TOKYO INSTITUTE OF TECHNOLOGY

MASTER THESIS

# Pyrlang: A High Performance Erlang Virtual Machine Based on RPython

*Author:*
Ruochen HUANG
*14M54036*

*Supervisor:*
Prof. Hidehiko
MASUHARA

*A thesis submitted in fulfillment of the requirements for the degree of Master of Science*

*in the*

Department of Mathematical and Computing Science
Graduate School of Information Science and Engineering

August 29, 2016

# Declaration of Authorship

I, Ruochen HUANG, declare that this thesis titled, "Pyrlang: A High Performance Erlang Virtual Machine Based on RPython" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

TOKYO INSTITUTE OF TECHNOLOGY

# *Abstract*

Masuhara Laboratory
Graduate School of Information Science and Engineering

Master of Science

**Pyrlang: A High Performance Erlang Virtual Machine Based on RPython**

by Ruochen HUANG

In widely-used actor-based programming languages, such as Erlang, sequential execution performance is as important as scalability of concurrency. In order to improve sequential performance of Erlang, we develop Pyrlang, an Erlang virtual machine with a just-in-time (JIT) compiler by applying an existing meta-tracing JIT compiler. In this paper, we overview our implementation and present the optimization techniques for Erlang programs, most of which heavily rely on function recursion. Our preliminary evaluation showed approximately 38% speedup over the standard Erlang interpreter.

# *Acknowledgements*

I would first like to thank my thesis advisor Prof. Hidehiko Masuhara of the department of mathematical and computing science at Tokyo Institute of Technology. The door to Prof. Masuhara office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this paper to be my own work, but steered me in the right the direction whenever he thought I needed it.

I would also like to acknowledge Assistant Prof. Tomoyuki Aotani of the department of mathematical and computing science at Tokyo Institute of Technology as the second reader of this thesis, and I am gratefully indebted to him for his very valuable comments on this thesis.

My sincere thanks also goes to all memebers from PyPy and Pycket projects. Thank you for your kind answers about RPython, meta-tracing JIT and different kinds of optimization approaches on mailing-list and IRC.

I thank my fellow labmates in Programming Research Group: Izumi Asakura, Hirotada Kiriyama, Tomoki Imai, Kenta Fujita, Atsushi Taya, Ryo Okugawa, and Keita Watanabe, for the stimulating discussions and the valuable comments.

Finally, I must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Ruochen Huang

# Contents

# Chapter 1

# Introduction

Erlang (Armstrong and Virding, 1990) is a dynamically-typed, functional and concurrent programming language based on the actor model (Agha, 1986). It is widely used for practical applications that require distribution, concurrency and availability. The application area ranges from telecommunication, banking, electric commerce to instant messaging (Armstrong, 1996), and recently expanding to server side like Cowboy[1], Chicago Boss[2], and MochiWeb[3].

We consider that sequential execution performance in Erlang is as important as scalability of concurrency. In this regard, the two mainstream implementations of Erlang, namely the BEAM virtual machine (or BEAM in short) (Armstrong, 1997) and the HiPE compiler (or HiPE in short) (Johansson et al., 1999b), are either less efficient or less portable. BEAM is a bytecode interpreter, and guarantees bytecode level portability across different platforms. Its sequential execution is however slow due to the interpreter-based execution[4]. HiPE is a static native code compiler, and is faster than BEAM[5]. However, despite of the performance improvement, the compiled code using HiPE loses the compatibility, which means we cannot use it cross-platform. Moreover, users cannot re-compile libraries without source-code into native code using HiPE.

Alternatively, we propose Pyrlang, a virtual machine for the BEAM bytecode with a just-in-time (JIT) compiler. We use the RPython's meta-tracing JIT compiler (Bolz et al., 2009) as a back-end. Although the back-end is primarily designed for imperative programming languages like Python (as known as the PyPy project), Pyrlang achieved approximately 38% speedup over BEAM.

We consider that the implementation cost is also important for JIT compiler research. In this thesis we also compared the implementation cost of different approaches to help to build JIT compiler in Chapter 5. The result showed that meta-tracing JIT has significant benefits because it makes language developers concentrate on interpreter implementation, and then one can generating a tracing JIT compiler without touching any low-level details.

[1] https://github.com/ninenines/cowboy

[2] https://github.com/ChicagoBoss/ChicagoBoss

[3] https://github.com/mochi/mochiweb

[4] According to the Computer Language Benchmarks Game (http://benchmarksgame.alioth.debian.org/), BEAM is slower than C by the factors of 4–95 with 10 benchmark programs.

[5] Though HiPE is known to exhibit largely different performance improvements depending on the types of application programs (Johansson et al., 1999a), it speeds up by the factors from 1.8 to 3.5 according to the benchmark results in a literature (Pettersson, Sagonas, and Johansson, 2002) and our experiments.

**Contributions** The contributions of the thesis can be explained from the two viewpoints: as an alternative implementation of Erlang, and as an application of a meta-tracing JIT compiler to a mostly-functional language.

As an alternative implementation of Erlang, Pyrlang demonstrates a potential of JIT compilation for Erlang[6]. The performance was comparable to an existing static Erlang compiler. This suggests that, by reusing a quality back-end, we could provide Erlang a JIT compiler with a number of modern optimization.

From a viewpoint of tracing JIT compilers, Pyrlang is equipped with a new more strict tracing JIT policy, which focuses on detecting more frequently executed path under conditional branch. In our research, we found that a naive application of a tracing JIT compiler suffers overheads when the compiler chooses less frequent paths. We showed that a new tracing policy reduces the overheads by the factor of 2.9% on average.

**Organization of the Thesis** The thesis is organized as follows. Chapter 2 introduces the instruction set of BEAM as well as an overview of different approaches to help to build a JIT compiler, including meta-tracing JIT. Chapter 3 describes the key design decisions in Pyrlang. Chapter 4 describes several performance optimization we applied to Pyrlang. Chapter 5 evaluates both the performance of Pyrlang by comparing against the existing Erlang implementations, and the implementation cost for different approaches to build a JIT compiler. Chapter 6 discusses related work. Chapter 7 concludes the thesis with discussing future work.

[6]Other than BEAM and HiPE, there are a few attempts to support JIT compilation for Erlang, which we discuss in the later section of the thesis.

# Chapter 2

# Background

## 2.1 Erlang and BEAM Bytecode

The mainstream Erlang implementation compiles an Erlang program to bytecode, and executes on BEAM. We here briefly explain the architecture of the bytecode language by using a few examples.

BEAM is a bytecode-based register machine, whose instruction set includes register transfers (e.g., move), conditional jumps (e.g., is_eq_exact and is_lt_exact), arithmetic operations (expressed as calls to built-in functions like "gc_bif erlang:+/2"), and function calls (e.g., call and call_only). There are three sets of registers, namely X, Y and F, which are denoted as $x(i)$, $y(i)$ and $f(i)$, respectively. The X and F registers store values of any types other than floating point numbers, and values of floating point values, respectively. They are used for passing parameters to and returning results from functions, and can also be used as caller-saved temporary variables. The Y registers are callee-saved, and can be used for storing local variables in a function body. There are instructions to save (allocate_zero) and restore (deallocate) Y registers.

Figures 2.1, 2.2 and 2.3 show three simple functions in BEAM bytecode.

