Extending a Meta-Tracing Compiler to Mix Method and Tracing Compilation

Yusuke Izawa
Department of Mathematical and Computing Science
Tokyo Institute of Technology
izawa@prg.is.titech.ac.jp

Hidehiko Masuhara
Department of Mathematical and Computing Science
Tokyo Institute of Technology
masuhara@acm.org

Tomoyuki Aotani
Department of Mathematical and Computing Science
Tokyo Institute of Technology
aotani@is.titech.ac.jp

ABSTRACT

Meta-interpreter-based just-in-time compiler frameworks provide a convenient way for language designers to implement efficient virtual machines. Those frameworks either employ tracing-based or method- (or partial evaluation) based strategies, which have their own pros and cons. This paper proposes an approach to enable both tracing- and method-based compilation so that the runtime can selectively apply an appropriate strategy to different parts of a program. The proposal basically extends a meta-tracing compiler to method-based compilation by roll backing at conditional branches, trace-splitting at loop entries, and not following at function calls. As a proof-of-concept, we implemented a tiny meta-tracing compiler in MinCaml by following the RPython's architecture and extended it to support both tracing- and method-based compilation.

CCS CONCEPTS

• **Software and its engineering → Just-in-time compilers**; *Source code generation.*

KEYWORDS

Language implementation frameworks, tracing JIT compilation, RPython

ACM Reference Format:
Yusuke Izawa, Hidehiko Masuhara, and Tomoyuki Aotani. 2019. Extending a Meta-Tracing Compiler to Mix Method and Tracing Compilation. In *Companion of the 3rd International Conference on Art, Science, and Engineering of Programming (Programming '19), April 1–4, 2019, Genova, Italy.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3328433.3328439

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

Two of the most successful frameworks, namely RPython [3] and Truffle/Graal [13], employ different strategies in terms of compilation units. RPython utilizes the trace-based strategy that compiles a straightline execution path, inlines method calls and ignores untaken branches. Truffle/Graal takes the method-based strategy that mainly compiles all execution paths in a method.

These strategies have their own advantages and disadvantages. The trace-based strategy is good at compiling programs with many branching possibilities, which are common in dynamically-typed languages. However, it sometimes works poorly for programs that have varying control flow, for example the Fibonacci function[6], which often takes different execution paths from the one traced for compilation. The method-based strategy is more robust with those kinds of programs. However, it relies on carefully planned method inline to achieve good performance.

2 META-HYBRID COMPILATION APPROACH

In this paper, we propose a *meta-hybrid JIT compilation framework*, and its experimental implementation called BacCaml. The goal of the framework is to enable both method- and trace-based compilation by using a single interpreter definition. It compiles different parts of a program with different strategies. Choosing a compilation strategy is left for future work, though we plan to apply the trace-based compilation first, and to apply the method-based compilation for methods that causes frequent guard failures.

While there are many approaches to support the two strategies, our framework extends a trace-based meta-compilation framework to realize method-based compilation as well. BacCaml, a proof-of-concept implementation of our framework, follows the basic architecture of RPython, but implemented in OCaml [1]. The tracing and code generation parts are written by modifying the MinCaml compiler [10].

Currently, we designed and implemented the core compiler parts of BacCaml, while most of the runtime support (e.g., profiler and dispatcher) and optimizations are left as future work. The rest of the paper explains how we achieve method-based compilation by using a meta-tracing compiler.

3 COMPILING A METHOD BY USING A META-TRACING COMPILER

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

[1]Since we did not know the requirements to the compilation frameworks in order to support method-based compilation, we implmnted from scratch rather than extending existing frameworks like RPython.

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 generate code only for one of two subsequent paths of a conditional branch, we modify a tracer so that it can roll back its states including the values in the registers and the heap, at the branch at the branch, and generate code for the untaken branch after it has traced the taken branch. Note that our tracer only roll backs at conditional branches in the base-program (which shall be annotated in the interpreter definition).

