TOKYO INSTITUTE OF TECHNOLOGY

MASTER THESIS

STACK HYBRIDIZATION: A Mechanism for Bridging Two Compilation Strategies in a Meta Compiler Framework

Author:
Yusuke IZAWA

Student Number:
18M30019

Supervisor:
Prof. Hidehiko MASUHARA

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Programming Research Group
Department of Mathematical and Computing Science

February 20, 2020

Declaration of Authorship

I, Yusuke IZAWA, declare that this thesis titled, "STACK HYBRIDIZATION: A Mechanism for Bridging Two Compilation Strategies in a Meta Compiler Framework " and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

TOKYO INSTITUTE OF TECHNOLOGY

Abstract

School of Computing
Department of Mathematical and Computing Science

Master of Science

STACK HYBRIDIZATION: A Mechanism for Bridging Two Compilation Strategies in a Meta Compiler Framework

by Yusuke IZAWA

Most virtual machines employ just-in-time (JIT) compilers to achieve their high-performance. Trace-based compilation and method-based compilation are two major compilation strategies in JIT compilers. In general, the former excels in compiling programs with more in-depth method calls and more dynamic branches, while the latter is suitable for a wide range of applications.

Some previous studies have suggested that each trace- and method-based compilation has its pros and cons. A trace-based compilation is quite suitable for programs with many branching possibilities because it records the really-executed path of an application and translates it into machine code. It is also able to compile only the necessary and sufficient parts of an application. However, it performs much worse in a tracing complicated control flow because of the occurrence of many side exits. On the other hand, a method-based compilation is so robust that we can apply it to any program. Nonetheless, we need more careful management for generating code. This is because a method-based strategy usually compiles functions, including less-executed parts. The side of the generated code tends to be bigger than that in the trace-based compilation.

In this thesis, we present a new approach, namely meta-hybrid JIT compilation combining trace- and method-based compilations to utilize the advantages of both JIT compilation strategies, and it is realized as a meta JIT compiler framework. It is so hybrid that the runtime can selectively apply different compilation strategies for different program parts. Furthermore, using our framework, language developers can realize a virtual machine empowered by a hybrid JIT compiler by merely writing an interpreter definition. Besides, we propose a mechanism to integrate two kinds of JIT compilation strategies, namely stack hybridization. Stack hybridization enables the runtime to go back and forth between the native code generated from different strategies as a single execution.

When we apply trace- and method-based compilations for a program with branching possibilities and a program with complicated control flow respectively, the runtime is about 1.1x faster than using a single JIT strategy for our microbenchmark results. To solve the trace-offs, we have to use a suitable approach depending on the structure of a program.

Acknowledgements

I would first like to thank my thesis advisor Prof. Hidehiko Masuhara; professor of the Department of Mathematical and Computing Science at Tokyo Institute of Technology, and Dr. Youyou Cong; assistant professor of the Department of Mathematical and Computing Science at Tokyo Institute of Technology. The doors to Prof. Masuhara and Dr. Youyou offices were always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this paper to be my own work, but steered me in the right the direction whenever he thought I needed it.

I would like to thank all members of our research group; Yudai Tanabe, Aochi Shu, Jizhe Chenxin, Fathul Asrar Alfansuri, Lubis Luthfan, Kazuki Niimi, and Tomoki Ogushi. Passionate discussions with them greatly help to advance my research.

I also would like to express my gratitude to Haruna Abe and Ichigo Hoshimiya. They kept me going on and this work would not have been possible without their input.

Finally, I must express my very profound gratitude to my parents and to my labmates for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Yusuke Izawa

Contents

List of Figures

Chapter 1

Introduction

Just-in-time (JIT) compilation is widely used for modern programming language implementations including Java [31, 13], JavaScript [14, 24, 16], and PHP [1, 30] to name a few. While there are a variety of JIT compilers, they commonly identify "hot spots", or frequently executed parts of a program at runtime, collect runtime information, and generate compiled code. By compiling only frequently-executed parts of a program, the compilation is fast enough to be performed at runtime. By exploiting runtime information, those compilers can perform aggressive optimization techniques such as inlining and type specialization and produce as efficient—sometimes more efficient—code as the code generated traditional static compilers.

JIT compilers can be classified by the strategy of selecting compilation targets, namely the method-based and the trace-based strategies. The method-based one uses methods (or functions) as a unit of compilation, which has been used in many JIT compilers [10, 40, 31]. It shares many optimization techniques with traditional static compilers. The trace-based one uses a *trace* of an execution of a program, which is a sequence of instructions during a particular run of a program, as a compilation unit [2, 4, 9, 14]. It effectively performs inlining, loop unrolling and type specialization.

There is no clear winner between those two strategies; instead, each of them works better for different kinds of programs. While the method-based strategy works well on average, the trace-based strategy exhibits polarized performance characteristics. It works better for programs with a lot of biased conditional branches and method calls [2, 9, 14, 21]. However, the trace-based strategy can cause severe overheads when the path of an execution is highly varying, which is known as the *path-divergence problem* [19, 20].

It is straightforward to combine those two strategies; i.e., when compiling a different part of a program, using a strategy that works better for that part. Then the questions are (1) how we can construct such a compilation engine without actually creating two very different compilers, (2) how the code fragments that are compiled by different strategies interact with each other, and (3) how we can determine a compilation strategy to compile a part of a program. There are not many studies on these regards. As far as the authors know, HHVM [30] is the only VM that tries to support both strategies, which is designed for PHP, and implemented by making a method-based compiler more flexible on selecting compilation targets.

We propose a new approach called *meta-hybrid* JIT *compilation* to combining trace- and method-based strategies in order to study the above questions (1) and (2), while leaving the question (3) for future work. One major difference from previous work is that we design it as a *meta*-JIT compilation framework, which can generate a JIT compiler for a new language by merely writing an interpreter of the

language. Moreover, we design it by extending meta-*tracing* compilation framework by following the RPython's [9, 11] architecture.

There are many possible approaches to promote the two kinds of JIT compilations in a meta JIT compiler framework, such as building and coordinating two types of framework, or building a single framework and developing two types of compilation approaches on that framework. For simplicity, we decided to choose the latter. We build a method-based compilation based on a meta-tracing JIT compiler and make the two compilations coexist in a meta JIT compiler framework. From the perspective of language developers, a meta-tracing compiler approach is more natural than a partial evaluation approach, since developers only are only concerned about the behaviors of interpreters [23].

The basic idea of realizing this compilation is to just cover a method by tracing. Since a trace has no control flow and inlines a function, we realize a method-based compilation by customizing the tracing JITs' features to cover all the paths in a method. In this way, our meta-hybrid JIT compiler shares its implementations.

In addition to the leverage strength of the two JIT compilation policies, we aim to resolve the path divergence problem [1] by selectively applying method-based compilation to the functions that cause the problem, while applying trace compilation to the other parts of a program.

Since the two meta compilation strategies require stack frames in different ways, we cannot naively connect the native code from distinct strategies. This difference makes the runtime go back and forth between the native codes generated from distinct strategies.

To overcome this, we also propose Stack Hybridization, a mechanism to bridge the native codes generated from different strategies. Stack Hybridization manages different kinds of stack frames and makes the compiler generate machine code that can be mutually executed in trace-JIT and method-JIT contexts. To support Stack Hybridization, a language developer needs to write a meta-interpreter in a specific way: (1) For executing a call instruction in the base language, which the developers are going to realize, they put a special flag to indicate which stack frame is used in a self-prepared stack data structure. (2) For executing a return, they have to branch to the return instruction of the base language corresponding to the call by checking the already pushed flag.

We implemented BacCaml, a proof-of-concept implementation of the framework. Though its architecture is based on the RPython's, it is a completely different and much simpler implementation written in OCaml [2]. As the compiler backend, we modified the MinCaml compiler [37].

Contributions and Outline This thesis has the following contributions.

- We proposed an idea of the meta-hybrid JIT compilation framework that can apply both trace- and method-based compilation strategies to different parts of a program under a meta-compilation framework.

[1] Illustrated in Appendix B.

[2] Since we did not know the requirements of the compilation frameworks in order to support method-based compilation, we implemented them from scratch instead of extending the existing frameworks like RPython.

- We presented a technique to achieve method-based compilation with a meta-tracing compiler by controlling the tracing compiler to cover all the paths in a method.
- We identified a problem of interconnecting code fragments compiled by the different strategies, and proposed a solution that dynamically switches the usage of call stacks.
- We implemented a prototype framework called BacCaml, and confirmed that there are programs where the hybrid compilation strategy performs better.

The remainder of this paper is organized as follows. Chapter 2 gives a background of our research. Chapter 3 describes the architecture and provides an overview of the dynamic compilation steps of our meta-hybrid JIT compiler framework. Chapter 4 shows how to implement method-based compilation based on a meta-tracing JIT compiler. Chapter 5 illustrates how we bridge trace- and method-based compilations in a meta JIT compiler framework and describes a meta-interpreter design supporting this idea. Chapter 6 evaluates the application performance of our implementation using a small and simple language built with BacCaml. Chapter 7 compares our approach with other frameworks and JIT compilers. Finally, Chapter 8 concludes the thesis.

Chapter 2

Background

Before presenting the concept and implementation of our meta-hybrid compiler, we briefly review the background of our work.

2.1 Just-in-Time Compilation

Traditionally, there are two approaches to translate programs: compilation and interpretation. Compilation translates a whole language into another, e.g., C to assembly, and reproduce native code by repeating such steps. Finally, we execute the produced native code. Interpretation removes such intermediate steps and directly performs execution immediately. Interpretation is a straightforward approach to build a programming language runtime, and many kinds of scripting programming languages such as Ruby, Python, and JavaScript adopted this technology for realizing themselves. In fact, Interpretation is much slower than ahead-of-time (AOT) compilers. Therefore, interpretation approach is not suited for developing the performance-critical software. However, by employing just-in-time (JIT) compilation techniques solves such a serious performance problem. Roughly speaking, JIT compilation combines compilation and interpretation: it interprets input sources at first, compiles frequently-executed parts into native code, and runs it lately [1]. By applying many optimization techniques such as eliminating type checking, constant folding, and inlining depending on runtime information, JIT compilers can produce highly optimized native code. Recent VMs, such as Java Virtual Machine (JVM) or Common Language Runtime, employed JIT compilation techniques and succeeded in multiple-order speed up comparing to merely interpretation strategy.

We can classify JIT compilation strategies into several types in terms of compilation unit: basic blocks (QEMU [5], and 1st generation of HHVM [1].), regions (IMPACT [17], IBM's Java region JIT [36], and 2nd generation of HHVM [30]), methods (SELF [40], and HotSpot [31]), and traces (Dynamo [2], and TraceMonkey [14]). Among these variety of JIT compilation strategies, method-based JIT and trace-based JIT are widely used JIT approaches, we employ them in our meta JIT compiler framework. Therefore, we give some overviews of method- and trace-based JIT compilers in the next sections.

[1] In other words, JIT compilation makes flexibility with interpretation and the speed of compiled code.

