BacCaml: The Meta-Hybrid Just-In-Time Compiler

Yusuke Izawa
Department of Methematical and Computing Science
Tokyo Institute of Technology
yizawa@acm.org

1 INTRODUCTION

Meta-interpreter-based just-in-time compiler frameworks [2, 3, 12, 13] are useful to conveniently build a language runtime with reasonable execution performance [1].

Two most successful frameworks, RPython [3] and Truffle/Graal [13], employ different strategies in terms of compilation units. RPython takes the trace-based strategy that compiles a straight-forwarded execution-path, inlining function calls and ignoring untaken branches. Truffle/Graal takes the method-based strategy that basically compiles all execution paths in a method.

Those strategies have their own pros and cons. The trace-based approach compiles programs with many ramification possibilities well, which are common in dynamically-typed languages. However, it is sometimes work poorly for programs with varying control flow, for example the Fibonacci function[6], as such program can take different execution path from the one used for compilation quite often. The method-based strategy is rather robust with those kinds of programs. However, it relies on carefully planned method inline to achieve good performance.

2 META-HYBRID COMPILATION APPROACH

We propose a *meta-hybrid JIT compilation framework*, and its experimental implementation BacCaml. The goal is to enable both method- and trace-based compilation by using a single interpreter definition, and to compiles different parts of a program with different strategies.

[1]successfully implemented for Smalltalk [4, 7], Racket [1], Python [8, 11], and Ruby [5, 9].

Programming '19, April 1–4, 2019, Genova, Italy
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6257-3/19/04.
https://doi.org/10.1145/3328433.3328466

While there are many possibilities for supporting the two strategies, our framework extends a trace-based meta-compilation framework to support method-based compilation as well. BacCaml is a proof-of-concept implementation of our framework. The meta-tracing compiler of BacCaml follows the basic architecture of the RPython's one, but implemented in OCaml.

Currently, we designed and implemented the core compiler parts of BacCaml, while most of the runtime supports (e.g., profiler and dispatcher) and optimizations are left as future work.

3 COMPILING A METHOD BY USING A META-TRACING COMPILER

The compiler engine of our framework performs both method- and tracing-compilation by using the same interpreter definition. Furthermore, the compiler engine shares the large part of the implementation for method- and tracing-compilation, which is basically achieved by applying a tracing-compiler to all possible paths in a method.

Of course, it is not trivial to compile a method by using a meta-tracing compiler. The following are the techniques we devised for that purpose.

Conditional Branches. Since tracing compilers basically generates code only for one of two subsequent paths of a conditional branch, we modify a tracer so that it can rollback its states, including the values in the registers and the heap, at the branch, and generate code for the un-taken branch after traced the taken branch.

Loops. Our compiler compiles a loop in a base-program in this way. First, we assume that the interpreter explicitly handles loops in the base-program so that we know where the entry-point and the back-edge of the loop. Second, the tracer splits traces at the entry-point of a loop. It then traces the loop body until it reaches a back-edge of a loop. Instead of following the back-edge, it finishes tracing with generating a jump instruction to the trace entry of the loop body.

Function Calls. Basically, we compile a function call in the base-program by not tracing into the destination of the respective call instruction in the interpreter, and generating the call instruction in the compiled code. Though it is

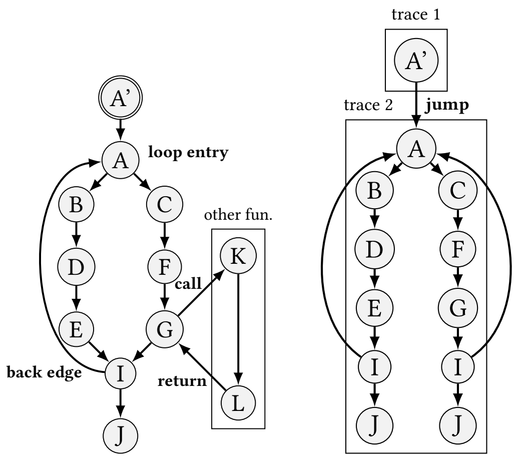

Figure 1: An example of the method-based compilation. The left- and right-hand sides are the control-flows of the base- and the compiled programs, respectively. As can be seen in the figure, the compiled code consists of two traces, has duplicated code after the conditional branch, and does not inline the function call.

a simple idea, it requires the interpreter to use the host language's stack for function calling, which does not work well with trace-based compilation. To overcome this, we provide a special syntax for defining two versions of a method-call handler in an interpreter. It was possible to define such an interpreter with reasonable amount of efforts as far as we experimented.

Figure 1 shows an example of method-based compilation of a function with one conditional branch with a function call in a loop.

4 PRELIMINARY BENCHMARK TEST

To confirm our framework can perform both trace- and method-based compilation, we wrote a small interpreter that executes two microbenchmarks (sum and fib). The latter has two non-tail recursions which cause the path divergence problem with tracing compilers.

Figure 2 shows the execution times of the two programs compiled by the method- and trace-based compilation. [2]. The numbers are relative to the programs directly compiled by MinCaml [3].