Loops. In order to compile a loop in a base-program without guard failures, our compiler compiles in the following way [2]. 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 are. 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 on generating a jump instruction to the trace entry of the loop body.

Function Calls. Fundamentally, we compile a function call in the base-program by not tracing into the destination of the respective call instruction in the interpreter, but by generating a call instruction in the compiled code. Though it is 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 effort as far as we experimented.

Figure 1 shows an example of method-based compilation of a function with one conditional branch and a function call in a loop. The left- and right-hand sides are respectively 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: (1) duplicated code after the conditional branch (2) the function call that isn't being inlined.

4 PRELIMINARY BENCHMARK TEST

In order to confirm that our framework can perform both two strategies, we wrote a small interpreter that executed two microbenchmark programs, namely sum and fib. The latter has two cases of non-tail recursion, which causes the path divergence problem with tracing compilers. Note that this preliminary bechmark tests merely use two compilation strategies separately. Mixed compilation is left for future work.

We only implemented the core part of BacCaml at this moment. For both tracing- and method-based compilation, we ran the tracer offline by manually specifying the entry/exit points of the traces. Therefore, the execution times do not include the time for profiling and dynamic compilation. For trace-based compilation of fib, we specified two traces as it was not possible to compile all the possible traces.

[2]The existing meta-tracing compilers like RPython would generate similar compiled code after sufficient amount of tracing at failed guards. Our method-based compilation aims at generating the compiled code for a whole method body at once.

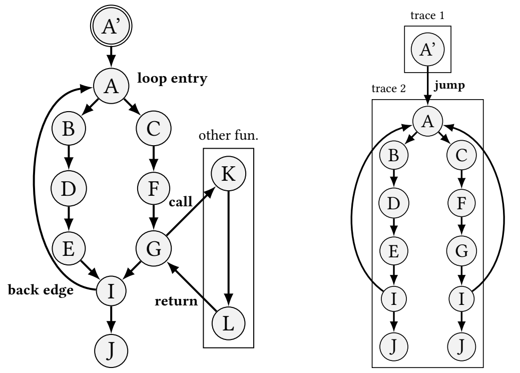

Figure 1: An example of the method-based compilation. A base-program with the control-flow depicted on the left-hand side will be compiled into a code with the control-flow depicted on the right-hand side.

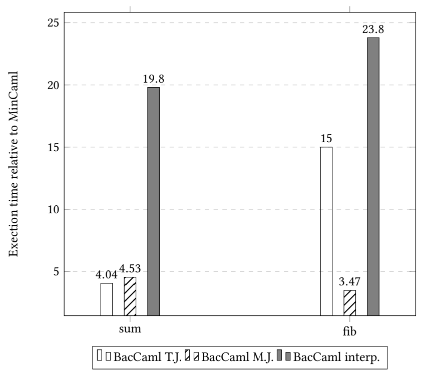

Figure 2: Execution times of microbenchmark programs. The bars for each program show the execution times of the trace-based compiled, method-based compiled and interpreted code, relative to the MinCaml compiled code (shorter is better).

Figure 2 shows the execution times of the two programs compiled by the method- and trace-based compilation strategies [3]. The numbers are relative to the programs directly compiled by MinCaml [4].

From the figure, we can see that the code compiled by the method-based compilation is faster than the interpreted execution by more than a 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 performance of the compiled code is still worse than the MinCaml compiled code by the factors of 3 to 4. This is mainly because we have not implemented trace-optimizers, which leaves a lot of unnecessary memory and register accesses in the compiled code.

[3]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.

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

5 CONCLUSION

We propose a hybrid meta-compilation framework that can perform method-based and trace-based compilation by using a single interpreter definition, implemented the core part of the framework, and demonstrated both method- and tracing-based compilation for a very small language. Further, we plan to complete the implementation of the framework called BacCaml as well as investigate strategies of switching compilers and a good programming interface for defining interpreters.

ACKNOWLEDGMENTS

This work was supported by KAKENHI (18H03219).

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