2.1.1 Method JIT Compilation

Traditional JIT compilers, such as SELF and JVM, adopt method-based compilation; they compile blocks of code, such as methods, or loop bodies, to native code. By using the profiling data, the compilers identify frequently-executed methods and compile it into native code. The advantages of method-based compilation are as follows: Firstly, the same optimization techniques employed in AOT can be applied for method-based compilation. Thus method-based JIT compilation can leverage already existing AOT compilers. This also makes compiler infrastructures more useful: JIT compiler backends such as GCC [15] and LLVM [12] provide JIT compilation APIs, and they can be used as the backend of method-based JIT compilers. Furthermore, method-based compilation can be applied for various types of programs, since method-based JITs translate the structure of a target method as it is. However, because they do not include type information, it is difficult to use method JITs for extensive method inlining.

2.1.2 Tracing JIT Compilation

Tracing optimization was initially investigated by the Dynamo project [2], and its technique was adopted for implementing compilers for many languages such as Lua [32], JavaScript [14], Java trace-JIT [13, 21] and the SPUR project [4].

Tracing JIT compilers track the execution of a program and generate a machine code with hot paths. They convert a sequential code path called *trace* into native code while interprets others [6]. "Trace" is a straight-line program; therefore, every possible branch is selected as the actually-executed one. In order to ensure that the condition in tracing and execution is the same, a *guard* code is placed at every possible point (e.g., if statements) that go to another direction. The guard code performs to determine 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.

Tracing JIT compilers usually consist of the following components: (1) *profiler*: it profiles runtime information and detecting a hot path of a target program. (2) *tracer*: it records really-executed instructions of a target. It stops to record when when reaching the point where it stared taking the trace. (3) *trace specializer*: It optimizes the resulting trace and converts it into machine code.

Tracing JIT compilers track the execution of a program and generate native code of hot paths. They convert a sequential code path called *trace* into native, while interprets others [6]. "Trace" is a straight-line of a program, so every possible branch is selected only actually-executed one. In order to make sure that the condition in tracing and execution is the same, a *guard* code is placed at every possible point (e.g., if statements) that go to another direction. The guard checks to determine 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.

Please look up a simple example. The program counts numbers from 1 to 100000; if a number is a multiple of 41, the function adds the number multiplied by 41. In Figure 2.1, the hot spot is the `for` loop in the function `f`, so this part becomes a trace. Then this trace converted into machine code and executed. Note that the function call to `strange_num` is inlined, and the `if` statement is converted into a `guard_false` instruction. The `guard_false` instruction checks the whether

```
def strange_num(a):
  if a % 41 == 0:
    return a * 41
  else:
    return a

def f(n):
  i = 0
  for j in range(n):
    i += strange_num(j)
  return i

f(100000)
```

```
# corresponding trace:
loop_header(i0, j0)
j1 = int_mod(j0, Const(41))
j2 = int_eq(j1, Const(0))
guard_false(j2)
j3 = int_add(j0, Const(1))
i1 = int_add(i0, j3)
jump(i1, j3)
```

FIGURE 2.1: A simple Python program and its' recorded trace.

variable `j2` is false. If `j2` is false, the execution continues. However, if `j2` is true, the execution is stopped, and other programs are interpreted.

Tracing JIT compilation is suitable for dynamically typed languages. The compilers for statically typed languages use annotated type information for generating efficient machine code. In dynamically typed languages, the types of variables vary at runtime; therefore, the compilers cannot perform type-specific optimizations for some instructions depending on the types at compilation time. However, tracing JIT can collect the runtime type information of variables and generate efficient machine code. Since compilation codes include only one execution path, compilers can apply many optimizations techniques [8] such as constant-subexpression elimination, dead-code elimination, constant-folding, and register allocation removal [7]. Moreover, tracing JITs can perform aggressive function inlining calls at a low cost. A tracing JIT compiler follows the execution of a program so that a resulting trace will include an inlined function call [13]. This leads to reducing overheads of a function call and making chances for further optimization.

2.2 Meta JIT Compiler Framework

n JIT compilers have a great effect on the performance of VMs, but implementing them require painful and error-prone engineering tasks of language developers. Recently, some researches proposed a *meta* JIT *compiler framework* to overcome this problem. A meta JIT compiler frameworks provide a convenient and effective way to implement a high-performance VM empowered by a JIT compiler. There are two approaches in meta JIT compiler frameworks. Trace-based meta compilation, i.e., *meta-tracing*, is the first practical attempt for general interpreters. Meta-tracing technique was firstly used for implementing PyPy [35], but its system turned out to be usable for implementing other programming languages, such as JavaScript [38], Ruby [39], and Smalltalk [11]. Meta-tracing system enforces a language developer to write a bytecode-formatted *meta*-interpreter, since its meta-tracing JIT compiler is highly optimized for operating sequential data structure.

Practical method-based (or partial evaluation-based) meta compilation, on the other hand, was recently proposed by Würthinger et al. [43, 41] called *self-optimizing interpreter*. Its meta JIT compiler translates abstract-syntax-tree nodes selected by a

```
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
```

FIGURE 2.2: An example meta-interpreter definition written in RPython.

partial evaluator into native code. This approach is used for implementing Ruby [28], R [29], JavaScript [27] and Smalltalk [26]. In contrast to meta-tracing, an user of self-optimizing interpreter has to write AST-based *meta*-interpreter definition.

In the next sections, we overview the meta-tracing JIT compiler framework that is the foundation of our work.

2.2.1 Meta-tracing JIT Compiler

Typically, tracing JIT compilers record a representation of the program; however, a meta-tracing JIT compiler traces the execution of a *meta-interpreter* defined by a language builder. Meta-tracing compilation is just tracing compilation: in the sense that it compiles a *path* of a base program, even if it has conditional branches. If it has, the compiled code will contain *guards*, each of which is a conditional branch to the interpreter execution from that point.

RPython [9, 6], a statically typed subset of Python programming language, is a tool-chain for creating high-performance VMs empowered by a trace-based JIT compiler. It requires a language builder for implementing a bytecode compiler and a meta-interpreter definition for the bytecode. BacCaml is based on RPython's architecture. Before describing the details of BacCaml, let us give an overview of RPython's meta-tracing JIT compilation.

To leverage the RPython JIT, a meta-interpreter developer should annotate to help identify the loops in the *base-program* that is going to be interpreted. Figure 2.2 shows an example of a meta-interpreter defined by a programming language builder. The example uses two annotations, `jit_merge_point` and `can_enter_jit`. A developer should put `jit_merge_point` at the top of a dispatch loop to identify which part is the main loop, and `can_enter_jit` at the point where a back-edge instruction can occur (where meta-tracing compilation might start).

Algorithms 1 and 2 illustrate the meta-tracing compilation algorithm in pseudocode. The procedure *JitMetaTracing* takes the following arguments: *rep* – a representation for the meta-interpreter and *states* – the state of the meta-interpreter just in starting to trace. A meta-tracing JIT compiler records the execution and checks the operands in the executed operations. It uses *red* and *green* colors for recognizing runtime information. The color *red* means "a variable in a base language"; hence, red variables are used for calculating the result of a base program.

Algorithm 1: JitMetaTracing(rep, states)

```
input : Representations of a meta-interpreter itself
input : States (e.g., virtual registers and memories) of a meta-interpreter
        itself
output: The resulting trace of the hot spot in a base program
entry_states ← states;
repeat
    residue ← [ ] ; // A data to store the result
    op ← rep.current_operation(states);
    if op = conditional branch then
        if op has red variables then
            guard ← op.mk_guard(states);
            residue.append(guard);
        eval(op, states, residue);
    else if op = function call to f then
        inline f;
    else
        eval(op, states, residue);
until op = jit_merge_point ∧ entry_states = states;
return residue;
```

Algorithm 2: Eval(op, states, residue)

```
if op has red variable then
    op.const_fold(states);
    residue.append(op);
else
    op.execute(states);
```

The color *green* indicates "a variable in a meta-interpreter", then the compiler will optimize this variable by constant-folding or inlining. If all the operands in one operation are green, the operation is only used for calculation in an interpreter, and therefore the compiler executes it. If at least one variable is red, the compiler recognizes the operation is in a base program and writes to the *residue*.

One significant advantage of the meta-tracing compilation strategy depicted by RPython is that it is more comfortable to write meta-interpreters in comparison to the AST-rewriting specialization in Truffle. In [23], S. Marr and S. Ducasse stated that a significant difference between RPython and Truffle is the number of optimizations a language implementer needs to apply to reach the same level of performance. In their experiments Michael et al. found that SOM [18] built with RPython achieves excellent performance without adding many optimizations. On the other hand, SOM built with Truffle without any additional optimizations performs one-order worse than the meta-tracing. By adding some optimizations, SOM with Truffle reaches the same level as that of SOM with RPython. From these experiments, they concluded that the meta-tracing strategy has significant benefits from an engineering perspective [23].

Chapter 3

Meta-hybrid JIT Compilation Approach

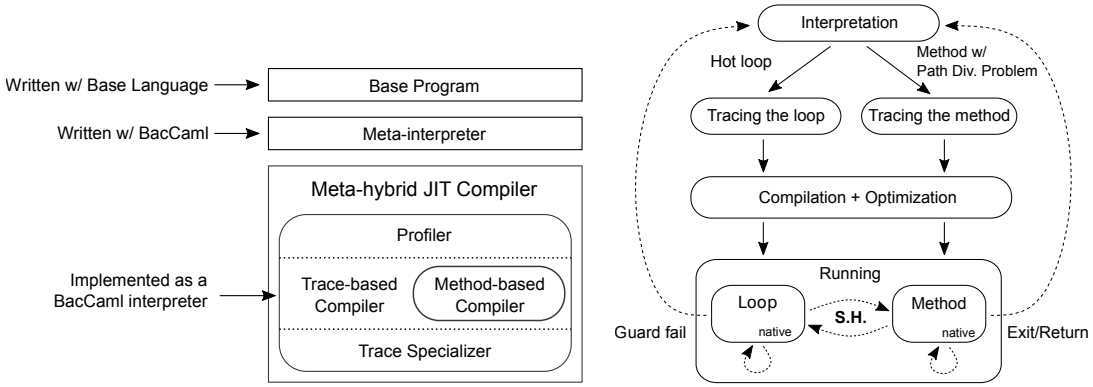
In this section, we explain the trade-offs between trace- and method-based compilation, and introduce our approach to solve the trade-offs.

3.1 Trade-offs between Trace- and Method-based Compilation

The advantages of method-based compilation are the following: Firstly, the same optimization techniques employed in AOT can be applied for method-based compilation. Thus, it can leverage already existing AOT compiler engines such as GCC [15] and LLVM [12]. Secondly, method-based compilation is so mature and well-studied that it can be applied for various shaped programs, since a method-based JIT compile a method without changing its structure. Thus, it performs better on average than trace-based compilation [21, 23]. However, when it compiles a method with many biased branches, the compiled code includes colds spots of the method, causing compilation time longer than the case of applying trace-based compilation to it. Furthermore, it needs a well-planned method inlining for reducing the overhead of a function call. When it applies aggressive inlining to a method with deeply-nested function calls, the size of the compiled code goes big, and it puts high pressure on the runtime.

On the other hand, trace-based compilation can apply many optimizations techniques [8] including constant-subexpression elimination, dead-code elimination, constant-folding, and register allocation removal [7], since compilation code represent only one execution path. Thus, trace-based compilation get better results with certain programs with branching possibilities or loops [3, 20]. Moreover, it can perform aggressive function inlining calls at a low cost; this is because a trace-based JIT compiler follows the execution of a program so that a resulting trace will include an inlined function call [13]. This leads to reducing overheads of a function call and making chances for further optimization. However, trace-based compilation works worse at programs with complex control flow. This problem is caused by the mismatch between tracing and execution [20] [1] than the case of applying method-based compilation for it.

