A Lightweight Method for Generating Multi-Tier JIT Compilation Virtual Machine in a Meta-Tracing Compiler Framework (Extended Version)

Yusuke Izawa
Tokyo Metropolitan University, Japan

Hidehiko Masuhara
Institute of Science Tokyo, Japan

Carl Friedrich Bolz-Tereick
Heinrich-Heine-Universität Düsseldorf, Germany

Abstract

Meta-compiler frameworks, such as RPython and Graal/Truffle, generate high-performance virtual machines (VMs) from interpreter definitions. Although they generate VMs with high-quality just-in-time (JIT) compilers, they still lack an important feature that dedicated VMs (i.e., VMs that are developed for specific languages) have, namely *multi-tier compilation*. Multi-tier compilation uses light-weight compilers at early stages and highly optimizing compilers at later stages in order to balance between compilation overheads and code quality.

We propose a novel approach to enabling multi-tier compilation in the VMs generated by a meta-compiler framework. Instead of extending the JIT compiler backend of the framework, our approach drives an existing (heavyweight) compiler backend in the framework to quickly generate unoptimized native code by merely embedding directives and compile-time operations into interpreter definitions.

As a validation of the approach, we developed 2SOM, a Simple Object Machine with a two-tier JIT compiler based on RPython. 2SOM first applies the tier-1 threaded code generator that is generated by our proposed technique, then, to the loops that exceed a threshold, applies the tier-2 tracing JIT compiler that is generated by the original RPython framework. Our performance evaluation that runs a program with a realistic workload showed that 2SOM improved, when compared against an RPython-based VM, warm-up performance by 15%, with merely a 5% reduction in peak performance.

2012 ACM Subject Classification Software and its engineering → Interpreters; Software and its engineering → Just-in-time compilers

Keywords and phrases virtual machine, JIT compiler, multi-tier JIT compiler, meta-tracing JIT compiler, RPython

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.8

Supplementary Material *Software (Source Code)*: `https://doi.org/10.5281/zenodo.15286979`
Software (Source Code): `https://doi.org/10.5281/zenodo.15286954`
Software (Source Code): `https://doi.org/10.5281/zenodo.15287001`

Funding This research was supported by KAKENHI grant nos. 25K21180, 24K23866, 23H03368, and 23K28058, and JST ACT-X grant no. JPMJAX2003.

Acknowledgements We would like to thank Stefan Marr for his valuable advice on the measurement methodology and the evaluation results of 2SOM. We are also grateful to the anonymous reviewers for their insightful comments, which helped improve this manuscript.

39th European Conference on Object-Oriented Programming (ECOOP 2025).
Editors: Jonathan Aldrich and Alexandra Silva; Article No. 8; pp. 8:1–8:32
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

A *meta-compiler framework* [7, 38] is a system that generates high-performance virtual machines (VMs) from interpreter definitions. Traditionally, VMs were manually implemented by VM developers from scratch, including interpreters, just-in-time (JIT) compilers, and garbage collectors. However, by using a meta-compiler framework, it is possible to generate a VM of comparable quality to traditional VMs simply by defining the interpreter. Currently, there are two frameworks in this domain: RPython [6] and Graal/Truffle [39]. These frameworks have demonstrated their practical effectiveness by generating high-performance VMs such as PyPy [36], Pycket [33], TruffleSqueak [31], and TruffleRuby [38].

Given the importance of multi-tier JIT compilation VMs, efforts have been made with Graal/Truffle to enable a meta-compiler framework to perform multi-tier JIT compilation [35]. The multi-tier JIT compilation VM has multiple execution tiers using different compilation methods or optimization levels to balance code quality and compilation time [1]. In Graal/Truffle's methodology, a single JIT backend is used, and first-tier and second-tier compilations are achieved by gradually enabling or disabling optimizations. However, it has not yet achieved the incorporation of tiers with fundamentally different compilation methods, as has been done in traditional VMs.

Furthermore, as with traditional VMs, enabling a meta-compiler framework to generate a multi-tier JIT compilation VM requires a significant development effort. In traditional VMs, developing a multi-tier JIT compilation VM is more complex than simply implementing multiple compilers. At the least, a VM developer develops multiple compilers with different compilation behaviors—a compiler for rapid compilation speed but generating code with moderate quality, and a compiler for generating quality code—plus a profiler to gather runtime information for switching between different compilers, and a mechanism to transition between code generated by different compilers. Therefore, a framework developer needs to implement these components at the level of a meta-compiler framework.

To address the implementation cost, we propose a novel methodology that enables a meta-compiler framework to generate a multi-tier JIT compilation VM without requiring the development of new JIT compilation backends. Our methodology treats interpreter definitions not only as semantic specifications of the source language but also as specifications of compilation strategies of the meta-compiler. Under our methodology, a VM developer provides interpreter definitions for each compilation tier. Then, RPython can automatically generate the necessary components for multi-tier JIT compilation.

Specifically, the proposed methodology:

(Multiple compilers:) generates a lightweight compiler for rapid code generation during the warm-up phase of a source program, and a heavyweight compiler for optimizing the hot spots of a source program,

(Profiling mechanism:) generates profiling code fragments that dynamically identify hot spots and trigger tier transitions, and

(Transition mechanism:) switches between code generated by different compilers.

To validate our methodology, we implemented a two-level Simple Object Machine, which we call 2SOM for short, on top of the RPython framework. 2SOM is an extended version of PySOM [27] that is an implementation of Simple Object Machine [19] by RPython. 2SOM has two JIT compilers:

(Tier-1:) A threaded code generator, which quickly generates subroutine-threaded code for frequently invoked methods. The threaded code generator is generated by using the

RPython's mechanism based on the seminal ideas of Izawa et al. [25] along with several techniques proposed in this paper to make it practical.

(Tier-2:) A tracing JIT compiler, which is generated by RPython [6, 16, 17] and compiles traces with aggressive optimizations to frequently executed loops in the threaded code.

Our approach effectively enables multi-tier JIT compilation for meta-compiler frameworks. Although this paper demonstrates an application to SOM, the proposed approach should also be applicable to other RPython-based language implementations.

To evaluate the effectiveness of the proposed approach, we analyzed the performance of 2SOM with respect to (1) improvement of the warm-up performance for a realistic workload, (2) degradation of peak performance, and (3) quality of the threaded code generator. Our evaluation demonstrates that (1) two-level compilation runs about 15% faster than tracing JIT-only compilation, (2) the degradation is up to about 5% compared to tracing JIT-only compilation, and (3) our contribution improves the code quality of about 10% compared to interpreter execution.

This paper makes the following contributions:

- A novel approach for creating a multi-tier JIT compilation VM using the RPython framework by extending interpreter definitions.
- The optimization of meta-compiler–based threaded code generation [25], addressing practical issues and proposing solutions for integration with meta-tracing JIT compiler frameworks.
- The implementation of 2SOM, the first VM with two JIT compilers—a threaded code generator and a tracing compiler—generated entirely from interpreter definitions.
- A methodology for synthesizing large-scale application programs to evaluate the warm-up performance of the JIT-compiler-based VMs.
- The improvement of warm-up execution speed in 2SOM for synthesized application programs.

The remainder of this paper is organized as follows. Section 2 introduces the techniques underlying our proposal, including JIT and multi-tier JIT compilation, meta-compilation, meta-tracing compilation, and meta-compiler–based threaded code generation. Section 3 discusses the challenges of realizing a multi-tier JIT compilation VM within a meta-compiler framework. Section 4 presents the architecture and technical details of 2SOM. Section 5 identifies practical problems in meta-compiler-based threaded code generation and proposes solutions. Section 6 evaluates 2SOM's performance and discusses the results. Section 7 sets the results into the context of related work, and Section 8 concludes the paper.

2 Background

In this section, we demonstrate the overview and implementation challenges in a multi-tier JIT compiler. Subsequently, we present an overview of the meta-compiler framework. Finally, we introduce meta-compiler–based threaded code generation, a promising technique for lightweight compilation. Finally, we provide an overview of the Simple Object Machine that we used to implement our proposal.

2.1 JIT and Multi-Tier JIT Compilation

In general, most VMs have interpreters as their first execution tier. This is because the interpreter is easy to implement and can perform profiling tasks such as type information collection. For many languages, a VM developer can likely develop a lightweight JIT compiler with moderate effort. A lightweight JIT compiler generates code quickly, but the performance of the generated code is moderate. To achieve the desired performance, heavyweight JIT compilers are implemented. The code that they generate is highly optimized but compilation consumes markedly more time than lightweight JIT compilation.

Most traditional VMs such as Java and JavaScript VMs [32, 26, 13, 18, 34] use multi-tier JIT compilation [1]. A multi-tier JIT compilation VM involves a hierarchy of JIT compilers, each offering different optimization tiers. As the optimization tiers grow in number, compilation time also increases, and the resulting code achieves higher performance. Most multi-tier JIT compilation VMs include at least two lightweight and heavyweight compilers.

Multi-tier JIT compilation dynamically profiles the runtime information of a source program to select the most appropriate compiler for compilation and execution. Additionally, it transitions execution to higher optimization levels when beneficial, thereby enhancing the overall JIT performance.

The early execution phase, evaluated by *warm-up performance*, and the computationally intensive phase, evaluated by *peak performance*, are crucial for improving the VM's overall performance. The lightweight compiler is typically used to accelerate warm-up, whereas the heavyweight compiler focuses on maximizing peak performance. Below, we discuss the relationship between VM execution, warm-up performance, peak performance, and multi-tier JIT compilation in detail.

