JOT
JOURNAL OF
OBJECT TECHNOLOGY

Threaded Code Generation with a Meta-Tracing JIT Compiler

Yusuke Izawa[*], **Hidehiko Masuhara**[*], **Carl Friedrich Bolz-Tereick**[†], **and Youyou Cong**[*]
[*]Tokyo Institute of Technology, Japan
[†]Heinrich-Heine-Universität Düsseldorf, Germany

ABSTRACT Language implementation frameworks, e.g., RPython and Truffle/Graal, are practical tools for creating efficient virtual machines, including a well-functioning just-in-time (JIT) compiler. It is demanding to support multitier JIT compilation in such a framework for language developers. This paper presents an idea to generate threaded code by reusing an existing meta-tracing JIT compiler, as well as an interpreter design for it. Our approach does not largely modify RPython itself but constructs an effective interpreter definition to enable threaded code generation in RPython. We expect our system to be extended to support multilevel JIT compilation in the RPython framework. We measured the potential performance of our threaded code generation by simulating its behavior in PyPy. We confirmed that our approach reduced code sizes by 80 % and compilation times by 60 % compared to PyPy's JIT compiler on average, and ran about 7 % faster than the interpreter-only execution.

KEYWORDS JIT compiler, meta-tracing JIT compiler, RPython, threaded code

1. Introduction

Language implementation frameworks, such as RPython (Bolz et al. 2009) and Truffle/Graal (Würthinger et al. 2012), help language developers in building a full-fledged virtual machine with a smaller amount of implementation effort. Those language implementation frameworks have a mechanism that takes an interpreter definition of a language and yields a virtual machine (VM) with advanced features, including a quality just-in-time (JIT) compiler. The effectiveness and usefulness of language implementation frameworks are demonstrated by efficient implementations of many programming language implementations including PyPy (Bolz et al. 2009), GraalPython (Oracle Labs. 2018), Topaz (Gaynor et al. 2013), TruffleRuby (Oracle Lab. 2013), RSqueak (Felgentreff et al. 2016), and TruffleSqueak (Niephaus et al. 2019). Not only those implementations are realized by writing interpreters, many of them exhibit better performance than their interpreter-based counterparts (e.g., CPython and CRuby).

It is challenging for language implementation frameworks to support multitier JIT compilation. Multitier JIT compilation is a technique that compiles different parts of programs at different optimization levels to balance between compilation overheads and the efficiency of compiled code[1]. Naïvely, multitier compilation requires a compiler for each optimization level. It is also possible to construct one compiler that can yield code at different optimization levels, but implementing such a compiler would require more effort of language developers. Since language implementation frameworks need to *generate* such a compiler from interpreter definitions, it is more challenging to support multitier compilation with the same interpreter definition.

In this paper, we propose a technique that generates threaded code by reusing an existing meta-tracing JIT compiler, namely RPython. Threaded code generation is a compilation technique that simply converts each operation of a source program into

JOT reference format:
Yusuke Izawa, Hidehiko Masuhara, Carl Friedrich Bolz-Tereick, and Youyou Cong. *Threaded Code Generation with a Meta-Tracing JIT Compiler.* Journal of Object Technology. Vol. 21, No. 2, 2022.
http://dx.doi.org/10.5381/jot.2022.21.2.a1

[1] For example, the Jalapeño Java VM has a baseline and an optimizing compiler with three optimization levels (Alpern et al. 1999). In addition, The four tire JIT in the JavaScript engine in Webkit (**?**) has four different optimization levels.

a call to a respective handler function. Some multitier VMs use threaded code generation as the baseline compiler since its compilation speed is extremely fast. Our idea is to use existing RPython's engine (i.e., the meta-tracing compilation machinery) for threaded code generation so that it will serve as the baseline JIT compiler for RPython-based language implementations. We expect that our threaded code generation should be placed between an interpreter execution and a tracing JIT execution since it is just baseline compilation. Given that context, the threaded code compilation should reduce a compilation code size to compile it fast. Although this paper focuses on threaded code generation, we hope that our approach would be able to be extended to compilation at different optimization levels in the future.

The proposal of the paper is positioned as an application of our *meta-hybrid JIT compiler framework* project (Izawa & Masuhara 2020) to a production-level language implementation framework, namely the RPython-backend for PyPy (Rigo & Pedroni 2006) in the context of threaded code generation. In contrast to our original proposal (Izawa & Masuhara 2020) that is based on a simple experimental language implementation framework, this paper realizes threaded code generation on a production level language implementation framework by mostly reusing the existing implementation that does not consider threaded code generation or other levels of optimization at all. In other words, this approach doesn't need to modify the RPython's compilation engine too much, but we realize it just by preparing a specific interpreter definition and implement a new trace compilation engine that share almost all the code base (details are explained in Section 3).

In this paper, we make the following contributions:

- an idea to build a method-based threaded code generator on top of RPython,
- an implementation design to realize a method-based threaded code generation with a meta-tracing JIT compiler, and
- measuring the potential performance of the generated threaded code through preliminary experiments in PyPy.

The rest of this paper is organized as follows. Section 2 shows the background. Section 3 explains the idea of realizing a method-based baseline JIT compiler on top of RPython. In Section 4, through preliminary benchmark experiments, we discuss how well our threaded code generation performs in practice, and what kind of programs it should be applied to. Section 5 presents the related work and Section 6 concludes this paper.

2. Background

This section briefly gives an overview of a language implementation framework, and the meta-tracing JIT compiler in PyPy/RPython as well as threaded code.

2.1. Language Implementation Framework

A language implementation framework is a tool that generates a high-performance VM from an interpreter definition. In a traditional development way, programming language developers have to implement VM components, such as an interpreter, JIT compiler, memory management model, etc., from scratch for each language. However, by using a language implementation framework developers need to write only an interpreter by using a language implementation framework when they build a language.

```
1  jitdriver = JitDriver(reds=['self'],
2                        greens=['pc', 'bytecode'])
3
4  def interp(self):
5      pc = 0
6      while True:
7          jitdriver.jit_merge_point(
8              self=self,pc=pc,
9              bytecode=bytecode)
10         opcode = bytecode[pc]
11         pc += 1
12         if opcode == ADD:
13             ...
14         elif opcode == JUMP:
15             t = ord(bytecode[pc])
16             pc += 1
17             if t < pc:
18                 jitdriver.can_enter_jit(
19                     self=self,pc=t,
20                     bytecode=bytecode)
21             pc = t
22         ...
```

Listing 1 A simple example of a bytecode interpreter written in RPython

There are two state-of-the-art frameworks called RPython and Truffle/Graal. RPython (Rigo & Pedroni 2006) is a part of the PyPy project; PyPy is generated from the RPython framework. On the other hand, Truffle/Graal (Würthinger et al. 2012) is a part of GraalVM project that is being developed by Oracle Lab. They are successful in generating high-performance language implementations for Python (**?**Oracle Labs. 2018), PHP (Fijałkowski et al. 2014), Ruby (Gaynor et al. 2013; Oracle Lab. 2013), R (Oracle Lab. 2015), and so on.