[1] Details are explained in Appendix B



(A) Structure overview of the meta-hybrid JIT compiler framework.

(B) Dynamic compilation overview by the meta-hybrid JIT compiler. Black line is data flow, and dotted line is control flow.

FIGURE 3.1: The architecture and compilation steps overview of a meta-hybrid JIT compiler framework.

3.2 Proposal: Meta-hybrid JIT Compiler Framework

To overcome the trade-offs explained above, we propose a *meta-hybrid* JIT *compiler framework*. The framework is a *meta*-JIT compiler framework; therefore the language developer needs to write an interpreter definition to enable JIT compilation. It is a hybrid of the trace- and method-based compilations as it can compile both an execution trace and a function of a base program. The compiled code from the two types of strategies can work together in a single execution. Moreover, this compiler is based on a meta-tracing compiler; therefore, a method-based compilation is created by extending a tracing compilation.

The basic idea of realizing hybrid compilation is covering a method by tracing. Since a trace has no control flow and inlines a function, we realize a method-based compilation by customizing the tracing JITs' features to cover all the paths in a method. In this way, our meta-hybrid JIT compiler shares its implementations. In addition to the leverage strength of the two JIT compilation policies, we aim to resolve the path divergence problem by selectively applying method-based compilation to the functions that cause the problem, while applying trace compilation to the other parts of a program.

Since the two meta compilation strategies require stack frames in different ways, we cannot naively connect the native code generated from different strategies. To overcome this, we also propose Stack Hybridization, a mechanism to bridge the native codes generated from different strategies. Stack Hybridization manages different kinds of stack frames and makes the compiler generate machine code that can be mutually executed in trace-JIT and method-JIT contexts. To support Stack Hybridization, a language developer needs to write a meta-interpreter in the specific way: (1) For executing a call instruction in the base language, developers put a special flag to indicate which stack frame is used in a self-prepared stack data structure. (2) For executing a return, they have to branch to the return instruction of the base language corresponding to the call by checking the already pushed flag.

Figure 3.1a overviews the structure of our meta-hybrid JIT compiler framework. A base language is a programming language that a language developer

is going to create. A meta-interpreter is the interpreter for a base language written with the framework. The framework is based on a meta-tracing JIT compiler framework. As a result, the method-based compiler is retrofitted from the trace-based compiler and shares many parts of the implementations. It consists of the following components. (1) *profiler:* It profiles the runtime information of the interpreter and detects which compilation strategy (method- or trace-based) is preferred. Furthermore, it chooses a compiled trace to jump. (2) *hybrid* JIT *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 meta-interpreter. (3) *trace specializer*: It performs optimization of the resulting traces and converts them into native code. Both trace- and method-based compilers share this specializer.

Figure 3.1b shows the outline of BacCaml's dynamic compilation steps. First, in the interpreting phase, the profiler monitors the runtime information and detects which program parts are hot. At first, the hybrid JIT compiler applies trace compilation for the hot spot, and compiles it into native code. When many guard failures happen in some program code, the profiler tries to find a function, including that part. If it succeeds, instead of the trace-based compiler, the method-based compiler is launched to compile that function into native code. With *Stack Hybridization*, the runtime can use the appropriate mode of the stack frame for executing calls, and the loops and methods can call each other.

In Chapter 4, we describe how to construct a method-based compilation based on a meta-tracing JIT compiler. Chapter 5.3 illustrates how Stack Hybridization works and how to implement a meta-interpreter for supporting Stack Hybridization.

Chapter 4

Method JIT by Tracing

In this chapter, we provide the details of method JIT compilation based on tracing JIT compilation and discuss the problem of implementing an interpreter in the meta-hybrid JIT compiler in one definition.

To construct method JIT by utilizing trace-based compilation, we have to cover all paths of a function. In other words, we need to determine the *true path* and decrease the numbers of guard failures occur to solve the performance degradation problem while tracing JITs. We propose such a method-based JIT compilation by customizing the following features:

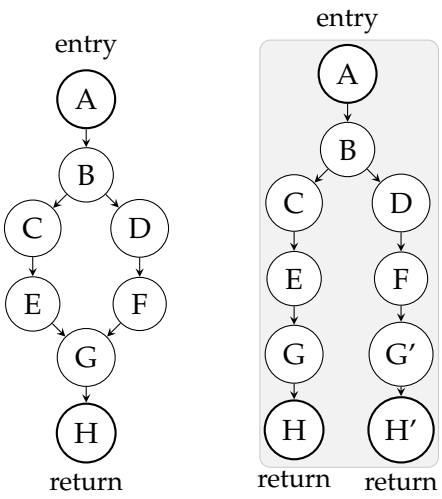
1. trace entry/exit points
2. conditional branches
3. loops
4. function calls

In the following paragraphs, we explain in detail how to trace a method by modifying these features.

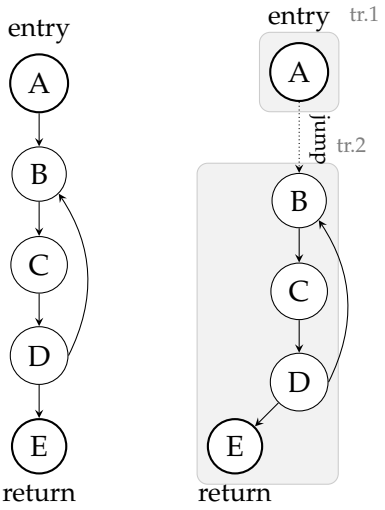
Trace entry/exit points Tracing JIT compilers [13, 14] generally compile loops in the base program; therefore, they start to trace at the top of a loop and end when the execution returns to the entry point. To assemble the entire body of a function, we modify this behavior to trace from the top of a method body until a `return` instruction is reached (see Algorithm 3).

Conditional branches When handling a conditional branch, the tracing JIT compilers convert the code into a guard instruction and collect the instructions that are executed. In method JIT, however, we must compile both sides of the conditional branches. In order to achieve this, the tracer must return to the branch point and restart to trace the other side as well. As shown in Algorithm 4, the tracer in the method JIT policy has to trace both then and else sides, so that it *backtracks* to the beginning of a conditional branch when it reaches the end of one side and continues to trace the other side. At tracing one side, the tracer stores its states (e.g., the data stored in the tracer's virtual registers and memories) in already prepared arrays. For just backtracking, the tracer *restores* those states and continues to the other side.

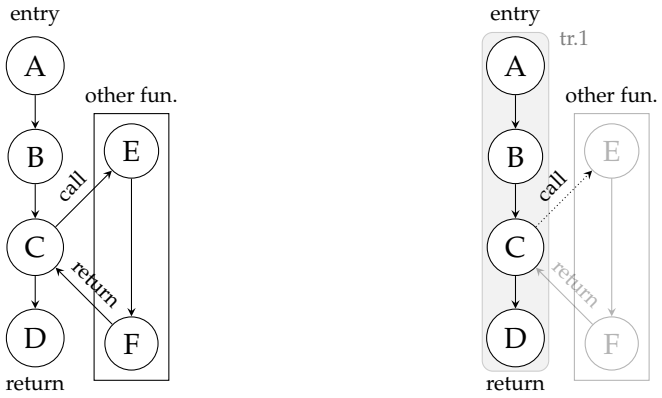
Figure 4.1a shows an example to describe how the method JIT tracer works. on the left-hand side, node A is the method entry, nodes B – C – D form a conditional branch, and node E is the end of this method. At first, the tracer starts to trace at



(A) Handling of a conditional branch. In this program, A is a method entry, B – C – D is a conditional branch.



(B) Handling of a loop. B – C – D represents a loop in this program.



(C) Handling of a function call. A – B – C – D and E – F are functions. In this program, (A – B – C – D) calls (E – F) at C. Note that only target function (A – B – C –D) is compiled.

FIGURE 4.1: Examples of method JIT by trace. Each left-hand side is the control-flow of a target base-program, and each right-hand side is a result.

A. On reaching a conditional branch (B), the tracer stores its state and follows one side (B – C – E – G – H). On reaching *return* instruction (H), the tracer backtracks to B and resumes to trace the other side (B – D – F – G – H) by restoring the already saved data.

Loops The method compiler compiles loops [1] without guard failure. Algorithm 5 and 5 illustrate how the tracer traces a loop in method-based compilation. At first, the tracer for method-based compilation analyzes the body of a function and tries to find a back-edge instruction. In order to find the loops, the meta-method analyzer symbolically executes the body of a program before recording instructions and tries to find a back-edge instruction. We utilize the fact that the developers

[1] Strictly speaking, loops indicate a control flow like `for` or `while`, without including a recursive call.

Algorithm 3: JitMetaMethod(rep, states) METHOD JIT

```
input : Representations of a meta-interpreter itself
input : States (e.g., virtual registers and memories) of a meta-interpreter
        itself
output: The resulting trace of the method in a base program
if op = method_entry then
    do
        /* trace the body of this method                    */
        if op = conditional branch then
            TraceCond (rep, states, residue);
        else if op = loop entry then
            TraceLoop (rep, states, residue);
        else if op = function call to f then
            TraceFunction (rep, states, residue);
        else
            eval(op, states, residue);
    while op = return;
else
    return;
```

of a meta-tracing compiler framework should add `can_enter_jit` when a back-edge instruction occurs. When the analyzer encounters `can_enter_jit`, it saves the entry points of a loop. Then, when all the entry points are gathered, the meta-method "tracer" starts to trace the body of a method. When it encounters the entry of a loop, it splits the trace and connects it with the successors by emitting a `jump` instruction. The rest of the program is traced as in the other cases. However, a back-edge instruction is converted into a `jump` instruction to the trace, including the loop [2].

Figure 4.1b shows an example of how to handle a loop. In this example, the method JIT compiler compiles a single loop into two trace parts. The first one (tr.1) is up to the loop entry, and the second one is the loop itself and the successors (tr.2) of the loop.

Function calls Whereas a tracing JIT will inline function calls, the method JIT will emit a call instruction code and continue tracing. Therefore, we make the tracer distinguish the part represented for the base program's function call. To inform the tracer which part is the base language's function call, the framework requires language developers of the meta-interpreter style shown in the left-hand side of Figure 4.2. By writing as such, the tracer can find the representation equivalent to the base program's function call from a meta-interpreter definition (`res = self.interp(addr)`), and continue to trace the successors of the function call. Figure 4.1c shows how the tracer compiles a function call. In this example, the tracer eventually generates two traces, the caller (tr.1) and callee (tr.2).

For optimizing a base program's non-tail recursive style function, developers should implement a meta-interpreter as shown in the right-hand side of Figure 4.2 and apply trace-based compilation for it since such a meta-interpreter definition

[2] Note that a loop condition is compiled as a conditional branch.

