Two-level Just-in-Time Compilation with One Interpreter and One Engine

Yusuke Izawa
izawa@prg.is.titech.ac.jp
Tokyo Institute of Technology
Tokyo, Japan

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

Carl Friedrich Bolz-Tereick
cfbolz@gmx.de
Heinrich-Heine-Universität
Düsseldorf
North Rhine-Westphalia, Germany

Abstract

Modern, powerful virtual machines such as those running Java or JavaScript support multi-tier JIT compilation and optimization features to achieve their high performance. However, implementing and maintaining several compilers/optimizers that interact with each other requires hard-working of VM developers. In this paper, we propose a technique to realize two-level JIT compilation in RPython without implementing several interpreters or compilers from scratch. As a preliminary realization, we created adaptive RPython which performs both baseline JIT compilation based on threaded code and tracing JIT compilation. We also implemented a small programming language with it. Furthermore, we preliminarily evaluated the performance of that small language, and our baseline JIT compilation achieved ran 1.77x faster than the interpreter-only execution. Furthermore, we observed that when we apply an optimal JIT compilation for different target methods, the performance was mostly the same as the one optimizing JIT compilation strategy, saving about 40 % of the compilation code size.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**.

Keywords: JIT compiler, threaded code, RPython, language implementation framework

1 Introduction

Language implementation frameworks, e.g., RPython [6] and Truffle/Graal [24], are tools to build a virtual machine (VM) with a highly efficient just-in-time (JIT) compiler by merely providing an interpreter of the language. For example, PyPy [5], which is a fully compatible Python implementation, achieved 4.5x speedup from the original CPython 3.7 interpreter [23]. The other successful examples include Topaz [9], Hippy VM [8], TruffleRuby [20], and GraalPython [21].

One of the limitations of RPython and Truffle/Graal is that they don't support a multi-tier JIT compilation strategy in contrast to existing language-specific VMs such as Java or JavaScript. One naïve approach for multi-tier compilation is to create compilers for each optimization level. However, this approach makes existing implementations more complex and requires more developing efforts to implement and maintain – we need at least to consider how to share components, and how to exchange profiling information and compiled code between each compiler. To avoid such a situation, we have to find another efficient and reasonable way to support multi-tier JIT compilation in a framework. Several works in RPython [2, 4, 14] found that specifying special annotations in an interpreter definition can influence how RPython's meta-tracing JIT compiler works. In other words, we can view the interpreter definition as a specification of a compiler. We believe those approaches can be extended to achieve an adaptive optimization system in meta-level.

As a proof-of-concept of language-agnostic multi-tier JIT compilation, we propose *adaptive RPython*. Adaptive RPython can generate a VM with two-level JIT compilation. We do not create two separated compilers in adaptive RPython but generate two different interpreters – the one is for the first level of compilation and the other is for the second level. As first-level compilation, we support threaded code [3, 13] in a meta-tracing compiler (we call this technique *threaded code generation* [15]). The second one is RPython's original tracing JIT compilation. We can switch optimization levels by moving between the two interpreters. In addition, adaptive RPython generates two interpreters from one definition. It will reduce implementation costs that the user of adaptive RPython does not need to pay. In the future, we plan to extend such a system to realize a nonstop optimization-level changing mechanism in a language implementation framework – we can use an appropriate and efficient optimization level or a compilation strategy depending on the executed program.

The contributions of this paper can be summarized as follows:

- An approach to generate multiple interpreter implementations from one common interpreter definition to obtain JIT compilers with different optimization levels.
- The technical details of enabling threaded code generation as a first-level JIT compilation by driving an existing meta-tracing JIT compiler engine.
- The preliminary evaluation of our two-level JIT compilation on a simple programming language.

PEPM '22, January 17–18, 2022, Philadelphia, Pennsylvania, United States
2022.

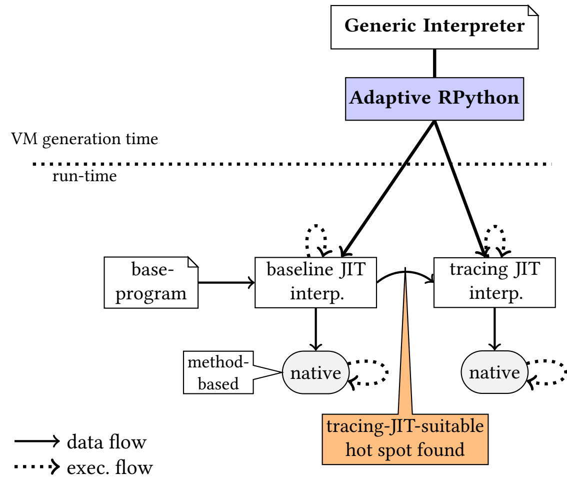


Figure 1. An overview of adaptive RPython: two different interpreters are generated from a single generic interpreter by adaptive RPython at VM generation time. At run-time, the two interpreters, baseline JIT and tracing JIT interpreters, behave level 1 and level 2, respectively.

The rest of this paper is organized as follows. Section 2 proposes an idea and technique to support two-level JIT compilation with one interpreter in RPython without crafting a compiler from scratch. We evaluate our preliminary implementation in Section 3. Section 4 discusses related work, and we conclude the paper and denote the future work in Section 5.