RPython and Truffle/Graal require different interpreter definition styles; a bytecode interpreter and an abstract-syntax-tree (AST) interpreter, respectively. In addition, to enable JIT compilation and other optimizations, framework users have to follow the implementation manners that are provided by the frameworks. For example, at least, RPython users have to write hint functions at a right place (details are shown in Section 2.2), and Truffle/Graal users should define AST nodes by inheriting `Node` class which provides Truffle/Graal and override `execute` method inside their defined AST nodes.

2.2. PyPy/RPython and Meta-tracing JIT Compiler

PyPy is an implementation of Python language, based on the RPython (Rigo & Pedroni 2006) compiler. It has a high-performance tracing JIT compiler, which is not directly implemented but generated by the RPython compiler. The RPython compiler accepts a bytecode interpreter written in RPython. A meta-tracing JIT compiler (Bolz et al. 2009) keeps track of the execution of a user-defined interpreter and compiles a hot loop of a target language.

Listing 1 is an example definition that a language developer needs to write in RPython. Note that a language developer needs

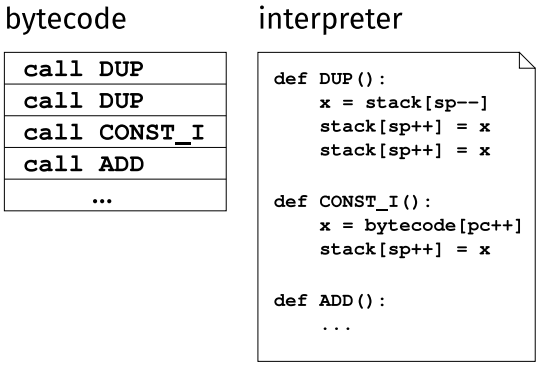

Figure 1 An overview of how threaded code works.

to define `jitdriver` for telling the necessary information to RPython's meta-tracing JIT compiler. There are also special functions, namely `jit_merge_point` and `can_enter_jit`. `jit_merge_point` and `can_enter_jit` should be placed at the beginnings of a bytecode dispatch loop and where back-edge instruction occurs (e.g., the end of `JUMP` definition in Listing 1), respectively.

2.3. Threaded Code

Threaded code (Bell 1973; Hong 1992) is a technique to improve the performance of a bytecode interpreter. The interpreter separately defines *handler functions* for all bytecode instructions as shown on the right-hand side in Figure 1. A program is a sequence of call instructions to handlers as shown on the left-hand side. Executing a threaded code-based program reduces the number of indirect branching that significantly pose a performance penalty at runtime because of branch mispredictions (Ertl & Gregg 2003).

3. The Compilation Tactic of RPython's Baseline JIT

In this section, we present how to realize method-based baseline JIT strategy on top of RPython without implementing a compilation engine from scratch.

The objective of introducing threaded code is for less start-up and compilation time in the RPython[2]. In general, a tracing JIT compiler automatically inlines function calls and applies several optimizations to a trace. The longer the trace you get and the better native code you want to generate, the longer the compilation time. In contrast, threaded code generation only leaves the call instruction to a subroutine, so tracing doesn't consume much time. We also apply only simple optimizations such as constant-folding and removing duplicated operations to the obtained trace from the threaded code generator, so we can reduce the compilation time than a normal tracing mode.

[2] An alternative approach to improve warm-up performance is to improve the dispatching mechanism of an interpreter, for example, by using threaded jumps. It would not be easy to realize such approaches in RPython as it currently assumes more straightforwardly written interpreters.

3.1. The Compilation Principle

Our threaded code generation is achieved by carefully controlling the RPython's meta-tracing compiler and reconstructing a control flow from the resulted trace. We realize it just by preparing a model of a specific interpreter definition (called *method-traversal interpreter*) and a new trace compilation mechanism (called *trace-stitching*). Below, we explain our approach by comparing our approach against a typical JIT compilation process in RPython.

When RPython compiles a base-program executed by an interpreter, it

starts compiling at the beginning of a loop, which is dynamically detected;

for each operation in the base program, follows into the respective handler body in the interpreter, which effectively eliminates "interpretation" (i.e., code dispatching and operand manipulation) by the interpreter;

at a function call in the base program, follows into the body of the callee function, which effectively achieves function inlining;

at a conditional expression in the base program, follows only one of the branch with emitting a *guard* operation for other branches; and

at a conditional branch in the handler (including selection of an arithmetic operation based on operands' runtime types), traces only one of the branch to achieve effective type-specialization;

finishes compiling at the end of the loop.

Our threaded code generation operates the RPython compiler so that it

starts compiling at the beginning of a method/function in the base-program[3];

for each operation in the base-program, follows the code dispatching part of the interpreter, but does not trace into the handler body but emits a call instruction to the respective handler;

at a function call in the base-program, emits a call instruction and continues tracing of the operations after the functional call;

at a conditional expression in the base-program, follows *all* branches;

at a conditional branch in the handler — this will not happen since the compiler does not trace the inside of handlers; and

finishes compiling at the end of the method/function.

[3] Whether the system uses threaded code generation or not is an open issue that we will consider in the future. For the time being, we merely assume that the threaded code generator is invoked for a particular base-program method/function.