2.1.1 Warm-up and Peak Performance in Multi-Tier JIT Compilation

Warm-up performance refers to the efficiency during the phase required for a VM to transition from its initial execution. This phase typically involves interpretation or lightweight compilation to a state where the optimized machine code generated by the JIT compiler is actively used. This transition is critical for interactive or short-lived applications, for which execution speed during the early stages of program execution exerts a significant impact on user experience. For long-running server applications, however, warm-up performance is less critical than peak performance.

Peak performance, conversely, represents the maximum level of processing power and resource utilization that a VM can achieve under optimal conditions. This phase is usually reached after the warm-up phase, when the VM is executing fully optimized code. This is particularly important for long-running applications, such as server processes, for which sustained high performance is essential. The C2 compiler in HotSpot JVM [32] is designed to maximize peak performance by leveraging the profiling data gathered during the warm-up phase to generate highly optimized machine code. However, achieving peak performance comes with trade-offs including the overhead of profiling, the complexity of code transitions, and potential inefficiencies caused by suboptimal profiling parameters or compilation strategies.

In the case of HotSpot JVM, the C1 compiler [26] generates moderately optimized code quickly, allowing the application to execute at a reasonable speed while gathering profiling information. This profiling data is then used by the C2 compiler [32] to produce highly optimized code, improving performance over time. However, the extra overhead introduced by tiered compilation, such when as profiling and transitioning between different levels of compiled code, can undermine warm-up performance if not carefully managed.

Multi-tier JIT compilation technique ensures that frequently executed code is greatly optimized and achieves faster peak performance. On the other hand, the code parts that are not frequently executed benefit from quicker compilation, leading to the improvement of warm-up performance. Correspondingly, overall execution performance is improved by tailoring the level of optimization to the usage frequency of each part of the code.

2.2 Meta-Compilation

While interpreters are easy to implement, understand, and extend, implementing a JIT compiler is an error-prone task and requires engineering work-hours summing, at times, to dozens of years.

Meta-compilation reduces the amount of work in implementing an individual JIT compiler by making the JIT compiler reusable and language-independent. A meta-compiler compiles the source program together with the interpreter that runs it. By using both the source code and the interpreter for compilation, meta-compilers can produce highly optimized code that is equivalent to state-of-the-art JIT compilers.

In general, a meta-compiler compiles a source program together with an interpreter that executes the source program. The process involves the following steps:

1. **Hot Spot Identification**: identifying hot spots by executing the interpreter,
2. **Behavior Extraction**: extracting the selected hot spot's behavior, and
3. **Native Code Translation**: translating this behavior into native code.

The current meta-compilation system differs from the method of extracting the hot spot's behavior. The meta-tracing JIT compilation system in RPython [6] traces the interpreter as it executes the hot spot, whereas the self-optimizing interpreter in Graal/Truffle [39] applies partial evaluation to it.

Next, we introduce RPython, which we use as the basis of our proposal.

2.2.1 RPython: A Meta-Tracing Compiler Framework

While traditional JIT compilers compile a frequently executed method as their compilation unit, tracing JIT compilers [2, 16, 3] compile the execution path of a source program called *trace*. Typically, tracing JIT compilers compile a loop in a source program.

The RPython framework [36, 6] uses tracing JIT compilation for its meta-compilation technique. A meta-tracing JIT compiler doesn't trace a source program directly but instead traces the executing interpreter. The RPython framework provides annotations to indicate which part of the interpreter can be a compilation target. For instance, to find a loop in a source program, an annotation is inserted where the back edge of the loop occurs in the interpreter. The meta-tracing compiler starts to trace there when a profiling counter on a back edge exceeds a threshold. Then, tracing continues until it reaches the inserted annotation again. The resulting trace is compiled into machine code, and the subsequent executions of the compiled source program are run in the generated machine code.

To further improve performance, VM developers can use annotations to convey more information to the meta-tracing JIT compiler. For instance, VM developers can specify that certain variables in the interpreter implementation should be constant, allowing for more aggressive optimizations. In addition, by specifying a function as side-effect–free, the meta-tracing JIT compiler can replace the function call with a resulting constant if all of the passed arguments are constant. If not, when the meta-tracing JIT compiler sees the same operation on the same arguments again later, it reuses the result from the previous operation's result. In addition, if inlining a particular function defined in an interpreter would make the performance worse, the developers can avoid inlining that function by specifying an annotation to leave a call instruction on it.

2.3 Meta-Compiler–Based Threaded Code Generation

Threaded code [4, 12, 9, 10] is a form of machine code generated from a bytecode-based source program and its interpreter. Among various types of threading (which include direct threading [4] and indirect threading [9, 10], and subroutine threading [9, 10]) compiles a sequence of bytecode instructions into corresponding call instructions to their respective handlers. Threaded code generation is a lightweight compilation technique, simplifying the process and eliminating the overhead of bytecode fetching and dispatching [14].

Meta-compiler–based threaded code generation [25] is a compilation technique for lightweight compilation that generates subroutine-threaded code using a meta-tracing JIT compiler. Unlike other threaded code generators that develop a compiler from scratch, this approach leverages the interpreter definitions used in the compilation of the meta-tracing JIT compiler. Specifically, it makes use of the annotation mechanisms provided by the RPython framework. For instance, annotations are added to prevent inlining within all handler functions in the interpreter definitions, allowing call instructions to bytecode handlers to be aligned sequentially in the resulting trace. Additionally, while tracing JIT compilers trace only one side of a branch; this technique introduces mechanisms to trace all possible paths in the source program in one shot to compile the entire method. In this way, a new lightweight compilation technique is achieved without creating an entirely new compilation pipeline within the meta-tracing compiler.

The workflow for threaded code generation is illustrated in Figure 2. First, the threaded code generator traces `strange_add` by executing its bytecode, as shown in Listing 2. During this process, it records call instructions to the corresponding handler functions in the trace. When it encounters a branch instruction, such as `JUMP_IF L1`, it first traces the false branch. Upon reaching `RET`, a dummy instruction, `cut_here(return)`, is temporarily added to the trace. The technique then returns to trace the true branch, stopping at `RET` again. The resulting trace, shown in Listing 3, includes only calls to handlers, along with guards, labels, and pseudo instructions such as `cut_here`.

Next, the technique splits the trace into two parts, `Trace A` and `Trace B`, replacing the dummy instructions with actual RPython intermediate representations (e.g., `finish()`). The split traces are shown in Listing 4. Finally, it stitches `Trace B` back into `Trace A` at the guard point, `guard_false(i2)`. This stitched trace is then compiled into the final assembly code, as displayed in Listing 5.

Meta-compiler–based threaded code generation demonstrates promise as a foundation for multi-tier JIT compilation in a meta-compiler framework, because it simplifies the integration of a lightweight JIT compiler within the RPython framework. However, meta-compiler–based threaded code generation was limited to *offline* code generation, and the generated code was not optimized sufficiently. This paper, therefore, proposes optimizations to enable practical code generation (discussed in Section 5).

2.4 Simple Object Machine

Simple Object Machine (SOM) is a dynamically typed language similar to Smalltalk. It is designed as a subset of Smalltalk for the purpose of teaching and researching the implementation of a VM [19]. SOM supports class inheritance, closures, and non-local returns. The SOM family comprises many VM implementations written in C/C++, Rust, Java, Python, and Smalltalk.

Our proposed two-level SOM is based on PySOM. PySOM is a SOM VM that has AST and bytecode interpreters written in RPython. We are going to adopt the bytecode

Listing (1) Example SOM program.

```
C = (
  calc: n = (
    | x |
    # for ...
    1 to: n do: [ |i|
      x :=
        "Addition: call strange_add"
        x + strange_add: i m: n.
    ] )

  "Newly added method"
  strange_add: n m: m = (
    [ n % 42 == 0 ]
      ifTrue:  [ ^ (m - 42) ]
      ifFalse: [ ^ (n + m) ] )

  "Entry point of this program"
  run = (
    calc: 10000.
  ) )
```

Listing (2) Pseudo bytecode format corresponding to the program shown in Listing 1.

```
calc:
  DUP     # n
  CONST 1 # i
L0:
  CALL "strange_add"
  # ... Same to Figure 2 ...
  RET
strange_add:
  DUP        # n
  CONST 42
  MOD        # n % 42
  JUMP_IF L1
# false branch
  DUP        # n
  DUP        # m
  ADD        # n + m
  RET
# true branch
L1:
  CONST 42
  DUP        # n
  SUB        # n - 42
  RET
```

Figure 1 Example source program and corresponding bytecode representation in 2SOM.

Listing (3) Temporal trace from `strange_add` traced by the threaded code generation.

```
# p0 means a red var.
# in interpreter
label(strange_add)
call(DUP, p0)
call(CONST,p0,  42)
i2 = call(MOD, p0)
guard_false(i2)
# false branch
call(DUP, p0)
call(DUP, p0)
call(ADD)
cut_here(return)
# true branch
call(CONST, p0, 42)
call(DUP, p0)
call(SUB, p0)
finish(p0)
```

Listing (4) Traces after reconstructing the control flow.

```
# Trace A
label(strange_add)
call(DUP, p0)
call(CONST, p0, 42)
i2 = call(MOD)
# go to L1 if failed
guard_false(i2)  [L1]
# false branch
call(DUP, p0)
call(DUP, p0)
call(ADD, p0)
finish()

# Trace B
# true branch
label(L1)
call(CONST, p0, 42)
call(DUP, p0)
call(SUB, p0)
finish()
```

Listing (5) Pseudo-assembly code compiled from Listing 4.

```
strange_add:
  push p0
  call DUP
  push 42
  push p0
  call CONST
  push p0
  call MOD
  jnz L1
  push p0
  call DUP
  push p0
  call DUP
  push p0
  call ADD
  ret
L1:
  push 42
  push p0
  call CONST
  push p0
  call DUP
  push p0
  call SUB
  ret
```

Figure 2 Compilation steps of the meta-compiler–based threaded code generation.

interpreter to implement 2SOM.

