Taming an Interpreter for Threaded Code Generation with a Tracing JIT Compiler

Yusuke Izawa
yizawa@acm.org
Tokyo Institute of Technology
Tokyo, Japan

Hidehiko Masuhara
masuhara@acm.org
Tokyo Insitute of Technology
Tokyo, Japan

ABSTRACT

Modern virtual machines support a multitier JIT compilation strategy to balance the code quality and compilation time. This strategy brings many benefits to the user. However, it is hard for virtual machine developers to build and maintain multiple compilers in a single managed runtime. In this work-in-progress paper, we describe the problem that occurred by our use of a meta-tracing JIT compiler and propose a solution to it based on our previous work. Our preliminary performance evaluation of the compilation and execution times suggests that the proposed threaded code generation is promising as a lower-tier runtime compiler in a multitier JIT compilation system.

KEYWORDS

JIT compiler, interpreter, meta-tracing JIT compiler, multitier JIT compilation, RPython

1 INTRODUCTION AND BACKGROUND

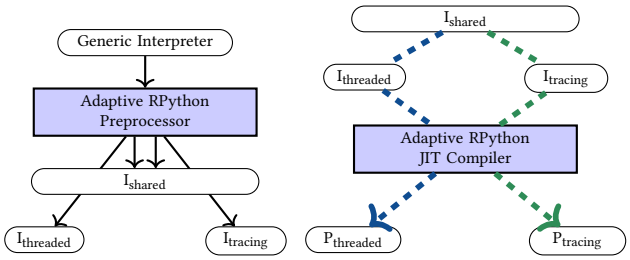
Many managed language runtimes provide multiple optimization levels to balance code quality and compilation time. Not only research-oriented virtual machines (VMs) like Jikes RVM [1], but also recent practical VMs, such as OpenJDK[1], SpiderMonkey[2], and V8[3], have multiple optimization levels. In particular, OpenJDK has two different compilers, namely C1 [8] and C2 [9], and four different levels of optimization.

Creating more than one compiler and optimizer from scratch and maintaining them long-term requires language developers a significant amount of effort. For example, Mozilla developers struggled with the complexity and pain of managing two different JIT compilers, namely, TraceMonkey and JaegerMonkey, as a single JavaScript VM[4]. However, that mechanism must be needed in a "wild" situation where a VM runs a variety of software; from batch processing to GUI-based programs, and from data processing to business-oriented server-side applications.

In this work-in-progress paper, we improve our threaded code generation technique [7] by proposing the *shallow tracing*, which lets the tracing compiler yield only handler calls without performing side effects. Moreover, we implement a small language with our proposed framework called Adaptive RPython, by extending our previous work [5, 6] that enables a meta-tracing compiler to emit code at different optimization levels.

[1]https://openjdk.java.net/
[2]https://spidermonkey.dev/
[3]https://v8.dev/
[4]https://blog.mozilla.org/javascript/2013/04/05/the-baseline-compiler-has-landed/

ICOOOLPS '22, June 07,2022, Berlin, Germany
.



(a) At the preprocessing step. **(b) At the JIT compilation phase.**

Figure 1: At the preprocessing phase, we generate interpreters from the generic interpreter. Next, at the JIT compilation phase, we choose an optimal interpreter depending on the current runtime situation.

The rest of the paper is organized as follows. First, section 3 explains the problem when we naïvely apply threaded code generation to an interpreter and the solution called shallow tracing. Second, section 2 shows the overview of Adaptive RPython and its compilation steps. Third, section 4 measures the compilation and execution times of our threaded code generation to compare with other code generation techniques. Finally, section 5 concludes the paper.

2 OUR APPROACH: ADAPTIVE RPYTHON

This section briefly introduces Adaptive RPython which can generate code of different code quality and show its compilation steps, namely, preprocessing and JIT compilation steps.

Adaptive RPython [6] can change its level of optimization depending on the corresponding interpreter definition. For example, we can let the meta-tracing JIT compiler perform threaded code generation (tier1) by providing a specially annotated interpreter to the meta-tracing JIT [7]. The compilation steps can be divided into the following steps;

Preprocessing step. Figure 1a illustrates the preprocessing step of Adaptive RPython in the case of preparing the two-level JIT compilation. At the top of this figure, the adaptive RPython preprocessor generates interpreters corresponding to each level from a common interpreter definition called *generic interpreter*. The preprocessor produces a shared interpreter that collects common definitions and other level-specific primitives[5] to reduce virtual machine footprints.