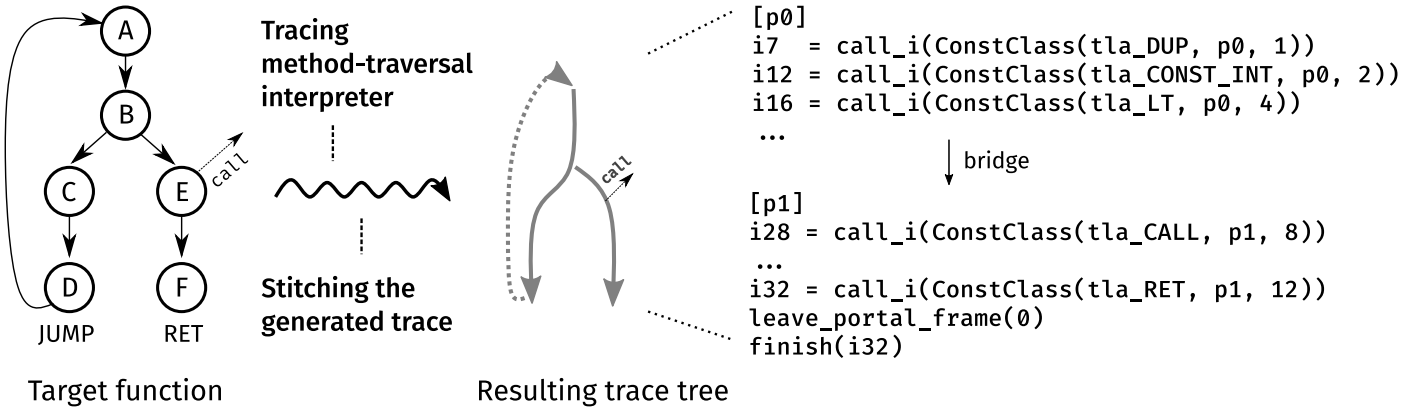

Figure 2 A sketch of how RPython method-based baseline JIT compiler works. From the target function in the left-hand side, it generates the trace tree shown in the right-hand side.

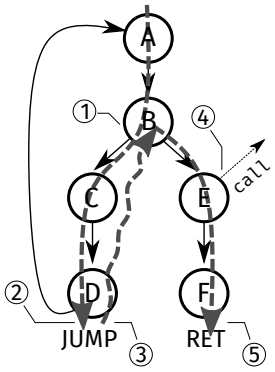

Figure 3 Tracing the entire of a function with method-traversal interpreter.

To drive the RPython compiler like that, our proposal consists of the following three techniques:

The *method-traversal interpreter* technique. We write an interpreter to let the tracing mechanism of RPython traverse all execution paths in a base-program method/function. We achieve this behavior by merely defining the interpreter in a specific way, but not modifying the existing RPython infrastructure.

The hinting technique. We let the RPython compiler not trace inside of handlers and the callee of a function/method call in a base-program. This is also achieved by placing existing RPython annotations into the interpreter definition.

The *trace stitching* technique. We reconstruct the original control flow of a base-program function/method from a recorded trace. Since the method-traversal interpreter technique will yield a straight-line trace that covers all the execution paths, this technique will split the trace into basic blocks and then connect them together by using branch and jump instructions. This is achieved by adding a post-processing module into the RPython tracer.

Figure 2 shows a high-level example of RPython baseline JIT compiler. The left-hand side of Figure 2 represents the control flow of a target function. B – C – E is a conditional branch, D is a back-edge instruction, and F is a return. The compiler finally generates a trace tree [4], which covers a function body as shown in the right-hand side of Figure 2. In contrast to trace-based compilation, it keeps the original control flow, we can see that the bodies of subroutines are not inlined but call instructions to them are left.

To produce such a trace tree, the tracer of RPython baseline JIT has to sew and stitch generated traces. We call this behavior *trace tailoring*. Technically speaking, the compiler traces a special instrumented interpreter namely *method-traversal interpreter*. Since the obtained trace from the method-traversal interpreter ignores the original control flow, we have to restore

```
@dont_look_inside
def tla_ADD(self, pc):
    x, y = self.pop(), self.pop()
    self.push(y.add(x))
    return pc

@dont_look_inside
def tla_CONST_INT(self, pc):
    arg = ord(self.bytecode[pc])
    self.push(W_IntObject(int(arg)))
    return pc + 1

driver = JitDriver(reds=['self'],
                   greens=['pc','bytecode','traverse_stack'])

class Frame:
    def interp(self, pc, traverse_stack):
        while True:
            driver.jit_merge_point(
                bytecode=self.bytecode,pc=pc,self=self,
                traverse_stack=traverse_stack)
            opcode = ord(self.bytecode[pc])
            pc += 1
            if opcode == ADD:
                pc = self.tla_ADD(pc)
            elif opcode == JUMP:
                ...
            elif opcode == RET:
                ...
            elif opcode == JUMP_IF:
                ...
```

Listing 2 Skeleton of method-traversal interpreter and subroutines decorated with `dont_look_inside`.

[4] Each trace has a linear control flow, but they are compiled as a bridge.

```
DUP,
CONST_INT, 1,
GT,
JUMP_IF, 10,
CONST_INT, 1
SUB,
JUMP, 0
CALL, 23,
EXIT,
```

Listing 3 An example bytecode with the control flow shown in Figure 3.

```
while True:
    if x > 1:
        x -= 1
    else:
        x = call g(x)
        return x
```

Listing 4 An example program corresponding to Listing 3.

```
if opcode == JUMP_IF:
    target = ord(self.bytecode[pc])
    e = self.pop()
    if self._is_true(e):
        if we_are_jitted():
            pc += 1
            # save another direction
            traverse_stack = t_push(
                pc, traverse_stack)
        else:
            if t < pc:
               driver.can_enter_jit(pc=target,
                 bytecode=self.bytecode,self=self,
                 traverse_stack=traverse_stack)
        pc = target
    else:
        if we_are_jitted():
            # save another direction
            traverse_stack = t_push(target,
                traverse_stack)
        pc += 1
```

Listing 5 Definition of JUMP_IF.

```
@dont_look_inside
def cut_here(self, pc):
    "A pseudo function for trace stitching"
    return pc

if opcode == JUMP:
    t = ord(self.bytecode[pc])
    if we_are_jitted():
        if t_is_empty(traverse_stack):
            pc = t
        else:
            pc, traverse_stack = traverse_stack.t_pop()
            # call pseudo function
            pc = cut_here(pc)
    else:
       if t < pc:
           jitdriver.can_enter_jit(
               bytecode=self.bytecode,pc=t,self=self,
               traverse_stack=traverse_stack)
       pc = t
```

Listing 6 Definition of JUMP.

```
if opcode == RET:
    if we_are_jitted():
        if t_is_empty(traverse_stack):
            return self.tla_RET(pc)
        else:
            pc, traverse_stack = traverse_stack.t_pop()
    else:
        return self.tla_RET(pc)
```

Listing 7 Definition of RET.

it. To rebuild the original control flow, in the next phase, the baseline JIT compiler stitches the generated trace. We call this technique *trace stitching*. In the next sections, we will explain method-traversal interpreter and trace stitching, respectively.