3 Problem

While multi-tier JIT is used to improve performance in widely used VMs, implementing a multi-tier JIT compilation VM is harder than implementing a single-tier JIT compilation VM. The multi-tier option requires at least the following components of VM developers:

(Multiple compiler implementations:) Both lightweight and heavyweight compilers are required, each generating code at different optimization levels, balancing code generation time and execution efficiency.

(Profiling mechanism:) A profiling mechanism is necessary to dynamically determine which compiler to invoke. The profiler measures the execution frequency of methods or loops in an interpreter or in code generated by a lightweight compiler. Profiling data are collected to trigger higher optimization levels for frequently executed code.

(Transition mechanism from code to code generated by different compilers:) A mechanism is required to transition between code generated from different compilers, enabling seamless transitions between different compilation tiers.

Due to these complexities, it is not trivial to generate a multi-tier JIT compilation VM using a meta-compiler framework. Thus, a naïve approach may unnecessarily increase the implementation costs. The primary problem lies in the absence of a mechanism for integrating multiple compilers into such frameworks. One potential approach is to separately implement both lightweight and heavyweight compilers within the framework, coupled with a profiler to trigger transitions and manage the control flow between code from different compilers. While this approach appears feasible, it incurs significant implementation costs comparable to those already associated with traditional VMs featuring multi-tier JIT compilation.

4 Two-Level Compilation by Integrating Meta-Compiler–Based Threaded Code Generation

To address the problems outlined in Section 3, we propose a novel methodology for generating a multi-tier JIT compilation VM with minimal implementation costs. This methodology leverages the RPython framework to generate a multi-tier JIT compiler VM directly from interpreter definitions, reducing the need to implement several necessary components for multi-tier JIT compilation from scratch. As a proof of concept, we generate 2SOM, a two-level multi-tier JIT compiler VM for Simple Object Machine [19], using RPython. We will begin by showing the architecture and behavior of 2SOM in Section 4.1 and then describe how we realize it in Section 4.2.

4.1 Architecture and Behavior of 2SOM

The architecture of 2SOM, illustrated in Figure 3, uses a single meta-tracing JIT compiler for multi-tier JIT compilation to simplify the implementation of multiple compilers. In this architecture, the interpreter controls whether lightweight or heavyweight compilation is applied. The interpreter for lightweight compilation, which is called the *lightweight interpreter* here, generates threaded code [25] in collaboration with the meta-tracing JIT compiler to improve warm-up performance. Conversely, the interpreter for heavyweight compilation (the *heavyweight interpreter*) generates trace-based, highly optimized code

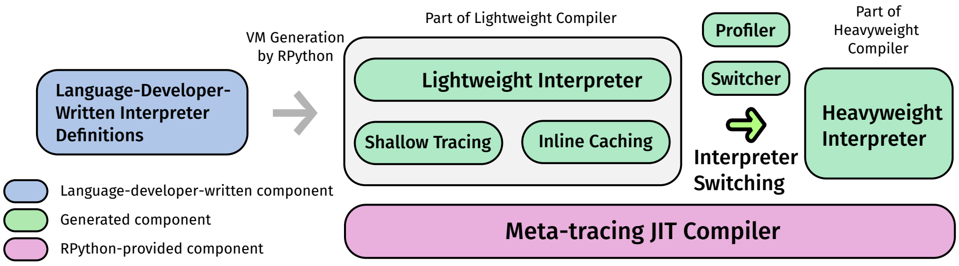


Figure 3 Architecture of 2SOM.

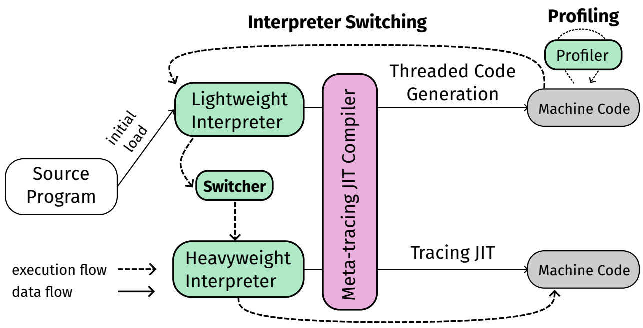


Figure 4 Execution flow of 2SOM.

Listing (6) Obtained trace from the lightweight compilation tier.

```
# Trace A
label(strange_add)
call(DUP)
call(CONST, 42)
i2 = call(MOD)
# go to L1 if failed
guard_false(i2) [L1]
# false branch
call(DUP)
call(DUP)
call(ADD)
finish()

# Trace B
# true branch
label(L1)
call(CONST, 42)
call(DUP)
call(SUB)
finish()
```

Listing (7) Obtained trace from the lightweight compilation tier.

```
# p0 means a red var. in interpreter
label(loop)
i0 = getarrayitem(p0, 0) # get n
i1 = getarrayitem(p0, 1) # get i
i2 = getarrayitem(p0, 2) # get x
# inlining strange_add
i3 = int_mod(i0, 42)     # n % 42
guard_false(i3)          # n % 42 == 0
i4 = int_add(i1, i0)     # ^(n + m)
# inlining finished
i5 = int_add(i2, i4)     # x + strange..
i6 = int_add(i1, 1)      # incr i
i6 = int_le(i6, i0)
guard_true(i6)
setarrayitem(p0, 1, i6) # set to x
setarrayitem(p0, 2, i5) # set to i
jump(loop)
```

Figure 5 Obtained traces from 2SOM when executing the program shown in Figure 1.

for peak performance. To manage runtime integration, 2SOM incorporates a profiler and switcher. The profiler identifies hot spots in code compiled by a lightweight JIT compiler, and the switcher transitions execution to heavyweight compilation when necessary, ensuring efficient optimization and seamless control flow. The profiler and switcher are implemented as interpreter definitions for the sake of reducing the need to develop complex components from scratch.

Next, we illustrate how 2SOM's multi-tier JIT compilation works, using the program shown in Figure 5 and the traces presented in Figure 5. An overview of 2SOM's execution flow is provided in Figure 4.

As depicted in Figure 4, the execution of 2SOM begins in the lightweight interpreter with the `run` function, which serves as the program's entry point. This function calls `calc:` with an argument of 10,000. The lightweight interpreter then processes the `calc:` function, starting with `initialization: x`. Subsequently, the `initialization` function is compiled into a subroutine-threaded code by the lightweight compiler. Subroutine-threaded code replaces bytecode instructions with direct calls to operation handlers, ensuring efficient execution with minimal overhead compared to heavyweight compilation. The trace obtained from this process is shown in Listing 6.

After initialization, the `calc` function enters a loop that iterates from 1 to `n`. Within this loop, the result of `strange_add: i m: n` is repeatedly added to `x`. The `strange_add: i m: n` function is initially compiled by the lightweight compiler. This loop involves frequent backward jumps, which are monitored by an embedded profiler in the code compiled by the lightweight interpreter, as depicted by Figure 4. Once the frequency of these backward jumps surpass a predefined threshold, the profiler identifies the loop as a "hot" spot suitable for more aggressive optimization.

At this stage, the interpreter switcher transfers control from the lightweight interpreter to the heavyweight interpreter. The interpreter switcher recovers the frame used by the lightweight interpreter and passes it to the heavyweight interpreter for continued execution. The heavyweight interpreter performs trace-based compilation, applying advanced techniques such as loop unrolling and inlining. The tracing JIT compiler then compiles the entire loop, inlining and optimizing the `strange_add` function. Hence, the remainder of the loop executes using highly optimized machine code generated by the heavyweight compilation tier, thereby improving efficiency while maintaining correctness. The corresponding trace is presented in Listing 7.

4.2 Technical Details

In this section, we describe the technical details of realizing 2SOM. We describe how each of the three elements essential for realizing two-level JIT compilation in 2SOM is achieved, specifically using interpreter definitions and the RPython framework.

4.2.1 Multiple Compiler Implementations

To enable multi-tier JIT compilation without the complexity of implementing multiple distinct compilers, we introduce meta-compiler–based threaded code generation [25] into RPython. Consequently, RPython can generate both generator and compiler from the two interpreter definitions with different specifications of compiler strategies. In this setup, the meta-compiler-based threaded code generation works as a lightweight compiler, while the tracing JIT compiler serves as the heavyweight compiler.

Using this technique minimizes the development and maintenance overhead associated with creating and supporting multiple compilers. Additionally, VM developers can leverage the interpreter annotations and hints provided by RPython to further customize the compilation process, tailoring optimization levels to the unique requirements of their language.

To illustrate this technique, we provide examples of implementing a heavyweight compiler, which will be followed later by a lightweight compiler example.