[5]The primitives consist of the handler of `CALL/RET`, `JUMP`, and `JUMP_IF` (i.e. branching) instructions.

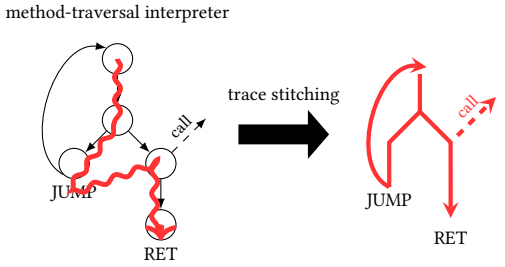


Figure 2: Overview of the threaded code generation technique. The tree on the left-hand side shows an example of the target program, and the final output is shown on the right-hand side.

JIT compilation step. In the preprocessing step, Adaptive RPython generated interpreters that correspond to each optimization level. The JIT compiler has to choose the appropriate interpreter definitions for the current optimization level. Looking at Figure 1b, if threaded code generation is to be performed at runtime, the Adaptive RPython JIT compiler chooses the appropriate interpreter definition I_{shared} and $I_{threaded}$) (blue dashed line). Furthermore, if we want to move to tier2, we let the JIT compiler select I_{shared} and $I_{tracing}$ (the green dashed line).

3 INCONSISTENCY PROBLEM AND OUR SOLUTION IN THREADED CODE GENERATION

In this section, we introduce a problem called *inconsistency problem*, which is caused by our use of a meta-tracing JIT compiler, and our solution to it. To illustrate them, first of all, we briefly explain how we realize our threaded code generation by exploiting a meta-tracing JIT compiler.

3.1 Overview of Threaded Code Generation

Usually, in the context of meta-tracing JIT compilation, an interpreter definition is a target of a partial evaluator. In the case of RPython, its meta-tracing JIT compiler produces trace-based compiled code by specializing the interpreter with respect to the source program at runtime.

$$\text{RPython}[\text{interp}_{\text{tracing}}, \text{source}] = \text{executable}_{\text{tracing}}$$

Since the meta-tracing compiler generates a specialized interpreter with respect to a trace of the interpretation of a source, it can be considered as a specialized case of the first Futamura projection [4].

Our technique obtains executables of the source at different optimization levels by providing different interpreter definitions. The following equations illustrate how we want to explain those in the above sentences.

$$\text{RPython}[\text{interp}_{\alpha}, \text{source}] = \text{executable}_{\alpha}$$
$$\text{RPython}[\text{interp}_{\beta}, \text{source}] = \text{executable}_{\beta}$$
$$\cdots$$

Threaded code generation [7] is realized by an interpreter that is instrumented to perform threaded code [3]. The running overview is shown in Figure 2. First, RPython's tracer follows the execution of an instrumented interpreter called *method-traversal interpreter*. After tracing it, the temporarily generated trace does not keep the original structure. We apply *trace stitching* to retrieve the original.

On the interpreter side, this technique is achieved by placing RPython hint functions according to the following policies (P1) — (P4).

(P1) *Leave only a CALL instruction.* the method-traversal interpreter suppresses inlining by decorating all handlers with a hint `dont_look_inside`.

(P2) *Start at the beginning of a function and do not inline a function call.* Tracing compilation usually starts at the top of a loop or function, but the method-traversal interpreter lets the RPython's tracer (1) start at the top of a function and (2) not inline a function call.

(P3) *Follow all sides of a branch.* the method-traversal interpreter tells the RPython tracer to follow all sides of the branch. This behavior is enabled by the traversal stack technique [7], which saves another side of the branch to trace the side later. The saved pcs are retrieved when the tracer reaches the end of a function, for example, RET or JUMP.

(P4) *Basically do not end the tracing in JUMP or RET.* the method-traversal interpreter does not finish tracing there but requires the tracer to do more things: trace another branch saved at P3. Moreover, the threaded code interpreter requires language developers to put pseudo-functions `emit_JUMP` or `emit_RET` for trace stitching.