Figure 2.1 shows a function that adds two parameters (from L2) and a code fragment that calls the function with parameters 3 and 5 (from L5). The function expects parameters in registers $x(0)$ and $x(1)$, and returns a result by storing it in $x(0)$. The instruction immediately after L2 (gc_bif2 erlang:+/2) is a built-in function that stores the sum of two registers into a register. To invoke a function, the caller sets parameters on X registers, and then executes the call instruction. As can be seen in the code, the caller and the callee share the X registers.

```
;function my_module:add/2
L2:     ; x(0) := x(0) + x(1)
        gc_bif2 erlang:+/2 x(0) x(1) x(0)
        return
        ...
L5:     move #3 x(0)
        move #5 x(1)
        call L2
        ...
```

FIGURE 2.1: An add function and its invocation in BEAM bytecode

```
;function fact:fact/2
L2:     ; if x(0)==0, jump to L3
        is_eq_exact L3, x(0), #0
        ; x(0) := x(1)
        move x(1), x(0)
        return
L3:     ; x(2) := x(0) - 1
        gc_bif2 erlang:-/2, x(0), #1, x(2)
        ; x(1) := x(0) * x(1)
        gc_bif2 erlang:*/2, x(0), x(1), x(1)
        move x(2), x(0)
        call_only L2 ; tail call
```

FIGURE 2.2: A tail-recursive factorial function in BEAM bytecode

```
;function fact:fact/1
L2:     ; if x(0)==0, jump to L3
        is_eq_exact L3, x(0), #0
        move #1, x(0) ; x(0) := 1
        return
L3:     allocate_zero 1, 1 ;save Y registers
        ; x(1) := x(0) - #1
        gc_bif2 erlang:-/2, x(0), #1, x(1)
        move x(0), y(0); save x(0) to y(0)
        move x(1), x(0)
        call 1, L2          ;non-tail call
        ; x(0) := y(0) * x(0)
        gc_bif2 erlang:*/2, y(0), x(0), x(0)
        deallocate 1 ;restore Y registers
        return
```

FIGURE 2.3: A non-tail recursive factorial function in BEAM bytecode

Figure 2.2 shows a factorial function written in a tail recursive manner, where the second parameter accumulates the product of numbers computed so far. The first instruction from L2 (is_eq_exact) compares two parameters and jumps if they are the same. The last instruction of the function (call_only) is a tail-call. Note that BEAM uses different instructions for tail (call_only) and non-tail calls (call).

Figure 2.3 shows a non-tail recursive factorial function. Since the function multiplies the result from a recursive call by the given argument, it saves the argument ($x(0)$) into a callee-saved register ($y(0)$) before the recursive invocation. The block from L3 saves and restores the Y registers at the beginning and the end of the block, respectively.

## 2.2 Approaches to Help to build JIT Compiler

### 2.2.1 Meta-tracing JIT Compiler

#### Tracing JIT Compiler

A tracing JIT compiler works by (1) detecting a frequently executed instruction (called *a JIT merge point*) in an execution of a program, which is usually a backward jump instruction, (2) then recording a series of executed instructions from the merge point (which we call a *trace*), and (3) compiling the trace to optimized code. When the control reaches the merge point again, the optimized code runs instead of the original one. Since a trace is a straight-line code fragment spanning over multiple functions, the compiler effectively achieves aggressive inlining with low-level optimization like constant propagation.

When the counter of a JIT merge point hits a threshold, the compiler records a trace, which is a series of executed instructions, until the control comes back to the same JIT merge point. The compiler converts the trace into native code by applying optimization like the constant propagation. When a conditional branch remains in a trace, it is converted to a *guard*, which checks the condition and jumps back to the original program when the condition code holds differently from the recorded trace.

In real world programming languages, a trace recorded by a JIT compiler is usually either an iteration of some loop or recursive function, since this kind of structures satisfy the definition of trace once we put a JIT merge point in the entry point of them, and are usually executed frequently enough, make it worth to pick them up and compile into native code.

**False Loop Problem & Two State Tracing** The false loop problem was originally proposed by (Hayashizaki et al., 2012), it is a problem which makes a trace into pieces when there are nested traces where the inner trace is executed more than once.

Consider a recursive function $a$, as Figure 2.4 shows, which calls a very short function $b$ in it body for twice, and recursive $a$ itself again. Here we expect to record the function body of $a$, since it is a tail recursion and we can reuse that trace efficiently. However, in this situation, JIT tracing will start from $b$ since $b$ is executed more frequently than $a$. As a result, trace of $b$ break the body of $a$ into two pieces, it may start from the first invocation, and end up with the second invocation, or it may start at the second invocation, then go to the next iteration of $a$, then end up with the first invocation of $b$. Neither of these two patterns are not good, in fact, $b$ here are totally neither a loop nor a recursion. This is what means a "false" loop.

Currently, there are several approaches that try to solve this problem, like by inspecting function call stack to detect a false loop and eliminate it (Hayashizaki et al., 2012), or by adding another tracing state to get rid of increasing counter, namely *two state tracing*, which is originally proposed by Pycket (Bolz et al., 2014).

Here we introduce the two state tracing since it is relative to our research. The basic idea of two state tracing is to partly distinguish different execution contexts of a function invocation, and therefore make JIT compiler judge a piece of code as a trace more strictly. In Pycket, the previous

```
a() ->
        b(),
        b(),
        a().

b() ->
        ok.
```

FIGURE 2.4: An example to show false loop problem

AST (Abstract Syntax Tree) node is recorded as a part of JIT merge point at every old JIT merge point, and therefore function invocations from different contexts may be considered as different.

Consider previous example with function *a* and *b* again. This time the first invocation of *b* can be regarded as different with the second one, because they have different previous states. As a result, the body of function *a* can be covered by one single trace, in other word, the false loop is eliminated.

We also applied two state tracing in Pyrlang. We introduce the implementation detail in Chapter 4.

#### RPython meta-tracing JIT Compiler

RPython (Ancona et al., 2007) is a subset of Python. It remains most of Python's core datatype like list, dictionary, tuple, and features like OOP, exception handling, closure, but removed some dynamic features like dynamic typing, generator, dynamic field lookup. As as result, types of all variables in a RPython program can be determined statically. With a special compilation tool, namely RPython tool chain, a RPython program can be transformed to different kinds of backend code like Java, CLI, and C, and has a comparable performance with CLI and Java.

RPython is originally developed and used by pypy project to build a high performance python virtual machine, but afterwards grows up to a general-used programming language. RPython has lots of low-level libraries including number datatype, string I/O, socket, random, and so on, which can help language developers implement their own language virtual machine easier. For example, if a developer want to implement big int datatype it their own language, he can just simply reuse the implementation in *rbigint* library, so there is no need to touch any low-level implementation details in this case.

RPython can also support meta-programming. That is because the RPython tool chain only begins collecting type information and transforming code after python's loading time (which is a phrase that Python virtual machine trying to executing every top-level command at each module in order to load those function or class definitions). In other words, developers can fully use Python's dynamic features during the loading time, which remains a good chance to do meta-programming. We also show an example in Chapter 4 about using this feature to implement optimization of datatype representation.

Besides those features, RPython can also provide a tracing JIT compiler to language developer, this technique is called meta-tracing JIT compiler (Bolz et al., 2009; Bolz and Tratt, 2015). We therefore introduce this technique here.

Meta-tracing JIT compilers are tracing JIT compilers that optimize an *interpreter program* (which is the interpreter written by RPython in our case). Their mechanisms for monitoring and tracing an execution of the subject program are the same as the one in general tracing JIT compilers, except for the notion of program locations. While general tracing JIT compilers select a trace from the loops in the interpreter program, meta-tracing JIT compilers do so from the loops in an subject program. To do so, they recognize the *program counter* variable (denoted as *pc* hereafter) in the interpreter, and assign a different execution frequency counter to different *pc* values.

With this extension, the compilers detect frequently executed instruction in the subject program, and record the interpreter's execution until it evaluates the same instruction in the subject program. As a result, the technique enables to build a JIT compiler of a language by writing an interpreter of that language with proper annotations. In the case of Pyrlang, we use a meta-tracing JIT compiler for interpreters written in RPython, a subset of Python. In other words, we write a BEAM bytecode interpreter in RPython.

In the rest of the thesis, we simply refer JIT merge points as locations in subject programs. Except that the program locations are indirectly recognized through variables in the interpreter, the readers can understand the subsequent discussion in the paper as if we are working with a dedicated tracing JIT compiler for BEAM.

### 2.2.2 Compilation to JIT Languages

Instead of generating a target language specific JIT compiler, another approach to build a JIT compiler is compilation to JIT languages.

For the popular dynamic typed languages, there are projects like Jython (*Jython: Python for the Java Platform*) and JRuby (*JRuby, the Ruby Programming Language on the JVM*), which are pure Java implementations for Python and Ruby, respectively. However, The main goal of these projects is to build those dynamic language implementations with better compatibility with Java, such as manipulating Java class directly. As for the performance, unfortunately, although there is an existing JIT compiler in most of modern JVM implementations, which can improve Java program performance very much, performance of such Java implementations of dynamic languages are not as good as their official implementations. One of the most important reasons is due to the mismatch between the dynamic features of such dynamic languages, and the JVM static object modal.