Listing 8 Lightweight interpreter for threaded code generation.

```
# greens: names of constant
# reds: varying variables (reds)
threadeddriver = JitDriver(
  greens=['bytecodes','pc'],
  reds=['frame'],
  threaded_code_gen=True)

def threaded_interpret(frame):
  # entry point of lightweight compilation
  threadedriver.can_enter_jit(frame=frame,
   bytecodes=frame.bytecodes, pc=frame.pc)

  while True:
    # indicate dispatch loop to
    # the meta-tracing compiler
    threadedriver.jit_merge_point(
      frame=frame,
      bytecodes=frame.bytecodes,
      pc=frame.pc)

    opcode = frame.bytecodes[frame.pc]
    if opcode == ADD:
      handler_add(frame)
    elif opcode == SUB:
      handler_sub(frame)
    # ... other handlers ...
    frame.pc += 1

@enable_threaded_code
def handler_add(frame):
  w_y = frame.pop()
  w_x = frame.pop()
  frame.push(w_x.add(w_y))

# ... other handler functions ...
```

Listing 9 Heavyweight interpreter for tracing compilation.

```
tracingdriver = JitDriver(
  greens=['bytecodes','pc'], reds=['frame'])

def tracing_interpret(frame):
  while True:
    # indicate dispatch loop to
    # meta-tracing compiler
    tracingdriver.jit_merge_point(
      frame=frame,
      bytecodes=frame.bytecodes,
      pc=frame.pc)
    opcode = frame.bytecodes[frame.pc]

    # handle back-ward jump
    if opcode == JUMP_BACKWARD:
      # entry point of heavyweight
      # compilation
      tracingdriver.can_enter_jit(
        frame=frame,
        bytecodes=frame.bytecodes,
        pc=frame.pc)

    elif bytecode == ADD:
      handler_add(frame)
    elif bytecode == SUB:
      handler_sub(frame)
    # ... other handlers ...
    frame.pc += 1

def handler_add(frame):
  w_y = frame.pop()
  w_x = frame.pop()
  frame.push(w_x.add(w_y))

# ... other handler functions ...
```

For the heavyweight compiler, we use the original interpreter definition originally provided by the RPython framework. In this interpreter, a `jitdriver` is created to configure the constant variables (greens) and varying variables (reds) in the dispatching loop. Also, hints are used to guide the JIT compiler in optimizing frequently executed code paths. The hint `jit_merge_point` tells the meta-tracing JIT compiler the part of interpreter-dispatching loop, and the `can_enter_jit` informs an entry point of the JIT compilation to the meta-tracing JIT compiler.

The heavyweight interpreter is defined as displayed by Listing 9. The interpreter guides the meta-tracing JIT compiler to perform heavyweight compilation. To indicate the place of the dispatching loop, the `jit_merge_point` hint is placed just after `while True: ...` To compile the loop of a source program, `can_enter_jit` hint is placed at the handler for `JUMP_BACKWARD`, which corresponds to the back-edge jump in a source program.

For the lightweight compiler, we define an interpreter that enables the meta-tracing JIT compiler to perform meta-compiler-based threaded code generation. The major differences from the heavyweight interpreter are that all handlers are annotated with `enable_threaded_code` and the placement of `can_enter_jit`. The `enable_threaded_code` annotation serves as an instruction to retain calls to handlers in the trace. Moreover, to compile the method body, `can_enter_jit` is placed at the very beginning of the method call–that is, just before the

Listing 10 Embedded profiler in the interpreter

```
HOT_THRESHOLD = 1000

def threaded_interpret(frame):
  while True:
    opcode = frame.bytecodes[frame.pc]

    # backward jump should be
    # compiled by heavyweight compiler
    if opcode == JUMP_BACKWARD:
      # increment the counting value
      count = frame.counts.get(frame.pc,0)+1
      frame.counts[frame.pc] = count

      if count == HOT_THRESHOLD:
        # Trigger heavyweight compilation
        raise ContinueInTier2(frame)

    # ... Execute other opcodes ...
    frame.pc += 1
```

Listing 11 Interpreter switcher that dispatches from lightweight to heavyweight interpreters

```
def interpet_switcher(bytecode):
  frame = Frame(bytecode)
  while True:
    try:
      # Start interpretation first
      # in lightweight interpreter
      w_x = threaded_interpret(frame)
    except ContinueInTier2 as e:
      # Catch thrown ContinueTier2
      # from a generated threaded code,
      # and Continue to execute in
      # heavyweight interpreter
      w_x = tracing_interpret(e.frame)
    return w_x
```

interpreter is invoked and enters the dispatching loop.

The lightweight interpreter is defined as shown in Listing 8. The handlers `handler_add` and `handler_sub` are decorated with the annotation `enable_threaded_code`, and the placement of `can_enter_jit` is moved to just before the entry of the dispatching loop.

4.2.2 Profiling Mechanism

Integrating different compilers requires a mechanism to decide when to trigger heavyweight compilation during executing code generated by a lightweight JIT compiler. We achieve this by generating code fragments that can profile runtime information in the code compiled by a lightweight JIT compiler. These generated profilers are responsible for monitoring the execution frequency of code spots and triggering recompilation at higher optimization levels when necessary.

The profiler works using a simple frequency counter that increments each time a particular code spot, such as a method or loop, is executed. When the counter exceeds a predefined threshold, the profiler determines that the code spot is "hot" and would benefit from additional optimization. At this point, the profiler triggers the heavyweight tracing JIT compilation for that spot.

Under our methodology, this profiling mechanism provides a flexible method to manage multiple compilers generated by RPython. VM developers can adjust the thresholds and profiling strategies to suit the dynamic behavior of their language in their interpreter definitions.

The example definition of the profiler is depicted in Listing 10. This profiler monitors the execution of specific opcodes. In this definition, backward jumps—typically indicating loops—are identified as key candidates for heavyweight compilation. When a backward jump opcode (`JUMP_BACKWARD`) is encountered, the profiler increases a counter tied to the program counter (`frame.pc`) stored in the `frame.counts` dictionary, which tracks execution frequencies for each opcode.

If this counter exceeds the predefined `HOT_THRESHOLD` (set to 1000 in this case), the profiler marks the code spot as "hot" and suitable for further optimization. At this point, the profiler triggers heavyweight compilation by using a global jump. Because RPython does not provide

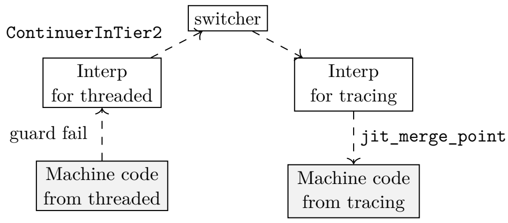


Figure 6 Overview of interpreter switching technique.

global jumping, we use an exception mechanism instead. This exception, called `ContinueTier2`, halts the lightweight interpreter and switches control to the heavyweight interpreter. To make the transition work correctly, the exception contains an execution context. The exception is used in the transition mechanism described in Section 4.2.3.

4.2.3 Transition Mechanism Between Generated Code

Finally, we need a mechanism to effect transitions from code generated by a lightweight JIT compiler to that generated by a heavyweight JIT compiler. This is achieved through the *interpreter switching* technique, which redirects the execution in code compiled by the lightweight compiler to the code compiled by the heavyweight compiler.

We overview the technique in Figure 6. When the profiler, the function of which is explained in Section 4.2.2, detects a hot spot suitable for tracing JIT compilation, a guard fail occurs. Then, a global jump from code generated by a lightweight JIT compiler to the lightweight interpreter happens. Next, the lightweight interpreter throws an exception to move to the interpreter switcher. The switcher catches it and resumes the execution context. Next, it launches the heavyweight interpreter with the resumed execution context. If the running program is already compiled by the heavyweight compiler, the control goes to the compiled machine code via `jit_merge_point`. Otherwise, the program is interpreted and will be compiled via `can_enter_jit` in the heavyweight interpreter.

The interpreter switcher is implemented as described in Listing 11. The `interpret_switcher` function handles this transition. It starts with the lightweight interpreter and, upon catching a `ContinueTier2` exception thrown from `threaded_interpret`, passes the execution context (`e.frame`) to the heavyweight interpreter `tracing_interpret`. Both interpreters return a result, `w_x`, to the caller, ensuring consistent execution flow.

Together with the profiling and interpreter switching technique explained in sections 4.2.2 and 4.2.3, we can maintain correctness while optimizing performance. By leveraging the interpreter for managing transitions, the framework provides a basis for multi-tier JIT compilation.

However, this mechanism can introduce significant runtime overhead because the control bypasses the interpreter when it goes to other code. We evaluate the overhead by measuring the peak performance of 2SOM's two-level compilation in section 6.

5 Improving Meta-Compiler–Based Threaded Code Generation

In 2SOM, we use meta-compiler–based threaded code generation [25] for lightweight compilation. However, its integration with a meta-tracing JIT compiler in 2SOM has exposed problems, as the earlier implementation [25] focused solely on code generation. Therefore, we analyzed the technical problems that arise when introducing meta-compiler–based threaded

Listing 12 Example of SOM program that has a side effect.

```
"Compute the sum of arr"
strange_sum_arr: arr index: i
  sum: n = (
  [ i <= arr length ]
    ifTrue: [
     sum_array: arr
      index: (i + 1)
      sum: (n + (arr at: i)).
    ]
    ifFalse: [
     "Clear all the elements"
     arr clear. ^ n  ] )

"Entry point"
run = (
  | arr |
  "Create an array where"
  "all elements are 1"
  arr :=
    Array new: 30 putAll: 1.
  strage_sum_arr: arr
    index: 1 sum: 0. )
```

Listing 13 Simplified bytecode from Listing 12.

```
strange_sum_arr:
 DUP0 # arr
      # arr length
 CALL(length, frame)
 DUP2 # i
 LE   # i <= arr length
 JUMP_IF_FALSE L1
 DUP1 # i
 CONST 1
 ADD  # i + 1
 DUP1 # arr
 CALL(at, frame)
      # arr at: i
 DUP2 # n
 ADD  # n + (arr at: i)
 JUMP strange_sum_arr
L1:
 DUP0 # arr
 CALL(clear, frame)
 DUP2 # n
 RET
```

Listing 14 Handler function that interprets `CALL` instruction.

```
def threaded_interpret(frame):
  ...
  while True:
    ...
    if opcode == CALL:
      handler_CALL(frame)
```

```
    # ... XXX ...

@dont_look_inside
def handler_CALL(frame):
  method = frame.bytecodes[frame.pc+1]
  newframe = crate_frame(frame, method)
  w_x = threaded_interpret(newframe)
  frame.push(w_x)
```

code generation to the meta-tracing compiler framework and introduced a new technique and optimization to enhance its efficiency and compatibility within the meta-tracing JIT compiler.