2 Two-level Just-in-Time Compilation with RPython

In this section, we propose a multi-tier meta-tracing JIT compiler framework called *adaptive RPython*, and its technique to support two different compilation levels with one interpreter and one engine. We firstly introduce adaptive RPython in Section 2.1. Then, in Section 2.2 and 2.3 we explain the technical details to realize two-level JIT compilation in RPython. Sections 2.2 and 2.3 correspond to "one interpreter" and "one engine", respectively.

2.1 Adaptive RPython

Adaptive RPython steers the existing meta-tracing engine to behave in two ways, as both a baseline JIT compiler and a tracing JIT compiler. To realize this behavior, the most obvious approach would be to implement two different compilers or interpreters. However, this approach increases the amount of implementation necessary. Thus, we do not craft compilers from scratch but steer an existing compilation engine by providing a specializing interpreter called the *generic interpreter*. Figure 1 shows an overview of how adaptive RPython and the generic interpreter work. At VM generation time (the upper half of Figure 1), the developer writes the *generic interpreter*. Adaptive RPython generates two different interpreters that support different JIT compilation level, e.g., baseline JIT and tracing JIT interpreters work as level-1 and level-2, respectively. At run-time (the bottom half of Figure 1), a generated baseline JIT interpreter firstly accepts and runs a base-program. While running the program, the execution switches to a generated tracing JIT interpreter if the hot spot is turned out to be suitable for tracing JIT compilation.

```
1  jittierdriver = JitTierDriver(pc='pc')
2
3  class Frame:
4      def interp(self);
5          pc = 0
6          bytecode = self.bytecode
7          jitdriver.jit_merge_point(pc=pc,
8              bytecode=bytecode,self=self)
9          opcode = ord(bytecode[pc])
10         pc += 1
11         if opcode == JUMP_IF:
12             target = bytecode[pc]
13             jittierdriver.can_enter_tier1_branch(
14                 true_path=target,false_path=pc+1,
15                 cond=self.is_true)
16             if we_are_in_tier2(kind='branch'):
17                 # do stuff
18
19         elif opcode == JUMP:
20             target = bytecode[pc]
21              jittierdriver.can_enter_tier1_jump(target=target)
22              if we_are_in_tier2(kind='jump'):
23                  # do stuff
24
25         elif opcode == RET:
26             w_x = self.pop()
27             jittierdriver.can_enter_tier1_jump(ret_value=w_x)
28             if we_are_in_tier2(kind='ret'):
29                 # do stuff
```

Listing 1. A skeleton of the generic interpreter definition.

2.2 Generic Interpreter

Adaptive RPython takes the generic interpreter as input. The generic interpreter is converted into an interpreter that includes two definitions that are individually for baseline JIT and tracing JIT compilations. Listing 1 illustrates a skeleton of the generic interpreter definition. To remove the task of manually writing redundant definitions in the method-traversal interpreter [15], we internally generate both the method-traversal interpreter (for baseline JIT compilation) and a normal interpreter (for tracing JIT compilation) from the generic interpreter definition.

To convert the generic interpreter into several interpreters with two different definitions, we need to tell adaptive RPython some necessary information. For that reason, we implement several hint functions for the generic interpreter along with RPython's original hints. The skeleton is shown in Listing 1. When developers write the generic interpreter, they firstly declare an instance of `JitTierDriver` class that has a field pc. It tells adaptive RPython fundamental information such as the variable name of the program counter. Furthermore, transforming hints should be defined in a specific handler. `can_enter_tier1_branch`, `can_enter_tier1_jump` and `can_enter_tier1_ret` tell the adaptive RPython's

transformer the necessary information to generate the method-traversal interpreter. The method-traversal interpreter requires a particular kind of code in `JUMP_IF`, `JUMP`, and `RET` bytecode handlers, so that hints are called in those handlers. The requisite information in each handler and hint is the following:

can_enter_tier1_branch. The method-traversal interpreter needs to drive the engine to trace both sides of a branch instruction. Thus, we have to pass the following variables/expressions to the transformer:

- the true and false paths of program counters: to manage them in the traverse stack
- the conditional expression: to generate `if` expression.

In the handler of `JUMP_IF` in Listing 1, we pass the following statements/expressions: `target` as `true_path`, `pc+1` as a `false_path` and `self.is_true` as `cond`. We also remember and mark the `cond` to utilize at trace-stitching inside of adaptive RPython (details are explained in Section 2.3.1).

can_enter_tier1_jump. A jump instruction possibly placed at the end of a function/method. In such a case, the engine does not follow the destination but takes out a program counter from top of `traverse_stack`. Otherwise, the jump instruction performs as defined in the original interpreter . Thus, the method-traversal interpreter requires the program counter of a jump target to manage it in the `traverse_stack`. As shown in the handler of `JUMP` in Listing 1, we pass `target` to the `transform_jump` function.

can_enter_tier1_ret. A return instruction is invoked at the end of a function/method. The method-traversal interpreter requires a return value, so we have to pass `w_x` via `transform_ret` as illustrated in the handler of `RET` in Listing 1.

we_are_in_tier2. This is a hint function to tell the area where a definition for tracing JIT compilation is written to the transformer. One thing we need to do is to specify the type of handler function that is defined immediately above. For example, we add the keyword argument `kind='branch'` in the handler of `JUMP_IF` in Listing 1.