3.2 Shallow Tracing to Solve Inconsistency Problem

When we naïvely apply the threaded code generation technique to an interpreter of a practical language, we will encounter the *inconsistency problem*, where the tracing compiler makes the interpreter's state inconsistent after tracing. This is caused by the nature of the tracing compiler, which actually executes the program during the tracing. It is not a problem when the tracing compiler only follows the trace in the actual program execution. However, with our threaded code generation technique, we let the tracing compiler execute all paths in the method, which includes paths that might not be executed in the actual execution. This makes the interpreter's state (like the operand stack) inconsistent, which will result in incorrectly compiled code. The problem is even worse when the interpreter performs global side effects, such as I/O operations and function calls.

To avoid this problem, we propose *shallow tracing technique*, allowing the tracer to follow the code without also executing it. The key to this technique is to add a *dummy* flag in the last argument of each bytecode handler. The value of that flag is set to `True` during tracing, but after the trace has been recorded it gets rewritten to `False` in the trace.

First, we explain that technique from the interpreter definition side. The example definition is shown in the Listing 1. Looking into

```
@dont_look_inside
def ADD(dummy):
  # do nothing
  if dummy: return
  w_y, x_x = pop(), pop()
  push(w_x.add(w_y))

def interp():
  while True:
    jit_merge_point(..)
    if opcode == ADD:
      if we_are_jitted():
        ADD(dummy=True)
      else:
        ADD(dummy=False)
    ...
```

```
def f(n):
    if n < 1:
        return g(n)
    return n + 1

# bytecode
# "f":
#     DUP,
#     CONST_I, 1,
#     LT,
#     JUMP_IF, L2,
# L1:
#     CALL "g",
#     RET
# L2:
#     CONST_I, 1,
#     ADD,
#     RET
```

Listing 1: Example of an interpreter definition for the shallow-tracing technique (left-side). In addition, the function f and its bytecode that is applied to the shallow tracing technique (right-side).

```
# Just after shallow tracing.
# Flags are still activated.

# Loop 0
call_n('DUP', p0, 1)
call_n('CONST_I, 1', p0, 1)
i0 = call_n('LT', p0, 1)
guard_true(i0) [p0]
call_n('CALL', p0, 'g', 1)
p31 = call_n('RET', p0, 1)
finish(p31)

# Bridge 0
call_n('CONST_I, 1', p0, 1)
call_n('ADD', p0, 1)
p31 = call_n('RET', p0, 1)
finish(p31)
```

```
# Just after deactivating flags.

# Loop 0
call_n('DUP', p0, 0)
call_n('CONST_I, 1', p0, 0)
i0 = call_n('LT', p0, 0)
guard_true(i0) [p0]
call_n('CALL', p0, 'g', 0)
p31 = call_n('RET', p0, 0)
finish(p31)

# Bridge 0
call_n('CONST_I, 1', p0, 0)
call_n('ADD', p0, 0)
p31 = call_n('RET', p0, 0)
finish(p31)
```

Listing 2: Before and after applying shallow tracing to the function f are shown on the right-hand side of Listing 1. During shallow tracing, all flags are activated (left side), but they are finally deactivated in the resulting trace (right side).

the ADD handler, on the handler side, all handlers need to have the extra flag dummy. During tracing, it should be True to do nothing but leave only a call operation to the handler. On the dispatching the loop side, we manage the dummy's state with the we_are_jitted hint function. we_are_jitted returns true during tracing, so we turn on the dummy flag at then branch while turning off at else branch.

Then, we explain how that technique works with several examples. The running example is shown in Listings 2. First, the RPython tracer shallowly traverses all paths of the function f, which is displayed on the right-hand side of Listing 1. Then, we get the trace as shown on the left-hand side of Listing 2. Next, we deactivated all extra flags dummy to make the trace runnable. We already know where those flags are. Therefore, we can automatically turn them off. Finally, we get the executable traces as shown on the right-hand side of Listing 2.

4 PRELIMINARY EVALUATION

In this section, we preliminarily evaluate whether threaded code generation can be tier1 JIT in Adaptive RPython. Tier1 JIT is in a very early stage in all optimization tiers, so we compare the startup speeds of an interpreter, the generation of threaded code, and the tracing JIT executions. Then, we discuss an adaptive compilation strategy to appropriately apply an optimal compilation strategy from the point of view of running speed.