5.1 Runtime Problem of the Meta-Compiler-Based Threaded Code Generation

Meta-compiler–based threaded code generation exhibits unsatisfactory runtime execution and performance. In particular, the following technical problems occur when we use it in 2SOM.

1. If the definition of the lightweight interpreter has side effects, tracing the lightweight interpreter leads to an incorrect state due to tracing both conditional branches in a row.
2. Function calls always become indirect in the compiled code.

The first problem occurs because meta-compiler–based threaded code generation and tracing JIT compilation use different compilation scopes (method-based for the threaded code generation and trace-based for tracing compilation) but rely on the same compilation engine. For example, when performing threaded code generation, as described in the previous section, branches in the code are linearized into a single sequence during tracing. This means that even branches that would not be executed during actual program runs are included in the trace. If these unnecessary branches are executed during tracing, it can lead to an invalid runtime state.

Consider a simple `if-then-else` statement where the `then` branch is always taken during execution. During threaded code generation, both branches are traced at compilation time as described in Section 2. Although the meta-tracing JIT compiler executes the source program with the interpreter at compilation, tracing the `else` branch can modify variables or states, causing inconsistencies. For example, the program in Listing 12[1] computes the sum of an array: the `then` clause (receiver of `ifTrue`) adds each value to `n`, while the `else` clause (receiver of `ifFalse`) returns `n`. However, executing the `clear` function in the `else` clause, which sets the array to zeros, corrupts the runtime state if na"ively traced by the meta-compiler–based threaded code generation.

The second problem arises because the meta-tracing JIT compiler could not identify the target of a handler function wrapped in a call instruction during tracing, when the handler represented a method invocation. In general, calling a dynamically bound method takes longer time than calling a statically bound method. This is because a system needs to look up a correct method by using the runtime class of a callee method. The same kind of problem occurs in threaded code generation.

For example, consider a handler function `handler_CALL` that dynamically dispatches a method call as illustrated in Listing 14. During tracing, the meta-tracing JIT compiler encounters the call instruction for `handler_CALL` but cannot identify the callee method compiled at tracing. Without knowing whether the target method is compiled, the compiler cannot optimize the method call, which leads to worse performance.

In meta-compiler–based threaded code generation, this could result in repeated interpretation or inefficient indirect calls, even if the invoked method is a commonly used one that would benefit from inlining or specialized optimizations. This lack of optimization is particularly problematic for workloads with frequent method invocations, as it leads to missed performance improvements.

5.2 Our Solutions

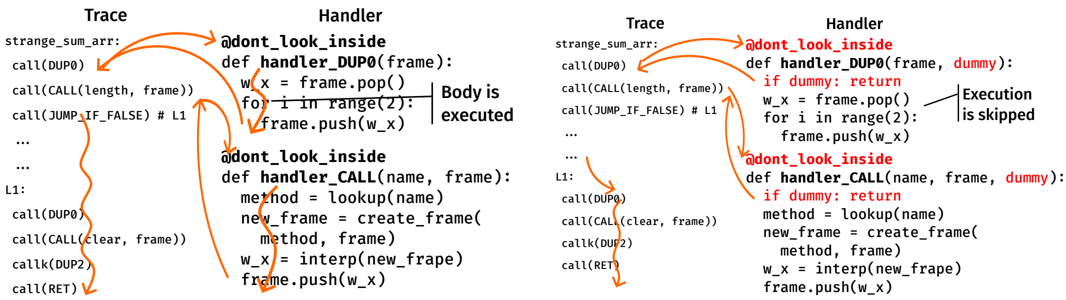
In this section, we show the solutions, namely, shallow tracing and direct calls by inline caching to the first and second problems.

5.2.1 Shallow Tracing

For the first problem, we propose the *shallow tracing* technique. Shallow tracing prevents side effects during tracing by introducing lightweight annotations within the interpreter. These annotations guide the tracing process to leave only call instructions to handler functions, while the actual execution of these handlers is avoided. In contrast, during the interpretation and execution of generated machine code, the annotations have no effect, ensuring seamless runtime execution.

The fundamental reason for needing shallow tracing lies in the fact that, while the `dont_look_inside` annotation retains the call instruction to the handler function, it still executes the function's body, using the interpreter's state. This is not a problem in standard tracing compilation. However, in method-based threaded code generation, both branches of a conditional are traced. If side effects occur during the tracing of one branch, they propagate to the tracing of the other branch, corrupting the interpreter's state.

[1] Its simplified bytecode appears in Listing 13.



(a) Flow of tracing before applying shallow tracing. The left-hand side and right-hand side show the trace and the handler functions, respectively.

(b) Flow of tracing after applying shallow tracing. The left-hand side and right-hand side show the trace and the handler functions, respectively.

Figure 7 The problem with naïvely integrating meta-compiler-based threaded code generation with a meta-tracing JIT compiler (left-hand side), and our solution to it (right-hand-side). In each side, orange lines are the flow of tracing. In naïve integration, the body of the handler function is executed during tracing, which can lead to side effects that corrupt the interpreter's state. To resolve this, we introduce shallow tracing, a technique that prevents the execution of the handler function's body during tracing, thereby avoiding such side effects.

Shallow tracing suppresses the execution of the body that would otherwise occur under `dont_look_inside`. This technique relies on the following two key annotations and the dummy flag mechanism:

- `we_are_jitted`: Checks whether the execution is currently within a traced and compiled context, and
- `dont_look_inside`: Ensures that only call instructions to annotated functions are traced without inlining their bodies of annotated functions, and
- `dummy` flag mechanism: the dummy flag is inserted into every handler. It is turned into `True` inside the then branch conditioned with `we_are_jitted`. When this flag is on, the dummy return is triggered. When this trace is compiled and executed, the dummy flag is turned into `False`, allowing the handler's body to execute correctly.

Its overall execution flows are compared in Figure 7: the left-hand side illustrates the trace without shallow tracing, while the right-hand side shows the execution flow with shallow tracing applied. Shallow tracing introduces a *dummy flag* and a *dummy return* for each handler (except for control-flow–related handlers like `JUMP_IF_FALSE`). Also, the implementation in an interpreter and the resulting trace are shown in Listings 15 and 16, respectively.

5.2.1.1 Optimizing Shallow Tracing

Manually implementing the code in Listing 15 is complex and time-consuming for VM developers. Na"ive insertion of a dummy flag into handlers also risks performance overhead. To address this, we automate the interpreter generation at VM generation time, while optimizing shallow-tracing code by eliminating redundant `dummy` flag checks.

We introduce a new annotation, `enable_threaded_code`, to automate this process. Annotated handlers are automatically marked with `dont_look_inside` and a dummy flag, and their call sites are wrapped with `we_are_jitted`, leveraging the RPython VM generation process [36]. Developers only need to annotate each handler function, as shown in Listing 17, and the annotated interpreter is transformed into the structure in Listing 18.

Listing 15 Naïve version of an interpreter that supports shallow tracing.

```
def threaded_interpret(frame):
  while True:
    opcode = frame.bytecodes[frame.pc]
    if opcode == ADD:
      # returns true while tracing and
      # executing machine code
      if we_are_jitted():
        # set 'dummy' True
        # to avoid side-effects
        handler_add(frame, dummy=True)
      else:
        # set 'dummy' False
        # execute body
        handler_add(frame, dummy=False)

    elif opcode == SUB:
      if we_are_jitted():
        handler_sub(frame, dummy=True)
      else:
        handler_sub(frame, dummy=False)
    # ... other handlers ...
    frame.pc += 1

# leaves only handler call but
# executes the body
@dont_look_inside
def handler_add(frame, dummy):
  # immediately return when dummy
  # is true to avoid side-effects
  if dummy: return
  w_y, w_x = frame.pop(), frame.pop()
  frame.push(w_x.add(w_y))

@dont_look_inside
def handler_sub(frame, dummy):
  if dummy: return
  # ... subtract two elements in
  # frame.stack ...
```

Listing 16 Naïve version of a generated trace by shallow tracing.

```
label(strange_sum_arr)
call(handler_DUP, p0, True)
call(handler_CALL, "length", p0, True)
call(handler_DUP2, p0, True)
i1 = call(handler_LE, p0, True)
# i1 is initialized with a default value (0)
guard_false(i1)
call(handler_DUP1, p0, True)
call(handler_CONST, p0, 1, True)
call(handler_ADD, p0, True)
call(handler_ADD, p0, "at", True)
call(handler_DUP2, p0, True)
call(handler_ADD, p0, True)
jump(strange_sum_arr)

label(L1)
call(handler_DUP0, p0, True)
call(handler_CALL, "clear", p0, True)
finish()
```

To mitigate the dummy flag overhead, we apply a two-step optimization. First, for each annotated handler, we generate both a stub handler (with a dummy check) and a real handler (without the check) under the same name. Second, during shallow tracing, only stub handlers are traced to build the instruction-to-handler mapping; after tracing, they are replaced with real handlers at compile time. Listings 19 and 20 show the traces before and after this optimization.