2.3 Just-in-Time Trace Stitching

The *just-in-time trace stitching* technique builds back up the original control of a trace yielded from the method-traversal interpreter to emit an executable JITted code. This is the essential component in baseline JIT compilation of adaptive RPython.

To reconstruct the original control flow, we need to connect a correct guard operation and bridging trace. Therefore, we should specially treat the destination of a "false" branch while doing trace-stitching. In tracing JIT compilers,

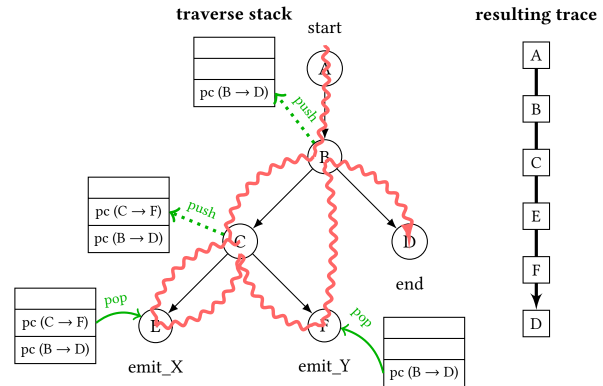


Figure 2. An overview of method-traversing in a program with nested branches. The left-hand side shows how we drive the meta-tracing engine by using traverse stack. The snake line represents the trail of tracing. The right-hand side is a resulting trace.

at branching instructions guards are inserted.[1] After a guard operation fails many times, a tracing JIT compiler will trace the destination path starting from the failing guard, i.e., trace the other branch. The resulting trace from a guard failure is called a bridge, which will be connected to the loop body. On the other hand, in trace-stitching, we perform such a sequence of generating and connecting a bridge in one go. For the sake of ease and reducing code size, we generate a trace essentially consisting of call operations by threaded code generation.[2]

The left-hand side of Figure 2 illustrates the process of traversing an example target method that has nested branches. In tracing a branch instruction, we record the other destination, e.g., a program counter from B to D in node B, in the traverse stack. For example, when tracing B and C, we push each program counter from B to D and from C to F to traverse stack. Then, we pop a program counter from the traversal stack and set it to E and F. After traversing all paths, we obtain a single trace that does not keep the structure of the target as shown in the right-hand side of Figure 2.

2.3.1 Resolving Guard Failures and Bridges.

Intending to resolve the relations between guards and bridges, we utilize the nature of the method-traversal interpreter: it manages base-program's branches as a stack data structure, so the connections are first-in-last-out. Thus, we implement *guard failure stack* to manage each guard's failure. The guard failure stack saves guard failures in each guard operation

[1] Technically speaking, type-checking guards are inserted to optimize the obtained trace based on the observed run-time type information. However, we currently handle branching guards only and leave type optimization to the last JIT tier.

[2] Note that threaded code generation yields essentially call and guard operations. In contrast to normal tracing JIT compilation, it can reduce the code size and compilation time by 80 % and 60 % [15].

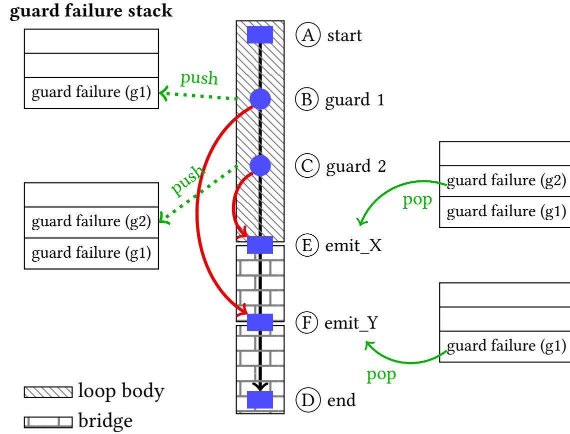


Figure 3. An overview of just-in-time trace-stitching. This shows how we resolve the relations between guard failures and bridges as taking Figure 2 for example.

and pops them at the start of a bridge, that is, right after `emit_jump` or `emit_ret` operations that tell a cut-here line.

Before explaining the details of the algorithm, we give a high level overview of how we resolve the connections between guard failures and bridges by using Figure 3 as an example. First of all, the necessary information is that which guard failure goes to which bridges. We resolve it by sequentially reading and utilizing the guard failure stack. When we start to stitch the trace shown in Figure 3, we sequentially read the operations from the trace. In node B, when it turned out that the guard operation is already marked as cond in the generic interpreter, we take its `guard failure (g1)` and push it to the guard failure stack. In node C, we do the same thing in the case of node B. Next, in node E, we finish the tracing an operation and cut the current trace. In addition, we pop a guard failure and connect it to the bridge that we are going to retrieve, since the bridges are lined up below with a depth-first approach. In node F, we do the same thing in the case of node F. In node D, finally, we finish reading and produce one trace and two bridges. The connections are illustrated as red arrows in Figure 3.