3.2. Method-traversal Interpreter

We propose method-traversal interpreter, a specially instrumented interpreter for the baseline JIT compiler. It works as an abstract interpreter because it follows complete control flow graph by exploring both sides of a conditional branch.

The skeleton of method-traversal interpreter is shown in Listing 2. All handlers that are shown at the top of the listing, are decorated by `dont_look_inside` hint which tells the tracer not to trace the function body. Furthermore, specific areas are written in the `then` block of `we_are_jitted`. This hint function returns `True` after entering tracing. Therefore, the resulting trace has only call instructions to subroutines.

Figure 3 shows how method-traversal interpreter traverses a function body with respect to the bytecode denotes in Listing 3 and 4. In Figure 3, the gray-colored dotted line means a generated trace with the method-traversal interpreter. Normally, a tracing JIT only follows an executed side of the conditional branch. In contrast, the baseline JIT tracer follows the both sides. To enable it, method-traversal interpreter manages a special stack data structure called `traverse_stack`. It only stores program counters, so it is marked as *green* and finally removed from the resulting trace.

We explain the behavior of method-traversal interpreter with respect to the examples. The differences from a normal tracing JIT compiler are: (1) conditional branch, (2) back-edge instruction, (3) function call, and (4) function return.

3.2.1. *Conditional branch.*

Our baseline JIT tracer follows both sides of a conditional branch; firstly, tracing `then` branch, and tracing `else` branch next.

When tracing a conditional branch ① in Figure 3, it saves the program counter in another direction of a conditional branch to the `traverse_stack`. Listing 5 shows the handler for the `JUMP_IF`. You can see that `traverse_stack` saves another directions in lines 8 and 19.

3.2.2. *Back-edge instruction.*

Upon a back-edge instruction, the baseline JIT tracer jumps to one of the remaining branches. RPython's original tracer follows a back-edge instruction and finishes tracing when it reaches the beginning of tracing. We modify such a behavior not to finish tracing until the tracer reaches the end of a target method and visits its all paths.

When tracing a back-edge instruction at ②, it does not follow the jump target. Instead, at ③, it pops a program counter from `traverse_stack` and goes to the other branch which is an unfollowed branch of a previous conditional jump (E in the Figure 2).

Seeing the implementation of `JUMP` in Listing 6, before jumping to somewhere, it checks whether `traverse_stack` is empty or not. If empty, the baseline tracer normally executes `JUMP`. Otherwise, it restores the saved program counter

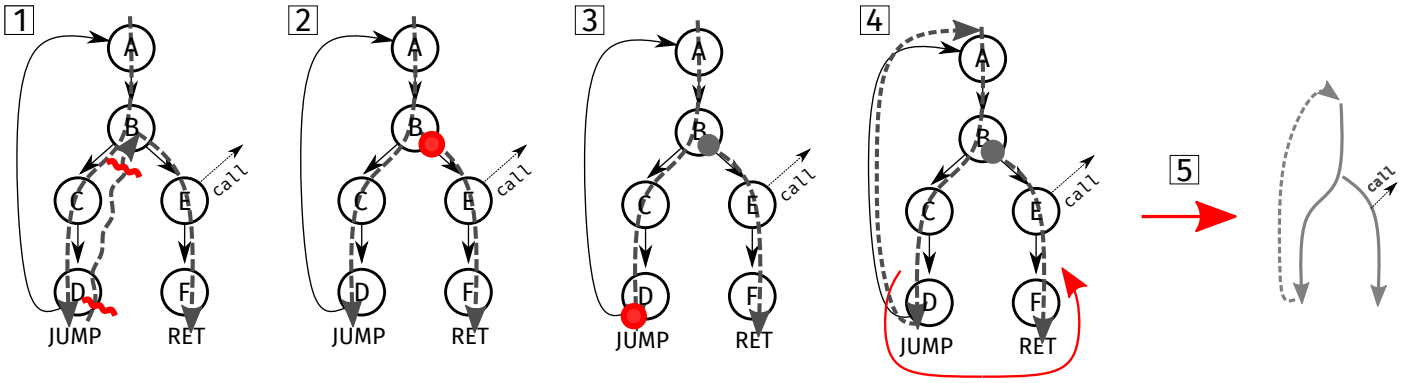

Figure 4 The working flow of trace stitching.

```
[p0]
i1 = call_i(ConstClass(tla_DUP, p0))
i2 = call_i(ConstClass(tla_CONST_INT, p0, 1))
i3 = call_i(ConstClass(tla_GT, p0, 2))
i4 = call_i(ConstClass(_is_true, p0, 4))
guard_true(i4) [p0]
i5 = call_i(ConstClass(tla_CONST_INT, p0, 7))
i6 = call_i(ConstClass(tla_SUB, p0))
i7 = call_i(ConstClass(cut_here, 8))
i8 = call_i(ConstClass(tla_CALL, p0, 10))
i9 = call_i(ConstClass(tla_RET, p0, i8))
leave_portal_frame(0)
finish(i9)
```

Listing 8 The temporarily generated trace from a method-traversal interpreter.

from `traverse_stack` and goes to that place. To tell the place of a back-edge instruction, we have to call a pseudo function `cut_here`. It is used in trace-stitching to restore the original control flow.

3.2.3. Function call.

To reduce the compilation code size, our baseline JIT compiler does not inline a function call.

When tracing `CALL` instruction at ④, it does not follow the destination of `CALL` but emits only a call instruction since sub-routines are decorated with `dont_look_inside`.

3.2.4. Function return.

When tracing `RET` at ⑤, first, the baseline tracer checks whether `traverse_stack` is empty or not. If not empty, it restores a saved program counter and continues to trace. Otherwise, it executes `RET` instruction. The implementation is shown in Listing 7, and the behavior is almost same to `JUMP`.

We finally get the following trace as shown in Listing 8. Note that it is still linear, so we will cut and stitch the generated trace to restore the original control flow.

3.3. Trace Stitching

The obtained trace by tracing method-traversal interpreter is a linear execution path, since the tracer is led to track all paths by the interpreter. For correct execution, we propose trace stitching, which is a technique to reconstruct the original control flow.

Figure 4 shows how trace stitching works, and 1 – 5 indicate its working flow.

1: the tailor cuts where `cut_here` indicates to handle each branch as a separate trace. In Figure 4, the tailor cuts the node B in Figure 4 that `cut_here` points to;

2: the tailor restores the conditional branch by compiling the trace E – F as a bridge. When compiling as a bridge, the tailor emits a label L and rewrites the definition of an original guard failure that is placed at B;

3: the tailor restores `JUMP` instruction at the bottom of D. After that,

4: it copies variables and instructions that are not in the scope of the branch B – E – F for run-time correctness. Finally,

5: the tailor folds or removes constants or unused variables/instructions, respectively.