For those popular functional programming languages, there are projects like OCaml-Java (Clerc, 2012) and Kawa (Bothner, 1998), which are pure Java implementations for OCaml and Scheme, respectively. In the case of OCaml-Java, performance data showed that it can be faster than OCaml interpreter, but still slower than OCaml native code compiler. The result

showed OCaml's static typed object is somehow suitable in JVM, and therefore has a comparable performance. On the other hand, however, performance of Kawa is not as good as other Scheme implementations like Scheme48.

We therefore consider the dynamic typing feature is a large factor to effect the performance of compiled Java code.

### 2.2.3 Partial-evaluation-based Meta-compilation

Besides typical JIT compiler, there is another runtime optimization Technology called partial-evaluation-based meta-compilation. This technology was firstly proposed by Truffle project (Würthinger et al., 2012) (Würthinger et al., 2013), which is a Java framework for self-optimizing interpreter. Different from RPython, which use meta-tracing JIT to generate optimized code during the runtime, Truffle uses partial evaluation (Futamura, 1999) technology to optimize "hot" method of target language.

There are two main differences between meta-tracing JIT and partial evaluation. (1) A compilation unit in partial evaluation includes entire control flow of a hot method, while there is no control flow in a trace generated by meta-tracing JIT compiler. (remember that all conditional branches in a trace are reduced to guard instructions) (2) Compilation units in partial evaluation is strictly independent of a concrete execution, while meta-tracing JIT relies on runtime information (such as assumption that some variables are changed rarely during runtime) to generate optimized code.

There are also existing researches to compare meta-tracing JIT and partial evaluation like ("Tracing vs. partial evaluation: Comparing meta-compilation approaches for self-optimizing interpreters"). In this paper authors showed that Truffle's partial evaluation has a comparable performance that RPython's meta-tracing JIT in their experiment target language SOM (Simple Object Machine) (Haupt et al., 2010). However, this paper also showed that much more work is needed to implement a target language in Truffle since language developers need to provide more information to guide optimistic Optimization.

# Chapter 3

# Implementation

This section overviews the design of Pyrlang's JIT compiler, which is embodied as a BEAM bytecode interpreter written in RPython. We first show the representation of data structures in the interpreter, and then describes the design of the dispatch loop (i.e., the interpreter's main loop).

We use C language as our backend for RPython tool chain, since this backend can gain the best performance of the interpreter. As a result, all RPython datatypes are transferred into C datatypes. For example, a RPython integer is a C integer, which means we do get overflow if the integer is large enough.

## 3.1 Representation of Internal Data of the Interpreter

### 3.1.1 Erlang datatype

As Figure 3.1 shows, we represent each Erlang datatype as an object of RPython class. All of these datatypes inherit from a common abstract class, namely *W_Root*, we therefore can realize polymorphism in related functions to handle different kinds of datatypes.

Erlang number datatype including both integer and float-point numbers, both of them inherit from *W_AbstractNumber* class. Integer is implemented by two different RPython object. When the value of integer is little enough, we use *W_IntObject*, which is a wrap of RPython integer. When the value of integer is as large as resulting an overflow, we transfer this *W_IntObject* into a *W_BigIntObject*, which is a wrap of rpython's big int library. Float is simply a wrap of RPython float-point number.

As for other important Erlang datatypes, Erlang atom is object including an index of some atom table. Erlang list is implemented as a linked-list of RPython object. Erlang closure is a RPython object including a pointer to bytecode address of that function, an arity field stroing arity of that function, and a list of *W_Root* object to store free variables. A general Erlang tuple object is a wrap of a RPython list of *W_Root*, for tuples with small size, we use some specialized tuple objects, which we introduce in Chapter 4.

We mark as many fields as possible with RPython *_immutable_fields_* hint function, which tells RPython that these fields can never be changed so RPython JIT compiler can perform further optimization like constant propagation during optimizing recorded trace.

```
class W_Root:
  # makes sure field of a W_Root is atually empty
  _attrs_ = ()

  def clone(self):
    return W_Root()

  def is_equal(self, other):
    # a trick to simulate abstract function
    raise NotImplementedError

class W_AbstractNumberObject(W_Root):
  def abs(self):
    raise NotImplementedError

  def tofloat(self):
    raise NotImplementedError

  def toint(self):
    raise NotImplementedError

  def neg(self):
    raise NotImplementedError

class W_AbstractIntObject(W_AbstractNumberObject):
  pass

class W_IntObject(W_AbstractIntObject):
  _immutable_fields_ = ['intval']
  ...
```

FIGURE 3.1: part of implementations of Erlang datatypes by using RPython

### 3.1.2 Built-In Functions and User-Defined Functions

There are two kinds of functions in Erlang, namely built-in functions and user-defined functions. Built-in functions (BIF) are implemented using native code (that is, RPython code in our research), while user-defined functions are implemented using Erlang code. Each function also belongs to an actual Erlang module. An function without any context namespace belongs to *Erlang* module, implicitly.

As Figure 3.2 shows, We use a RPython class called *BaseBIF* to represent built-in functions. Each built-in function is a subclass of *BaseBIF*, which has a common method called *invoke*. We can get the execution context via arguments past to this *invoke* function. A *unroll_safe* hint is placed at the head of *apply_bif* function, which suggests RPython to unroll loop inside this function (that is, the list-comprehension of operands) to generate better native code during JIT compilation. Unrolling is safe here because we confirm one BIF can only take limited number of operands.

```
class BaseBIF(BaseFunc):
  def invoke(self, args):
    raise NotImplementedError

class AddFunc(BaseBIF):
  def invoke(self, args):
    (a,b) = args
    return a.add(b)

@jit.unroll_safe
def apply_bif(cp, bif, rands, dst_reg):
  # evaluate each operand into actual value
  args = [get_basic_value(cp, rand) for rand in rands]
  # a trick to simulate type casting
  assert isinstance(bif, BaseBIF)
  res = bif.invoke(args)
  # store result into destination register
  store_basereg(dst_reg, res)
```

FIGURE 3.2: Definition of built-in functions and an example of add function

When loading an Erlang module, Pyrlang will firstly parse the module header information to find what BIFs the module needs, than initiate corresponding BIF objects, storing them into an array, then replace each reference of these BIFs in the module with corresponding index in this array, so we can find each BIF with O(1).

For user-defined functions, however, we need to take care of both tail-invocation and nontail-invocation.

Tail-invocation is invoked by a BEAM instruction namely CALL_ONLY. In this situation, similar with any other functional languages, there is no need to handle call-stack, so we simply make the execution jumps to destination address by modifying the value of program counter(*pc*).

Nontail-invocation is invoked by several BEAM instructions such as CALL, CALL_LAST, and CALL_EXT. In this situation, we need to handle the call-stack of user-defined functions by ourselves, that is because there is a stack limitation in RPython function call-stack, while in Erlang, the limitation of call-stack should be as large as the memory size of local machine, which is different with RPython.

### 3.1.3 Instructions and Literal Values

An Erlang program is a module with at least one function. Each Erlang module has head information, including at least (1) a table of all atoms occurred in this module file, (2) a list of functions that are exported from this module so any other module can access them, (3) a table of literal values used in this module.

When executing an Erlang program, an user firstly specify the module file and the entry function, which must be an exported function in that

```
class interpreter:
  ...

  # handler for CALL instruction
  def call(self, frame, label):
    self.call_stack.push(frame)
    return frame[0].label_to_addr(label)
  ...
```

FIGURE 3.3: Definition of built-in functions and an example of add function

```
class Instruction:
  _immutable_fields_ = ['opcode', 'args[*]']
  def __init__(self, opcode, args):
    self.opcode = opcode
    self.args = args
```

FIGURE 3.4: Definition of basic BEAM instruction

module, with valid argument(s). A module may include other modules as well, these modules will be pre-loaded before interpreter starts.

We represent a BEAM bytecode program as an RPython array of instruction objects. As Figure 3.4 shows, an instruction object is an RPython object that contains operands in its fields. The *opcode* field suggests the exact instruction, while *args* field suggests the arguments of this instruction.

Literal values are stored in literal tables, whose indices are used in the operands. We made the following design decisions in order to let the JIT compiler perform obvious operations at compile time. (1) We separately manage the literal table for integers, and the table for other types. This will eliminate dynamic type checking for integer literals. (2) We mark the instruction array and all fields of instruction objects as "immutable." This will eliminate operations for fetching instructions and operands from the generated code.

