A Stack Hybridization for Meta-hybrid Just-in-time Compilation

Yusuke Izawa Hidehiko Masuhara Tomoyuki Aotani Youyou Cong

Meta-interpreter-based language implementation frameworks, such as RPython and Truffle/Graal, are convenient tool for implementing state-of-the-art virtual machines. Those frameworks are classified into trace-based and method- (or AST-) based strategies. RPython uses a trace-based policy to compile straight execution paths, while Truffle/Graal leverages method invocation to compile entire method bodies. Each approach has its own advantages and disadvantages. The trace-based strategy is good at compiling programs with many branching possibilities and able to reduce the size of compiled code, but it is weak at programs with varying control-flow. The method-based strategy is robust with the latter type of programs but it needs thorough method-inlining management to achieve excellent performance.

To take advantage of both strategies, we propose a meta-hybrid compilation technique to integrate trace- and method-based compilations, as well as a proof-of-concept implementation called BacCaml. To achieve this goal, we develop a stack hybridization mechanism which makes it possible to coordinate trace- and method-based meta JIT compilation. In the implementation, we extend RPython's architecture and introduced a special syntax for realizing this system in a single interpreter definition.

1 Introduction

Meta-interpreter-based language implementation frameworks are getting more and more popular as they allow language developers to leverage convenient components such as a just-in-time (JIT) compiler and garbage collectors. Many programming languages, such as Python [3][11], Ruby [14][10], R [9], and Erlang [6], have been implemented with language implementation frameworks, and each of implementations achieves good performance as well as ahead-of-time compilation.

Language mplementation frameworks can be classified into two compilation approaches from the viewpoint of compilation units. RPython [3][4], a language implementation framework as part of the PyPy [3] project, adopts tracing compilation, requiring bytecode interpreters and utilizing straight execution paths commonly executed loops. In contrast, Truffle/Graal [16][15] choose method-based approach, rewriting abstract-syntax-tree (AST) nodes and applying partial evaluation to enhance run-time performance.

Each strategy has its own pros and cons. The trace-based strategy is particularly good at compiling programs with branching possibilities, which are common in dynamically typed languages. However, it performs poorly in the case of compiling Fibonacci like programs [6], where the execution path actually taken at run-time is often different from the one trace for compilation. We call this *the path divergence problem*. On the other hand, the method-based strategy is so robust that we can apply it for a variety of programs. However, it needs careful function-inlining to achieve excellent run-time performance.

To take advantage of both compilation strategies, we propose a language implementation framework where the language designer writes a single meta-interpreter definition; the framework compiles a part of programs in either compilation strategy, and the run-time can switch execution between code fragments compiled by both strategies. The key

* This is an unrefereed paper. Copyrights belong to the Author(s). An erlier version of the paper was presented at the MoreVM'19 workshop.

Yusuke Izawa, Hidehiko Masuhara, Tomoyuki Aotani, Youyou Cong, Tokyo Institute of Technology, Dept. of Mathematical and Computing Science.

technique in our proposal is *stack hybridization*, as the two strategies require the meta-interpreter to use stack frames in different ways.

In this paper, we describe how to take advantage of both meta compilation strategies and our design choice to implement our meta-hybrid JIT compiler which we call BacCaml.

2 Background

Before presenting the implementation of our meta-hybrid compiler, we briefly review tracing and meta-tracing compilation. We also explain *the path divergence problem*, a performance degradation in tracing JIT compilers.

2.1 Tracing Compilation

A tracing optimization was firstly investigated by Dynamo project [1], and its technique is adopted to implement JIT compilers for dynamic languages, e.g. TraceMonkey JavaScript VM [5] and SPUR, a tracing JIT compiler for CIL [2].

Generally, tracing JIT compilation is separated into following phases:

- *Profiling*: The profiler detects commonly executed loops (*hot loop*). Typically, this is done by counts how many times a backward jump instruction is executed, and if the number is greater than a threshold, the path is considered as a hot loop.
- *Tracing*: The interpreter records all executed operations as an intermediate representation (IR). Therefore function calls in hot loops are automatically inlined.
- *Code generation*: The compiler optimizes the trace in several ways, e.g. common-subexpression elimination, dead-code elimination, and constant folding, and compiles it into native code.
- *Execution*: After the trace has been compiled to native code, it is executed when the interpreter runs the hot code once again.