Algorithm 4: TraceCond(rep, states, residue) METHOD JIT

```
regs, mems ← [ ], [ ];
do
    if op = cond branch then
        regs.store(states.get_reg());
        mems.store(states.get_mem());
        trace_then ← JITMETAMETHOD(states);
        states.restore(regs, mems);
        trace_else ← JITMETAMETHOD(states);
        /* construct if exp including trace_then and trace_else
           */
        trace_ifexp ← begin
            if op.const_fold(states) then
                trace_then;
            else
                trace_else
        residue.append(trace_ifexp);
    rep.next();
while op = return;
```

Algorithm 5: TraceLoop(rep, states, residue) METHOD JIT

```
loop_states ← FindLoopEntry(rep, state);
do
    if loop_states.contains(state) then
        start_state, end_state ← loop_states.get_state_by(state);
        loop ← JITMETHOD(rep, start_state, residue);
        residue.append(jump to loop);
        residue.append(loop);
while op = return;
/* a helper function                                          */
Function FindLoopEntry(rep, state) do
    entry_state ← state;
    op ← rep.get_current_states();
    do
        if op = can_enter_jit then
            /* get the state of a loop entry by can_enter_jit  */
            entry_state ← rep.next_of(op).get_state();
            /* add a pair of the state of the "entry"          */
            results.append(entry_state);
        else
            rep.next_of(op);
    while op = return;
    return results
```

Algorithm 6: TraceFunction(rep, states, residue) METHOD JIT

```
do
    if op = function call to f then
        /* not following but leaving the instruction "call f"  */
        residue.append(call to f);
        /* trace successors                                    */
while op = return;
```

```
if opcode == CALL:
  addr = ord(self.bytecode[pc])
  # call the `interp' recursively
  res = self.interp(addr)
  user_stack.push(res)
  pc += 1
elif opcode == RETURN:
  # just return a top of `user-stack'
  return user_stack.pop()
```

```
if opcode == CALL:
  addr = ord(self.bytecode[pc])
  pc += 1
  # push a return address to
  # the user-stack
  user_stack.push(W_IntObject(pc))
  if addr < pc:
    can_enter_jit(..)
  # jump to a callee function
  pc = t
elif opcode == RETURN:
  v = user_stack.pop()
  # restore the already pushed
  # return address
  addr = user_stack.pop()
  user_stack.push(v)
  if addr < pc:
    can_enter_jit(..)
  # jump back to the caller
  ↪ function
  pc = addr
```

FIGURE 4.2: Meta-interpreter definition styles. For managing a return address/value, left-hand side style uses a host-language's (system provided) stack, but right-hand side uses a developer-prepared stack data structurpe.

allows the tracer to convert a non-tail recursive call into a loop (it is represented as a series of a meta-interpreter loop). The tracer can consider that this part is a loop when it comes back to the beginning of the function.

Considering the integration of the two compilation strategies, the difference between the meta-interpreter definition styles (Figure 4.2) causes an inability to traverse code compiled with different compilation methods (we discuss this in Section 5.2). To solve this problem, we propose a technique to execute trace- and method-compiled native codes in a single runtime (we explain this in Section 5).

Chapter 5

Bridging Tracing and Method Compilation in Meta-level

In this section, we first explain the problem that we cannot naively integrate trace- and method-based compilation strategies. Next, we introduce a mechanism to overcome this problem.

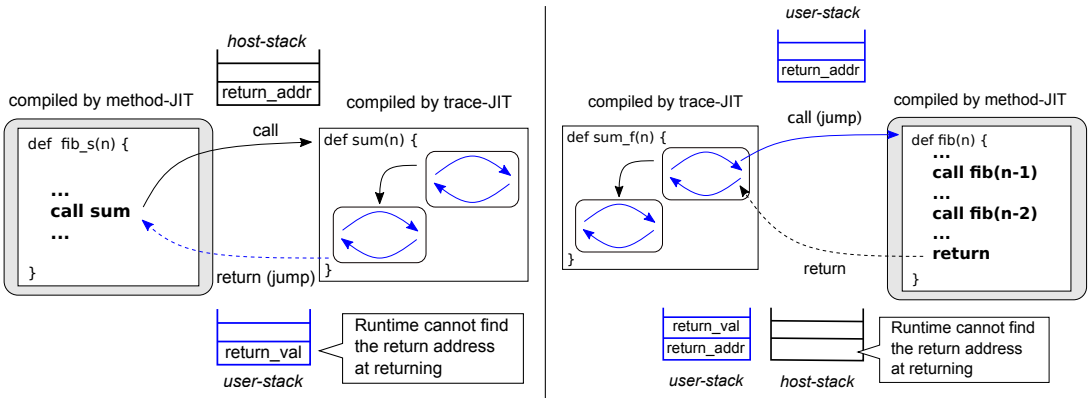
5.1 Two Meta-interpreter Definition Styles

To fully bring out the potential of meta JIT compilation, programming language developers are forced to implement a meta-interpreter in a specific way. RPython requires language developers to write a bytecode-based meta-interpreter definition, while the Truffle framework requires the implementation of an AST-based meta-interpreter. Riding on the rails provided by the frameworks is important, but choosing a suitable meta-interpreter implementation method depending on the style of the base language (e.g., functional or object-oriented programming language; bytecode-, AST-, or IR-based format, etc. . .) is more important. In the case of AST-rewriting interpreters, Sulong [34] and GraalSqueak [26] followed a bytecode-based AST execution since their base language specifications included unstructured control flow or a well-defined bytecode set, and therefore they designed it to run on a bytecode interpreter.

In a meta-tracing JIT compiler, we discovered that there are two possible ways: one is managing the states in a stack defined by a language developer (*user-stack style*), and the other is managing the states in a stack by a host language (*host-stack style*). With regard to the novel implementations by RPython, PyPy [1] and Topaz [2] employ the host-stack style for defining a method invocation. On the other hand, other implementations for functional programming languages, such as Pycket [3] and Pyrlang [4], employ the user-stack style.

The difference between the two styles (shown in Figure 4.2, and their trade-offs are explained in Appendix A) is in the way they manage a return address, return value, and callee address. We call the left-hand side one as *host-stack style*, and the right-hand one as *user-stack style*. In the user-stack style, all of them are managed by a user-defined stack data structure. In contrast, in the host-stack style, the return value is only managed by a user-defined stack, and the stack provided by a host-language manages other variables.

[1] https://bitbucket.org/pypy/pypy/src/default/
[2] https://github.com/topazproject/topaz
[3] https://github.com/pycket/pycket
[4] https://bitbucket.org/hrc706/pyrlang/src/master/



(A) Calling a trace-compiled code from a method-compiled function.

(B) Calling a method-compiled function from a trace-compiled code.

FIGURE 5.1: Example of Integration Problem. Gray background code is compiled by method JIT, and blue lined code is compiled by tracing JIT.

As a result, the user-stack style is suitable for trace-based compilation, while the host-stack style is suitable for method-based compilation. The reasons are as follows: Firstly, a trace-based compiler can automatically inline a function and also convert a non-tail recursive call into a loop, since the definition of function call in a meta-interpreter is implemented as a "jump" (left-hand side of Figure 4.2). Secondly, method-based compilation can distinguish which part is a function call of a "base-program" and spontaneously follow the successors of the function call.

5.2 Integration Problem

Naive integration of trace- and method-based compilations is not allowed, because of the following reason: the two types of compilations require different meta-interpreter implementation styles. Trace-based compilation requires the *user-stack style*, while method-based compilation requires the *host-stack style*. In other words, the native codes from different types of compilations use different stack frames. Because of this gap, the runtime cannot call back and forth between native codes generated from the two compilations. We call this problem *Integration Problem*.

Trace-based compilation inlines a function call; therefore, there is no function call instruction in the resulting trace. On the other hand, method-based compilation "leaves" a function call instruction in the resulting trace. We explain this problem by using Figures 5.1a and 5.1b. This problem happens when calling a trace-compiled function from a method-compiled function, and vise versa.

In the case that `fib_s` (method-compiled) calls `sum` (trace-compiled) as shown in Figure 5.1a, the runtime puts a return address in the host-stack. In `sum`, the return value and return address are stored in the user-stack. On returning from `sum`, since the semantics of return is defined as shown in Figure 4.2, the runtime tries to find a return address from a user-stack. However, the return address is stored in a host-stack, and the runtime cannot return to the correct place.

On the other hand, in the case that `sum_f` (trace-compiled) calls `fib` (method-compiled), first, the runtime puts its return address in the user-stack. When runtime returns from `fib`, it tries to find the return address from the host-stack, but it

```
if instr == CALL:
   addr = bytecode[pc]
   # get an annotation from bytecode
   if mj_context(addr):
     # push JIT flag to user-defined
     ↪ stack (HS)
     user_stack.push(HS)
     return_value = interp(addr)
     user_stack.push(return_value)
   else:
     # push JIT flag to user-defined
     ↪ stack (US)
     user_stack.push(US)
     user_stack.push(pc + 1)
     pc = addr
```

```
elif instr == RETURN:
  return_value = user_stack.pop()
  # get JIT context flag from the
  ↪ user-defined stack
  JIT_flag = user_stack.pop()
  # check the JTI context and branch
  if JIT_flag == HS:
    return return_value
  else:
    return_addr = user_stack.pop()
    user_stack.push(return_value)
    pc = return_addr
```

FIGURE 5.2: A sketch of meta-interpreter definition with Stack Hybridization. Some hint functions (e.g., `can_enter_jit` and `jit_merge_point`), and other definitions are omitted for simplicity.

fails to find the address, resulting in a runtime-error because the return address is pushed to the user-stack.

5.3 Stack Hybridization

To address the Integration Problem, we propose *Stack Hybridization*: a meta-interpreter design to enable runtime to switch the context of function call. Roughly speaking, the meta-interpreter handles the 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 using the host-stack.

For supporting these behaviors, we propose a meta-interpreter implementation pattern: we embed both styles into a single meta-interpreter and switch its behavior depending on the flag. Figure 5.2 shows a sketch of some special syntax to support Stack Hybridization. For implementing `call`, the developer separates the host- and user-stack style using the `mj_context(addr)` function. This function returns `true` under the method JIT context, otherwise `false`. `addr` is used for emitting trace (described in Section 5.4). The host-stack styled definition is placed in

```
# method-compiled fun.
def fib(n):
  if n <= 2:
    return n
  else:
    return fib(n-1) + fib(n-2)

# trace-compiled fun.
def summary(n):
  if n <= 2:
    return n
  else:
    return fib(n) + summary(n-1)

summary(30)
```

(A) Calling method-compiled code from method-compiled function.

```
# method-compiled fun.
def fib(n):
  if n <= 2:
    return summary(n)
  else:
    return fib(n-1) + fib(n-2)

# trace-compiled fun.
def summary(n):
  if n <= 2:
    return n
  else:
    return n + summary(n-1)

fib(30)
```

(B) Calling method-compiled code from method-compiled function.

FIGURE 5.3: Example base-programs, `fib` and `sum`. `fib` is compiled with method-based compilation, but `sum` is compiled with trace-based compilation.

the `then` branch, and user-stack styled definition is places in the `else` branch. The flags `US` and `HS` indicate under "user-stack" context, and under "host-stack" context, respectively. The developer needs to write an instruction to leave these flags at the top of each branch. For implementing `return`, first of all, the developer should first check the flag in the user-stack and then branch the process according to the flag.

5.4 Compilation with Stack Hybridization

By implementing a meta-interpreter as in Figure 5.2, the compiler can generate traces as explained in Section 5.3. When the compiler compiles the base programs shown in Figure 5.3, the tracer works as follows:

- When the tracer traces `mj_context(addr)`, it checks whether there is a compiled code at `addr`. If there is a native code compiled by method-based compilation, it does not inline a function call, but emits a jump instruction to the `addr`. This avoids the following problem: if the tracer continues to trace and enters a function causing the path-divergence problem, it keeps tracing almost infinitely, which results in a high runtime overhead.
- When the tracer traces the flag check part at `return`, it is converted into a guard function. If the method-compiled code is called under the context of trace-compile, then it lets the interpreter complete the remaining instructions before returning (and vise versa).