5.2.2 Direct Calls with Inline Caching

Inline caching [11] is a well-known runtime optimization technique used to accelerate method dispatch in object-oriented programming languages such as Smalltalk-80, SELF, and Java. It optimizes method calls by caching method lookups directly at the call site, avoiding redundant type resolution when the same types are repeatedly encountered. If the cached type matches the runtime type of the receiver, a fast path is taken; otherwise, the system falls back to a slow path for dynamic method resolution [20].

In the absence of inline caching in meta-compiler–based threaded code generation, method

Listing 17 Annotated version of the interpreter design for shallow tracing.

```
def threaded_interpret(frame):
  while True:
    opcode = frame.bytecodes[frame.pc]
    if opcode == ADD:
      handler_add(frame)
    elif opcode == SUB:
      handler_sub(frame)
    frame.pc += 1

@enable_threaded_code
def handler_ADD(frame):
  w_y = frame.pop()
  w_x = frame.pop()
  frame.push(w_x.add(w_y))

@enable_threaded_code
def handler_SUB(frame):
  # subtract two elements in frame.stack
```

Listing 18 Generated stub and original handlers.

```
def threaded_interpret(frame):
  while True:
    opcode = frame.bytecodes[frame.pc]
    if opcode == ADD:
      if we_are_jitted():
        stub_ADD(frame, True)
      else:
        ADD(frame)
    elif opcode == SUB:
      if we_are_jitted():
        stub_SUB(frame, True)
      else:
        SUB(frame)
    frame.pc += 1

@dont_look_inside
def stub_ADD(frame, dummy):
  if dummy: return
  ADD(frame)

@dont_look_inside
def stub_SUB(frame, dummy):
  if dummy: return
  SUB(frame)

def ADD(frame):
  # ... add two elements ...

def SUB(frame):
  # ... subtract two elements ...
```

Listing 19 Before applying the handler optimization in shallow tracing.

```
label(strange_sum_arr)
call(stub_DUP, p0, True)
call(stub_CALL, "length", p0, True)
call(stub_DUP2, p0, True)
i1 = call(stub_LE, p0, True)
# i1 is initialized with a default value (0)
guard_false(i1)
call(stub_DUP1, p0, True)
call(stub_CONST, p0, 1, True)
call(stub_ADD, p0, True)
call(stub_CALL, p0, "at", True)
call(stub_DUP2, p0, True)
call(stub_ADD, p0, True)
jump(strange_sum_arr)

label(L1)
call(handler_DUP0, p0, True)
call(handler_CALL, "clear", p0, True)
finish()
```

Listing 20 After applying the handler optimization in shallow tracing.

```
label(strange_sum_arr)
call(DUP, p0)
call(CALL, "length", p0)
call(DUP2, p0)
i1 = call(LE, p0)
# i1 is initialized with a default value (0)
guard_false(i1)
call(DUP1, p0)
call(CONST, p0, 1)
call(ADD, p0)
call(CALL, p0, "at")
call(DUP2, p0)
call(ADD, p0)
jump(strange_sum_arr)

label(L1)
call(handler_DUP0, p0)
call(handler_CALL, "clear", p0)
finish()
```

calls to compiled code are executed as indirect calls. Such calls bypass `jit_merge_point`, which can lead to significant performance degradation when methods are invoked repeatedly. Direct calls with inline caching alleviate this by converting indirect calls into direct calls, thereby improving execution speed. Figure 8 provides an overview of this transformation at

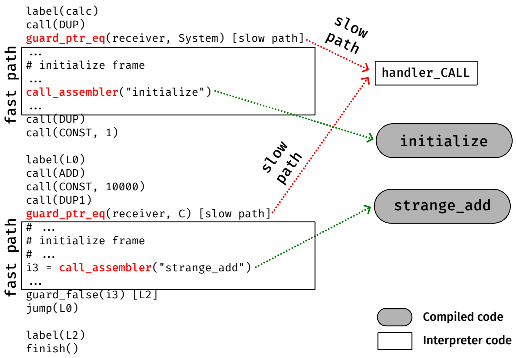


Figure 8 Overview of direct calls with inline caching.

Listing 21 Interpreter definition instrumented for inline caching

```
def threaded_interpret(frame):
  while True:
    jit_merge_point(..)
    ...
    if opcode == CALL:
      # check whether the runtime type of
      # this method equals to the
      # recorded runtime type
      method = frame.stack[frame.pc]
      if ‘check_type‘(method, frame.pc):
        # if it matches, calling the
        # method with call_assembler
        ‘call_assembler‘(interp(frame))
      else:
        # otherwise, call handler_CALL
        handler_CALL(frame)

def handler_CALL(frame):
  method = pop(stack)
  if not we_are_jitted():
     # during interpreting (not jitted),
     # record runtime method types at pc
     ‘record_type‘(pc, method.type)
  r = threaded_interpret(frame)
  frame.push(r)
```

runtime, revealing the transition from indirect to direct calls enabled by inline caching.

To integrate inline caching efficiently, we leverage the annotation mechanism of the RPython framework instead of implementing it from scratch. Inline caching is realized by instrumenting the interpreter with runtime type collection, validation, and direct call conversion. During interpretation, runtime type information—such as the receiver's type and the associated method—is collected. This collected type information is then validated at runtime, enabling the conversion of indirect calls into direct calls.

At runtime, our inline caching implementation works as follows:

- Fast Path: If the runtime type matches the cached type, the system directly calls the compiled method using the `call_assembler` instruction.
- Slow Path: If the type validation fails, the system falls back to the `handler_CALL` function, which dynamically resolves the method and records type information for future optimizations.

The implementation uses the RPython annotation mechanism. Two annotations were originally developed:

- `record_type`: records the runtime type of a method during interpretation.
- `check_type`: validates the runtime type against the cached type.
- `call_assembler`: performs the direct call to the compiled method.

Listing 21 illustrates how these annotations are used in the interpreter:

Based on the implementation, the compilation with direct calls with inline caching works as follows:

1. During interpretation, the `handler_CALL` function records the runtime type of the receiver using the `record_type` annotation.

2. During tracing, the `CALL` handler, `check_type` validates the runtime type of the receiver. If the validation succeeds, the `call_assembler` instruction is invoked to perform a direct call to the compiled method.
3. During compilation, `check_type` is transformed into a guard instruction that ensures the receiver type matches the recorded type. Similarly, `call_assembler` is converted directly into an assembler call instruction in the resulting trace.

6 Evaluation

In this section, we address the three questions introduced at the beginning of this paper by evaluating the performance of the two-level compilation implemented in 2SOM. We begin by describing the benchmarking programs and the execution environment used in the evaluation. Each research question is then addressed by analyzing both the warmup performance and peak performance of two-level compilation and meta-compiler–based threaded code generation using 2SOM. Since the measurement methods differ for each case, they are described in separate sections of their own.

6.1 Methodology

Throughout this evaluation, we basically use the SOM benchmarks [28], an extended version of the Are We Fast Yet benchmark [30], which includes programs such as PageRank and graph search algorithms.

To answer Q1 and Q2 in particular, we should analyze the performance of two-level JIT using real-world large-scale programs. However, SOM benchmark suite does not have such programs. Thus, we synthesize a program that has workloads similar to real-world large-scale programs.

First, we examined the relationship when methods in the DaCapo benchmark [5] were sorted in descending order based on the number of method calls. Using the least squares method to analyze this relationship, we found that the R^2 value exceeded 0.98, indicating a high correlation. Therefore, we aimed to approximate the ranking correlation between DaCapo's method call counts and the method order using the SOM benchmark suite.

We generated 20 variants of *synthesized benchmark program* using the following procedure: (1) place each program SOM's benchmark suite in order, (2) manually tune the number of internal iterations for each program so that the correlation between DaCapo's method call counts and method ranks is as close as possible. (3) Finally, from the program created in steps (1) and (2), generate the remaining 19 variants by randomly shuffling the program execution order.The details are described and discussed in Appendix A.

In Q1, we measure the elapsed time obtained from the first iteration of each program and repeat this measurement 2,000 times. We calculate their medians, averages, and variances.

We evaluate the warm-up and peak performances against six execution modes:

- Interpreter-only execution: execution using 2SOM's interpreter without any form of compilation.
- Threaded code generation: meta-compiler–based threaded code generation implemented in 2SOM, offering a lightweight approach to improving execution efficiency.
- Tracing JIT: execution utilizing RPython's meta-tracing compiler.
- Tracing JIT with high threshold: A variation of the tracing compilation where the compilation threshold is increased by a factor of three. This configuration is designed to

investigate the fundamental differences between a higher-threshold tracing and two-level JIT compilations.

- Two-level JIT: A combined strategy that combines both threaded code generation and tracing compilation, leveraging the strengths of both techniques for improved execution performance.
- TrufleSOM: execution using TruffleSOM, which is another SOM built with the Truffle framework [39], to compare the performance of 2SOM with a different language implementation. Note that multi-tier JIT compilation [35] is enabled.

In Q2, we evaluate the peak performance of 2SOM. We use the synthesized programs used in Q1 in addition to the SOM microbenchmark programs. Each program is executed 2,000 times, and the average elapsed times are calculated based on the last half of the iterations to omit the effect of JIT compilation. We compare the following execution models: tracing JIT, tracing JIT with a higher threshold, and two-level JIT.

In Q3 (performance of threaded code), we assess the improvements introduced by meta-compiler–based threaded code generation by utilizing the SOM microbenchmarks. Specifically, we measure the peak performance of these benchmarks to evaluate the effectiveness of meta-compiler–based threaded code generation enhancements.