Among all possible branches, only actually executed one is selected. To make sure that the condition of tracing and execution are the same, a special instruction (*guard*) is placed at every possible point (e.g. `if` statements) that goes to another direction. The guard checks whether the original condition is still valid. If the condition is false, the execution in the machine code is quit and continues to execute by falling back to the interpreter.

2.2 Meta-Tracing Compilation

BacCaml is based on RPython's architecture. Before describing the implementation details, let us give an overview of RPython's meta-tracing JIT compilation.

RPython, a subset of Python, is a tool-chain for creating programming languages with a trace-based JIT compiler. RPython's trace-based JIT compiler traces the execution of an user-defined, interpreter instead of tracing the program that an interpreter executes. It requires an user to implement a bytecode compiler and an interpreter definition for the bytecode.

In Figure 1, we show an example of a user-defined interpreter. The example uses two annotations, `jit_merge_point` and `can_enter_jit`. By adding special annotations to an user-defined interpreter as shown in Figure 1, language implementers can make their VM more efficient. The key annotations that RPython provides are `jit_merge_point` and `can_enter_jit`. We put `jit_merge_point` at the top of a dispatch loop to indicate it. `can_enter_jit` at the point which a back-edge instruction can be occurred. Furthermore, we have to tell the compiler whether the variables are "red" or "green". "red" means that a variable is relevant to the result of a calculation, hence red-colored variables are recorded in resulting traces. "green" variables which are not related to the result, are executed when tracing the execution.

In the profiling phase, the profiler monitors the program counter at `can_enter_jit`, and counts how many times a back-edge instruction (e.g. `jump` from `204` to `20`) is occurred. When the counter gets over the threshold, the JIT compiler goes into the tracing phase. Basically, the tracer records or executes instructions the user-defined interpreter executed. In this tracing phase, only "green" variables (such as `bytecode`, and `pc`) are executed and other "red" variables are recorded. The resulting traces are optimized and converted into native code. When control reaches again, it goes to the compiled code.

2.3 The Path Divergence Problem

The tracing JIT compilers have weakness in compiling a particular kind of programs [6], which we

```
def push(stack, sp, v):
  stack[sp] = v
  return sp + 1

def pop(stack, sp):
  v = stack[sp - 1]
  return v, sp - 1

def interp(bytecode):
  stack = []; sp = 0; pc = 0
  while True:
    jit_merge_point(
      reds=['stack', 'sp'],
      greens=['bytecode','pc'])
    inst = bytecode[pc]
    if inst == ADD:
      v2, sp = pop(stack, sp)
      v1, sp = pop(stack, sp)
      sp = push(stack, sp, v1 + v2)
    elif inst == JUMP_IF:
      pc += 1
      addr = bytecode[pc]
      if addr < pc: # backward jump
        can_enter_jit(
          reds=['stack', 'sp'],
          greens=['bytecode','pc'])
      pc = addr

```

Fig. 1: An example interpreter definition written in RPython.

call *the path divergence problem*. The problem is observed as frequent guard failures (and compilation of the subsequent traces) when it runs such kind of programs. Since it spends most of time for JIT compilation, the entire execution can be slower than an interpreted execution.

Programs that cause the path divergence problem often take different execution paths when they are executed. Functions that have multiple non-tail recursive calls, e.g., Fibonacci, are examples.

Let us look at the problem with a Fibonacci function whose control flow graph is shown in Figure 2. Each node in the graph is a basic block. Since tracing JIT compilers †1 basically inline function calls,

†1 As explained above, meta-tracing JIT compilers effectively compile traces of the base program by tracing the meta-interpreter. Therefore we discuss the problem in a context when a tracing compiler handles a base program.