The objective of this preliminary evaluation is to find the break-even points between the three tiers; interpreter, threaded code, and tracing JIT executions. Therefore, we decided to measure the cumulative execution times of each microbenchmark program.

Methodology. According to Barret et al.'s method [2], we ran every microbenchmark program with 2000 in-process iterations and 30 out-process iterations. To compare the earlier performance characteristics, we plot the cumulative speeds of each program's first 30 in-process iterations.

Target. As a microbenchmark program, we took the same programs that we used in our previous project called BacCaml [5] [6]. We use a small language called TLA that has integer, floating, and array variables. The source code can be accessed at Heptapod[7], and the bytecode compiler is hosted on GitHub[8].

Environment. We ran all microbenchmarks on the standalone server machine, which equips the Ryzen 9 5950X CPU, 32 GB DDR4-3200MHz Memory, and the Ubuntu 20.04.3 LTS operating system with a 64-bit Linux kernel 5.11.0-34-generic.

Threat to validity. The tested language is smaller than widely used ones, so the results may change if we test on production-level languages like PyPy.

4.1 Results and Discussion: Can Threaded Code Generation Be a Tier1 JIT?

Figure 3 shows the cumulative execution times. From these plots, we can find the following four performance patterns;

- ♢ **Tracing.** Tracing JIT is the fastest at the beginning. fib-tail, gcd, random, sum-tail are in this pattern. Those programs are loop-heavy.
- ♣ **Threaded-Tracing.** The threaded code is dominant first and the trace JIT finally becomes the fastest. ack, fib, square, sum, tak, and tarai are in this pattern. Those programs have at least one recursive call.
- ♠ **Interp-Threaded-Tracing.** First, the interpreter is predominant, the next threaded code is predominant, and finally, the trace JIT becomes the fastest. ary and prefix_sum are in this pattern. Those programs manipulate an array of data structures.
- ♡ **Exception.** In fact, the interpreter is first predominant, and finally tracing JIT becomes the fastest. While in the sieve, threaded code is predominant and fastest.

By looking into these patterns, we discuss the performance characteristics of threaded code generation from the point of view of startup performance. Especially in ♣ and ♠ patterns, threaded code is predominant in early iterations within the first 0.3 s. Otherwise, as in the pattern ♢, the threaded code does not show good performance even in the starting situation for loop-heavy programs. These

[6] They are chosen from all runnable programs from the shootout benchmark suite. The original shootout benchmark suite is here: https://dada.perl.it/shootout/.
[7] https://foss.heptapod.net/pypy/pypy/-/tree/branch/threaded-code-generation
[8] https://github.com/prg-titech/prg-caml