[2]For taking data, we used Mac Pro (Late 2013) with CPU: 3.5 GHz 6-Core Intel Xeon E5, and Mem: 16 GB 1866 MHz DDR3, running macOS Mojave version 10.14.2.

[3]MinCaml is known to generate as efficient code as the mainstream optimizing compilers like GCC and OCamlOpt [10]

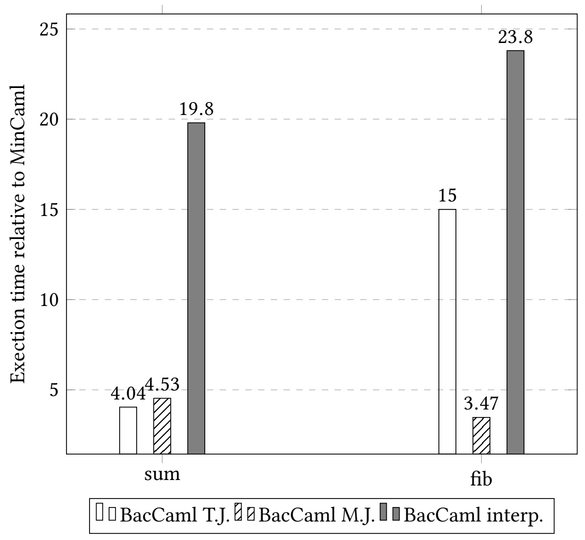

Figure 2: Execution times of microbenchmark programs. The bars for each program show the execution times of the trace-based compiled, method-based compiled and interpreted code, relative to the MinCaml compiled code (shorter is better).

From the figure, we can see the code compiled by the method-based compilation is faster than the interpreted execution by more than the factor of 4. In contrast, the code compiled by the trace-based compilation is slow for fib, because the compiled trace can only cover some of the execution paths. The reason why the performance is still worse than MinCaml's execution is that we haven't implemented trace-optimizers.

5 CONCLUSION

We propose a hybrid meta-compilation framework that compiles in method-based and trace-based strategy using a single interpreter definition. We plan to complete the implementation of the framework, and investigate strategies of switching compilers and a good programming interface for defining interpreters.

REFERENCES

[1] Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. 2015. Pycket: a tracing JIT for a functional language. *ACM SIGPLAN Notices* 50, 9 (2015), 22–34. https://doi.org/10.1145/2858949.2784740

[2] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. 2011. Runtime feedback in a meta-tracing JIT for efficient dynamic languages. *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems - ICOOOLPS '11* (2011), 1–8. https://doi.org/10.1145/2069172.2069181

[3] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems - ICOOOLPS '09* (2009), 18–25. https://doi.org/10.1145/1565824.1565827

[4] Carl Friedrich Bolz, Adrian Kuhn, Adrian Lienhard, Nicholas D. Matsakis, Oscar Nierstrasz, Lukas Renggli, Armin Rigo, and Toon Verwaest. 2008. Back to the Future in One Week — Implementing a Smalltalk VM in PyPy. In *Self-Sustaining Systems*, Robert Hirschfeld and Kim Rose (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 123–139.

[5] Tim Felgentreff. 2013. Topaz Ruby. https://github.com/topazproject/topaz

[6] Ruochen Huang, Hidehiko Masuhara, and Tomoyuki Aotani. 2016. Improving Sequential Performance of Erlang Based on a Meta-tracing Just-In-Time Compiler. In *Post-Proceeding of the 17th Symposium on Trends in Functional Programming*. https://tfp2016.org/papers/TFP_2016_paper_16.pdf

[7] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2018. GraalSqueak A Fast Smalltalk Bytecode Interpreter Written in an AST Interpreter Framework. *ICOOOLPS* 18 (2018). https://doi.org/10.1145/3242947.3242948

[8] Armin Rigo and Samuele Pedroni. 2006. PyPy ' s Approach to Virtual Machine Construction. *Companion to the 21st ACM SIGPLAN symposium* (2006), 944–953. https://doi.org/10.1145/1176617.1176753

[9] Chris Seaton, Benoit Daloze, Kevin Menard, Petr Chalupa, Brandon Fish, and Duncan MacGregor. 2017. TruffleRuby – A High Performance Implementation of the Ruby Programming Language. https://www.graalvm.org/docs/reference-manual/languages/ruby/

[10] Eijiro Sumii. 2005. MinCaml: A Simple and Efficient Compiler for a Minimal Functional Language. *FDPE: Workshop on Functional and Declaritive Programming in Education* (2005), 27–38. https://doi.org/10.1145/1085114.1085122

[11] Christian Wimmer and Stefan Brunthaler. 2013. ZipPy on Truffle: A Fast and Simple Implementation of Python. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity (SPLASH '13)*. ACM, New York, NY, USA, 17–18. https://doi.org/10.1145/2508075.2514572

[12] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12)*. ACM, New York, NY, USA, 13–14. https://doi.org/10.1145/2384716.2384723

[13] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. *Proceedings of the 8th symposium on Dynamic languages - DLS '12* (2012), 73. https://doi.org/10.1145/2384577.2384587