Figure 5.4 shows an overview of the resulting traces compiled with stack hybridization. Hence, we assume that `summation` and `fib` are determined to be compiled by the tracing JIT and method JIT, respectively, by the profiler. Since a resulting trace and the native code generated from the trace correspond one-to-one, we use the resulting traces for explaining how to compile.

In Figure 5.4a, given that `fib` has already been compiled by method JIT, the tracer traces `summation` under the tracing JIT context. When it reaches the `CALL` part, `mj_context` checks whether there is any compiled code and finds that `fib` has been already compiled. Then, it traces the user-stack styled `CALL` definition, but leaves the `jmp` instruction to `fib` and exits without tracing.

In Figure 5.4b, on the other hand, the tracer first traces `fib`. When it encounters the part of calling `summation`, the tracer does not follow the destination of calling `summation`. Because it is under method JIT context, it traces the host-stack styled `CALL` definition. Then, it leaves the instruction to call `summation` and continues to trace in the context of method JIT. Subsequently, `summation` is traced under the tracing JIT context, and its `CALL` and `RETURN` parts are traced as `jmp` instruction.

```
# trace: summation
loop_header("summation0",stack0,sp0)
...
store(stack0,HS,sp5)
sp6 = int_add(sp5,Const(1))
jmp("fib0",stack0,sp6)
```

```
# trace: fib
function_header("fib0",stack0,sp9)
...
store(stack0,HS,sp12)
sp13 = int_add(sp12,Const(1))
call("fib",stack0,sp13)
...
store(stack0,HS,sp19)
sp20 = int_add(sp12,Cons(1))
call("fib",stack0,sp20)
...
JIT_flag0 = load(stack0,sp23)
guard_eq(JIT_flag0,HS)
ret
```

(A) Resulting trace from Figure 5.3b.

```
# trace: fib
function_header("fib1",stack0,sp0)
...
store(stack0,HS,sp2)
sp3 = int_add(sp2,Const(1))
call("fib1",stack0,sp3)
...
store(stack0,HS,sp12)
sp13 = int_add(sp12,Const(1))
call("fib1",stack0,sp13)
...
store(stack0,HS,sp18)
sp19 = int_add(sp2,Const(1))
call("summation1",stack0,sp19)
...
ret
```

```
# trace: summation (the first
↪  recursive call part)
loop_header("summation1",stack0,sp0)
...
store(stack0,US,sp12)
sp13 = int_add(stack0,Const(1))
jmp("summation1",stack0,sp13)
```

```
# trace: summation (the return part)
loop_header("summation2",stack0,sp0)
...
JIT_flag1 = load(stack0,sp31)
sp32 = int_sub(sp31,Const(1))
guard_eq(JIT_flag1,US)
jmp("summation2",stack0,sp32)
```

(B) Resulting trace from Figure 5.3a.

FIGURE 5.4: Overview of resulting traces compiled with Stack Hybridization.

Chapter 6

Evaluation

In this section, we briefly introduce our implementation of a meta-hybrid JIT compiler framework and evaluate BacCaml by using two different types of microbenchmarks. Firstly, we compare the performance of each JIT compilation strategy. Next, based on the previous result, we assess the performance of the hybrid JIT compilation approach and discuss the effects of combining the two compilation strategies. In order to obtain data, we implemented a small functional programming language with BacCaml.

6.1 Implementation

As a proof-of-concept, we implemented BacCaml based on the MinCaml compiler. MinCaml is a ML compiler created by E.Sumii [37] for education-purpose. It generates native code almost as fast as other notable compilers such as GCC or OCamlOpt. The reason why we did not extend RPython itself is that the implementation of RPython is too huge to comprehend: initially, we created a subset of RPython on a compiler with reasonable implementation size. BacCaml itself is written in OCaml, and its implementation can be accessed at GitHub (`https://github.com/prg-titech/BacCaml`).

In order to evaluate the performance of BacCaml, we also created a small functional programming language, we call it MinCaml−− here, with BacCaml. This language consists of a meta-interpreter and a bytecode compiler for the interpreter. This is almost similar to MinCaml, but is limited to use unit integer variables. The meta-interpreter and the bytecode compiler are also available at GitHub (`https://github.com/prg-titech/MinCaml`).

6.2 Benchmarking

To assess the performance of the hybrid JIT compilation, we prepared two kinds of benchmarks. The first one is to compare the standalone performance of trace- and method-based compilations, and the second one is comparing hybrid, trace-, and method-based compilations [1].

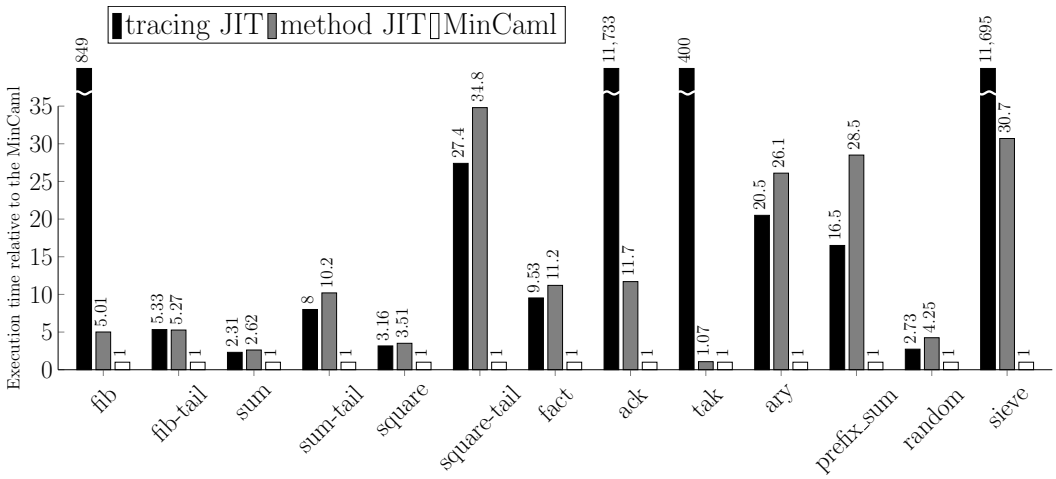


FIGURE 6.1: Standalone JIT compilation microbenchmark results in elapsed time relative to the MinCaml compiler (lower is better). Each benchmark program runs 100 times and the first run was ignored.

6.2.1 Standalone JIT Microbenchmarking Result

For comparing the standalone performance of trace- and method-based compilations, we first applied the both compilations separately for programs written in MinCaml−−, and compared the performance of MinCaml−− and MinCaml.

The standalone benchmark results are shown in Figure 6.1. The trace-based compilation performs 1.13x to 1.72x better than the method-based compilation in straight-forward programs (sum, sum-tail, square, fact, ary, prefix_sum and random). Sieve has a simple control flow; however it works worse than the method-based compilation. This is because our implementation of the trace-based compilation cannot retrace at the point where guard failure occurs several times, and therefore the tracer generates only one branch in sieve [2]. In contrast, when it is applied for programs with several recursive function calls (fib, tak, ack), the performance becomes worse; it is around 100x to 1000x slower than the method-based compilation because of the path-divergence problem.

Instead, the method-based compilation shows good performance on average compared to the trace-based compilation (1.07x to 34.8x slower than MinCaml). Above all, the method-based compilation performs from around 100x to 400x better than the trace-based compilation in programs with complex control-flow (in the case of fib, tack, ack).

The main reason why MinCaml−− is still around 10x slower than MinCaml mainly is that we use dynamic loading for realizing JIT compilation as BacCaml is a proof-of-concept. To improve the performance, we should adopt some just-in-time native code generation framework such as libgccjit [3] or GNU Lightning [4]. If

[1] We ran all the microbenchmarks on Arch Linux with Linux kernel version 5.4.6-arch-3-1 and dedicated hardware based on the MacBook Pro 13 inch model from Early 2015 (CPU: 2.7 GHz Intel Core i5 (I5-5257U) processor; Memory: 8 GB 1866MHz LPDDR3). We ran each program 100 times, and the first run was ignored to exclude the warm-up process.

[2] When we implement a retracing function, the performance in sieve will reach or surpass the method-based compilation.

[3] https://gcc.gnu.org/onlinedocs/jit/

[4] https://www.gnu.org/software/lightning/

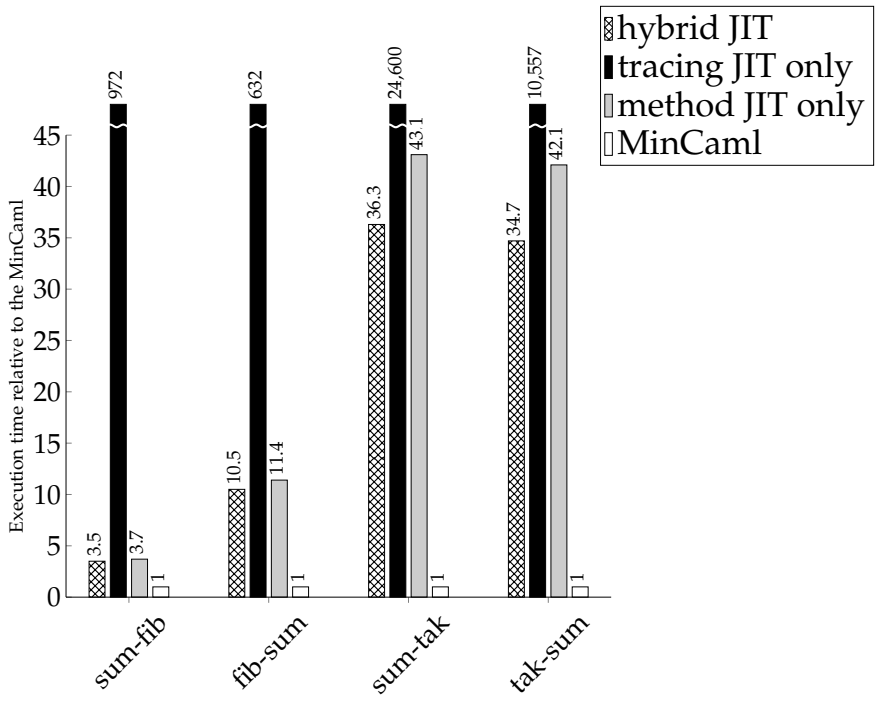


FIGURE 6.2: Hybrid JIT compilation microbenchmark results in elapsed time relative to the MinCaml compiler (lower is better). Each benchmark program runs 100 times and the first run was ignored.

we realize it, the performance of the compiled code from BacCaml will be almost similar to that from MinCaml.

From the result, we can conclude that trace-based compilation is suitable for straight-forwarded control flow (1.28x better than method JIT). Alternatively, method-based compilation is better for complex control flow, which causes the path-divergence problem (396x faster than tracing JIT).

6.2.2 Hybrid JIT Microbenchmarking Result

According to the result of standalone microbenchmarks, we composed functions suitable for trace- and method-based compilations. Then, we appled a hybrid compilation for them, and compare the performance with standalone strategies and MinCaml. The hybrid JIT microbenchmark program are shown and explained in Appendix C, and results are shown in Figure 6.2.