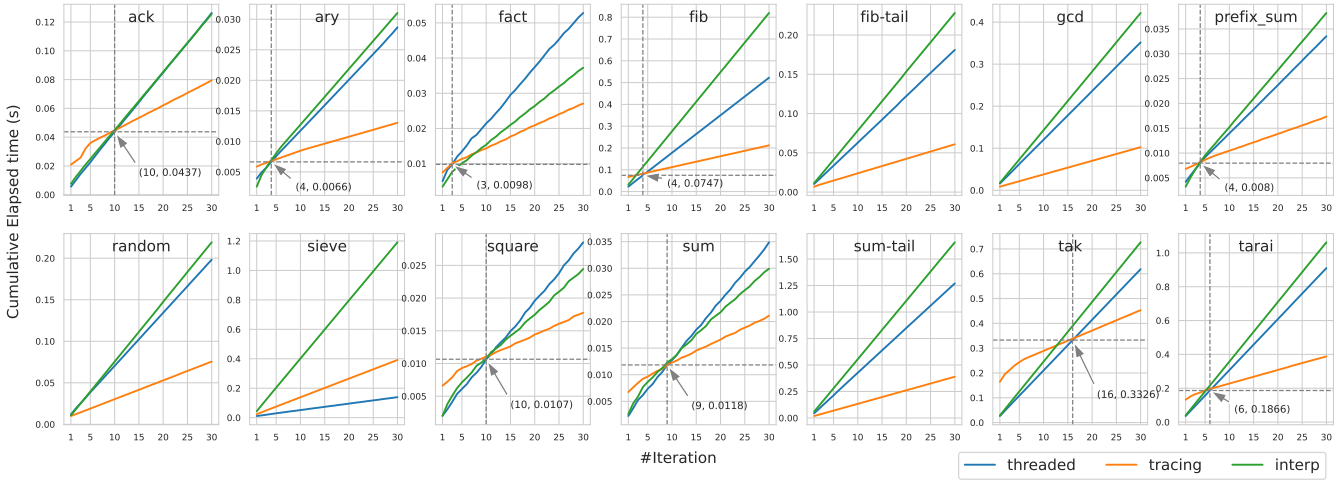


Figure 3: Result of cumulative execution times of each microbenchmark program. Because of the space, break-even points between threaded code and tracing JIT are only plotted.

results imply that threaded code can be useful in the initial stage of execution, but the use of trace JIT is better from the start when programs are run in loops heavily. Meanwhile, we find the unusual pattern ♡ and, in fact, the sieve. fact uses multiplication, and sieve has a point where many equally probable branchings occur. Investigating why such situations occur is left as future work since there is not enough material to judge.

Those results suggest a promising sign that threaded code generation can be used as tier-1 JIT in RPython. When we use threaded code generation as a tier1 JIT, its threshold should be much lower than that of tracing JIT's. In the case of RPython, its tracing JIT's threshold is 1039 for loops. Given this context, that of threaded code generation should be 5.

5 CONCLUSION AND FUTURE WORK

In this paper, we propose a new technique called shallow tracing, which avoids an inconsistency between the states of interpretation and tracing in threaded code generation. Next, we illustrate the overview of Adaptive RPython and its compilation steps. These steps can be divided into preprocessing and JIT compilation steps. At the preprocessing step, Adaptive RPython generates a shared interpreter and other primitives for each optimization level from a common interpreter definition called the generic interpreter. Then, we preliminarily evaluated the performance of threaded code generation as a tier1 JIT compilation strategy. From the result, we got the implication that the threaded code generation can be useful in the early stage of execution, especially for programs with at least one recursive call.

We have a lot of work to do in the future. First, we need to evaluate the performance of threaded code generation in larger languages. As a reasonable next step, we believe that PySOM[9] is a good option to change from TLA since it already has a well-written interpreter implementation and many benchmark programs. Second, we must develop a level-shifting strategy in the context of adaptive compilation. Of course, we are investigating a reasonable way to realize our tier1 JIT on PyPy. Finally, we need to develop a strategy for adaptive compilation in Adaptive RPython. For example, we should consider a more concrete condition when shifting from interpreter to threaded code generation, or from threaded code generation to tracing JIT.

[9]https://github.com/smarr/PySOM

REFERENCES

[1] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2000. Adaptive Optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Minneapolis, Minnesota, USA) *(OOPSLA '00)*. Association for Computing Machinery, New York, NY, USA, 47–65. https://doi.org/10.1145/353171.353175

[2] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 52 (oct 2017), 27 pages. https://doi.org/10.1145/3133876

[3] James R. Bell. 1973. Threaded Code. *Commun. ACM* 16, 6 (June 1973), 370–372. https://doi.org/10.1145/362248.362270

[4] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler. *Higher Order Symbol. Comput.* 12, 4 (Dec. 1999), 381–391. https://doi.org/10.1023/A:1010095604496

[5] Yusuke Izawa and Hidehiko Masuhara. 2020. Amalgamating Different JIT Compilations in a Meta-Tracing JIT Compiler Framework. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages* (Virtual, USA) *(DLS '20)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3426422.3426977

[6] Yusuke Izawa, Hidehiko Masuhara, and Carl Friedrich Bolz-Tereick. 2022. Two-level Just-in-Time Compilation with One Interpreter and One Engine. https://doi.org/10.48550/ARXIV.2201.09268

[7] Yusuke Izawa, Hidehiko Masuhara, Carl Friedrich Bolz-Tereick, and Youyou Cong. 2022. Threaded Code Generation with a Meta-Tracing JIT Compiler. *Journal of Object Technology* 21, 2 (2022), a1. arXiv:2106.12496 In press.

[8] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot™ Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.* 5, 1, Article 7 (May 2008), 32 pages. https://doi.org/10.1145/1369396.1370017

[9] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java Hotspot™ Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1* (Monterey, California) *(JVM '01)*. USENIX Association, USA, 1.