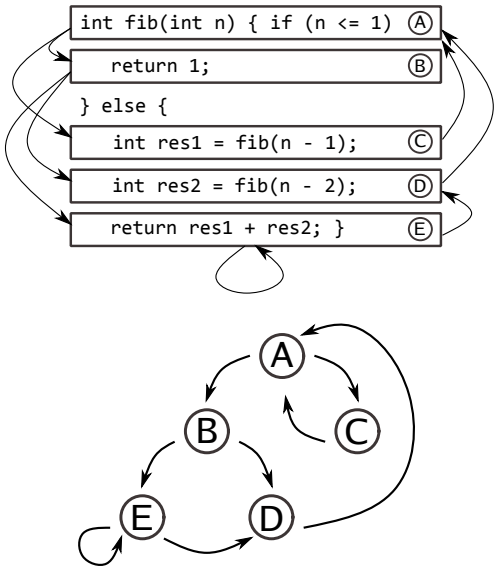

Fig. 2: The Fibonacci function and its control flow.

the nodes end with a function call is connected to the entry of the function. Also, the nodes end with a return statement is connected to the next basic blocks of its caller, which can be more than one.

Tracing compilers rely on the fact that many program executions contain a subsequence (i.e., a sequence of control flow nodes) that appear frequently in the entire execution. However, the execution of Fibonacci rarely contain such a subsequence. This is because the branching nodes in the graph, namely `A`, `B` and `E` in the graph, take one of two following nodes almost the same probability. As a result, no matter what path the tracing compiler chooses, the next execution of the compiled trace will likely to cause guard failure in its middle of execution.

3 Meta-hybrid JIT Compilation Framework

We propose a meta-hybrid JIT compilation framework. It is a *meta*-compiler framework since the language developer merely needs to write a single interpreter definition to enable JIT compilation. It is hybrid as it can compile both an execution trace of a base program (*trace compliation*) and a function (or a method) of the base program (*method compilation*). The compiled code from two types of compilation can work together in a single execution.

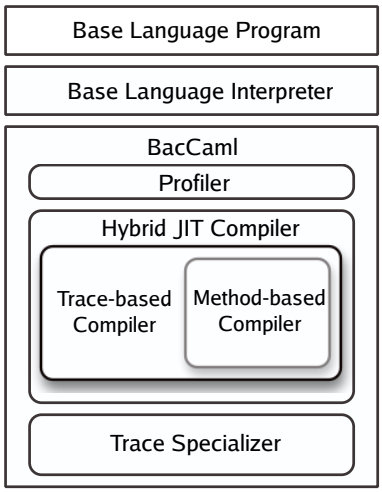

Fig. 3: The architecture of BacCaml

We implemented BacCaml based on the MinCaml [13] compiler, which is a subset of ML programming language. It is implemented in OCaml and consists of about 2000 lines of code, yet the compiled code runs as fast as GCC or OCamlOpt. Since RPython's implementation is huge and highly complex, we don't extend it and instead create a proof-of-concept implementation to show our research idea.

Figure 3 overviews the structure of our meta-hybrid JIT compiler. A base language is a language that a language implementer is going to create. A base language interpreter is written in BacCaml. The key idea that enables meta-hybrid compilation in a single interpreter definition is *stack hybridization*. Specifically we use a user-defined stack and a stack of the host language in the definitions of `call` and `return`. A base language program is run on the interpreter. The implementation language, BacCaml, consists of following components:

- *Profilier*: It profiles a run-time information of the interpreter and detects which compilation strategy (method- or trace-based) is preferred. Furthermore, it chooses a compiled trace to jump to.
- *Hybrid JIT Compiler*: This compiler is based on a trace-based compiler. It consists of a *trace-based compiler* and a *method-based compiler*. Both compilers have a tracer, implemented as an interpreter for BacCaml, to track the execution of a base language interpreter. A *trace-based compiler* compiles a straight line of the execution of a base language interpreter. On the other hand, a *method-based compiler* is constructed as an extension of a *tracing compiler*.
- *Trace Specializer*: It performs optimization of resulting traces and compiles them into native code.

Figure 4 gives an overview of BacCaml's VM generation and compilation steps. First, a language implementer writes a base language interpreter using our meta-hybrid JIT compiler called BacCaml. Next, a base language VM with a hybrid JIT compiler is generated. When a base language program is running on the VM, BacCaml's profiler monitors the execution, and detects which parts are frequently executed. Our compilation strategy is simple. We start by applying trace-based compilation and executing resulting traces. If there are frequently occurrence of guard failure, the compiler switches to method-based compilation. It also finds the method with the path divergence problem and applies method-based compilation for that program.

3.1 Method JIT Compilation by Tracing

In this section, we provide the details of method JIT compilation based on the tracing JIT compilation.