For all cases, the hybrid JIT strategy is 1.05x to 1.21x faster than the method JIT only. The hybrid JIT strategy applies trace-based compilation to sum, and method-based compilation to fib and tak. This avoids the overhead of function calling when executing the native code of sum which trace-based optimization is applied for. Furthermore, the hybrid JIT is about 100x faster than the tracing JIT only. This is because the tracing JIT has arose many times of side exits in executing the native code of fib, and the runtime performance is quite worse than hybrid JIT and other strategies.

From the results, the hybrid JIT is 1.1x faster than the method JIT only and over 300x faster than the tracing JIT only strategies. Hence, We can conclude that

the hybrid JIT strategy improves the runtime performance when we can apply appropriate JIT compilation strategies according to the structure of program parts.

Chapter 7

Related Work

In this chapter, we introduce some research projects related to our research and discuss and compare them with BacCaml.

7.1 Self-optimizing Interpreter

Self-optimizing abstract-syntax-tree (AST) interpreter [43, 42] enables language developers to implement effective virtual machines. The framework for building a language runtime is called Truffle, and the optimizer for a meta-interpreter written in Truffle is called Graal. The self-optimizing interpreter optimizes executing AST nodes of a meta-interpreter at execution time by profiling the runtime types and values. This prevents interpreters from executing unnecessary generic runtime operation and simplifies the control flow which leads to better performance of the compiled code.

For example, we consider the multiplying operation `a * b`. A self-optimizing interpreter represents the expression as `mul` node. It receives two child nodes (`a` and `b`) as arguments and will return the result of multiplying them. When the interpreter executes this expression, it first checks the type of arguments; `a` and `b` have some possibilities for having a type, e.g., integer or string at runtime. The self-optimizing interpreter will replace this `mul` node with a `mul-integer` when it proves that both child nodes are mainly integers during further execution. This makes it possible to reduce needless type checks and reduce the complexity of the generated native code. If two child nodes receive variables of other types, the self-optimizing interpreter changes `mul-integer` into a more generic node `mul`.

7.2 GraalSqueak

GraalSqueak [26] is a Squeak/Smalltalk VM implementation written in Truffle framework. In [25], Fabio Niephaus, et al. provided an efficient way to compile a bytecode-formatted program; that is, they showed a way to apply trace-based compilation with an AST-rewriting interpreter strategy.

We extend the meta-tracing JIT compilation framework to support method-based compilation, but their approach involves creating a meta-interpreter to enable trace-based compilation on a partial evaluation-based meta JIT compiler framework. Their idea is to implement a meta-interpreter with some specific hint annotations in order to expand the loop of an application program. Sulong has already demonstrated the same idea [34], and it was applied for implementing Squeak/Smalltalk VM.

7.3 HHVM JIT Compiler

HHVM [30] is a high-performance virtual machine for PHP and Hack programming languages. An important aspect of HHVM 2nd generation is its region-based JIT compiler. Region-based compilers [17] are not restricted to compile the entire body of methods, basic blocks, or straight-line traces; it can compile a combination of several program areas.

Their compilation strategy is more flexible than our hybrid compilation strategy, because HHVM region-based compiler can compile basic blocks, the entire body of methods, loops, and any combination of them. However, their approach is limited to a specific language system; our research aims to provide some flexibility of compilation as a meta JIT compiler framework.

7.4 IBM Trace-based Java JIT Compiler

IBM trace-based Java JIT compiler [21] was developed on the existing mature Java method-based JIT compiler. Their motivation was to investigate whether there are any opportunities to address the limitations of traditional JIT compilers, such as requiring careful management of inlining functions. As a result, this trace-based JIT compiler is as efficient as method-based JIT compiler, but they did not explore the coexistence of trace- and method-based compilations in one virtual machine.

7.5 Android Hybrid JIT

Pérez, et al. [33] combined trace- and method-based JIT compilers on the Android Dalvik VM. This hybrid JIT compiler achieved better performance than unmodified Dalvik VM by sharing profiling and compilation information. However, in comparison to our approach, their approach involved considerable engineering costs to realize the hybrid JIT compiler since they implemented the two kinds of compilers naively, which reduces many duplicated code bases and leads to less maintainability.

Chapter 8

Conclusion and Future Work

In this work, we presented a meta-hybrid JIT compilation approach to take advantage of trace- and method-based compilation strategies in a meta JIT compiler framework. For supporting the idea, we chose a meta-tracing JIT compiler and extended it to be able to perform method-based compilation using trace by customizing the following features: (1) trace entry/exit points, (2) conditional branches, (3) function calls, and (4) loops. Furthermore, we proposed Stack Hybridization: a meta-interpreter design to enable connecting native code generated from different strategies. The key concept of Stack Hybridization is (1) embedding two kinds of meta-interpreter implementation styles into a single definition, (2) selecting an appropriate style at just-in-time compilation time, and (3) putting a flag on the stack data structure to indicate whether it is under trace- or method-based compilation.

We implemented a meta-hybrid JIT compiler framework called BacCaml as a proof-of-concept. We created a small meta-tracing JIT compiler on the MinCaml compiler, supported method-based compilation by extending trace-based compilation, and realized stack hybridization mechanism on it.

Our benchmarks show that trace-based compilation performs 1.13x to 1.72x better than method-based compilation in the case of straightline control flow, but about 100x to 1000x worse in the case of complex control flow. On the other hand, method-based compilation performs well on average. By combining and applying them for appropriate program parts, the hybrid compilation approach shows about 1.1x better performance than individual compilation approaches.

We still need to specify some annotations to tell the compiler how to compile, but it requires additional engineering efforts from users of a programming language written in BacCaml. In the future, we are going to investigate a profiling scheme to detect which part is fitted for trace- or method-based compilation. Moreover, we want to apply our idea for RPython to measure the impact of hybrid JIT compilation on real-world applications. BacCaml is just a proof-of-concept meta JIT compiler framework; therefore, we have limitations to measure its performance. In order to return our ideas to the real world, implementing it on a practical meta JIT compiler is very important.

Bibliography

[1] Keith Adams et al. “The HipHop Virtual Machine”. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA’ 14. Portland, Oregon, USA, 2014, pp. 777–790. ISBN: 9781450325851. DOI: 10.1145/2660193.2660199.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. “Dynamo: a transparent dynamic optimization system”. In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. 2000. ISBN: 1-58113-199-2. DOI: 10.1145/349299.349303. arXiv: 1003.4074.

[3] Spenser Bauman et al. “Pycket: A Tracing JIT for a Functional Language”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. Vancouver, BC, Canada: ACM, 2015, pp. 22–34. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784740.

[4] Michael Bebenita et al. “SPUR: A Trace-based JIT Compiler for CIL”. In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’10. Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 708–725. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869517.

[5] Fabrice Bellard. “QEMU, a Fast and Portable Dynamic Translator”. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC ’05. Anaheim, CA: USENIX Association, 2005, p. 41.

[6] Carl Friedrich Bolz and Laurence Tratt. “The impact of meta-tracing on VM design and implementation”. In: *Science of Computer Programming* 98 (2015). Special Issue on Advances in Dynamic Languages, pp. 408–421. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2013.02.001.

[7] Carl Friedrich Bolz et al. “Allocation Removal by Partial Evaluation in a Tracing JIT”. In: *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM ’11. Austin, Texas, USA: ACM, 2011, pp. 43–52. ISBN: 978-1-4503-0485-6. DOI: 10.1145/1929501.1929508.

[8] Carl Friedrich Bolz et al. “Runtime Feedback in a Meta-tracing JIT for Efficient Dynamic Languages”. In: *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS ’11. Lancaster, United Kingdom: ACM, 2011, 9:1–9:8. ISBN: 978-1-4503-0894-6. DOI: 10.1145/2069172.2069181.

[9] Carl Friedrich Bolz et al. “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*. Genova, Italy: ACM, 2009, pp. 18–25. ISBN: 978-1-60558-541-3. DOI: 10.1145/1565824.1565827.

[10] L. Peter Deutsch and Allan M. Schiffman. "Efficient Implementation of the Smalltalk-80 System". In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '84. Salt Lake City, Utah, USA: Association for Computing Machinery, 1984, pp. 297–302. ISBN: 0897911253. DOI: 10.1145/800017.800542. URL: https://doi.org/10.1145/800017.800542.

[11] Tim Felgentreff et al. "How to Build a High-Performance VM for Squeak/Smalltalk in Your Spare Time: An Experience Report of Using the RPython Toolchain". In: *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies*. IWST'16. Prague, Czech Republic: Association for Computing Machinery, 2016. ISBN: 9781450345248. DOI: 10.1145/2991041.2991062.

[12] LLVM Foundation. *The LLVM Compiler Infrastucture*. 2019. URL: https://llvm.org/.

[13] Andreas Gal, Christian W. Probst, and Michael Franz. "HotpathVM: An Effective JIT Compiler for Resource-Constrained Devices". In: *Proceedings of the 2nd International Conference on Virtual Execution Environments*. VEE '06. Ottawa, Ontario, Canada: Association for Computing Machinery, 2006, pp. 144–153. ISBN: 1595933328. DOI: 10.1145/1134760.1134780.

[14] Andreas Gal et al. "Trace-Based Just-in-Time Type Specialization for Dynamic Languages". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: Association for Computing Machinery, 2009, pp. 465–478. ISBN: 9781605583921. DOI: 10.1145/1542476.1542528.

[15] *GCC, the GNU Compiler Collection*. 2019. URL: https://gcc.gnu.org/.

[16] Google. *Google's high-performance open source JavaScript and WebAssembly engine*. 2015. URL: https://v8.dev/.

[17] Richard E. Hank, Wen-Mei W. Hwu, and B. Ramakrishna Rau. "Region-Based Compilation: An Introduction and Motivation". In: *Proceedings of the 28th Annual International Symposium on Microarchitecture*. MICRO 28. Ann Arbor, Michigan, USA: IEEE Computer Society Press, 1995, pp. 158–168. ISBN: 0818673494.

[18] Michael Haupt et al. "The SOM Family: Virtual Machines for Teaching and Research". In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE '10. Bilkent, Ankara, Turkey: ACM, 2010, pp. 18–22. ISBN: 978-1-60558-820-9. DOI: 10.1145/1822090.1822098.

[19] Hiroshige Hayashizaki et al. "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*. ASPLOS XVI. Newport Beach, California, USA: ACM, 2011, pp. 405–418. ISBN: 978-1-4503-0266-1. DOI: 10.1145/1950365.1950412.

[20] Ruochen Huang, Hidehiko Masuhara, and Tomoyuki Aotani. "Improving Sequential Performance of Erlang Based on a Meta-tracing Just-In-Time Compiler". In: *International Symposium on Trends in Functional Programming*. Springer. 2016, pp. 44–58.

[21] Hiroshi Inoue et al. "A trace-based Java JIT compiler retrofitted from a method-based compiler". In: *Proceedings - International Symposium on Code Generation and Optimization* (2011), pp. 246–256. ISSN: 03621340. DOI: 10.1109/CGO.2011.5764692.

[22] Yusuke Izawa, Hidehiko Masuhara, and Tomoyuki Aotani. "Extending a Meta-tracing Compiler to Mix Method and Tracing Compilation". In: *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*. Programming '19. Genova, Italy: ACM, 2019, 5:1–5:3. ISBN: 978-1-4503-6257-3. DOI: 10.1145/3328433.3328439.