2.3.2 The Mechanism of Just-in-Time Trace-Stitching.

How JIT trace-stitching works is explained through Algorithm 1. First of all, the trace-stitcher DoTraceStitching prepares an associative array called `token_map`, and the (key, value) pair is (`program counter`, `target_token`).[3] Next, it declares `guard_failure_stack`, `trace` and `result`. `guard_failure_stack` is a key data structure to resolve the relation between guard failures and bridges. `trace` temporarily stores a handled operation, and `result` memorizes a pair of a `trace` and its corresponding `guard_falure`. After that, it manipulates each operation in the given `ops`. We specially handle the following operations: (1) a guard operation marked by the generic interpreter, (2) pseudo call operations `CallOp(emit_ret)` and `CallOp(emit_jump)`, and (3) RPython's return and jump operations. Note that almost all operations except guards are represented as a call operation since the resulting trace is produced from our threaded code generator.

Marked guard operation. To resolve the guard-bridge relation, we firstly collect the guard operations that we have marked in the generic interpreter. The reason we mark some guards is to distinguish between branching guards and others. When we encounter such a guard operation, we take its `guard_failure` and append it to the `guard_failure_stack`. Its algorithm is shown in the first branching block of Algorithm 1.

Pseudo call operations. Pseudo functions `emit_jump` and `emit_ret` are used as a sign of cutting the trace at this position and to start recording a bridge. They are represented as `Call(emit_jump)` and `Call(emit_ret)` in a trace.

In the case of `CallOp(emit_jump)`:

1. look up a `target_token` from the `token_map` using the target program counter as a key
2. try to retrieve a `guard_failure` from the `guard_failure_stack`. If it the stack is empty, we do nothing since the current recording trace is a body (not a bridge). Otherwise, pop a guard failure from the `guard_failure_stack`.
3. create a jump operation with the `inputargs` and `target_token` and append to the `trace`.
4. append the pair of the `trace` and `guard_failure` to the `result`.

In the case of `CallOp(emit_ret)`:

1. take a return value (`retval`) from the operation.
2. take a `guard_failure` if it is not first time.
3. create a return operation with the `retval` and append to the `trace`.
4. append the pair of the `trace` and `guard_failure`.

RPython's Jump and Return. These operations are placed at the end of a trace. When we read them, we append the pair of the operation and retrieve the guard failure to the `result`, and finish reading.

3 Preliminary Evaluation

In this section, we evaluate our implementations of JIT trace-stitching and baseline JIT compilation. Since our work is not finished, these can only be preliminary.

[3] `target_token` is an identifier for a trace/bridge. When we encounter `emit_jump` or `emit_ret` operations, we get the program counter (key) passed as an argument to `emit_jump` or `emit_ret` operations, and create a new `target_token` (value).

3.1 Implementation

We wrote a small interpreter called TLA in adaptive RPython and executed several micro benchmark programs on it. In the TLA language, we have both primitive and object types, and they are dynamically typed.

Simulating Multi-tier Compilation. To conduct this preliminary evaluation, we partially implemented two-level JIT compilation by separately defining interpreters for each execution level – baseline JIT compilation (level 1), and tracing JIT compilation (level 2). In other words, to support multi-tier JIT compilation, we prepared two interpreters that have different jitdrivers in TLA; the one is for baseline JIT compilation and the other is for tracing JIT compilation. For example, during executing a base-program, we call separate interpreters for different JIT compilation levels which are manually specified in a base-program.

Accessibility. The implementations of proof-of-concept adaptive RPython and TLA interpreter is hosted on Heptapod.[4] Moreover, the generic interpreter implementations are hosted on our GitHub organization.[5]

3.2 Targets

To verify that our JIT trace-stitching mechanism works on a program with some complex structure, we wrote a single loop program **loop** and nested loop program **loopabit** in TLA for the experiments.

Furthermore, to confirm the effectiveness of shifting different JIT compilation levels, we wrote **callabit** that has two different methods and each method has a single loop. According to the specified JIT compilation strategy, callabit has the following variants:

(a) **callabit_baseline_interp**: the main method is compiled by baseline JIT compilation, but the other is interpreted.
(b) **callabit_baseline_only**: the two methods are compiled by baseline JIT compilation.
(c) **callabit_baseline_tracing**: the main method is compiled by baseline JIT compilation, and the other is by tracing JIT compilation.
(d) **callabit_tracing_baseline**: the main method is compiled by tracing JIT compilation, and the other is by baseline JIT compilation.
(e) **callabit_tracing_only**: all methods are compiled by tracing JIT compilation.

All bytecode programs used for this evaluation are shown in Appendix C.2.

[4] https://foss.heptapod.net/pypy/pypy/-/tree/branch/threaded-code-generation/rpython/jit/tl/threadedcode
[5] https://github.com/prg-titech/mti_transformer

3.3 Methodology

We took two kinds of data: the times of stable and startup speeds. When we measured stable speed, we discarded the first iteration and accumulated the elapsed time of each 100 iterations. In contrast, we measured the startup speed; we iterated 100 times the spawning of an execution process. In addition, we took how many operations they emit and how much time they consumed in tracing and compiling in the case of callabit programs. Note that in every benchmark, we did not change the default threshold of the original RPython to enter JIT compilation.