### 3.1.4 Atoms

We represent an atom by an index of a global atom table, which contains the identifiers of dynamically created atoms. This will make the equality test between atoms constant time.

### 3.1.5 X and F Registers

We use two RPython lists for X and F registers, respectively. We also mark those lists *virtualizable*[1], which is an optimization hint in RPython. This hint encourages the compiler to perform scalar replacement of marked objects, so that they do not need to be allocated, as a result fields reads/writes can be treated as pure data dependencies and their data are ideally kept in registers only.

[1] https://pypy.readthedocs.org/en/release-2.4.x/jit/virtualizable.html

### 3.1.6 Y Registers and Stack Frame

We represent Y registers and the stack frame as a pair of resizable lists with stack pointers. The first list serves as a stack of Y registers whose indices are shifted by its stack pointer. The stack pointer is adjusted by the allocate_zero and deallocate instructions. The second list serves as a stack of return addresses. The runtime initially constructs those two lists with fixed lengths, yet re-allocates a new lists with twice length of the current one when the stack pointer reaches to the end of either list.

Our representation differs from a linked-list of frames, which is found in typical implementations of interpreters. The rationales behind our representation are as follows. (1) We use single list a fixed-length (yet resizable) list for avoiding allocation overheads of frames that were required at every function invocation in the linked-list representation. (2) We separately manage the local variables and the return addresses so as to give static types to the return addresses.

## 3.2 Dispatch Loop

The core of the BEAM bytecode interpreter is a single loop called *the dispatch loop*, which fetches a bytecode instruction at the program counter, and jumps to the *handler* code that corresponds to the instruction. A handler performs operations of the respective instruction, such as moving values between registers, performing arithmetic operations, and changing the value of the program counter.

The design of the dispatch loop is similar to typical bytecode interpreters, except for the following three Pyrlang specific points. (1) We use a local variable for managing the program counter, which is crucial for the JIT compiler to eliminate accesses to the program counter. (2) The handler for call_only merely changes the program counter value as the instruction is for tail calls, which effectively realizes tail call elimination. (3) The dispatch loop yields its execution for realizing the green threading. To do so, the loop has a yield counter, and the handlers of some instructions (those can become an end of a trace) terminates the dispatch loop when the counter is decremented to zero.

The green thread yield policy is different from Ericsson's implementation, which decrease the yield counter at each function invocation (including customized functions and built-in functions). This is because we want to limit the switch-out behavior happening only at the boundary of trace. Of course, this approach might bring "unfairness" between different process execution. However, in real world, this "unfairness" is usually considered acceptable compared with the performance improvement, and widely used in many main-stream programming language implementations such as PyPy and Ruby.

To complete context switching, we implement a scheduler to help manage different green threads (also known as Erlang processes), which are objects that include a dispatch loop of their own.

Figure 3.7 shows part of RPython code of our scheduler. Each Erlang process has a priority when created, the scheduler handles different priority in order. A Erlang process is context switched by one of the next three conditions: (1) process terminates because all of its code has been executed, (2)

process hangs out because its event listener does not receive any new message, (3) process consumes all yield counter value and therefore is paused and switched out.

```
class CallStack:
  def __init__(self):
    self.addrs = [(None, 0)] * init_y_reg_size
    self.addr_top = -1
    self.addr_size = jit.hint(init_y_reg_size, promote=True)
    ...

  def push(self, val): # (cp, pc)
    self.addr_top += 1
    if self.addr_top >= self.addr_size:
      self.addrs = self.addrs + [(None, 0)] * self.addr_size
      self.addr_size = self.addr_size << 1
    self.addrs[self.addr_top] = val

  def pop(self):
    val = self.addrs[self.addr_top]
    self.addr_top -= 1
    return val

  ...

class Y_Register(AbstractRegister):
  def __init__(self):
    self.vals = [None] * init_y_reg_size
    self.val_top = -1
    self.val_size = jit.hint(init_y_reg_size, promote=True)

  @jit.unroll_safe
  def allocate(self, n, init_val = None):
    self.val_top += n
    while self.val_top >= self.val_size:
      self.vals = self.vals + [None] * self.val_size
      self.val_size = self.val_size << 1
    if init_val:
      index = self.val_top
      for i in range(n):
        self.vals[index] = init_val
        index -= 1

  def deallocate(self, n):
    self.val_top -= n

  def get(self,n):
    idx = self.val_top - n
    assert idx >= 0
    return self.vals[idx]

  def store(self, n, val):
    self.vals[self.val_top - n] = val
```

FIGURE 3.5: Definition of Y registers and the stack frame

```
# the greens are the position keys
driver = jit.JitDriver(greens=['bytecode', 'pc'], reds=[...])
reduction = 2000 # the yield counter
...

while(True):
  # to tell jit where is the head of the dispatch loop
  driver.jit_merge_point(bytecodes, pc, ...)
  instr = bytecodes[pc];
  if instr == 'GC_BIF2':
    ...
    pc += 1
  elif instr == 'CALL':
    pc = bytecodes[pc+1]
    ...
    # to tell jit where should it put a counter
    driver.can_enter_jit(bytecodes, pc, ...)
    reduction -= 1
    if reduction < 0:
      ...
      break
  ...
```

FIGURE 3.6: Implementation of dispatch loop

```
class scheduler:
  ...
  def schedule(self):
    low_skip_times = 0
    while True:
      while not self.max_queue.empty():
        self._handle_one_process_from_queue(self.max_queue)
      while not self.high_queue.empty():
        self._handle_one_process_from_queue(self.high_queue)
      if self.normal_queue.empty():
        # we now still only have one scheduler so here
        # we just simply terminate the whole system
        break
      else:
        process = self.normal_queue.pop()
        # for low priority process:
        # skipping a low priority process
        # for a number of times before executing it.
        if process.priority == constant.PRIORITY_LOW:
          if low_skip_times >=
            constant.LOW_PRIORITY_PROCESS_SKIP_TIMES:
            self._handle_one_process(self.normal_queue, process)
            low_skip_times = 0
          else:
            low_skip_times += 1
        # for normal priority process
        else:
          self._handle_one_process(self.normal_queue, process)

  def _handle_one_process(self, queue, process, msg=None):
    state = process.execute(self.is_single_run,
            self.reduction, msg)
    if state == constant.STATE_SWITH:
      queue.append(process)
    elif state == constant.STATE_TERMINATE:
      pass
    elif state == constant.STATE_HANG_UP:
      if process.mail_box.is_empty():
        process.is_active = False
      else:
        self.push_to_priority_queue(process, process.priority)
```

FIGURE 3.7: Implementation of scheduler

# Chapter 4

# Optimization

## 4.1 Optimizing Data Representation Based on RPython Meta-programming

As we mentioned in chapter 2, thanks to the transformation mechanism of RPython tool chain, we can achieve meta-programming by generating any class/function definitions during the loading time using any Python dynamic features.

Here we give an example that we use RPython Meta-programming to implement BEAM tuple datatype representation optimization.

Similar with tuple datatype in Python and Lisp, a tuple datatype in Erlang is a nameless data record, whose fields are all immutable, and can be any type. As figure 4.1 shows, a naive implementation of a tuple object can be considered as an object with an array field, which stores pointers to its content objects. Since a tuple object can have arbitrary length, the array field is allocated dynamically, then it can store the content object pointers.

Due to that array field, two overheads can be considered in this implementation. (1) overhead when allocating list field. (2) overhead when accessing tuple field, since it has to be accessed indirectly by first accessing the array filed, and accessing content object with its pointer.

On the other hand, we observed that, for most Erlang programs, programmers trend to use a tuple object with a small size (say length less than 5), which gives us chance to eliminate overheads below in this situation. That is, to pre-define small, fix-sized tuple object class, and therefore we can eliminate the array field and store content objects as the fields in a tuple object (as figure 4.2 shows).

To do so, we firstly define a class factory function to help us to generate specialised tuple class with fix-sized (that is *tuple_factory* in figure 4.4). Although this function uses many Python dynamic features, like dynamic class definition and *getattr*, *setattr*, it is still valid in RPython since we only invoke this function during the loading time(the initialization code of *specialized_tuples* at top level).