[23] Stefan Marr and Stéphane Ducasse. "Tracing vs. Partial Evaluation: Comparing Meta-compilation Approaches for Self-optimizing Interpreters". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. Pittsburgh, PA, USA: ACM, 2015, pp. 821–839. ISBN: 978-1-4503-3689-5. DOI: 10.1145/2814270.2814275.

[24] Mozilla. *IonMonkey, the next generation JavaScript JIT for SpiderMonkey*. URL: https://wiki.mozilla.org/IonMonkey.

[25] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. "GraalSqueak: A Fast Smalltalk Bytecode Interpreter Written in an AST Interpreter Framework". In: *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS '18. Amsterdam, Netherlands: ACM, 2018, pp. 30–35. ISBN: 978-1-4503-5804-0. DOI: 10.1145/3242947.3242948.

[26] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. "GraalSqueak: Toward a Smalltalk-based Tooling Platform for Polyglot Programming". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. MPLR 2019. Athens, Greece: ACM, 2019, pp. 14–26. ISBN: 978-1-4503-6977-0. DOI: 10.1145/3357390.3361024.

[27] Ocacle Lab. *A ECMAScript 2019 compliant Javascript implementation built on GraalVM. With polyglot language interoperability support*. URL: https://github.com/graalvm/graaljs.

[28] Oracle Lab. *A high performance implementation of the Ruby programming language*. URL: https://github.com/oracle/truffleruby.

[29] Oracle Lab. *A high-performance implementation of the R programming language, built on GraalVM*. URL: https://github.com/oracle/fastr.

[30] Guilherme Ottoni. "HHVM JIT: A Profile-Guided, Region-Based Compiler for PHP and Hack". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: Association for Computing Machinery, 2018, pp. 151–165. ISBN: 9781450356985. DOI: 10.1145/3192366.3192374.

[31] Michael Paleczny, Christopher Vick, and Cliff Click. "The java hotspot TM server compiler". In: *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium*. Vol. 1. S 1. 2001.

[32] Mike Pall. *A Just-in-time Compiler for Lua Programming Language*. 2005. URL: http://luajit.org/index.html.

[33] Guillermo A. Perez et al. "A hybrid just-in-time compiler for android". In: *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems - CASES '12* (2012), p. 41. DOI: 10.1145/2380403.2380418.

[34] Manuel Rigger et al. "Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle". In: *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*. VMIL 2016. Amsterdam, Netherlands: ACM, 2016, pp. 6–15. ISBN: 978-1-4503-4645-0. DOI: 10.1145/2998415.2998416.

[35] Armin Rigo and Samuele Pedroni. "PyPy's Approach to Virtual Machine Construction". In: *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 944–953. ISBN: 159593491X. DOI: 10.1145/1176617.1176753.

[36] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. "A Region-Based Compilation Technique for Dynamic Compilers". In: vol. 28. 1. New York, NY, USA: Association for Computing Machinery, Jan. 2006, pp. 134–174. DOI: 10.1145/1111596.1111600.

[37] Eijiro Sumii. "MinCaml: A Simple and Efficient Compiler for a Minimal Functional Language". In: *FDPE: Workshop on Functional and Declaritive Programming in Education* (2005), pp. 27–38. DOI: 10.1145/1085114.1085122. URL: http://portal.acm.org/citation.cfm?doid=1085114.1085122.

[38] PyPy JS Team. *PyPy compiled into JavaScript*. URL: https://github.com/pypyjs/pypyjs.

[39] Topaz Project. *A high performance ruby, written in RPython*. URL: http://docs.topazruby.com/en/latest/.

[40] David Ungar and Randall B. Smith. "Self: The Power of Simplicity". In: vol. 22. 12. New York, NY, USA: Association for Computing Machinery, Dec. 1987, pp. 227–242. DOI: 10.1145/38807.38828.

[41] Thomas Würthinger et al. "One VM to Rule Them All". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 187–204. ISBN: 9781450324724. DOI: 10.1145/2509578.2509581.

[42] Thomas Würthinger et al. "Practical Partial Evaluation for High-performance Dynamic Language Runtimes". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 662–676. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062381.

[43] Thomas Würthinger et al. "Self-Optimizing AST Interpreters". In: *Proceedings of the 8th Symposium on Dynamic Languages*. DLS '12. Tucson, Arizona, USA: Association for Computing Machinery, 2012, pp. 73–82. ISBN: 9781450315647. DOI: 10.1145/2384577.2384587.

Appendix A

Trade-offs between the Two Stack Styles

As shown in Section 5.3, there are two possible choices to implement a meta-interpreter with a meta-tracing JIT compiler framework. The reason why such difference occurs is that, whether being able to optimize the base language program efficiently or not depends on how language developers manage the states of the base language program in a meta-interpreter. In this section, we organize the advantages and disadvantages between both styles: user- and host-stack meta-interpreter definition styles.

A.0.1 Host-stack Style

If a developer adopts only host-stack definition style, its implementation is easier than that of user-stack style, because it does not require managing a return address and value in a meta-interpreter. Moreover, the biggest advantage of host-stack style is that our proposed method JIT compilation, which is retrofitted from a meta-tracing JIT compilation, is based on host-stack style: The compiler can find which part is a base-program's method invocation in a meta-interpreter definition.

The drawback of employing only host-stack style for a meta-tracing compilation is that a meta-tracing compiler cannot efficiently optimize nontail-recursive function calls, that is, the compiler cannot convert them into loops. For example, when PyPy executes the `sum` function shown in the left of Figure A.1, its meta-tracing JIT compiler generates the trace shown in the right of Figure A.1. From the resulting trace, the compiler cannot recognize a nontail-recursive call as a loop in the viewpoint of a meta-interpreter's program counter. Basically, a meta-tracing JIT compiler closes a trace when its tracer comes to the start of tracing, by using hint functions (e.g., `can_enter_jit` and `jit_merge_point`). In this case, the states of the meta-interpreter between the beginning of trace recording and at the point where nontail-recursive function call occurs differ. Therefore, the compiler cannot optimize nontail recursive function calls and tries to continue tracing.

A.0.2 User-stack Style

On the other hand, if a developer employs only use-stack definition style, on the other hand, a meta-tracing JIT compiler can convert a nontail-recursive function call of the base program into a jump instruction (i.e., method inlining). Therefore, this style works better and enhances the performance of a meta-tracing JIT compilation in functional programming languages. The disadvantage is that the

path-divergence problem [20] occurs when it is applied to a program with a complex control flow as illustrated in Appendix **??**. To prevent this problem, a tracing JIT compiler has to apply method-based compilation as shown in our previous work [22]. However, if you decide to use only user-stack style, the tracer has to continue taking a trace to know the state in which a method invocation is performed completely. In the meantime, the tracer does not record instructions and merely follows the execution of a meta-interpreter. When the tracer enters a program with the path-divergence problem, it spends a considerable amount of time in the program, which results in a big performance penalty. For such a reason, sticking to using only user-stack style is not a good choice.

```
def sum(n):
  if n <= 1:
    return 1
  else:
    return n+sum(n-1)

sum(10000)
```

```
# resulting trace
debug_merge_point(0,0,#0 LOAD_FAST')
debug_merge_point(0,0,#3 LOAD_CONST')
debug_merge_point(0,0,#6 COMPARE_OP')
debug_merge_point(0,0,#9 POP_JUMP_IF_FALSE')
debug_merge_point(0,0,#16 LOAD_FAST')
debug_merge_point(0,0,#19 LOAD_GLOBAL')
debug_merge_point(0,0,#22 LOAD_FAST')
debug_merge_point(0,0,#25 LOAD_CONST')
debug_merge_point(0,0,#28 BINARY_SUBTRACT')
debug_merge_point(0,0,#29 CALL_FUNCTION')
debug_merge_point(1,1,#0 LOAD_FAST')
debug_merge_point(1,1,#3 LOAD_CONST')
...
...
debug_merge_point(6,6,#28 BINARY_SUBTRACT')
debug_merge_point(6,6,#29 CALL_FUNCTION')
debug_merge_point(7,7,#0 LOAD_FAST')
debug_merge_point(7,7,#3 LOAD_CONST')
debug_merge_point(7,7,#6 COMPARE_OP')
debug_merge_point(7,7,#9 POP_JUMP_IF_FALSE')
debug_merge_point(7,7,#16 LOAD_FAST')
debug_merge_point(7,7,#19 LOAD_GLOBAL')
debug_merge_point(7,7,#22 LOAD_FAST')
debug_merge_point(7,7,#25 LOAD_CONST')
debug_merge_point(7,7,#28 BINARY_SUBTRACT')
debug_merge_point(7,7,#29 CALL_FUNCTION')
debug_merge_point(7,7,#32 BINARY_ADD')
debug_merge_point(7,7,#33 RETURN_VALUE')
debug_merge_point(6,6,#32 BINARY_ADD')
debug_merge_point(6,6,#33 RETURN_VALUE')
debug_merge_point(5,5,#32 BINARY_ADD')
debug_merge_point(5,5,#33 RETURN_VALUE')
debug_merge_point(4,4,#32 BINARY_ADD')
debug_merge_point(4,4,#33 RETURN_VALUE')
debug_merge_point(3,3,#32 BINARY_ADD')
debug_merge_point(3,3,#33 RETURN_VALUE')
debug_merge_point(2,2,#32 BINARY_ADD')
debug_merge_point(2,2,#33 RETURN_VALUE')
debug_merge_point(1,1,#32 BINARY_ADD')
debug_merge_point(1,1,#33 RETURN_VALUE')
debug_merge_point(0,0,#32 BINARY_ADD')
debug_merge_point(0,0,#33 RETURN_VALUE')
```

FIGURE A.1: The left is a function calculating a summation of a given number, implemented with tail-recursive sytle. The right is the resulting trace of sum function by RPython and PyPy.

Appendix B

The Path-Divergence Problem

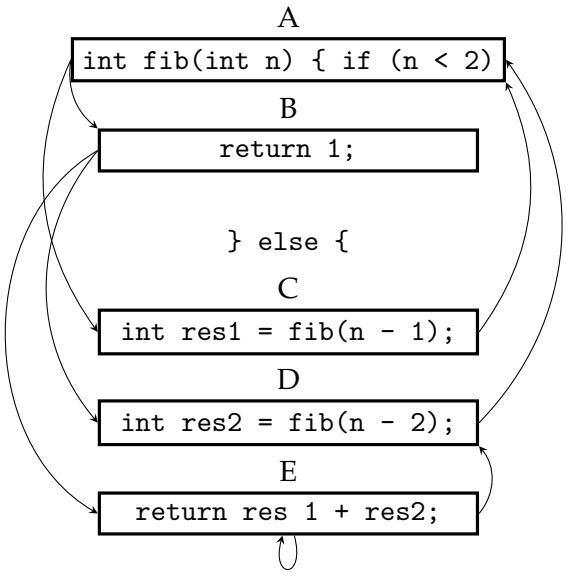


FIGURE B.1: The control flow of a Fibonacci function.

The tracing JIT compilers are not very efficient in compiling certain types of programs [20]; we call this *the path-divergence problem*. The problem is observed as frequent guard failures (and compilation of the subsequent traces) when it runs such kinds of programs. Since it spends most of the 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., the Fibonacci, are the examples.