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

3.4 Result

Compiling Single and Nested Loops. Figure 4 visualizes the resulting traces from loopabit.tla by baseline JIT compilation. We can initially confirm that our baseline JIT compilation works correctly when we look at this figure. Figure 5a and 5b show the result of stable and startup times in loop and loopabit programs, respectively. In stable speed, on average, baseline JIT and tracing JIT are 1.7x and 3.25x faster than the interpreter-only execution. Or more specifically, baseline JIT compilation is about 2x slower than tracing JIT compilation. In startup time (Figure 5b), baseline JIT compilation is about 1.9x faster and tracing JIT compilation is about 5x faster. The loop and loopabit programs are much suitable for tracing JIT compilation, so we consider that tracing JIT compilation should be dominant in executing such programs. Furthermore, at applying baseline JIT compilation for such a program, we should reduce the value of threshold to enter JIT compilation. Tuning the value is left for our future work.

Simulating Multi-tier JIT Compilation. Figure 6 shows how the simulated multi-tier JIT compilation works on a program with two different methods. The callabit has `main` method that repeatedly calls `sub_loop` that reduces the given number one by one. The call method is not implemented by using jump but invoking an interpreter, so the effectiveness of inlining by tracing JIT compilation is limited in this case. In other words, `main` is relatively suitable for baseline JIT compilation and `sub_loop` is for tracing JIT compilation.

In the stable and startup speeds (Figure 6a and 6b), callabit_baseline_interp is about 3 % slower than the interpreter-only execution. This means that repeating back and forth between native code and an interpreter execution leads to run-time overhead. Meanwhile, the combination of baseline JIT and tracing JIT compilations (callabit_baseline_tracing) is as fast as tracing JIT compilation-only strategy (callabit_tracing_only). Additionally, when seeing the Figure 7, the baseline-tracing JIT strategy's trace size is about 40 % smaller than the only tracing

JIT strategy. In contrast, the trace sizes are the same between baseline-tracing JIT and tracing-baseline JIT strategies, but the tracing-baseline JIT strategy is about 45 % slower than baseline-tracing JIT strategy, and the baseline-only strategy is about 5 % faster than tracing-baseline JIT strategy. From those results, we can deduce that there is a ceiling to using only a single JIT strategy. Furthermore, to leverage different levels of JIT compilations, we have to apply an appropriate compilation according to the structure or nature of the target program.

In summary, our baseline JIT compilation is about 1.77x faster than the interpreter-only execution in both stable and startup speeds.[6] Moreover, our baseline JIT compilation is only about 43 % slower than the tracing JIT compilation, even though it has very few optimizations such as inlining and type specialization. This means that our approach to enabling baseline JIT compilation alongside with tracing JIT compilation has enough potential to work as startup compilation if we carefully adjust the threshold to enter a baseline JIT compilation. This is left as one of the essential future work.

4 Related Work

Both well-developed VMs, such as Java VM or JavaScript VM, and research-oriented VMs with a certain size support multi-tier JIT compilation to balance among the startup speed, compilation time, and memory footprint. As far as the authors know, such VMs build at least two different compilers to realize multi-tier optimization. In contrast, our approach realizes it in one engine with a language implementation framework.

The Java HotSpot™ VM has the two different compilers, that are C1 [16] and C2 [22], and four optimization levels. The typical path is moving through the level 0, 3 to 4. Level 0 means interpreting. On level 3, the C1 compiler compiles a target with profiling information gathered by the interpreter. If the C2's compilation queue is not full and the target turns out hot, the C2 starts to optimize the method aggressively (level 4). Level 1 and 2 are used when the C2's compilation queue is full, or level 3 optimization cannot work.

The Firefox JavaScript VM called SpiderMonkey [17] has several interpreters and compilers to enable multi-tier optimization. For interpreters, it has normal and baseline interpreters [18]. The baseline interpreter supports inline caches [7, 12] to improve its performance. The baseline JIT compiler uses the same inline caching mechanism, but it translates the entire bytecode into machine code. In addition, a full-fledged compiler WarpMonkey [19] compiles a hot spot into fast machine code. Besides such a JavaScript engine, the SpiderMonkey VM has an interpreter and compiler called WASM-Baseline and WASM-Ion.

[6] We calculated the geometric mean of loop, loopabit, and callabit_baseline_only in both stable and startup speeds.

Google's JavaScript engine V8 that is included in the Chrome browser also supports a multi-tier compilation mechanism [10]. V8 sees it as a problem that the JIT-compiled code can consume much memory, but it runs only once. The baseline interpreter/compiler is called Ignition, and it is so highly optimized to collaborate with V8's JIT compiler engine Turbofan. It can reduce the code size up to 50 % by preserving the original Google said Ignition reduced the memory footprint of each Chrome tab by around 5 %.

