var("b")
PushAdd(1)
Assign("b")
...
```

As you can see we saved one instruction per incremented variable per iteration in the loop.

To get an even bigger performance increase on the increment_loop.imp program we can enable two superinstructions. As we can see in Figure 3 (PushAdd, AssignPushAdd) the run time was decreased to only 32.7 ms in threaded code. This is an improvement in speed of nearly 30% in threaded code and 10% in non-threaded code.
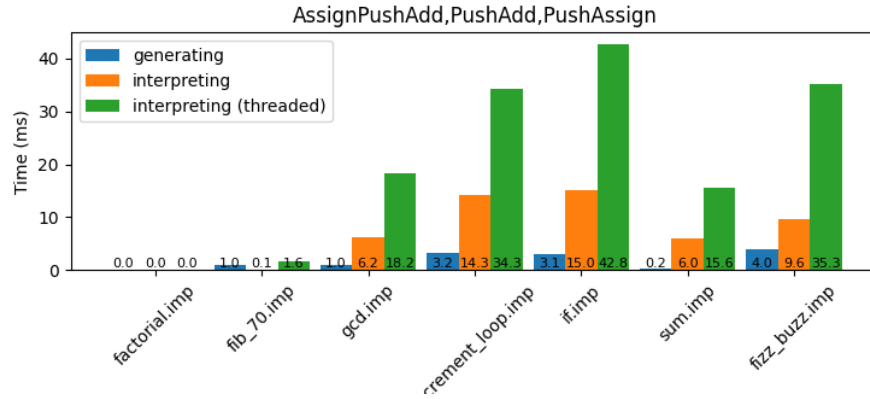
Figure 7: Benchmark results with the superinstructions PushAdd, Assign-PushAdd and PushAssign enabled

When we combine the previously build PushAdd superinstruction with the ASSIGN instruction we can reduce the number of instructions even more to just two instructions per incremented variable.

```
inside the loop...
```

```
    Var("a")
    AssignPushAdd { name: "a", value: 1 }
    Var("b")
    AssignPushAdd { name: "b", value: 1 }
    ...
```

Since none of the test programs results in a generated bytecode that exceeds 100 instructions the differences at generating are minimal but in some cases still notable.

When we compare the generation time of fizz_buzz.imp in Figure 1 and Figure 4 we can see that the time it took to generate the byte code increased by 0.2ms. This increase might seem very little and irrelevant but the generated byte code only consists of 57 instructions. If this would be a big program resulting in thousands of instructions the time of code generation would noticable increase as the time needed to create superinstructions grows linear to the byte code (see Section 4.4.1).

6 Summary

7 Future Work

The last section of this paper will focus on ways to improve this project in the future.

23

Looking at the title of this paper and compare it to the write up we can see that we are missing an essential part. The part of automation.

The idea behind the automation part is to give the user of this program the opportunity to provide a configuration file which can define all sorts of information that then will be used for the generation of the virutal machine interpreter.

Some of the information can be custom superinstructions for example. The idea is to let the user define their own superinstructions and write them down in a syntax we can understand to automatically implement them during generation.

Other ways to improve this project in the future is to think about a different way to model the byte code inside of the Rust code. We have already described why the approach we chose is not the most optimal way in Section 4.2.3.

To further improve the efficiency we can implement more optimization techniques in the future. To name a few optimizations that should be pretty easy to implement at this point are for example 'dead code elimination' as described in Section 1.

Another thing that can be done in the future is to add more superinstructions. For that it would be useful to create more benchmark examples and see what instructions are used frequently to create superinstructions out of them.

References

[1] M. Ertl and David Gregg. The structure and performance of efficient interpreters. *J. Instruction-Level Parallelism*, 5, 11 2003.

[2] M. Anton Ertl. Stack caching for interpreters. page 315–327, 1995.

[3] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen: A generator of efficient virtual machine interpreters. *Softw. Pract. Exper.*, 32(3):265–294, mar 2002.

[4] Roberto Ierusalimschy, Luiz Figueiredo, and Waldemar Celes. The implementation of lua 5.0. *J. UCS*, 11:1159–1176, 01 2005.

[5] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hrițcu, Vilhelm Sjöberg, and Brent Yorgey. *"Logical Foundations"*, volume *"1"* of *"Software Foundations"*. "Electronic textbook", "2023".