Pyrlang: A High Performance Erlang Virtual Machine Based on RPython

Ruochen Huang Hidehiko Masuhara Tomoyuki Aotani
Tokyo Institute of Technology, Japan
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. We are developing a virtual machine called Pyrlang for the Erlang BEAM bytecode with a just-in-time (JIT) compiler. By using RPythons tracing JIT compiler, our preliminary evaluation showed approximately twice speedup over the standard Erlang interpreter. In this poster, we overview the design of Pyrlang and the techniques to apply RPythons tracing JIT compiler to BEAM bytecode programs written in the Erlangs functional style of programming.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Interpreters

Keywords Meta-tracing, JIT, Erlang, BEAM

1. Introduction

Erlang is a dynamically-typed, functional, and concurrent programming language based on the actor model [1]. It is widely used for practical applications that require high scalability and availability.

Erlang has a compiler implementation called HiPE [3], which is 1.8 to 3.5 times faster than the commonly-used interpreter implementation called BEAM [2]. However, HiPE is a ahead-of-time compiler, and is not as widely used as BEAM.

We are developing a virtual machine called Pyrlang for the BEAM bytecode with a just-in-time (JIT) compiler. By using RPython's tracing JIT compiler [4], Pyrlang's performance is comparable to HiPE according to our preliminary evaluation.

SPLASH Companion'15, October 25–30, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3722-9/15/10
http://dx.doi.org/10.1145/2814189.2817267

2. Implementation Overview of Pyrlang

Pyrlang is a virtual machine compatible with BEAM written in RPython, which is a subset of Python and originally used in PyPy project.

In order to support functional programming features in Erlang, we apply the PyPy virtualizable optimization to the BEAM registers. We apply the RPython immutable annotation to many datatypes BEAM, so that RPython meta-tracing JIT can unbox the fields of built-in datatypes. We implement a call-stack structure at the RPython level so that we can implement an unlimited call-stack for non-tail calls.

Furthermore, we implement a scheduler to support the actor model. Currently, the scheduler supports multi-threading on a single CPU core.

3. Optimization Techniques for a Tracing JIT Compiler

3.1 Optimizing Memory Allocation

We carefully design runtime data structures so that the RPython optimizer eliminates redundant memory allocations. Memory allocations for runtime data structures are not automatically optimized by JIT optimizer because they survive across multiple traces. Therefore, we manually optimized by defining the following data structures so that they have minimal allocations at runtime. The remaining redundant memory allocations that RPython cannot optimize out include literal data type loading, symbol table management, and call-stack management. We therefore implement the following mechanisms to eliminate redundant memory allocations:

- A preprocess for building a literal data table and a constant number table.
- A runtime symbol manager for manipulating the symbol data type.
- A special data structure for the call stack. We use a pair of fixed-sized RPython arrays to implement the call-stack, one is for storing local variables, and the other is for storing return addresses. A dedicated array for return addresses is crucial to avoid extra object wrapping.

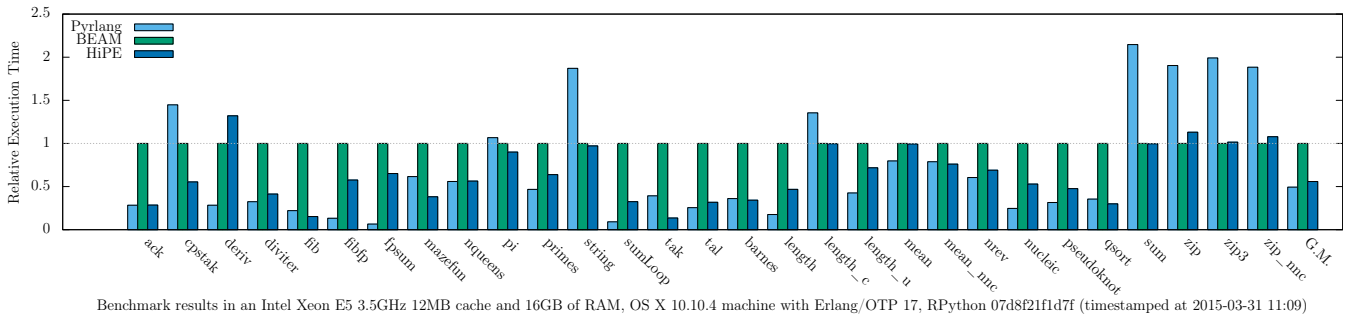

Figure 1. The execution time of pure functional programs translated from Scheme benchmark suite and ErLLVM benchmark suite, relative to those executed by BEAM.

3.2 Finer-grained Path Profiling

We proposed a new profiling policy called the *pattern-matching-trace* at the bytecode level to avoid overheads caused by shorter compiled traces detected in (non-tail) recursive functions. The idea is to profile runtime code at a finer granularity. In our experiment, the pattern-matching-trace shows a faster and more stable performance with different execution conditions.

4. Preliminary Performance Evaluation

As a preliminary experiment, we compared performance of BEAM, HiPE and Pyrlang by running a small benchmark. The selected benchmark programs are a subset of ErLLVM benchmark suites and a subset of the Scheme Larceny benchmark suite which is translated to Erlang by the authors.

As we can see in Figure 1, both HiPE and Pyrlang show better performance than BEAM for most benchmark program. Note that Pyrlang is not yet good at allocating a large amount of memory, such as closure generation in cpstak, and list building in string.

5. Conclusion and Future Work

We proposed Pyrlang, a virtual machine for the BEAM bytecode language with a tracing JIT compiler. In order to achieve reasonable runtime performance, we need to carefully make several design decisions and to apply optimization techniques, in particular for memory allocation and loop detection in recursive functions. The resulted implementation exhibited 2.03 performance improvement over BEAM.

As mentioned above, our implementation leaves some of the features to be implemented in future:

List Optimization. In our experiment, we observed that Pyrlang is still slow at processing huge lists in a non-tail recursion coding style. We want to optimize the list representation using techniques such as cdr-coding [8] or list unrolling [9].

Compatibility Improvement. We will extend Pyrlang to support all datatypes of BEAM, and implement more bytecode instructions and built-in functions.

Multi-core Support. BEAM supports multiple CPU cores. As for the implementation details, it requires multiple schedulers at each conceptual CPU core.

References

[1] G. Agha. *Actors: A model of concurrent computation in distributed systems.* MIT Press, Cambridge, 1986.

[2] J. Armstrong. "The development of Erlang." In *Proceedings of International Conference on Functional Programming '97*, pages 196–203. ACM, 1997.

[3] E. Johansson, et al. *HiPE: High Performance Erlang.* Technical Report ASTEC report 99/04, Uppsala University, 1999.

[4] C. F. Bolz, A. Cuni, M. Fijalkowski, & 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.

[5] Bolz, Carl Friedrich, et al. "Allocation removal by partial evaluation in a tracing JIT." *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation.* ACM, 2011.

[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] 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.

[8] K. Li and P. Hudak. A new list compaction method. *Software: Practice and Experience* 16.2:145-163, 1986.

[9] 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.

[10] K. Lundin. "About Erlang/OTP and Multi-core performance in particular." *Erlang Factory London* (2009).