Then we can dispatch the argument about size in PUT_TUPLE(the BEAM instruction used to create new tuple object), and prepare corresponding specialised tuple object for it. For those general-sized tuple object, we still use the naive implementation in figure 4.3.

## 4.2 Two State Tracing

In order to overcome the false loop problem, we also applied Pycket's two state tracing to Pyrlang. Different with Pycket's AST interpreter, Pyrlang is

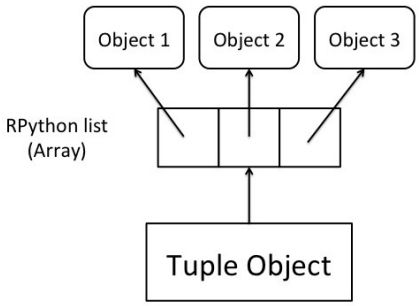


FIGURE 4.1: A naive implementation of BEAM tuple datatype

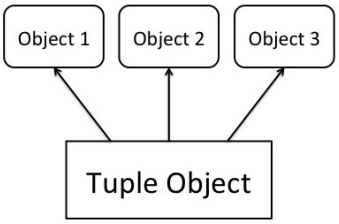


FIGURE 4.2: Optimized implementation by RPython meta-programming

```
class Tuple(AbstractTuple):
  def __init__(self, values):
    self.values = values

  def element_at(self, index):
    return self.values[i]
...
```

FIGURE 4.3: code of naive implementation of BEAM tuple datatype

```
def tuple_factory(tuple_size):
  class cls(AbstractTuple):
    def __init__(self, values):
      for i in range(tuple_size):
        setattr(self,
          'value%s'%i, values[i])

    def element_at(self, index):
      for i in range(tuple_size):
        if i == index:
          return getattr(self,
            'value%s'%i)
    ...
    return cls
specialised_tuples =
  [tuple_factory(i)
    for i in range(MAX_SIZE)]
```

FIGURE 4.4: code of optimized implementation by RPython meta-programming

```
# the greens are the position keys
driver = jit.JitDriver(greens=['bytecode', 'pc',
       'caller_pc'], # consider it as position key, too
       reds=[...])
reduction = 2000 # the yield counter
...

while(True):
  # to tell jit where is the head of the dispatch loop
  driver.jit_merge_point(bytecodes, pc, caller_pc, ...)
  instr = bytecodes[pc];
  if instr == 'GC_BIF2':
    ...
    pc += 1
  elif instr == 'CALL':
    caller_pc = pc
    pc = bytecodes[pc+1]
    ...
    # to tell jit where should it put a counter
    driver.can_enter_jit(bytecodes, pc, caller_pc, ...)
    reduction -= 1
    if reduction < 0:
      ...
      break
  ...
```

FIGURE 4.5: Implementation of two state tracing in Pyrlang

a BEAM bytecode interpreter. In this situation, we implement it by using the program counter as well as the caller's address (we refer as *caller-pc* hereafter) as the interpreted program's location.

Figure 4.5 shows the implementation of two state tracing in Pyrlang. The *caller_pc* is updated every time right before *pc* jumps to another address caused by function invocation. As a result, the loop judging rule of JIT compiler becomes "it's a loop only when both *pc* and *caller-pc* are same as before, respectively".

## 4.3 Finer-grained Path Profiling

### 4.3.1 The False Path Problem

A tracing JIT compiler sometimes chooses an execution path as a compilation target, even if it is not frequently executed. We call this problem *the false path problem*, as an analogy to the false loop problem (Hayashizaki et al., 2012).

One of the causes of the false path problem is mismatch between profiling and compilation. A tracing JIT compiler selects the first execution path executed from a merge point whose execution counter exceeds a threshold. When there are conditional branches after the merge point, the selected path can be different from the one that is frequently executed.

```
;function cd:cd/1
L2:
        is_eq_exact L3, x(0), #1        A
        move #10, x(0)                  }B
        call_only L2                    }
L3:
        gc_bif2 erlang:-/2, x(0), #1, x(0)  }C
        call_only L2                        }
```

FIGURE 4.6: A countdown function which restarts itself from 10

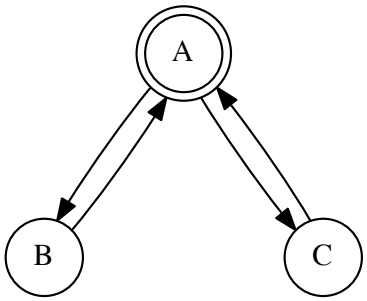


FIGURE 4.7: Control flow graph of the count-down function. (The alphabets in the nodes correspond to the basic blocks in Figure 4.6. The doubly-circled node denotes the JIT merge point.)

When a false path is selected and compiled, it puts a considerable amount of performance penalty on the frequently executed paths that share the same merge point. This is because the execution from the merge point has to follow the compiled false path, and then frequently fails; i.e., a conditional branch goes to a different target from the compiled path. Upon a failure, the runtime needs to reconstruct an intermediate interpreter state.

### 4.3.2 An Naive Profiling Policy for Functions

A naive profiling policy for functional programs can cause the false path problem. Let us illustrate this by using a simple concrete program after we introduced a naive profiling policy.

For functional programs where loops are realized by recursive function calls, a naive profiling policy places merge points at the beginnings of functions. In fact, we used this policy in our first implementation, which we refer as *pyrlang-naive* in the rest of the paper. Technically, pyrlang-naive actually marks call (non-tail function invocation), call_only (tail function invocation), and return as JIT merge points. Since we have tail call elimination as introduced in Chapter 3, tail recursions in Pyrlang are similar with typical loops in an imperative language, as a tracing JIT compiler expects.

Figures 4.6 and 4.7 show a countdown function in the BEAM bytecode

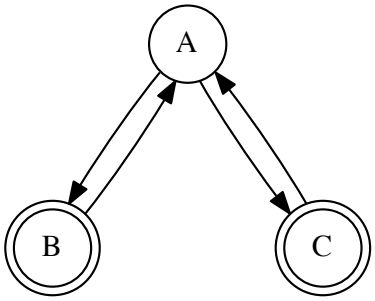


FIGURE 4.8: Control flow graph of the countdown function for pattern matching tracing

with its control flow graph. The function infinitely repeats counting numbers from 10 to 1. While imperative languages realize the computation double nested loops, functional languages typically realize it a recursive function with a conditional branch as can be seen in the control flow graph. In this case, node B is executed one out of ten iterations.

The two loops, namely A–B–A and A–C–A in the control flow graph, share the single JIT merge point, namely A. This means that the compiler has 10% of chance to select the less frequently path (i.e., A–B–A). Then, the subsequent executions from A use the compiled trace for the path A–B–A, and fail 9 out of 10 cases.

### 4.3.3 Pattern Matching Tracing

We propose an improved policy for tracing called *pattern matching tracing*. The basic idea is to place JIT merge points on the destinations of conditional branches, instead of the beginnings of functions so as to distinguish different paths as different traces. For the countdown function, we place JIT merge points on the target nodes of conditional branches, namely B and C as in the control-flow graph shown in Figure 4.8.

With this policy, the compiler will select more frequently executed paths because the merge points are placed in all branch targets. In the example, the merge point C will hit the threshold before B will, and the path starting from the next execution of C (i.e., C–A–C) will be compiled first.