As a result, we get the trace tree as shown in the rightest side of Figure 4. Inside the RPython, the trace tree is represented as two traces shown in Listing 9. There is no linear trace, but one trace and bridge are connected with a guard failure. If `guard_true(i4)` is failed, the control goes to the Bridge 1 and executes it.

4. Performance of Simulated Threaded Code Generation

In this section, we experimentally evaluate the potential performance of our threaded code generation by simulating the behavior with PyPy. We here compare the threaded code performance against the interpreter performance, although we are also interested in that against the best possible threaded code performance. It would be an interesting future work to implement and compare different threaded code generations in real-world languages like Python.

In Section 3, we described the idea of threaded code generation that enables a baseline compilation with a meta-tracing JIT compiler. The question arises whether the technique is effective

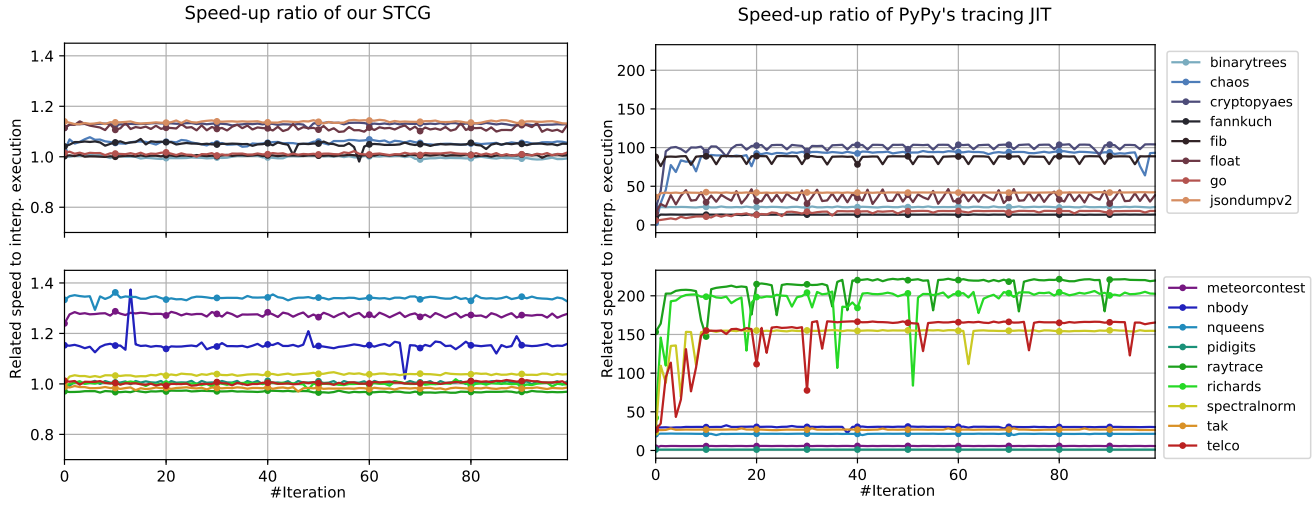

Figure 5 Speed-up ratio of STCG (left-hand side) and PyPy's tracing JIT compiler (right-hand side) related to the interpreter. They are executed on PyPy's original micro benchmark suite plus our original ones. X-axis and Y-axis mean every iteration and speed-up ratio standardized to interp. execution, respectively. Dots are plotted every five iterations.

```
# Loop 1, token number is 13458300
[p0]
i1 = call_i(ConstClass(tla_DUP, p0))
i2 = call_i(ConstClass(tla_CONST_INT, p0, 1))
i3 = call_i(ConstClass(tla_GT, p0, 2))
i4 = call_i(ConstClass(_is_true, p0, 4))
guard_true(i4) [p0] # pointing to Bridge 1
i5 = call_i(ConstClass(tla_CONST_INT, p0, 7))
i6 = call_i(ConstClass(tla_SUB, p0))
# targeting to its own top
jump(p0, descr=TargetToken(13458300))

# Bridge 1, token number is 1345340
[p0]
i8 = call_i(ConstClass(tla_CALL, p0, 10))
i9 = call_i(ConstClass(tla_RET, p0, i8))
leave_portal_frame(0)
finish(i9)
```

Listing 9 Tailored traces. One linear trace is converted into one trace and one bridge, and they are connected with a guard failure.

at runtime or not. To answer the question, we measured JIT compilation time and code size of traces of PyPy's tracing JIT compiler and our simulated threaded code generation (SMTG), respectively. In addition, we compared the potential performance of the two following executions: PyPy 3.7 with SMTG and interpreter-only execution.

4.1. Simulated Threaded Code Generation (STCG) in PyPy

To measure the potential performance of our threaded code generation, we need to reproduce its behavior on PyPy. The brief ideas of the *simulated threaded code generation (STCG)* are;

Idea 1. All subroutines are not inlined, but call instructions to subroutines are left.

Idea 2. Tracing all paths of a target program area *at once*.

Idea 1 can be easily reproduced by adding `dont_look_inside` to the PyPy interpreter manually. The problem is how to reproduce idea 2. The current PyPy doesn't have such a function, but it has a guard failure. A guard failure is a runtime check to ensure the correctness of the generated trace. When the number of failing a guard surpasses a threshold, a tracing JIT starts to trace the destination of a guard and connects the original trace and the generated trace from a guard. Then, if we run programs with enough time, all runtime paths are eventually traced by a guard failure. Therefore, we can reproduce the behavior of idea 2 by running the benchmarks for a long time.

4.2. Setup

In this section, we explain the environment and how we performed our preliminary experiments.

4.2.1. System We conducted the preliminary benchmark on the following environment; CPU: Ryzen 9 5950X, Mem: 32GB DDR4-3200MHz, OS: Ubuntu 20.04.3 LTS with a 64-bit Linux kernel 5.11.0-34-generic.

4.2.2. Implementation We used the original PyPy 3.7 versioned 7.3.5[5], and our modified PyPy 3.7 with STCG[6].

4.2.3. Programs for Experiments You can find all benchmark programs here[7]. We chose all benchmarks that can be executed without any other libraries. Especially, `fib` and `tak` are programs causing the path-divergence problem.