Google's V8 has another optimizing compiler called Liftoff [11]. The Liftoff compiler is designed for a startup compiler of WebAssembly and works alongside Turbofan. Turbofan is based on its intermediate representation (IR), so it needs to translate WebAssembly code into the IR, leading to a reduction in the startup performance of the Chrome browser. However, Liftoff instead directly compiles WebAssembly code into machine code. The liftoff compiler is tuned to quickly generate memory-efficient code to reduce the memory footprint at startup time.

The Jikes Java Research VM (originally called Jalapeño) [1] that was developed by IBM Research, is a research-oriented VM that is written in Java. It has baseline and an optimizing JIT compilers and supports an optimization strategy in three-level.

5 Conclusion and Future Work

In this paper, we proposed the concept and initial stage implementation of adaptive RPython, which can generate a VM that supports two-level compilation. In realizing adaptive RPython, we did not implement another compiler from scratch but drove the existing meta-tracing JIT compilation engine by a specially instrumented interpreter called the generic interpreter. The generic interpreter supports a fluent API that can be easily integrated with RPython's original hint function. The adaptive RPython compiler generates different interpreters that support a different compilation level. The JIT trace-stitching reconstructs the initial control flow of a generated trace from a baseline JIT interpreter to emit the executable native code. In our preliminary evaluation, when we manually apply a suitable compilation depending on the control flow of a target method, we confirmed that the baseline-tracing JIT compilation runs as fast as tracing JIT-only compilation and reduces 50 % of the trace size. From this result, selecting an appropriate compilation strategy according to a target program's control flow or nature is essential in the multi-tier compilation.

To implement an internal graph-to-graph conversion of the generic interpreter in RPython is something we plan to work on next. We currently implement the generic interpreter transformer as source-to-source since it is a proof-of-concept. For a smoother integration with RPython, we need to switch implementation strategies in the future.

To realize the technique to automatically shift JIT compilation levels in adaptive RPython also needs to be implemented, including the investigation of suitable heuristics of when to go from one tier to the next.

Finally, we would implement our adaptive RPython techniques in the PyPy programming language since it brings us many benefits. For example, we can obtain a lot of data by running our adaptive RPython on existing polished benchmark programs to determine a certain threshold to switch a JIT compilation. Furthermore, we could potentially bring our research results to many Python programmers.

Acknowledgments

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

References

[1] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. 1999. Implementing Jalapeño in Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, USA) *(OOPSLA '99)*. Association for Computing Machinery, New York, NY, USA, 314–324. https://doi.org/10.1145/320384.320418

[2] Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. 2015. Pycket: A Tracing JIT for a Functional Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. ACM, New York, NY, USA, 22–34. https://doi.org/10.1145/2784731.2784740

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

[4] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. 2011. Runtime Feedback in a Meta-tracing JIT for Efficient Dynamic Languages. In *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems* (Lancaster, United Kingdom) *(ICOOOLPS '11)*. ACM, New York, NY, USA, Article 9, 8 pages. https://doi.org/10.1145/2069172.2069181

[5] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, and Armin Rigo. 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* (Genova, Italy). ACM, New York, NY, USA, 18–25. https://doi.org/10.1145/1565824.1565827

[6] Carl Friedrich Bolz and Laurence Tratt. 2015. The Impact of Meta-tracing on VM Design and Implementation. *Science of Computer Programming* 98 (2015), 408 – 421. https://doi.org/10.1016/j.scico.2013.02.001 Special Issue on Advances in Dynamic Languages.

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

[8] Maciej Fijałkowski, Armin Rigo, Rafał Gałczyński, Ronan Lamy, Sebastian Pawluś, Ashwini Oruganti, and Edd Barrett. 2014. *HippyVM - an implementation of the PHP language in RPython.* Retrieved 2021-10-07 from http://hippyvm.baroquesoftware.com

[9] Alex Gaynor, Tim Felgentreff, Charles Nutter, Evan Phoenix, Brian Ford, and PyPy development team. 2013. *A high performance ruby, written in RPython.* Retrieved 2021-10-07 from http://docs.topazruby.com/en/latest/

[10] Google. 2016. *Firing up the Ignition interpreter.* Retrieved 2021-10-07 from https://v8.dev/blog/ignition-interpreter

[11] Google. 2018. *Liftoff: a new baseline compiler for WebAssembly in V8.* https://v8.dev/blog/liftoff

[12] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 European Conference on Object-Oriented Programming*, Pierre America (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 21–38.

[13] P. Joseph Hong. 1992. Threaded Code Designs for Forth Interpreters. *SIGFORTH Newsl.* 4, 2 (Oct. 1992), 11–16. https://doi.org/10.1145/146559.146561

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

[15] Yusuke Izawa, Hidehiko Masuhara, Carl Friedrich Bolz-Tereick, and Youyou Cong. 2021. Threaded Code Generation with a Meta-tracing JIT Compiler. (Sept. 2021). arXiv:2106.12496 submitted for publication.