We implemented this policy in the Pyrlang's dispatch loop by marking conditional branch destinations, instead of function entries, as JIT merge points.

```
# the greens are the position keys
driver = jit.JitDriver(greens=['bytecode', 'pc',
      'caller_pc'], # consider it as position key, too
      reds=[...])
reduction = 2000 # the yield counter
...

while(True):
  # to tell jit where is the head of the dispatch loop
  driver.jit_merge_point(bytecodes, pc, caller_pc, ...)
  instr = bytecodes[pc];
  if instr == 'GC_BIF2':
    ...
    pc += 1
  elif instr == 'CALL':
    caller_pc = pc
    pc = bytecodes[pc+1]
    ...
    # this time we don't invoke can_enter_jit here
    reduction -= 1
    if reduction < 0:
      ...
      break
  elif instr == 'IS_EQ_EXACT':
    v1 = evaluate(bytecodes[pc+1])
    v2 = evaluate(bytecodes[pc+2])
    label = evaluate(bytecodes[pc+3])
    if v1.is_equal_exact(v2):
      pc += 4
      ...
    else:
      pc = label_to_addr(label)
      ...
      # we invoke it at every conditional
      # branch instruction
      driver.can_enter_jit(bytecodes, pc, ...)
  ...
```

FIGURE 4.9: Implementation of pattern matching tracing in Pyrlang

# Chapter 5

# Evaluation

## 5.1 Performance Evaluation

### 5.1.1 Benchmark Environment

We evaluate the performance of Pyrlang and its optimization technique with subsets of two benchmark suites. The one is the Scheme Larceny benchmark suite[1] that is translated by the authors from Scheme to Erlang. The other is the ErLLVM benchmark suite[2], which is developed to evaluate the HiPE LLVM backend. Since the current implementation of Pyrlang supports a limited set of primitives, we excluded programs that test concurrency, binary data processing, and modules using the built-in functions that are not yet implemented in Pyrlang. Also, currently there is an unsolved bug in Pyrlang which causes the execution crashing when dealing with float datatype during context switching, so we have to excluded benchmark programs related to it like fibfp and fpsum, too.

We evaluate three different versions of Pyrlang, namely (*pyrlang*) the version using pattern matching tracing, (*pyrlang-two-state*) the version using only two state tracing, which is the same as *pyrlang* except JIT merge points are placed in function entities rather than functional branch destinations, (*pyrlang-naive*) the version using only naive tracing policy, as we introduced in Section 4.3.2.

We emphasize that Pyrlang does not apply low-level optimization such as calling convention for the x86 architecture (Pettersson, Sagonas, and Johansson, 2002), and the local type propagation (Sagonas et al., 2003) that are used in BEAM or HiPE. Furthermore, we use the original BEAM instruction set for the ease of implementation, unlike BEAM which internally uses superinstructions (Hausman, 1997, Section 3.15) in order to reduce interpretation overheads.

All the benchmark programs are executed on a 1.87 GHz Intel Core i7 processor with 8 MB cache memory and 8 GB main memory, running GNU/Linux 14.04. The version of the BEAM runtime and HiPE is Erlang R16B03, with the highest optimization option (`-o3`) for HiPE. The backend of Pyrlang is RPython revision b7e79d170a32 (timestamped at 2016-01-13 04:38).

Each program is executed inside a loop, whose number of iterations is manually selected so that the loop runs for at least 5 seconds. The benchmark results in this section are indicated by the execution times relative to the ones with the BEAM runtime.

[1] https://github.com/pnkfelix/larceny-pnk/tree/master/test/Benchmarking/CrossPlatform/src
[2] https://github.com/cstavr/erllvm-bench/tree/master/src

### 5.1.2 Overall Performance

Figure 5.1 shows the execution times of 30 programs in the benchmark suites, executed with the three versions of Pyrlang, HiPE and the BEAM runtime. The height of each bar shows the relative time to the execution time with BEAM. The rightmost 4 bars (geo_mean) are the geometric means.

As can be seen in Figure 5.1, pyrlang-match is 38.3% faster than the BEAM, yet still 25.2% slower than HiPE. With some benchmark programs that are relatively non-trivial, such as deriv, pi, nucleic, and qsort, pyrlang-match is the fastest among 5 implementations. There is a group of the programs (*string*, *length_c*, *pseudoknot*, *ring*, *stable*, *sum*, *zip*, *zip3*, and *zip_nnc*) where Pyrlang is slower than BEAM and HiPE. We conjecture that most of the slow cases are caused by the overhead of Erlang datatype allocation. This is expected since we simply use RPython objects to implement datatypes such as lists, function closures without any further low-level optimization. The programs *ring* and *stable* are the only two benchmark programs using green threads in our benchmark suite, which indicate room of optimization around thread implementations.

### 5.1.3 Effect of Pattern Matching Tracing

#### Improvements by Pattern Matching Tracing

In this section, we evaluate effectiveness of our proposed pattern matching tracing by comparing execution times of benchmark programs with three versions of Pyrlang, namely the one with pattern matching tracing (pyrlang), one with two state tracing (pyrlang-two-state) and naive tracing (pyrlang-naive). In pyrlang-naive, we mark merely 3 kinds of instructions as JIT merge points, namely call (non-tail invocation), call_only (tail invocation), and return. Also, in this version, the JIT merge points are identified by only *pc* but not by *caller-pc*, which we introduced in Chapter 4.

As we can see, pyrlang is 1.3% and 2.9% faster than pyrlang-two-state and pyrlang-naive on average, respectively.

There are programs where pyrlang is significantly slower, namely *sum* and *pseudoknot*. Sum is a program using two Erlang built-in functions, namely lists:seq and lists:sum, to generate a long Erlang list and to calculate the sum of the elements in the generated list, respectively. Pseudoknot is a program that generates pseudoknot matrix from a database of nucleotide conformations. With sum, we found Pyrlang only generates a trace for lists:sum, but not for lists:seq, which contributed to the significant performance degradation. Currently we are not clear why the lists:seq is missed, which remains as the future work to be resolved. With pseudoknot, we found a loop of a long series of conditional branches that serves as a dispatch loop. This control structure created overly many JIT merge points with Pyrlang-match, though only a few of them are compiled. We conjecture that we can reduce the number of JIT merge points so as to improve performance.

Also, there are two programs where Pyrlang is significantly faster, namely *deriv* and *qsort*. Deriv is a program that performs symbolic derivation of mathematical expressions. Qsort is a program that performs quick-sorting of a number sequence. We observed that, in both benchmark programs, Pyrlang generated much longer traces. In deriv, the longest compiled trace

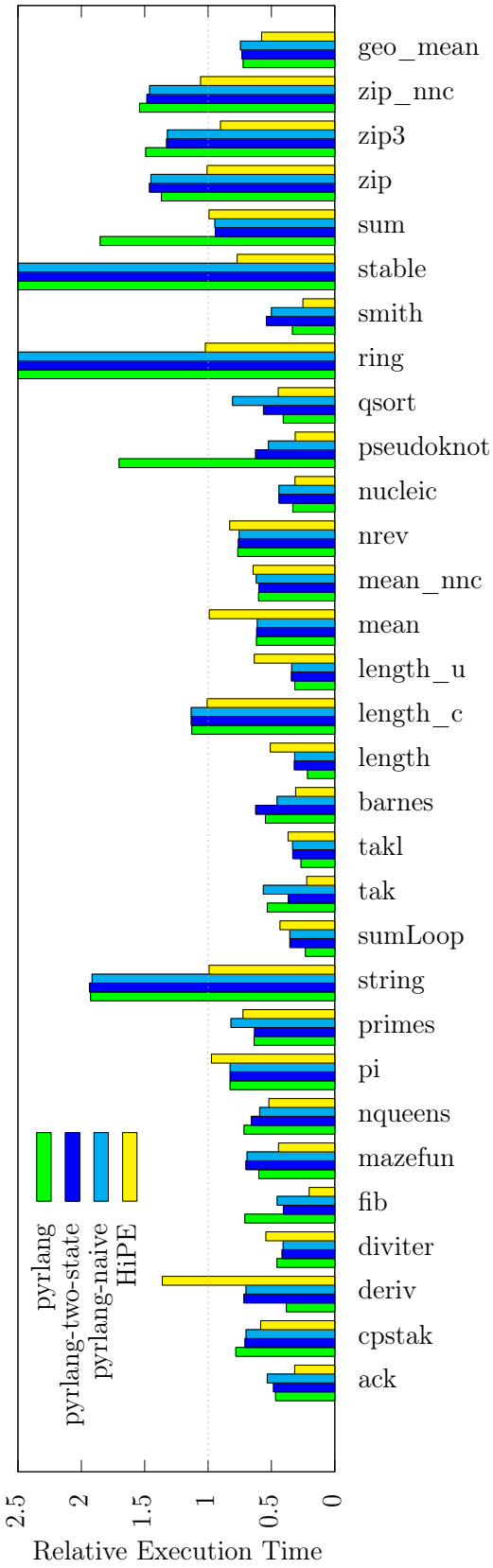


FIGURE 5.1: Execution Times with pyrlang, pyrlang-two-state, pyrlang-naive and HiPE, relative to BEAM (the shorter is the better)

TABLE 5.1: A comparison among different approaches to build a JIT compiler

| | interpreter implementation | optimization hints | JIT compiler implementation |
|---|---|---|---|
| manual implementation | High | High | High |
| meta-tracing JIT | High | Low | None |
| compilation to JIT languages | Low | None | None |
| partial-evaluation-based meta-compilation | High | High | None |

corresponds to an expression of a specific shape (namely multiplication of variables). We presume that the trace is significantly effective as the program has many sub-expressions in the same shape. In Qsort, the longest compiled trace corresponds to a loop in partition function. In fact, Pyrlang records a whole execution of the partition for a short sequence. We have to point out that these two cases heavily depend on the program inputs, which might not be as effective when inputs can vary among executions. In other words, those results indicate that Pyrlang with the pattern matching tracing is quite effective when a program follows a few specific paths in a complicated control structure.

