Improving Sequential Performance of Erlang Based on a Meta-tracing Just-In-Time Compiler

Ruochen Huang, Hidehiko Masuhara, and Tomoyuki Aotani

Tokyo Institute of Technology
huang.r.aa@m.titech.ac.jp
masuhara@acm.org
aotani@is.titech.ac.jp

Abstract. 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.

Keywords: Meta-tracing, JIT, Erlang, BEAM

1 Introduction

Erlang [4] is a dynamically-typed, functional and concurrent programming language based on the actor model [1]. 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 [2], 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) [3] and the HiPE compiler (or HiPE in short) [13], 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

[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.

is faster than BEAM[5]. However, it requires ahead-of-time compilation for each execution platform.

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 [5] 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.

Contributions The contributions of the paper 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]. Even though our initial implementation was the result of a few month's work, 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 optimizations.

From a viewpoint of tracing JIT compilers, Pyrlang is equipped with a new more strict tracing JIT policy, which focus 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 Paper The paper is organized as follows. Section 2 introduces the instruction set of BEAM as well as an overview of a meta-tracing JIT compiler. Section 3 describes the key design decisions in Pyrlang. Section 4 proposes an optimization technique for improving performance of functions with recursive calls. Section 5 evaluates the performance of Pyrlang by comparing against the existing Erlang implementations. Section 6 discusses related work. Section 7 concludes the paper with discussing future work.

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),

[5] Though HiPE is known to exhibit largely different performance improvements depending on the types of application programs [12], it speeds up by the factors from 1.8 to 3.5 according to the benchmark results in a literature [15] and our experiments.

[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 paper.

```
;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
        ...
```

Fig. 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
```

Fig. 2. A tail-recursive factorial function in BEAM bytecode

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 1, 2 and 3 show three simple functions in BEAM bytecode.

Figure 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 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
```

Fig. 3. A non-tail recursive factorial function in BEAM bytecode

Figure 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 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 Meta-tracing JIT Compiler

A meta-tracing JIT compiler [5, 7] monitors an execution of an interpreter of a target language, and generates optimized native code for a frequently executed instruction sequence (called a *hot trace*) of the *subject* program (which is the Erlang program in our work). Though it can be seen as an application of a tracing JIT compiler to an interpreter program, annotations specialized to interpreters optimizations make it possible to generate compiled code of the subject program, rather than compiled code of the interpreter. 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.

Meta-tracing JIT compilers basically have the same mechanism as tracing JIT compilers. We therefore explain the mechanisms of tracing JIT compilers first, and then the notions specific to meta-tracing JIT compilers.

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 straightline code fragment spanning over multiple functions, the compiler effectively achieves aggressive inlining with low-level optimizations 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 optimizations 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.

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.

In the rest of the paper, 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.

3 Implementation Overview

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).

3.1 Data Representation

Instructions and Literal Values We represent a BEAM bytecode program as an RPython array of instruction objects. An instruction object is an RPython

object that contains operands in its fields. 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.

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.

X and F Registers We use two RPython lists for X and F registers, respectively. We also mark those lists *virtualizable*[7], 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.

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.

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

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.

4 Finer-grained Path Profiling

4.1 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 [10].

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.

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.

4.2 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 Section 3, tail recursions in Pyrlang are similar with typical loops in an imperative language, as a tracing JIT compiler expects.

Figures 4 and 5 show a countdown function in the BEAM bytecode with its control flow graph. The function infinitely repeats counting numbers from 10 to 1. While imperative languages realize the computation double nested loops,

```
;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                        }
```

Fig. 4. A countdown function which restarts itself from 10

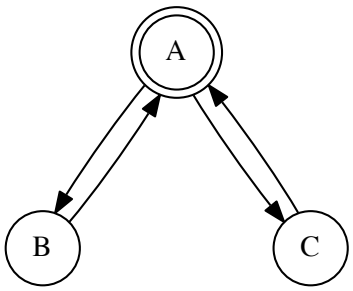

Fig. 5. Control flow graph of the count-down function. (The alphabets in the nodes correspond to the basic blocks in Figure 4. The doubly-circled node denotes the JIT merge point.)

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 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 6.

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

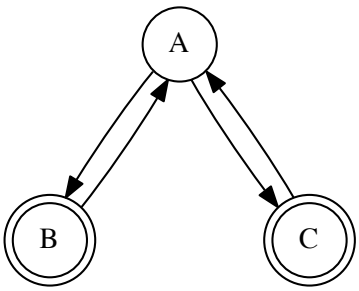

Fig. 6. Control flow graph of the countdown function for pattern matching tracing

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.

4.4 Two State Tracing

In addition to the pattern matching tracing, we also use two state tracing policy proposed for Pycket [6]. The basic idea of two state tracing is to distinguish entries of recursive functions by the caller instruction address. 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.

5 Evaluation

5.1 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[8] that is translated by the authors from Scheme to Erlang. The other is the ErLLVM benchmark suite[9], 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

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

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 4.2.

We emphasize that Pyrlang does not apply low-level optimizations such as calling convention for the x86 architecture [15], and the local type propagation [16] 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 [9, 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.

5.2 Overall Performance

Figure 7 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 7, 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 optimizations. 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.