4.2.4. Methodology We conducted two experiments on PyPy's original micro benchmark suite plus our original ones;

[5] https://downloads.python.org/pypy/pypy3.7-v7.3.5-linux64.tar.bz2
[6] https://foss.heptapod.net/pypy/pypy/-/tree/branch/py3.7-hack-measure-bytecode-dispatch
[7] https://foss.heptapod.net/pypy/benchmarks/-/tree/topic/python3_benchmarks/bitbucket-pr-5

Experiment 1. Measuring the overhead of tracing and compilation in our STCG.

Experiment 2. Measuring the stable speeds of our STCG.

Experiment 1. To measure the overhead of tracing and compilation, we used PyPy 3.7–7.3.5 with a tracing JIT and our STCG. We measured their compilation time and the size of traces to compile, and normalized them to PyPy with a tracing JIT. The compilation time includes tracing. Note that the implementation of our full-fledged threaded code generation on PyPy is ongoing, so we simulate the behavior (we describe how to do it in Section 4.1). The results are shown in Figure 6.

Experiment 2. To compare a stable speed, we compared the STCG on PyPy 3.7 with the interpreter-only execution. The interpreter-only execution means that we turn off the JIT compilation by passing `--jit off` when running scripts. We calculated the averages and standard deviations of the STCG normalized to the interpreter-only execution. The results are shown in Figure 7.

We set the max iteration count 100 from the results that plot the related speed-up ratio in the STCG and PyPy's tracing JIT as shown in Figure 5. From those results in the STCG, we can confirm that almost all programs except reach their stable state after the 5th iteration. In addition, the PyPy's tracing JIT reaches its stable speed after the 30th iteration. Experiment 1 requires a number of operations and times for compilation, and experiment 2 needs STCG's stable speed; in this context, we decide that the max iteration count 100 is enough to reach the stable speed. Thus, in experiment 2, we exclude the first 5 iterations for calculating the average value of every program's stable speed.

4.3. Result of Experiment 1: The Overhead of Our STCG

The objective of this experiment is to potentially evaluate the start-up time of our simulated threaded code generation. The results are shown in Figure 6. On average, in the case of trace sizes to compile, PyPy 3.7 with STCG is about 78 % smaller than PyPy 3.7–7.3.5 with a tracing JIT, and 13 of 17 programs are about 50 % smaller than PyPy's tracing JIT. In addition, in the case of compilation time, PyPy 3.7 with STCG is about 60 % shorter than PyPy 3.7–7.3.5 with a tracing JIT. 13 of 17 programs, that are same to the case of the size of traces to compile, are 60 % shorter than PyPy's tracing JIT. However, PyPy 3.7 with STCG size of traces and compilation time on `nbody` is almost the same as that of PyPy's tracing JIT. This program computes the N-body simulation with a matrix calculation. This calculation is implemented as a big for-loop, so there is less effect on performing threaded code generation than full-optimized tracing.

4.4. Result of Experiment 2: The Stable Speed

The results are summarized in Figure 7 (their values in every iteration are shown in Figure 5). The results of PyPy 3.7 simulated threaded code generation are normalized to the interpreter only execution. On average, STCG is 7% faster than the interpreter only. PyPy 3.7 with STCG is over 4 % faster in 9 of the 17 benchmarks, and $\pm$ 3 % faster in 8 of the 17 benchmarks. In particular, `meteorcontest` and `nqueens` are from about 27% to 34 % faster than the interpreter.

4.5. Discussion

In experiment 1, there is a relation between the size of traces and the compilation time. Our simulated threaded code generation can reduce the size of traces and compilation time, so we can use it for reducing the start-up time.

Moreover, programs with the path-divergence problem (`fib` and `tak`) are at least 96 % smaller and 80 % faster in trace size and compilation time, respectively. In general, when the path-divergence problem occurs, retracing often happens, and too many traces overlap each other and lead to high overhead in run-time performance. However, the result shows that the STCG traces and compiles only a primary hot function, so the trace sizes and compilation time are much smaller and shorter than a tracing JIT. Thus, we can say that a method-based threaded code can reduce the trace size and compilation time.

From both experiments, we can infer that our method-based threaded code generation will bring some benefits to a start-up performance. To make the technique more effective, we should select functions that have similar structure to `meteorcontes` and `nqueens` as well as programs with the path-divergence problem. In those programs, much part of one primary solver function with complex conditional branches is executed inside the main loop, but the other functions are not. In other words, during solving conditions, instead of running a main single region over and over, some regions are sometimes run randomly. This execution model potentially causes the path-divergence problem. Thus, a method-based threaded code generation can work effectively on such programs. To enhance the effectiveness of our method-based threaded code generation with this assumption, we need to select programs with complex conditional branches inside a long iteration in addition to programs which indeed cause the path-divergence problem.

Limitation of Threaded Code Generation. Threaded code generation is placed at the initial compilation tier, so the compilation limits further optimizations. For example, since the compilation does not inline an instruction handler but leaves a call instruction to that. We notice that there are gaps between this baseline JIT compilation and the tracing JIT that RPython provides. Thus, this gap suggests that we need several optimizations between the baseline and tracing JITs.

To allow further optimizations like allocation removal, we are going to implement higher levels of baseline JIT compilation. For instance, the tier-2 baseline JIT just inlines a stack manipulation, but other operations are not. The tier-3 baseline JIT inlines auxiliary methods but others are not inlined. We are able to realize those levels at low cost by placing `dont_look_inside` to each method header.

5. Related Work

The trade-off between compilation time and peak performance has been actively discussed in the context of compiler implementation. For long-running applications such as server-side