## 5.2 Implementation Cost Comparison

In this section, we compare several approaches help to build a JIT compiler. To to honest, it's very hard to evaluate these approaches in a quantitative way, since the implementation instances (that is, the different language implementation with a JIT compiler) have different integrity, using different programming languages and libraries.

Instead, we compare these approaches in a qualitative way. To do so, we separate the implementation with three parts as (1) interpreter implementation, (2) optimization hints, (3) JIT compiler implementation.

The interpreter implementation means the work spent to implement an AST/bytecode interpreter, including semantics of each node/instruction, built-in function, and scheduler (if that language support green-thread). The optimization hints mean the work spent to help a JIT compiler to do profiling and generate hotspot efficiently, for example, the JIT hints in meta-tracing compiler. The JIT compiler implementation means the work spent to implement a JIT compiler itself.

We also compare the traditional way to implement a language implementation with JIT compiler (that is, implementing each of those three parts manually), such as any modern JVM implementation, as a baseline.

Table 5.1 shows the comparison of different approaches to build a JIT compiler.

The manual implementation has the most implementation cost, trivially.

The meta-tracing JIT requires developers to rebuilt a language virtual machine by using RPython. As a result, developers need to implement each component as a normal virtual machine including interpreter, scheduler, built-in functions and so on. On the other hand, however, developers only need to insert a few JIT hint functions in their implementation code, so that RPython can generate a JIT compiler automatically, which means developers do not need to a new JIT compiler again.

Approach of compilation to JIT languages has the least implementation cost, because developers only need to use a JIT languages, such as Java, to implement target languages. During the implementation, however, many existing features and libraries, like scheduler and many built-in functions, in a JIT languages can also be reused in target language easily. There is also no need to add optimization hint or implement a JIT compiler again, because the JIT language implementation has already taken care of it. However, we also emphasize that, this approach can not really improve implementation performance, many implementations are even slower than the official one, instead, the main goal of these projects is to build a language implementation with better compatibility with the JIT languages, such as Java.

The partial-evaluation-based meta-compilation requires developers to rebuilt a language virtual machine by using using framework like Truffle. Besides it, developers need to provide both optimization hints for interpreter behavior and the compilation time. That is because the optimization of partial evaluation is performed strictly independent of actual execution, and therefore developers need to directly the partial evaluator to deal with multiple values. As a result, the implementation cost of optimization hints is considered higher than meta-tracing JIT. Note that the heavy optimization hints are not an entire disadvantage since it does help partial-evaluation-based meta-compilation to realize a relatively better performance.

# Chapter 6

# Related Work

PyPy (Bolz et al., 2009), an alternative implementation of Python, is the primary target system of the meta-tracing JIT compiler in RPython. It is reported to be faster than an interpreter implementation (CPython) by the factor of 7.1[1] while achieving high compatibility. This suggests that the meta-tracing JIT compiler is a realistic approach to provide an efficient and highly compatible language implementation with a JIT compiler. The meta-tracing JIT compiler's design is implicitly affected by the language features of PyPy (or Python), such as a stack-based bytecode language and use of loop constructs in source programs.

Pycket (Bolz et al., 2014) is an implementation of Racket runtime system based on the meta-tracing JIT compiler in RPython. Similar to Erlang, Racket is a Scheme-like functional programming language, in which user programs use recursive calls for iterations. It proposes a two-state-tracing (Bolz et al., 2014), which is a light-weight solution to the false loop problem (Hayashizaki et al., 2012). The basic idea of two-state-tracing is to record the previous abstract syntax tree node of the node in a function head, and use both previous node (1st state) and current node (2nd state) to identify a JIT merge point. It means a function head node in a control flow graph is duplicated. The pattern matching tracing extends the two-state-tracing by moving the 2nd state from a function head to a conditional jumping destination. By this approach, in addition to duplicate the function head nodes in a control flow graph, we also duplicate the nodes of conditional jumping destinations.

BEAMJIT (Drejhammar and Rasmusson, 2014) is a tracing JIT compiler implementation for Erlang. It extracts the basic handler code for each BEAM instruction from the BEAM runtime. The extracted handler code is then used to construct the content of a trace. The JIT therefore can be integrated in the BEAM runtime with high compatibility. The implementation work is quite different between our work and BEAMJIT because we already have a JIT compiler provided by RPython that almost for free, and target to build a BEAM VM using RPython that can match best to RPython JIT compiler, while BEAMJIT uses BEAM VM with full compatibility, and tries to build a JIT compiler that can match best to existing BEAM VM. BEAMJIT is reported 25% reduction in runtime in well-behaved programs.

ErLLVM (Sagonas, Stavrakakis, and Tsiouris, 2012) is a modified version of HiPE that uses LLVM as its back-end instead of the original code generator. Similar to Pyrlang, it uses an existing compiler backend, but it inherits other characteristics, like ahead-of-time compilation, from HiPE. It is reported that ErLLVM has almost the same performance as HiPE.

[1] According to the data from http://speed.pypy.org

PyHaskell (Thomassen, 2013) is another functional programming language implementation that uses the RPython meta-tracing JIT compiler. Similar to Pyrlang, it is implemented as an interpreter of an intermediate language (called the Core language). To the best of the authors' knowledge, current PyHaskell supports a few primitives, and still slower than an existing static compiler, GHC in most cases.

HappyJIT (Homescu Andrei, 2011) is a PHP implementation that uses the RPython meta-tracing JIT compiler. Its stack representation is a pre-allocated array of RPython objects, similar to the Y registers in Pyrlang. However, HappyJIT is slower than Zend Engine for the programs mainly performing recursive function calls. As far as the authors know, HappyJIT does not have a tracing policy specialized to recursive functions.

Marr and Ducasse implemented two implementations of SOM by using meta-tracing JIT and partial-evaluation-based meta-compilation, respectively in ("Tracing vs. partial evaluation: Comparing meta-compilation approaches for self-optimizing interpreters"). In this paper, they compared these two approaches with the respects of performance and implementation cost. According to it, meta-tracing JIT shows less implementation cost, while partial-evaluation-based meta-compilation is 30.4% faster than meta-tracing JIT.

# Chapter 7

# Conclusions and Future Work

We proposed Pyrlang, a virtual machine for the BEAM bytecode language with a meta-tracing just-in-time (JIT) compiler. At the high-level view, we merely needed to write a BEAM interpreter in RPython in order to adapt the compiler for Erlang. We however needed to make careful decisions in the interpreter design for achieving reasonable runtime performance. We also proposed an optimization technique called the *pattern matching tracing* for performance improvement. With the technique (and the two state tracing for Pycket), the compiler can select longer traces (which usually give better performance) by distinguishing different branches in a loop.

Our current implementation runs micro-benchmark programs 38.3% faster than the standard BEAM interpreter. Though it is still slower than HiPE in some benchmark programs, we believe that a JIT compiler with our approach can achieve similar level of performance by introducing further low-level optimization.

We also emphasize that most of the implementation decisions of Pyrlang in section 3 using RPython can be also easily applied to any other functional programming language implementations, because most functional programming languages share the similar feature in common.

Pattern Matching Tracing is also a general approach to improve tracing JIT policy, it focus on determining the path selection below some conditional branches, which means it can be also applied to any other tracing JIT compiler.

Our implementation is has several points of improvements and extensions as discussed below.