5.3 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),

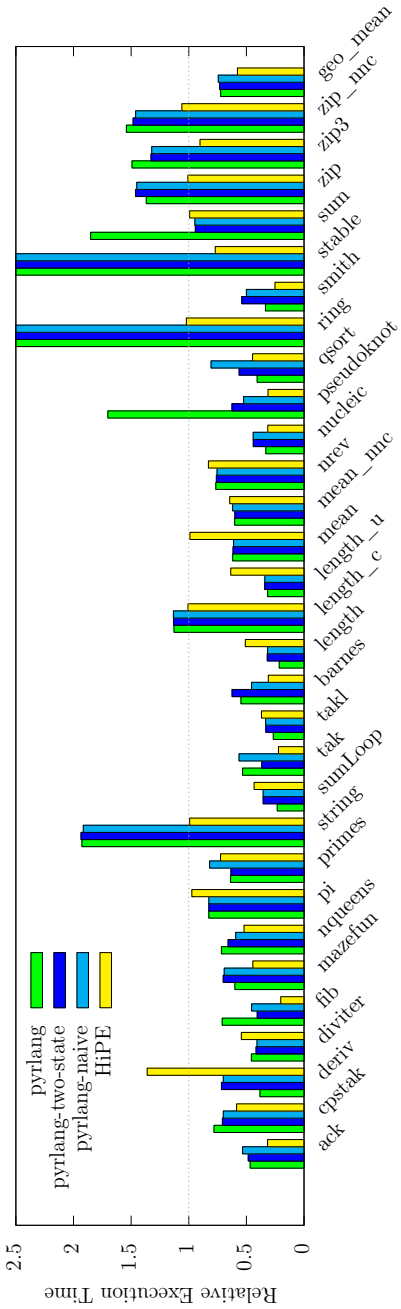

Fig. 7. Execution Times with pyrlang, pyrlang-two-state, pyrlang-naive and HiPE, relative to BEAM (the shorter is the better)

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 Section 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, respetively. 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 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.

6 Related Work

PyPy [5], 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[10] 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 [6] is an implementation of Racket runtime system based on the meta-tracing JIT compiler in RPython. Similar to Erlang, Racket is a Scheme-like

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

functional programming language, in which user programs use recursive calls for iterations. It proposes a two-state-tracing [6], which is a light-weight solution to the false loop problem [10]. 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 [8] 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 [17] 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.

PyHaskell [19] 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 [11] 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.

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 Pyret), 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 optimizations.

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

Improvement of Pattern Matching Tracing: As we explained in Section 5.2, 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 optimizations for functional style list processing, such as cdr-coding [14] and list unrolling [18]. 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.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, 1986.
2. J. Armstrong. Erlang: a survey of the language and its industrial applications. In *Proceedings of the symposium on industrial applications of Prolog (INAP96)*, pages 16–18, 1996.
3. J. Armstrong. The development of Erlang. In *Proceedings of International Conference on Functional Programming '97*, pages 196–203. ACM, 1997.
4. J. L. Armstrong and S. Virding. Erlang: an experimental telephony programming language. In *Proceedings of XIII International Switching Symposium*, pages 43–48, 1990.
5. C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. 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*, pages 18–25. ACM, 2009.
6. C. F. Bolz, T. Pape, J. Siek, and S. Tobin-Hochstadt. Meta-tracing makes a fast Racket. In *Proceedings of Workshop on Dynamic Languages and Applications*, 2014.

7. C. F. Bolz and L. Tratt. The impact of meta-tracing on VM design and implementation. *Science of Computer Programming*, 98:408–421, 2015.
8. F. Drejhammar and L. Rasmusson. BEAMJIT: a just-in-time compiling runtime for Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, pages 61–72. ACM, 2014.
9. B. Hausman. The Erlang BEAM virtual machine specification. Available at `http://www.cs-lab.org/historical_beam_instruction_set.html`, Oct. 1997. Rev. 1.2.
10. H. Hayashizaki, P. Wu, H. Inoue, M. J. Serrano, and T. Nakatani. 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*, pages 405–418. ACM, 2012.
11. A. Homescu Andrei, Şuhan. HappyJIT: a tracing JIT compiler for PHP. In *Proceedings of the 7th Symposium on Dynamic Languages*, pages 25–36. ACM, 2011.
12. E. Johansson, S.-O. Nyström, T. Lindgren, and C. Jonsson. Evaluation of HiPE, an Erlang native code compiler. Technical report, 99/03, Uppsala University ASTEC, 1999.
13. E. Johansson, S.-O. Nyström, M. Pettersson, and K. Sagonas. HiPE: High performance Erlang. Technical Report ASTEC 99/04, Uppsala University, 1999.
14. K. Li and P. Hudak. A new list compaction method. *Software: Practice and Experience*, 16(2):145–163, 1986.
15. M. Pettersson, K. Sagonas, and E. Johansson. The HiPE/x86 Erlang compiler: System description and performance evaluation. In *Proceedings of the 6th International Symposium on Functional and Logic Programming*, pages 228–244. Springer, 2002.
16. K. Sagonas, M. Pettersson, R. Carlsson, P. Gustafsson, and T. Lindahl. 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*, pages 36–42. ACM, 2003.
17. K. Sagonas, C. Stavrakakis, and Y. Tsiouris. ErLLVM: An LLVM backend for Erlang. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop*, pages 21–32. ACM, 2012.
18. Z. Shao, J. H. Reppy, and A. W. Appel. Unrolling lists. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 185–195, 1994.
19. E. W. Thomassen. Trace-based just-in-time compiler for Haskell with RPython. Master's thesis, Norwegian University of Science and Technology Trondheim, 2013.