To take advantage of trace- and method-based compilation strategies and to solve the performance degradation problem on tracing JITs, we need to determine the *true path* of the base program and decrease the number of occurring guard failures. For this reason, we implement our method-JIT compilation by changing the following features of tracing JIT compilation:

- Tracing entry / exit point
- Conditional branches
- Function calls
- Loops

Tracing JIT compilers [1] [5] generally compile loops in the base program, so they start tracing at the top of a loop and finish when execution returns to the entry point. To assemble the whole body of a function, we modify this behavior to trace from the top of a method body until a `return` instruction is reached. When handling a conditional branch, a

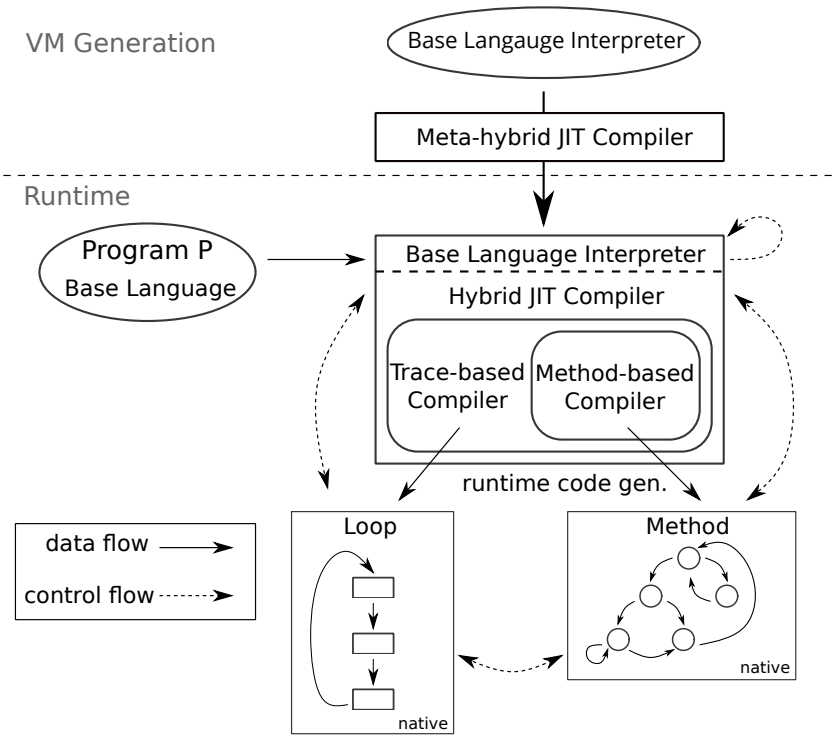

Fig. 4: Overview of BacCaml's VM generation and compilation steps.

tracing JIT compiler converts the code into a guard instruction and collect the instructions that are executed. In a method JIT compiler, however, we must compile both branches of conditionals. To do this, we must return to the branch point, and trace the other side as well. Additionally, whereas a tracing JIT inlines function calls, a method JIT emits a call instruction and continues tracing. For loops, we first transform the base program to use tail-calls, and compile loops as normal functions.

4 Stack Hybridization

In this section, we detail the *stack hybridization* mechanism for enabling meta-hybrid compilation in a single interpreter, along with the pseudo functions and macros for defining a meta-interpreter under this mechanism.

When defining a meta-interpreter, there are two possible ways to manage the call stack: the one is to use the call stack of the language of the interpreter (we hereafter call the *host-stack*), and the other is to use a stack data structure manipulated by the interpreter (we call the *user-stack*).

We discovered that a suitable definition for each `call` and `return` is different in trace- and method-based compilation. Figure 5 shows a definition of an interpreter using a user-defined stack (left) and the stack of a host-language (right). The difference between the two strategies is in the way they manage a return address, return value and callee address. As can be seen in Figure 5, all of them are managed by a user-defined stack in the implementation on the left. In the other, however, the return value is only managed by a user-defined stack, and other variables are managed by the stack of host-language.

An implication of this difference is that, the user-defined stack strategy is well suited for a trace-based compilation. This is because we can easily inline a function calling when we apply a trace-based compilation for a user-defined interpreter with this strategy. The host-language's stack stack strategy, on the other hand, is suitable for a method-based compilation. This is because we are able to find a function call, leave it to a resulting trace, and continue to trace successors.