**Improvement of Pattern Matching Tracing:** As we explained in Chapter 3, there are programs that are poorly optimized with the pattern matching tracing. While we identified the causes of overheads for some programs, we need to collect more cases and generalize the causes of overheads so that we can improve the strategy of trace selection.

**List Optimization:** Our experiments showed that Pyrlang performs more poorly than BEAM and HiPE with programs that allocate a large amount of objects (e.g., very long lists). While we are still investigating the underlying memory manager's performance in RPython, we plan to introduce well-known optimization for functional style list processing, such as cdr-coding (Li and Hudak, 1986) and list unrolling (Shao, Reppy, and Appel, 1994). We also consider to use the live variable hints in the BEAM bytecode during garbage collection.

**Compatibility:** There are still data types and operators that need to be implemented in Pyrlang. Those data types include bit strings and binaries. Though it is a matter of engineering, an (semi-)automated approach would be helpful to ensure compatibility.

# Appendix A

# Benchmark Program List

**ack** A version of the Ackermann function.

**cpstak** *tak* function with continuation-passing style.

**deriv** A Gabriel benchmark dealing with symbolic differentiation.

**diviter** A Gabriel benchmark dividing 1000 by 2 using lists as a unary notation for integers.

**fib** Calculating fibonacci number using doubly recursive computation.

**mazefun** Constructs a maze on a rectangular grid using purely functional style.

**nqueens** Computes the number of solutions to the 13-queens problem.

**pi** A bignum-intensive benchmark that calculates digits of pi.

**primes** Computes the primes less than 1000.

**string** Tests of ++ and *lists:sublist*.

**sumLoop** Sums the integers from 0 to argument, with tail-invocation style.

**tak** A triply recursive integer function related to the Takeuchi function.

**takl** A *tak* function but using lists to represent integers.

**barnes** Barnes-Hut algorithm for computing gravity against a brute-force direct summation.

**length** Computes the length of a list, with tail-invocation style.

**length_c** Computes the length of a list, with BIF implementation.

**length_u** Computes the length of a list, with an unrolled tail-recursive function.

**mean** Computes mean of a list with integer elements, using BIF *lists:sum* and *lists:duplicate*.

**mean_nnc** Computes mean of a list with integer elements, using customized *sum* and *duplicate* functions with tail-invocation style.

**nrev** Reverses a list using list concatenation.

**nucleic** A modified version of the program described in the paper (Feeley, Turcotte, and Lapalme, 1994).

**pseudoknot** Computes the 3D structure of a nucleic acid.

**qsort** Quick sort with the first element of list as pivot.

**ring** Creates N processes in a ring and sends a message round the ring M times so that a total of N * M messages get sent.

**smith** The Smith-Waterman DNA sequence matching algorithm.

**stable** Stable marriages problem with n men and n women.

**sum** Sums the integer from 1 to N using *lists:sum*.

**zip** Generates and sums a list from 2 lists using *lists:sum*, *lists:map*, *lists:zipwith*, *lists:seq*, and *lists:duplicate*.

**zip3** Generates and sums a list from 3 lists using *lists:sum*, *lists:map*, *lists:zipwith3*, *lists:seq*, and *lists:duplicate*.

**zip_nnc** Generates and sums a list from 2 lists using *lists:map*, *lists:zipwith2*, and manual implemetation of *sum*, *duplicate*, and *seq*.

# Bibliography


Agha, Gul (1986). *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge.

Ancona, Davide et al. (2007). "RPython: a step towards reconciling dynamically and statically typed OO languages". In: *Proceedings of the 2007 symposium on Dynamic languages*. ACM, pp. 53–64.

Armstrong, Joe (1996). "Erlang: a Survey of the Language and its Industrial Applications". In: *Proceedings of the symposium on industrial applications of Prolog (INAP96)*, pp. 16–18.

— (1997). "The development of Erlang". In: *Proceedings of International Conference on Functional Programming '97*. ACM, pp. 196–203.

Armstrong, Joe L and SR Virding (1990). "Erlang: an experimental telephony programming language". In: *Proceedings of XIII International Switching Symposium*, pp. 43–48.

Bolz, Carl Friedrich and Laurence Tratt (2015). "The impact of meta-tracing on VM design and implementation". In: *Science of Computer Programming* 98, pp. 408–421.

Bolz, Carl Friedrich et al. (2009). "Tracing the meta-level: PyPy's tracing JIT compiler". In: *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ACM, pp. 18–25.

Bolz, Carl Friedrich et al. (2014). "Meta-tracing makes a fast Racket". In: *Proceedings of Workshop on Dynamic Languages and Applications*. URL: http://www.lifl.fr/dyla14/.

Bothner, Per (1998). "Kawa: compiling dynamic languages to the Java VM". In: *Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association, pp. 41–41.

Clerc, Xavier (2012). "OCaml-Java: OCaml on the JVM". In: *International Symposium on Trends in Functional Programming*. Springer, pp. 167–181.

Drejhammar, Frej and Lars Rasmusson (2014). "BEAMJIT: a just-in-time compiling runtime for Erlang". In: *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*. ACM, pp. 61–72.

Feeley, Marc, Marcel Turcotte, and Guy Lapalme (1994). "Using Multilisp for solving constraint satisfaction problems: an application to nucleic acid 3D structure determination". In: *Lisp and symbolic computation* 7.2-3, pp. 231–247.

Futamura, Yoshihiko (1999). "Partial evaluation of computation process–an approach to a compiler-compiler". In: *Higher-Order and Symbolic Computation* 12.4, pp. 381–391.

Haupt, Michael et al. (2010). "The SOM family: virtual machines for teaching and research". In: *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*. ACM, pp. 18–22.

Hausman, Bogumil (1997). *The Erlang BEAM Virtual Machine Specification*. Available at http://www.cs-lab.org/historical_beam_instruction$_set.html$. Rev. 1.2.

Hayashizaki, Hiroshige et al. (2012). "Improving the performance of trace-based systems by false loop filtering". In: *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*. ACM, pp. 405–418.

Homescu Andrei Şuhan, Alex (2011). "HappyJIT: a tracing JIT compiler for PHP". In: *Proceedings of the 7th Symposium on Dynamic Languages*. ACM, pp. 25–36.

Johansson, Erik et al. (1999a). *Evaluation of HiPE, an Erlang native code compiler*. Tech. rep. 99/03, Uppsala University ASTEC.

Johansson, Erik et al. (1999b). *HiPE: High Performance Erlang*. Tech. rep. ASTEC 99/04. Uppsala University.

*JRuby, the Ruby Programming Language on the JVM*. http://jruby.org.

*Jython: Python for the Java Platform*. http://www.jython.org/.

Li, Kai and Paul Hudak (1986). "A new list compaction method". In: *Software: Practice and Experience* 16.2, pp. 145–163.

Marr, Stefan and Stéphane Ducasse. "Tracing vs. partial evaluation: Comparing meta-compilation approaches for self-optimizing interpreters". In: *In Proceedings of the 2015 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA'15. ACM, 2015. ISBN*. ACM.

Pettersson, Mikael, Konstantinos Sagonas, and Erik Johansson (2002). "The HiPE/x86 Erlang compiler: System description and performance evaluation". In: *Proceedings of the 6th International Symposium on Functional and Logic Programming*. Springer, pp. 228–244.

Sagonas, Konstantinos, Chris Stavrakakis, and Yiannis Tsiouris (2012). "ErLLVM: An LLVM backend for Erlang". In: *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop*. ACM, pp. 21–32.

Sagonas, Konstantinos et al. (2003). "All you wanted to know about the HiPE compiler:(but might have been afraid to ask)". In: *Proceedings of the third ACM SIGPLAN workshop on Erlang*. ACM, pp. 36–42.

Shao, Zhong, John H Reppy, and Andrew W Appel (1994). "Unrolling lists". In: *Proceedings of the ACM Conference on Lisp and Functional Programming*, pp. 185–195.

Thomassen, Even Wiik (2013). "Trace-based just-in-time compiler for Haskell with RPython". MA thesis. Norwegian University of Science and Technology Trondheim.

Würthinger, Thomas et al. (2012). "Self-optimizing AST interpreters". In: *Proceedings of the 8th symposium on Dynamic languages*. Vol. 48. 2. ACM, pp. 73–82.

Würthinger, Thomas et al. (2013). "One VM to rule them all". In: *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM, pp. 187–204.