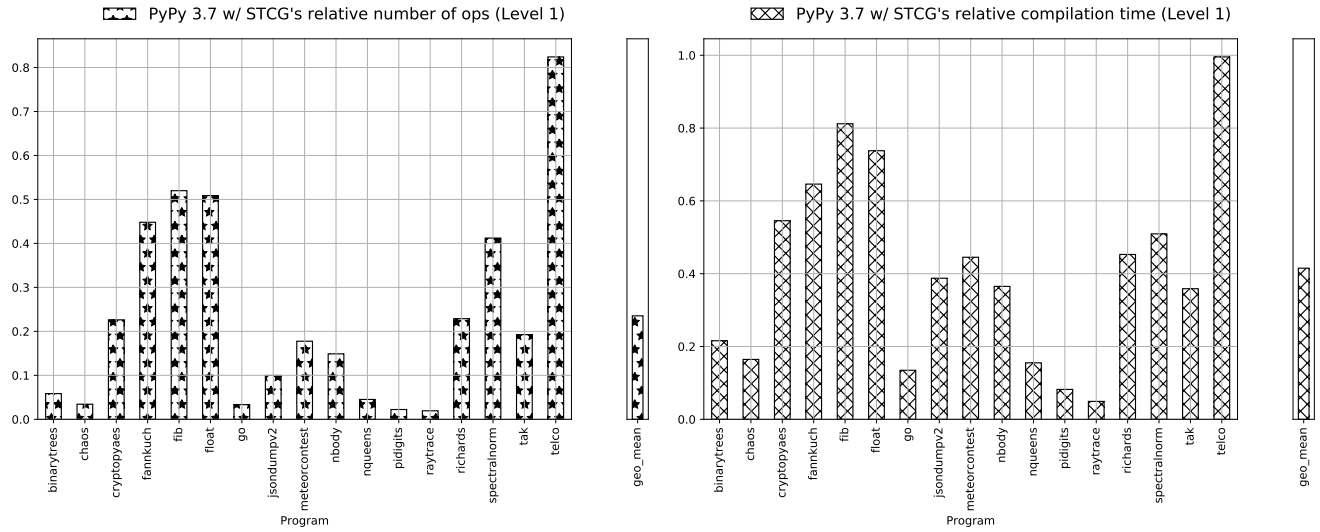

Figure 6 The results of the size of traces to compile and compilation time including tracing. In all results the Y-axis means PyPy 3.7 with our simulated threaded code generation (STCG)'s relative value to PyPy 3.7–7.3.5's tracing JIT compiler. The X-axis stands for the name of every program. The left-hand side shows the relative trace size, and the right-hand size is the relative compilation time. Lower is better.

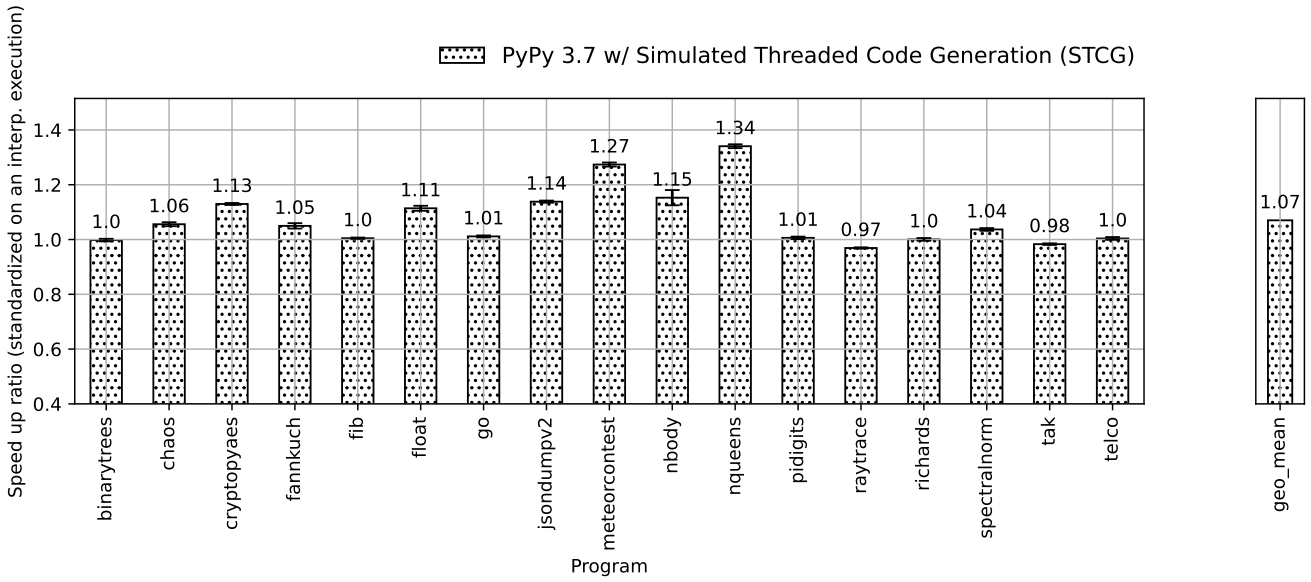

Figure 7 The results of a preliminary benchmark experiment. In all results the Y-axis means speed up ratio of the threaded code generation comparing to the interpreter-only execution, and the X-axis stands for the name of every program. The error bars mean standard deviations. Higher is better.

programs, we would accept a long compilation time. In contrast, for short-term applications such as GUI programs or batch processing programs, we would require a better response time, hence we usually apply a baseline JIT compiler at first.

The Java HotSpot™ VM has two JIT compilers: the server compiler (Paleczny et al. 2001) and the client compiler (Kotzmann et al. 2008). The server compiler is a highly optimizing compiler and is tuned to gain a faster peak-time performance with lower compilation speed. On the other hand, the client compiler is a JIT compiler designed for low start-up time and small memory footprint.

The Firefox baseline compiler (Vijayan 2013) is a warm-up compiler used in the IonMonkey JavaScript JIT compiler (Mozilla 2016). Firefox's baseline JIT is designed to work as an intermediate layer between interpretation and highly optimizing JIT compilation. Firefox used different JIT compilers, JaegerMonkey and IonMonkey, depending on the situation, but it had several issues. For example, the calling conventions of the two compilers are different. Moreover, JaegerMonkey itself has a too complex structure to easily extend. Firefox's baseline

JIT compiler is designed to solve these issues. Its baseline JIT compiler is simpler than other compilers, but runs 10–100 times faster than interpretation.

The Liftoff (Google 2018) is a baseline JIT compiler for V8 and WebAssembly. V8 has an older JIT compiler called TurboFan, but its compilation process is complicated, and it consumes longer compilation time. Liftoff makes the code quality secondary in order to achieve a faster start-up time, which is the key difference from the TurboFan compiler.

The Safari's JavaScript engine, JavaScriptCore, has 4-tier optimization levels in its VM (**?**). The engine consists of low-level interpreter (tier-1), baseline JIT (tier-2), data-flow graph JIT (DFG, tier-3), and forth-tier JIT (FTJ, tier-4) compilers. Execution firstly enters the interpreter-tier, and level-shifting between every JIT compiler can be executed by on-stack replacement (OSR) (**?**). In particular, the baseline JIT does not apply serious optimizations but just eliminates the interpretation overhead. Polymorphic inline caching (PIC) (**?**) is used in the baseline JIT a classic optimization technique to remove dynamic method dispatching, and profiling information gathered when PIC is performed are passed to higher-level JIT compilers.