To support both trace- and method-based compi-

```
/* user-defined stack strategy */
case CALL:
  // fetch a callee address
  addr = bytecode[pc]
  // push a return address
  stack[sp++] = pc + 1;
  pc = addr;
  break;

case RETURN:
  ret_value = stack[sp--];
  ret_address = stack[sp--];
  stack[sp] = ret_value;
  // jump to the return address
  pc = ret_address;
  break;
```

```
/* host-language's stack strategy */
case CALL:
  // fetching a callee address
  addr = bytecode[pc];
  // launch the interpreter
  ret_value = interp(addr);
  stack[sp] = ret_value;
  break;

case RETURN:
  // returning a value
  return stack[sp--];
```

Fig. 5: An example interpreter definition for `call` and `return` using pseudo codes.

```
case CALL:
  addr = bytecode[pc];
  if (is_mj ()) {
    // telling this call is made by method-based compilation
    stack[sp++] = MJ;
    ret_value = interp(addr);
    stack[sp++] = ret_value;
  } else {
    // telling this call is made by trace-based compilation
    stack[sp++] = TJ;
    stack[sp++] = pc + 1;
    pc = addr;
  }
  break;
```

```
case RETURN:
  ret_value = stack.pop();
  mode = stack.pop();
  // detecting compilation mode
  @if (mode == MJ) {
    return ret_value;
  } else {
    addr = stack.pop();
    stack.push(ret_value);
    pc = addr;
  }
  break;
```

Fig. 6: An interpreter definition using *stack hybridization*.

lation strategies in a single interpreter, we propose the stack hybridization mechanism that manages both the host- and user-stacks in the interpreter, and use one of them based on the current compilation strategy. Roughly speaking, the interpreter handles call and return operations in the following ways.

- When it calls a function under the trace-based compilation, it uses the user-stack; i.e., it saves the context information in the stack data structure, and iterates the interpreter loop. Additionally, it leaves a flag "user-stack" in the user-stack.
- When it calls a function under the method-based compilation, it uses the host-stack; i.e., it calls the interpreter function in the host language. Additionally, it leaves a flag "host-stack" in the user-stack.
- When it returns from a function, it first checks a flag in the user-stack. If the flag is "user-stack", it restores the context information from the user-stack. Otherwise, it returns from the interpreter function; i.e., using the host-stack.

In order to implement those behaviors, as shown in Figure 6, we introduce a pseudo function (`is_mj`) and special variables (`TJ` and `MJ`). When we apply

a trace-based compilation, `is_mj` returns `true` and the code in the `then` clause is traced. In the case of method-based compilation, `is_mj` returns `false` and the `else` clause is traced.

The special variables are used to detect compilation modes dynamically in `return`. This enables to cooperate resulting traces made from both method- and trace-based compilations. For example, there are two traces, one (A) is made from a trace-based compilation and the other (B) is from a method-based compilation. When a function call from A to B is occurred, a variable `TJ` is pushed to the user-defined stack. When executing a `return` instruction in B, the control executes a suitable definition by writing as shown in the right of Figure 6.

5 Preliminary Benchmark Test

Since the work is still ongoing, we organize what we have done and not done before showing data. Current, we have a complete implementation of method-based just-in-time compilation. We also have an implementation of trace-based compilation be able to trace, compile and execute dynamically, but we have not yet figure out how to make control.

In this section, we present the microbenchmark results of method-based compilation and interpreter-only execution here. We wrote a simple interpreter in BacCaml to execute small programs, sum and fib:

- **fib**: The function computes the Fibonacci number 30 and 40. It has two non-tail recursive function calls in its body which causes the path divergence problem.
- **sum**: The function computes the summation from 1 to 30000. It has two non-tail recursive function calls which do not cause the path divergence problem.

In our environment, we ran every program 100 times. We ignored the first run. The experiment was performed on a laptop with 2.70 GHz Interl(R) Core(TM) i5 processor and 8 GB memory, using Linux 5.2.5-arch1-1-ARCH.

In Figure 7, we present the results of our experiment. Note that we use shared library for just-in-time compilation, therefore it has many overheads in compilation time.