The 2SOM codebase is available at Zenodo [21]. To build 2SOM, we use GCC version 14.2.1 and our customized PyPy, which is available at Zenodo [22]. For measuring time, we use `monotonic_clock` by our RPython extension [23]. For TruffleSOM, we use revision e9d8032 of the repository hosted on GitHub[2], along with GraalVM Community Edition version 23.0.2.

All evaluations are conducted on a system running Ubuntu 22.04.1 LTS, equipped with a quad-core Intel Core i7-6700 CPU and 32 GB of RAM. To ensure reliable and consistent measurements, we use ReBench [29], a benchmarking framework designed to manage the evaluation process and minimize measurement noise.

6.2 Q1: Does Two-Level Compilation Improve Warm-Up Performance?

In this section, we answer Q1 by evaluating the warm-up performance of two-level JIT in 2SOM. The results of these measurements are presented in Figure 9, which shows the elapsed times with error bars displaying their variances.

As shown in Figure 9, two-level compilation demonstrated the best warm-up performance; it executed approximately 15% faster than tracing JIT alone. Furthermore, even when compared to tracing compilation with a threshold increased by a factor of three and TruffleSOM, two-level compilation outperformed by approximately 5%.

To statistically validate whether there were differences in the data for each pair—two-level and tracing JITs, as well as two-level JIT and tracing JIT with a high threshold—we conducted the Wilcoxon signed-rank test. The results revealed that for the two-level and tracing JITs pair, $p = 0.0314\%$, and for the two-level JIT and tracing JIT with a higher threshold pair, $p = 2.151\%$. In both cases, the null hypothesis was rejected.

These findings indicate that two-level JIT improves warm-up performance compared to RPython's single-tier compilation. Furthermore, they suggest that two-level JIT enhances warm-up performance more effectively than optimizing compilation strategies by limiting the compilation target scope, such as adjusting thresholds to reduce compilation time.

[2] `https://github.com/SOM-st/TruffleSOM`

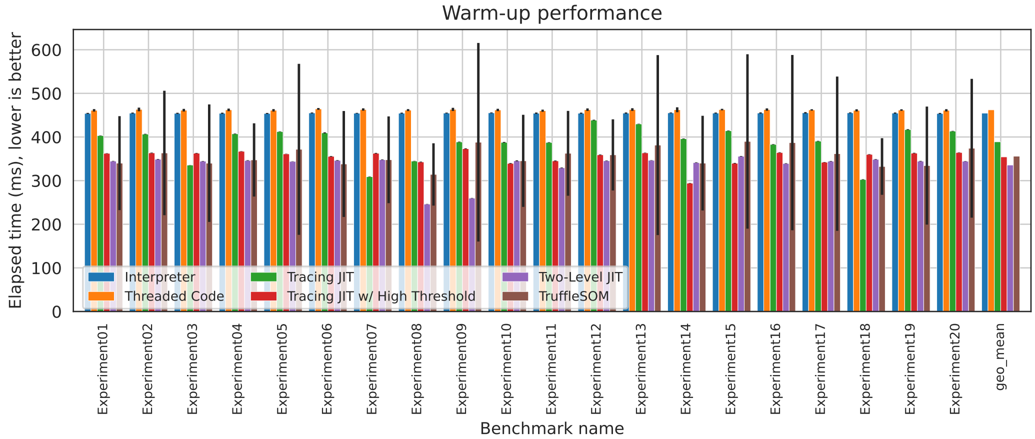


Figure 9 Warm-up performance (execution time of the first iteration)

In addition, we break down the traces generated from these strategies: two-level JIT, tracing JIT with high threshold, and tracing JIT. The results are shown in Figure 10.

The first breakdown (the left-hand side of Figure 10: number of traces) shows that two-level JIT generates fewer traces than tracing JIT but more than tracing JIT with a high threshold.

The second breakdown (the right-hand side of Figure 10: number of operations in traces) shows that two-level JIT generates fewer operations in traces than tracing JIT but slightly more than tracing JIT with a high threshold. In addition, the ratio of traces generated from threaded code overall number of traces are approximately 10%. These results stem from the characteristic of the lightweight compiler that it generates compact traces during early execution phases.

These results demonstrate the ability of the two-level JIT to balance warm-up speed and trace generation; lightweight compilation using threaded code can improve early performance. Therefore, two-level JIT effectively works interpreter execution, threaded code generation, and tracing together.

6.3 Q2: How Good is the Peak Performance of Two-Level Compilation?

In this section, we answer Q2 by describing the peak performance of two-level JIT in 2SOM. We measure the peak performance of synthesized programs and the SOM microbenchmark programs, and these results are shown in Figures 11 and 12.

As shown in Figure 11, the peak performance of two-level JIT is slower than approximately 3% against tracing JIT, and approximately 5% slower than tracing JIT with high threshold. In addition, as shown in Figure 12, the peak performance of two-level JIT is slower than approximately 7% against tracing compilation and approximately 10% slower than tracing JIT with high threshold.

This result indicates that introducing the meta-compiler–based threaded code generation to a meta-tracing JIT compiler does not significantly reduce the peak performance of meta-tracing JIT compilation. In particular, the introduction of interpreter switching does not result in a significant performance degradation in peak performance.

Figure 10 Number of operations and traces in the following strategies: two-level JIT, tracing JIT with high threshold, and tracing JIT in the evaluation of warm-up performance. The orange stitched bar inside the two-level JIT shows the number of operations and traces generated from the threaded code generation.

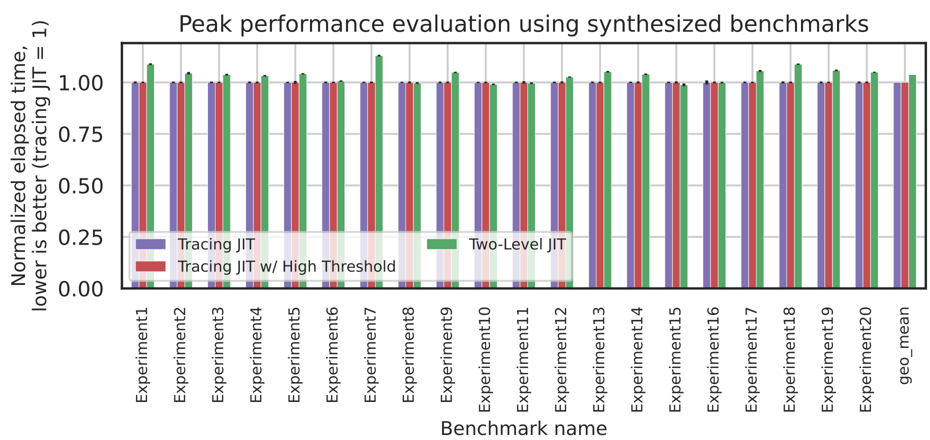


Figure 11 Peak performance results: Normalized elapsed time (lower is better) across synthesized programs for different JIT compilation strategies: (1) Tracing JIT, (2) Tracing JIT with a high threshold, and (3) Two-Level JIT. Results are normalized to the baseline Tracing JIT.

6.4 Q3: How Much is the Improvement of Threaded Code Generation?

In this section, we answer Q3 by describing the peak performance of the meta-compiler–based threaded code generation. To evaluate the impact of the optimizations implemented in threaded code generation, we measured its peak performance. Specifically, we compare three configurations: one with all optimization options enabled, one with only the direct calls with inline caching optimization disabled, and another with both direct calls with inline caching

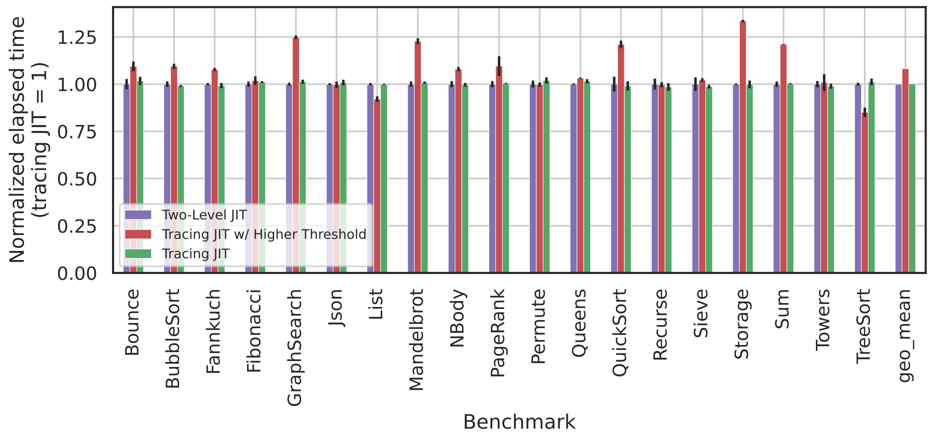


Figure 12 Peak performance results: Normalized elapsed time (lower is better) for SOM benchmarks, comparing three JIT compilation strategies: (1) Two-Level JIT, (2) Tracing JIT with a higher threshold, and (3) Tracing JIT. The results are normalized to the baseline Tracing JIT.

and shallow tracing handler optimization disabled. This comparison allowed us to assess the performance improvements contributed by each optimization individually. The result is shown in Figure 13.

As shown in Figure 13, the fully optimized configuration (Threaded Code, orange-colored bar) consistently achieves the best performance across all benchmarks, running about 10% faster than the interpreter. Disabling direct calls with inline caching (brown-colored bar) degrades performance by about 10% on average, highlighting its importance in optimizing frequent function calls and dynamic dispatch. Disabling both direct calls with inline caching and shallow tracing handler optimization (pink-colored bar) further degrades performance by about 5%, highlighting their complementary impact on efficiency. Overall, the results underscore the need for these optimizations to achieve optimal runtime performance.