[16] 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

[17] Mozilla. 2019. *Spider Monkey: Mozilla's JavaScript and WebAssembly Engine.* Retrieved 2021-10-07 from https://spidermonkey.dev

[18] Mozilla. 2019. *SpiderMonkey's JavaScript Interpreter and Compiler.* Retrieved 2021-09-27 from https://firefox-source-docs.mozilla.org/js

[19] Mozilla. 2020. *Warp: Improved JS performance in Firefox 83.* Retrieved 2021-10-07 from https://hacks.mozilla.org/2020/11/warp-improved-js-performance-in-firefox-83/

[20] Oracle Lab. 2013. *A high performance implementation of the Ruby programming language.* https://github.com/oracle/truffleruby

[21] Oracle Labs. 2018. *Graal/Truffle-based implementation of Python.* Retrieved 2021-10-07 from https://github.com/graalvm/graalpython

[22] 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.

[23] PyPy development team. 2009. *PyPy Speed Center.* Retrieved 2021-09-27 from https://speed.pypy.org

[24] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 662–676. https://doi.org/10.1145/3062341.3062381

A The Algorithm of Just-in-Time Trace-Stitching

Algorithm 1: DoTraceStitching(inputargs, ops)

```
input :red variables inputargs of the given trace
input :a list of operations ops taken from the given trace
/* Note that token_map and guard_failure_stack are global
   variables                                              */
token_map ← CreateTokenMap (ops);
guard_failure_stack, trace, result ← [], [], [];
for op in ops do
    if op is guard and marked then
        guard_failure ← GetGuardFailure (op);
        append guard_failure to guard_failure_stack;
    else if op is call then
        if op is CallOp(emit_jump) then
            trace, guard_failure ← HandleEmitJump(op, inputargs);
            append (trace, guard_failure) to result;
            trace ← [];
        else if op is CallOp(emit_ret) then
            trace, guard_failure ← HandleEmitRet(op);
            append (trace, guard_failure) to result;
            trace ← [];
        else
            append op to trace;
    else if op is JumpOp then
        append op to trace;
        guard_failure ← PopGuardFailure ();
        append (trace, guard_failure) to result;
        break;
    else if op is RetOp then
        append op to trace;
        guard_failure ← PopGuardFailure ();
        append (trace, guard_failure) to result;
        break;
    else
        append op to trace;
return result;
Function PopGuardFailure():
    if first pop? then
        return None;
    else
        failure ← pop the element from guard_failure_stack;
        return failure;
Function HandleEmitJump(op, inputargs):
    target ← GetProgramCounter (op);
    token ← token_map [target];
    guard_failure ← PopGuardFailure ();
    append JumpOp(args, token) to trace;
    return trace, guard_failure;
Function HandleEmitRet(op):
    retval ← GetRetVal(op);
    guard_failure ← PopGuardFailure();
    append RetOp(retval) to trace;
    return trace, guard_failure;
```

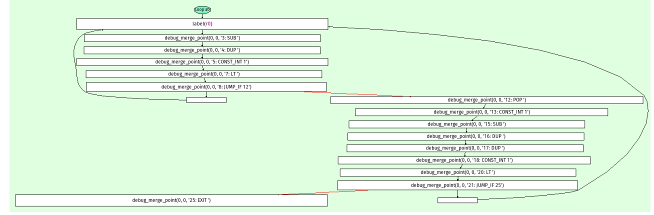
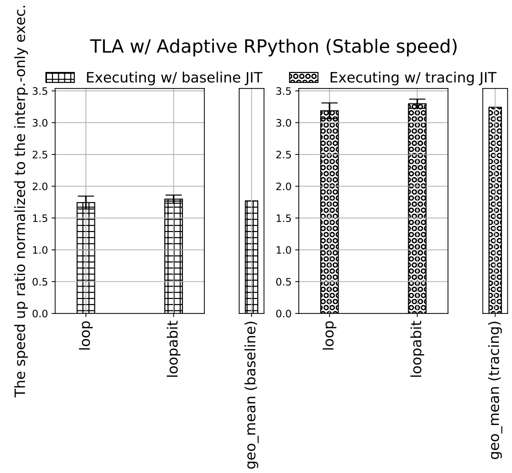
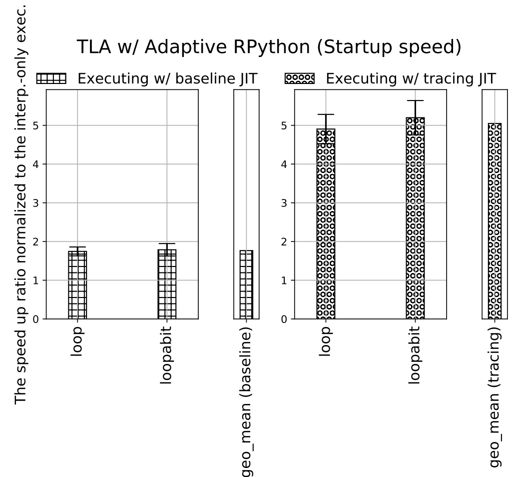


Figure 4. The visualization of the resulting traces from loopabit.tla compiled by baseline JIT compilation. Note that each trace was joined at one compile time.