The results show that fib actually causes the path divergence problem, and our method-based compilation strategy potentialy solves the problem by combining trace- and method-based compilation at run-time. This is evidenced by the fast that fib(40) is about 4 times slower than MinCaml, and that the interpreter-only execution occurre stack overflow. fib(30) is also about twice faster than interpreter, but still about 100 times slower than MinCaml in total. This implies that using GCC at run-time is quite slow, hence we need in-memory compiling and executing systems in the future.

In contrast, sum in method-based compilation is tremendously slower than interpreter only execution and MinCaml. From this result, we can conclude that method-based compilation isn't suitable for programs with simple control flow, and we should apply trace-based compilation for such programs.

6 Related work

6.1 Truffle / Graal

Truffle/Graal [16] [15] is an AST-based meta just-in-time compiler. Graal is a JIT compiler for Java, and Truffle is a framework for Graal. Building an interpreter with Truffle framework, Graal can apply partial evaluation for the language. RPython is based on a bytecode-based interpreter definition, but Truffle requires an AST-based interpreter definition. Truffle is also successful in implementing many languages, like Ruby, Python, and R.

6.2 IonMonkey

IonMonkey [8] is a Firefox's mature JavaScript engine. Mozilla created method JIT and tracing JIT compilers for JavaScript. It adopted a hybrid approach that method JIT was used in initial speeding up and tracing JIT is lately applied for hot loops.

6.3 Android Hybrid JIT

Pérez, et al. [12] cooperate a method JIT compiler and a tracing JIT compiler on the Android Dalvik VM. The hybrid JIT compiler achieved a high performance by sharing profiling and compilation information.

6.4 Pyrlang

Pyralng [6] is an Erlang virtual machine with a tracing JIT compiler by applying the meta-tracing

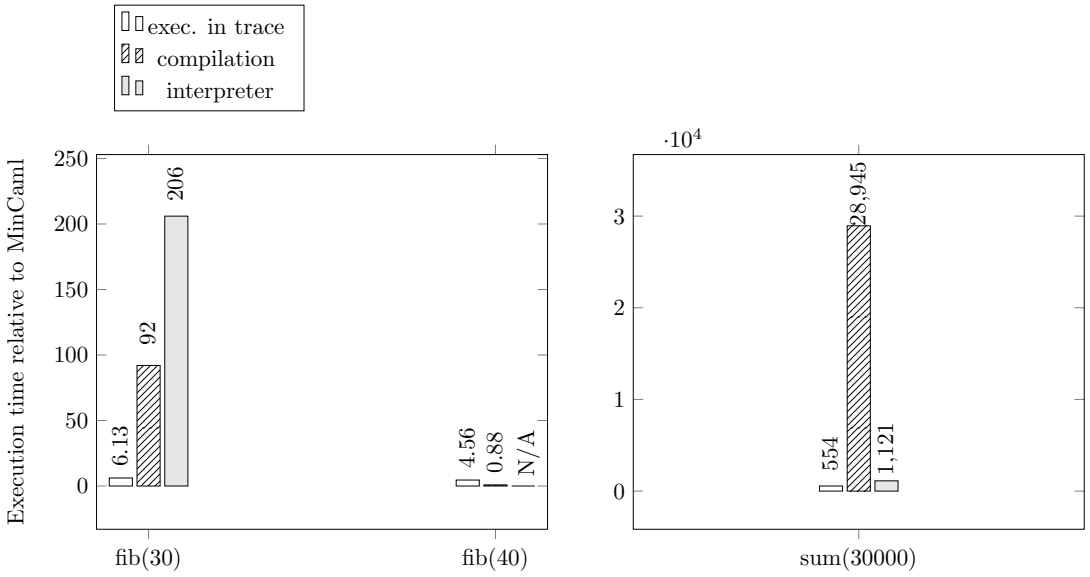

Fig. 7: Preliminary benchmark results of BacCaml's method JIT and interpreter only execution. Each bar shows the execution time of method-based JIT compilation and interpreter-only execution, related to MinCaml. *exec. in trace* is a execution time in the compiled code. *compilation* shows how much time dynamic compilation consumes in run-time. *interpreter* means all instructions are executed on the interpreter. N/A means stack overflow occurred. Lower is better in this chart.