Let us consider the problem with a Fibonacci function, whose control flow graph is shown in Figure B.1. Each node in the graph is a basic block. Since tracing JIT compilers[1] basically inline function calls, the node's end with a function call is connected to the entry of the function. Also, the node's 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 during the

[1] As explained in Section 2.2.1, meta-tracing JIT compilers effectively compile traces of the base program by means of tracing the meta-interpreter. Therefore we here discuss the problem in a context when a tracing compiler handles a base program.

entire execution. However, the execution of the Fibonacci rarely contains such a subsequence. This is because the branching nodes in the graph, namely A, B, and E in the graph, take one of the 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 cause guard failure in the middle of its execution.

Appendix C

Programs used for Evaluating Hybrid JIT Compilation

```
(* applying method-based
 * compilation for it *)
let%mj rec fib n =
  if n < 2 then 1 else
    fib (n-1) + fib (n-2) in

(* applying trace-based
 * compilation for it *)
let%tj rec sum i n =
  if i <= 1 then n else
    sum (i-1) (n + (fib i)) in

(* main function *)
print_int (sum 30 0)
```

(A) sum-fib

```
(* applying trace-based
 * compilation for it *)
let%tj rec sum n acc =
  if n <= 1 then acc else
    sum (n-1) (n+acc) in

(* applying method-based
 * compilation for it *)
let%mj rec fib n =
  if n < 2 then sum 1000 0 else
    fib (n-1) + fib (n-2) in

(* main function *)
print_int (fib 20)
```

(B) fib-sum

```
let%mj rec tak x y z =
  if x <= y then y else
  tak (tak (x-1) y z) (tak (y-1) z
  ↪  x) (tak (z-1) x y)
in
let%tj rec sum i n =
  if i <= 1 then n else
    let x = (tak 13 8 6) in
    sum (i - 1) (n + x) in
print_int (sum 20 1)
```

(C) sum-tak

```
let%tj rec sum i n =
  if i <= 0 then n else
    sum (i - 1) (n + i) in
let%mj rec tak x y z =
  if x <= y then sum 10 0 else
    tak (tak (x-1) y z)
      (tak (y-1) z x)
      (tak (z-1) x y)
in
print_int (tak 14 8 3)
```

(D) tak-sum

FIGURE C.1: Microbenchmark programs for evaluating the performance of hybird JIT compilation. sum is implemented in tail-recursive style, and fibs is in nontail-recursive style. Therefore sum and fib are fitted for trace-based and method-based compilation respectively.

Programs used for evaluating hybrid JIT compilation are shown in Figure C.1. `sum` is trace-based compilation-suited, but `fib` and `tak` are method-based compilation-suited programs. We applied trace-based compilation for `sum`, and method-based

compilation for `fib` and `tak` in taking a hybrid compilation strategy. For comparing the performance, we also applied trace- and method-based compilations individually and majored their execution time.

Appendix D

Interpreter Definition Example written in BacCaml

```
let rec frame_reset stack old_base new_base ret n i =
  if n = i then (stack.(old_base + n + 1) <- ret; old_base + n + 2)
  else (stack.(old_base + i) <- stack.(new_base + i);
        frame_reset stack old_base new_base ret n (i + 1)) in

(* declaring a casting function: int array -> int *)
let rec cast_fAII x = x in
(* declaring a casting function: int -> int array *)
let rec cast_fIAI x = x in

let rec frame_reset stack old_base new_base ret n i =
  if n = i then (stack.(old_base + n + 1) <- ret; old_base + n + 2)
  else (stack.(old_base + i) <- stack.(new_base + i);
        frame_reset stack old_base new_base ret n (i + 1)) in

let rec interp stack sp bytecode pc =
  jit_merge_point pc stack sp;
  let instr = bytecode.(pc) in
  if instr = 0 then             (* UNIT *)
    interp stack sp bytecode (pc + 1)
  else if instr = 1 then        (* ADD *)
    let v2 = stack.(sp - 1) in  (* sp: sp - 1 *)
    let v1 = stack.(sp - 2) in  (* sp: sp - 2 *)
    stack.(sp-2) <- (v1+v2);    (* sp: sp - 1 *)
    interp stack (sp - 1) bytecode (pc + 1)
  else if instr = 2 then        (* SUB *)
    let v2 = stack.(sp - 1) in
    let v1 = stack.(sp - 2) in
    stack.(sp - 2) <- (v1 - v2);
    interp stack (sp - 1) bytecode (pc + 1)
  else if instr = 3 then        (* MUL *)
    let v2 = stack.(sp - 1) in
    let v1 = stack.(sp - 2) in
    stack.(sp - 2) <- (v1 * v2);
    interp stack (sp - 1) bytecode (pc + 1)
  else if instr = 29 then       (* DIV *)
    let v2 = stack.(sp - 1) in
    let v1 = stack.(sp - 2) in
    stack.(sp - 2) <- (divide v1 v2);
    interp stack (sp - 1) bytecode (pc + 1)
  else if instr = 30 then       (* DIV *)
    let v2 = stack.(sp - 1) in
    let v1 = stack.(sp - 2) in
    stack.(sp - 2) <- (modulo v1 v2);
    interp stack (sp - 1) bytecode (pc + 1)
  else if instr = 4 then        (* NOT *)
    let v = stack.(sp - 1) in
    let n = (if v = 0 then 1 else 0)in
    stack.(sp - 1) <- n;
    interp stack sp bytecode (pc + 1)
  else if instr = 5 then        (* NEG *)
```

```
    let v = stack.(sp - 1) in
    stack.(sp - 1) <- (-v);
    interp stack sp bytecode (pc+1)
  else if instr = 6 then        (* LT *)
    let v2 = stack.(sp - 1) in
    let v1 = stack.(sp - 2) in
    let n = (@if v1 < v2 then 1 else 0) in
    stack.(sp - 2) <- n;
    interp stack (sp - 1) bytecode (pc + 1)
  else if instr = 7 then        (* EQ *)
    let v1 = stack.(sp - 1) in
    let v2 = stack.(sp - 2) in
    let v = (if v1 = v2 then 1 else 0) in
    stack.(sp - 2) <- v;
    interp stack (sp - 1) bytecode (pc + 1)
  else if instr = 15 then       (* CONST *)
    let c = bytecode.(pc + 1) in
    stack.(sp) <- c;
    interp stack (sp + 1) bytecode (pc + 2)
  else if instr = 8 then        (* JUMP_IF_ZERO *)
    let addr = bytecode.(pc + 1) in
    let v = stack.(sp - 1) in
    let sp2 = sp - 1 in
    @if v = 0 then (
      interp stack sp2 bytecode addr
    ) else
      interp stack sp2 bytecode (pc + 2)
  else if instr = 9 then        (* JUMP *)
    let addr = bytecode.(pc + 1) in
    if addr < pc then (
      can_enter_jit stack sp bytecode addr;
      interp stack sp bytecode addr
    ) else
      interp stack sp bytecode addr
  else if instr = 10 then        (* CALL *)
    let addr = bytecode.(pc + 1) in
    let rands = bytecode.(pc + 2) in
    if is_mj () then
      (stack.(sp) <- 100;       (* push jit flag *)
       let sp2 = sp+2 in
       let r = mj_call stack sp2 bytecode addr in
       stack.(sp - rands) <- r;
       interp stack (sp-rands+1) bytecode (pc+3))
    else
      (stack.(sp) <- pc + 3;
       stack.(sp + 1) <- 200;   (* push jit flag *)
       let sp2 = sp+2 in
       @if addr < pc then (
         can_enter_jit stack sp2 bytecode addr;
         interp stack sp2 bytecode addr
       ) else
         interp stack sp2 bytecode addr)
  else if instr = 11 then       (* RET *)
    let v = stack.(sp - 1) in
    let mode = stack.(sp-2) in  (* sp: sp-3 *)
    let addr = stack.(sp-3) in  (* sp: sp-3 *)
    if mode = 200 then          (* check jit flag *)
      (let n = bytecode.(pc + 1) in
       stack.(sp - n - 3) <- v; (* sp: sp-3-n+1 = sp-2-n *)
       let sp2 = sp - n - 2 in
       @if addr < pc then (
         can_enter_jit stack sp2 bytecode addr;
         interp stack sp2 bytecode addr
       ) else
         interp stack sp2 bytecode addr)
    else v
  else if instr = 12 then       (* HALT *)
    stack.(sp - 1)
  else if instr = 13 then       (* DUP *)
    let n = bytecode.(pc + 1) in
```

```
    let v = stack.(sp - n - 1) in
    stack.(sp) <- v;
    interp stack (sp + 1) bytecode (pc + 2)
  else if instr = 14 then       (* POP1 *)
    let v = stack.(sp - 1) in
    let _ = stack.(sp - 2) in
    stack.(sp - 2) <- v;
    interp stack (sp - 1) bytecode (pc + 1)
  else if instr = 16 then       (* GET *)
    let n = stack.(sp - 1) in
    let arr = cast_fIAI(stack.(sp - 2)) in
    stack.(sp - 2) <- arr.(n);
    interp stack (sp - 1) bytecode (pc + 1)
  else if instr = 17 then       (* PUT *)
    let i = stack.(sp - 1) in
    let arr = cast_fIAI(stack.(sp - 2)) in
    let n = stack.(sp - 3) in
    arr.(i) <- n;
    stack.(sp - 3) <- cast_fAII(arr);
    interp stack (sp - 2) bytecode (pc + 1)
  else if instr = 18 then       (* ARRAYMAKE *)
    let init = stack.(sp - 1) in
    let size = stack.(sp - 2) in
    let a = Array.make size init in
    stack.(sp - 2) <- cast_fAII(a);
    interp stack (sp - 1) bytecode (pc + 1)
  else if instr = 19 then       (* FRAME_RESET *)
    let o = bytecode.(pc + 1) in
    let l = bytecode.(pc + 2) in
    let n = bytecode.(pc + 3) in
    let ret = stack.(sp-n-l-1) in
    let old_base = sp - n - l - o - 1 in
    let new_base = sp - n in
    let sp2 = frame_reset stack old_base new_base ret n 0 in
    interp stack sp2 bytecode (pc + 4)
  else if instr = 20 then       (* PRINT_INT *)
    let v = stack.(sp - 1) in
    prerr_int v;
    interp stack (sp - 1) bytecode (pc + 1)
  else if instr = 21 then       (* PRINT_NEWLINE *)
    (print_newline ();
     interp stack sp bytecode (pc + 1))
  else if instr = 22 then       (* METHOD_ENTRY *)
    interp stack sp bytecode (pc + 1)
  else if instr = 23 then       (* CONSTO *)
    (stack.(sp) <- 0;
     interp stack (sp + 1) bytecode (pc + 1))
  else if instr = 24 then       (* DUPO *)
    let v = stack.(sp - 1) in
    stack.(sp) <- v;
    interp stack (sp + 1) bytecode (pc + 1)
  else if instr = 25 then       (* METHOD_COMP *)
    interp stack sp bytecode (pc+1)
  else if instr = 26 then       (* TRACING_COMP *)
    interp stack sp bytecode (pc+1)
  else if instr = 27 then       (* NOP *)
    interp stack sp bytecode (pc+1)
  else
    -1000 in
let stk  = Array.make 2000000 0 in
stk.(0) <- (-987); stk.(1) <- (-456);
let rec read_code i n arr =
  if i = n then arr
  else
    (arr.(i) <- read_int ();
     read_code (i+1) n arr) in
let n = read_int () in
let arr = Array.make n 0 in
let code = read_code 0 n arr in
```

```
interp stk 1 code 0
```