B Results of the Preliminary Evaluation

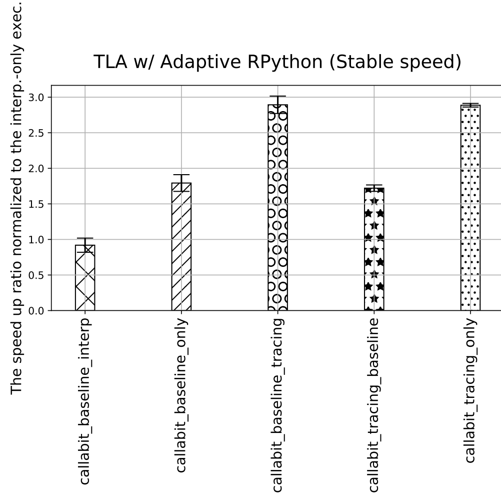


(a) The result of the stable speeds of loop and loopabit.

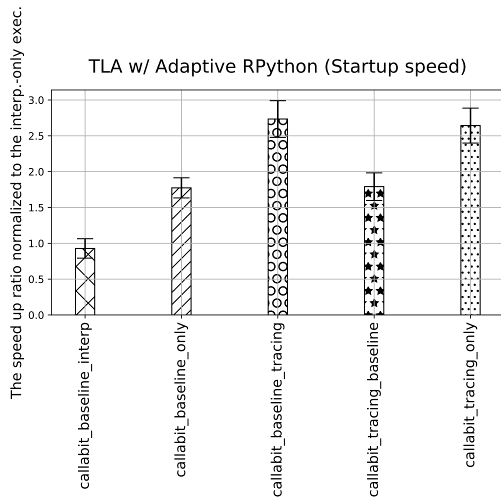


(b) The result of the startup speeds of loop and loopabit.

Figure 5. The results of loop and loopabit. Execution times are normalized to the interpreter-only execution. Higher is better



(a) The result of the stable speeds of callabit programs.



(b) The result of the startup speeds of callabit programs.

Figure 6. The results of callabit programs with simulated multi-tier JIT compilation. Every data is normalized to the interpreter-only execution. Higher is better.

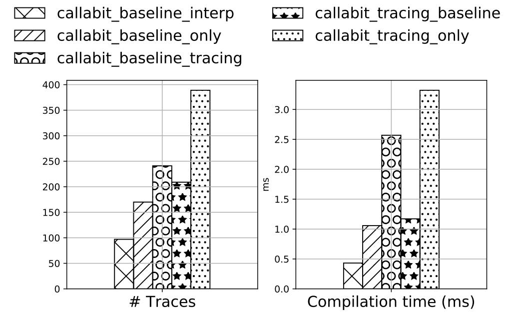


Figure 7. The trace sizes and compilation times in callabit programs. The program is so small that the compilation time is at most 3 % of the total.

C Programs

C.1 The Definition of Traverse Stack

```
1  class TraverseStack:
2      _immutable_fields_ = ['pc', 'next']
3
4      def __init__(self, pc, next):
5          self.pc = pc
6          self.next = next
7
8      def t_pop(self):
9          return self.pc, self.next
10
11     @elidable
12     def t_is_empty(self):
13         return self is _T_EMPTY
14
15 _T_EMPTY = None
16
17 @elidable
18 def t_empty():
19     return _T_EMPTY
20
21 memoization = {}
22
23 @elidable
24 def t_push(pc, next):
25     key = pc, next
26     if key in memoization:
27         return memoization[key]
28     result = TraverseStack(pc, next)
29     memoization[key] = result
30     return result
```

Listing 2. The definition of `traverse_stack`.

C.2 Bytecode Programs Used for Preliminary Evaluation

```
# loop.tla
tla.DUP,
tla.CONST_INT, 1,
tla.LT,
tla.JUMP_IF, 11,
tla.CONST_INT, 1,
tla.SUB,
tla.JUMP, 0,
tla.CONST_INT, 10
tla.SUB,
tla.EXIT,
```

```
# loopabit.tla
tla.DUP,
tla.CONST_INT, 1,
tla.SUB,
tla.DUP,
tla.CONST_INT, 1,
tla.LT,
tla.JUMP_IF, 12,
tla.JUMP, 1,
tla.POP,
tla.CONST_INT, 1,
tla.SUB,
tla.DUP,
tla.DUP,
tla.CONST_INT, 1,
tla.LT,
tla.JUMP_IF, 25,
tla.JUMP, 1,
tla.EXIT
```

Listing 3. The definitions of `loop`, `loopabit`.

```
# callabit.tla
# - callabit_baseline_interp replaces XXX with tla.CALL_NORMAL, 16
# - callabit_baseline_tracing and callabit_traing_only replace XXX
#   with tla.CALL_JIT, 16
# main(n)
tla.DUP,
tla.CALL, 16, # XXX
tla.POP,
tla.CONST_INT, 1,
tla.SUB,
tla.DUP,
tla.CONST_INT, 1,
tla.LT,
tla.JUMP_IF, 15,
tla.JUMP, 0,
tla.EXIT,
# sub_loop(n)
tla.CONST_INT, 1,
tla.SUB,
tla.DUP,
tla.CONST_INT, 1,
tla.LT,
tla.JUMP_IF, 27,
tla.JUMP, 16,
tla.RET, 1
```

Listing 4. The definition of `callabit`.