JIT compiler. Similar to us they have tried to approach to path divergence problem and introduced a pattern-matching-tracing. It is identical to Pycket's two-state-tracing; the basic idea is to place JIT merger points on the destination of conditional branches to distinguish different paths as different traces.

6. 5 Trace-based Java JIT Compiler

Inoue, et al. [7] presented the tracing JIT compiler. Opposite to our approach, they implemented by extending Java method JIT compiler by the compilation scope. They concluded that tracing JIT could reduce the method-invocation overhead, however, incurred additional runtime overhead than method JIT in their approach.

7 Conclusion and Future work

We have shown thequniques for realiizing trace- and method-based compilation as a meta JIT compiler. In such techniques, the stack hybridization is a key idea for enabling hybrid compilation in a single interpreter definition. Our preliminary benchmarks indicate that combining both compilation approaches will possibly solve the path divergence problem and improve the performance of tracing and meta-tracing compilations.

We are currently implementing gurad failure of BacCaml. As mentioned before, we use dynamic loading for just-in-time compilation. The challenging is that, when we apply tracing compilation, control should jump to compiled code, but dynamic loading, which we use for JIT compilation, does not provide any API to jump to native code. Once this is solved, we have to achieve run-time combination of both compilation strategies by using the stack hybridization, and test variety of programs on BacCaml to validate our work.

References

[1] Bala, V., Duesterwald, E., and Banerjia, S.: Dynamo: a transparent dynamic optimization system, *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

[2] Bebenita, M., Brandner, F., Fahndrich, M., Logozzo, F., Schulte, W., Tillmann, N., and Ven-

ter, H.: SPUR: A Trace-based JIT Compiler for CIL, *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, New York, NY, USA, ACM, 2010, pp. 708–725.

[3] Bolz, C. F., Cuni, A., Fijalkowski, M., and Rigo, A.: 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), pp. 18–25.

[4] Bolz, C. F. and Tratt, L.: The impact of meta-tracing on VM design and implementation, *Science of Computer Programming*, (2015).

[5] Gal, A., Orendorff, J., Ruderman, J., Smith, E. W., Reitmaier, R., Bebenita, M., Chang, M., Franz, M., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M. R., Kaplan, B., Hoare, G., and Zbarsky, B.: Trace-based just-in-time type specialization for dynamic languages, *ACM SIGPLAN Notices*, Vol. 44, No. 6(2009), pp. 465.

[6] Huang, R., Masuhara, H., and Aotani, T.: Improving Sequential Performance of Erlang Based on a Meta-tracing Just-In-Time Compiler, *The 17th Symposium on Trends in Functional Programming - TFP 2016*, 2016.

[7] Inoue, H., Hayashizaki, H., Wu, P., and Nakatani, T.: A trace-based Java JIT compiler retrofitted from a method-based compiler, *Proceedings - International Symposium on Code Generation and Optimization, CGO 2011*, (2011), pp. 246–256.

[8] Mozilla: IonMonkey, the next generation JavaScript JIT for SpiderMonkey.

[9] Oracle Lab.: A high-performance implementation of the R programming language, built on GraalVM.

[10] Oracle Lab.: A high performance implementation of the Ruby programming language.

[11] Oracle Labs.: Graal/Truffle-based implementation of Python., 2018.

[12] Perez, G. A., Kao, C.-M., Chung, Y.-C., and Hsu, W.-C.: A hybrid just-in-time compiler for android, *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems - CASES '12*, (2012), pp. 41.

[13] Sumii, E.: MinCaml: A Simple and Efficient Compiler for a Minimal Functional Language, *FDPE: Workshop on Functional and Declaritive Programming in Education*, (2005), pp. 27–38.

[14] Topaz Project: A high performance ruby, written in RPython.

[15] Würthinger, T., Wimmer, C., Humer, C., Wöß, A., Stadler, L., Seaton, C., Duboscq, G., Simon, D., and Grimmer, M.: Practical partial evaluation for high-performance dynamic language runtimes, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vol. 52, No. 6(2017), pp. 662–676.

[16] Würthinger, T., Wöß, A., Stadler, L., Duboscq, G., Simon, D., and Wimmer, C.: Self-optimizing AST interpreters, *Proceedings of the 8th symposium on Dynamic languages - DLS '12*, (2012), pp. 73.