6. Conclusion and Future Work

6.1. Conclusion

In this paper, we proposed the idea of a method- and threaded-code-based RPython's baseline JIT compiler and how to implement them on top of RPython. The essential technique is trace tailoring that consists of the method-traversal interpreter and trace stitching. A method-traversal interpreter is an interpreter design that tricks the trace to follow all paths of a target function. Trace stitching rebuilds a trace tree from a resulting trace generated from a method-traversal interpreter, aiming to restore the original control flow. In average, our experiments report that threaded code can reduce the size of traces and compilation time by about 80 % and 60 %, respectively. It can run 7 % faster than the interpreter-only execution in the case of a stable speed.

6.2. Future Work

Multitier Adaptive Compilation. In the viewpoint of code quality and compilation time, our threaded code generation is placed at an interpreter execution and tracing JIT compilation. We would connect them and shift the compilation level depending on a target program. For example, we start baseline JIT compilation before applying tracing JIT compilation. Then, when we find a program fragment that is suitable for tracing JIT compilation, we use a tracing JIT compiler instead of a baseline JIT compiler. In the future, we would realize such an adaptive compilation strategy on RPython.

Implementing Threaded Code Generation on PyPy. Currently, we designed a method-traversal interpreter for a tiny language and created a compiler that could emit a trace tree that contains only call instructions to subroutines. Our next task is to implement our idea on PyPy. By comparing with the original tracing JIT in RPython and PyPy, we will see how much start-up time and memory footprint can be reduced in practice. Finally, we will verify the effectiveness of our baseline JIT on production-level applications by using the PyPy that has a baseline, method (that will be extended from baseline JIT), and tracing compilation strategies. Given this context, we will implement it in Python with the PyPy interpreter to run production-level benchmarks.

Acknowledgments

We would like to thank the reviewers of the ICOOOLPS 2021 workshop for their valuable comments. This work was supported by JSPS KAKENHI grant number 18H03219, 21J10682, and JST ACT-X grant number JPMJAX2003.

References

Alpern, B., Attanasio, C. R., Cocchi, A., Lieber, D., Smith, S., Ngo, T., … Mergen, M. (1999). Implementing jalapeño in java. In *Proceedings of the 14th acm sigplan conference on object-oriented programming, systems, languages, and applications* (p. 314–324). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/320384.320418

Bell, J. R. (1973, June). Threaded code. *Commun. ACM*, *16*(6), 370–372. doi: 10.1145/362248.362270

Bolz, C. F., Cuni, A., Fijalkowski, M., & Rigo, A. (2009). 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* (pp. 18–25). New York, NY, USA: ACM. doi: 10.1145/1565824.1565827

Ertl, M. A., & Gregg, D. (2003). The structure and performance of efficient interpreters. *Journal of Instruction-level Parallelism*, *5*.

Felgentreff, T., Pape, T., Rein, P., & Hirschfeld, R. (2016). 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* (pp. 21:1–21:10). New York, NY, USA: ACM. doi: 10.1145/2991041.2991062

Fijałkowski, M., Rigo, A., Lamy, R. G. R., Pawluś, S., Oruganti, A., & Barrett, E. (2014). *Hippyvm - an implementation of the php language in rpython.* Retrieved from http://hippyvm.baroquesoftware.com/#performance

Gaynor, A., Felgentreff, T., Nutter, C., Phoenix, E., Ford, B., & PyPy development team. (2013). *A high performance ruby, written in RPython.* Retrieved from http://docs.topazruby.com/en/latest/

Google. (2018). *Liftoff: a new baseline compiler for webassembly in v8.* Retrieved from https://v8.dev/blog/liftoff

Hong, P. J. (1992, October). Threaded code designs for forth interpreters. *SIGFORTH Newsl.*, *4*(2), 11–16. doi: 10.1145/146559.146561

Izawa, Y., & Masuhara, H. (2020). Amalgamating different jit compilations in a meta-tracing jit compiler framework. In *Proceedings of the 16th acm sigplan international symposium on dynamic languages* (p. 1–15). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3426422.3426977

Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K., & Cox, D. (2008, May). Design of the java hotspot™ client compiler for java 6. *ACM Trans. Archit. Code Optim.*, *5*(1). Retrieved from https://doi.org/10.1145/1369396.1370017 doi: 10.1145/1369396.1370017

Mozilla. (2016). *IonMonkey, the Next Generation JavaScript JIT for SpiderMonkey.* Retrieved from https://wiki.mozilla.org/IonMonkey

Niephaus, F., Felgentreff, T., & Hirschfeld, R. (2019). 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* (pp. 14–26). New York, NY, USA: ACM. doi: 10.1145/3357390.3361024

Oracle Lab. (2013). *A high performance implementation of the ruby programming language.* Retrieved from https://github.com/oracle/truffleruby

Oracle Lab. (2015). *A high-performance implementation of the R programming language, built on GraalVM.* Retrieved from https://github.com/oracle/fastr

Oracle Labs. (2018). *Graal/Truffle-based implementation of Python.* Retrieved from https://github.com/graalvm/graalpython

Paleczny, M., Vick, C., & Click, C. (2001). The Java Hotspot™ Server Compiler. In *Proceedings of the 2001 symposium on javatm virtual machine research and technology symposium - volume 1* (p. 1). USA: USENIX Association.

Rigo, A., & Pedroni, S. (2006). PyPy's Approach to Virtual Machine Construction. In *Companion to the 21st acm sigplan symposium on object-oriented programming systems, languages, and applications* (pp. 944–953). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1176617.1176753

Vijayan, K. (2013). *The Baseline Compiler Has Landed.* Retrieved from https://blog.mozilla.org/javascript/2013/04/05/the-baseline-compiler-has-landed/ (Mozilla)

Würthinger, T., Wöundefined, A., Stadler, L., Duboscq, G., Simon, D., & Wimmer, C. (2012). Self-optimizing ast interpreters. In *Proceedings of the 8th symposium on dynamic languages* (p. 73–82). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2384577.2384587

About the authors

Yusuke Izawa is a Ph.D. student at the Tokyo Institute of Technology (Japan). You can contact the author at izawa@prg.is.titech.ac.jp or visit https://www.yuiza.org.

Hidehiko Masuhara is a professor at the Tokyo Institute of Technology. You can contact the author at masuhara@acm.org.

Carl Friedrich Bolz-Tereick is PyPy/RPython contributor and scientific employee at Heinrich-Heine-Universität Düsseldorf. You can contact the author at cfbolz@gmx.net or visit https://cfbolz.de.

Youyou Cong is an assistant professor at the Tokyo Institute of Technology. You can contact the author at cong@c.titech.ac.jp.