This result shows that our implemented optimization improves the performance of the meta-compiler–based threaded code generation. In particular, direct calls with inline caching significantly improve the performance in function call-intensive programs such as Fibonacci, Json, Recurse, Storage, Towers, and TreeSort. However, further optimization is needed to improve the warm-up performance more. For example, we believe that inline caching implemented this time can be optimized more efficiently by using polymorphic inline caching.

6.5 Threats to Validity

Since this evaluation was conducted on a subset of Smalltalk, the results may differ when it is performed on a larger language implementation such as PyPy and RSqueak [15].

7 Related Work

There exist several approaches to building a lightweight compiler in a meta-compiler framework. Copy-and-patch compilation [40] is a meta-compilation technique that performs template-based compilation. The template is generated from a bytecode instruction or an AST node of a high-level language by the MetaVar compiler. Although they build a new meta-compiler to generate a new baseline compiler, our approach is built on an existing

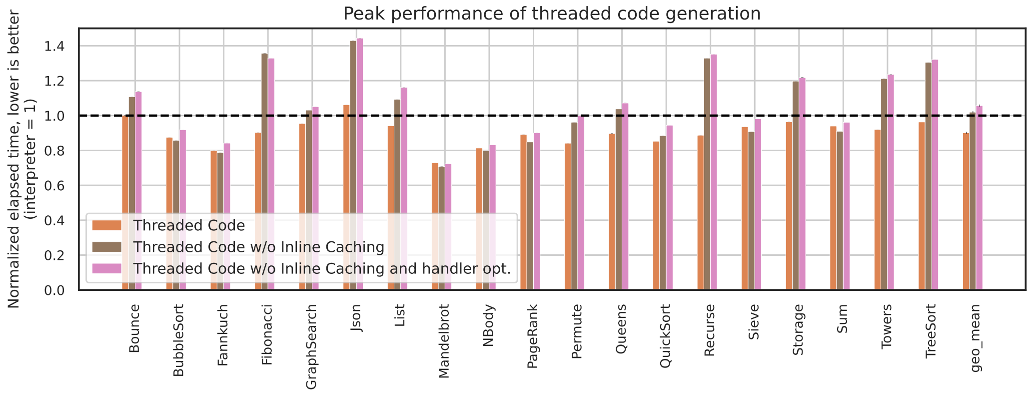


Figure 13 Comparison of peak performance for threaded code generation under different optimization configurations: (1) all optimizations enabled, (2) without inline caching, and (3) without both inline caching and shallow tracing handler optimization. Each data is normalized to interpreter-only execution.

meta-tracing compiler. In addition, although they use a parameterized binary code for generating a template, we use an interpreter definition with annotations to compile a source program.

There are also other methodologies for realizing a multi-tier JIT compilation VM using a meta-compiler framework. Graal/Truffle [38] has introduced a first-tier execution in addition to the second-tier execution that performs heavyweight compilation [35]. The approach is straightforward: they control the level of optimization in their meta-compilation system to enable multi-tier execution. While their methodology requires major modifications to the meta-compiler, our methodology is adding multiple tiers with different compilation methods in RPython.

Furthermore, other meta-circular VMs provide multi-tier JIT compilation, such as Jikes RVM [1] and Maxine VM [37]. These two VMs have lightweight and heavyweight compilers. In particular, they have original code generators for lightweight compilation: Jikes RVM's lightweight compiler translates Java bytecode into native code by simulating Java's operand stack. Maxine VM's lightweight compiler translates Java functions into native code using templates. In contrast, our lightweight compilation reuses the compiler of heavyweight compilation. To control the code quality and compilation time, we slightly modify the definition of an interpreter provided to the meta-tracing compiler.

8 Conclusion and Future Work

This paper introduces a lightweight methodology that enables a meta-compiler framework to generate a multi-tier JIT compilation VM. By treating interpreter definitions as specifications not only for language semantics but also for compilation strategies, we enable RPython to generate key components for multi-tier JIT compilation: multiple compilers, a profiler, and a transition mechanism.

To demonstrate our approach, we created a two-tier JIT compilation version of the Simple Object Machine called 2SOM. 2SOM incorporates two JIT compilers: a tier-1 threaded code generator and a tier-2 tracing JIT compiler. This integration was achieved by addressing practical challenges in combining meta-compiler-based threaded code generation [25] with

RPython.

Our evaluation shows that the proposed methodology significantly improves performance. Warm-up performance was improved by approximately 15%, while peak performance degradation was limited to approximately 5% compared to tracing JIT-only systems. Furthermore, enhancements to the threaded code generator improved execution speed by 10% compared to interpreter-based execution. These results demonstrated the potential of our methodology to simplify the development of multi-tier JIT compilers in other RPython-based language implementations.

Despite its advantages, our approach has certain limitations. First, it cannot add a tier that requires different low-level code generation and optimization mechanisms (e.g., register allocation or instruction-level scheduling) because it does not change the processes of RPython after generated traces. Second, it cannot add a tier that requires manipulating intermediate code. This limitation arises because the approach relies on running the interpreter to obtain traces of the intermediate code (RPython traces). The design does not assume modifications to these traces.

Future work is not only to solve these limitations, but also to apply our method to larger language systems than 2SOM and validate the effectiveness of multi-tier JIT compilation VMs using more realistic data.

One potential application is PyPy. Although we have conducted experiments using synthesized programs to evaluate the effectiveness of multi-tier JIT compilation VMs. However, we think that using a larger benchmark set would enable more detailed analysis. To achieve this in PyPy, two specific challenges must be addressed: a) How to reduce the effort required to annotate PyPy's interpreter, which consists of approximately 200,000 lines of code. b) How to handle cases where handlers in the PyPy interpreter may throw exceptions, given that our current tier-1 compiler does not support tracing handlers that can raise RPython-level exceptions. c) How to develop a lightweight compiler that can generate higher-quality code than threaded code generation, enabling the multi-tier JIT compilation VM to be effective for a broader range of workloads.

For challenge a), it will be necessary to allow at least two interpreters to be generated from a single interpreter. One idea is to improve PyPy's translation flow, which generates VMs, to automate the generation of interpreters.

For challenge b), we believe that adopting the *zero-cost exception mechanism* [8] introduced in Python 3.11–now being integrated into PyPy–will allow us to trace handlers in the PyPy interpreter. Unlike earlier versions of Python where try statements compiled to `SETUP_FINALLY` (incurring overhead even without exceptions), Python 3.11 uses `NOP` instructions for the non-exception path. This change should let our threaded code generation avoid triggering RPython-level exceptions and work smoothly in PyPy with this mechanism enabled.

For challenge c), we believe that a solution can be achieved through further refinement of meta-compiler–based threaded code generation. In general, interpreters implement the behavior of handler functions by calling various helper functions within those handlers. The current threaded code generator only leaves call instructions to the handler functions in the trace, but more efficient lightweight compilation could be achieved by inlining non-critical helper functions and leaving only those helper functions that are time-consuming to compile as call instructions.

References

1 Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2000, pages 47–65. Association for Computing Machinery, 2000. doi:10.1145/353171.353175.

2 Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000. doi:10.1145/349299.349303.

3 Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: A Trace-based JIT Compiler for CIL. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA 2010, pages 708–725. Association for Computing Machinery, 2010. doi:10.1145/1869459.1869517.

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

5 Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA 2006, pages 169–190. Association for Computing Machinery, 2006. doi:10.1145/1167473.1167488.

6 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 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*, ICOOOLPS 2009, page 18–25, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1565824.1565827.

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

8 CPython. Zero cost exception handling. URL: https://github.com/python/cpython/blob/main/InternalDocs/exception_handling.md.

9 Charles Curley. Life in the FastForth Lane. *Forth Dimensions*, 14(4):6–12, January/Feburary 1993.

10 Charles Curley. Optimizing in a BSR/JSR Threaded Forth. *Forth Dimensions*, 14(5):21–26, March/April 1993.

11 L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297—-302, New York, NY, USA, 1984. Association for Computing Machinery. doi:10.1145/800017.800542.

12 Robert B.K. Dewar. Indirect Threaded Code. *Communications of the ACM*, 18(6):330–331, June 1975.

13 Eclipse Foundation. Open J9: Unleash the power of java, 2017. URL: https://www.eclipse.org/openj9/.

14 M. Anton Ertl and David Gregg. The Structure and Performance of Efficient Interpreters. *Journal of Instruction-level Parallelism*, 5, November 2003.

15 Tim Felgentreff, Tobias Pape, Patrick Rein, and Robert Hirschfeld. How to Build a High-Performance VM for Squeak/Smalltalk in Your Spare Time: An Experience Report of Using the RPython Toolchain. In *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies*, IWST '16, pages 21:1–21:10. Association for Computing Machinery, 2016. RSqueak,RPython. doi:10.1145/2991041.2991062.

16 Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-Based Just-in-Time Type Specialization for Dynamic Languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2009, pages 465–478. Association for Computing Machinery, 2009. doi:10.1145/1542476.1542528.

17 Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: An Effective JIT Compiler for Resource-Constrained Devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, pages 144–153. Association for Computing Machinery, 2006. doi:10.1145/1134760.1134780.

18 Google. Google's High-performance Open Source JavaScript and WebAssembly Engine, 2015. URL: https://v8.dev/.

19 Michael Haupt, Robert Hirschfeld, Tobias Pape, Gregor Gabrysiak, Stefan Marr, Arne Bergmann, Arvid Heise, Matthias Kleine, and Robert Krahn. The som family: Virtual machines for teaching and research. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE 2010, pages 18–22. Association for Computing Machinery, 2010. doi:10.1145/1822090.1822098.

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

21 Yusuke Izawa. 2SOM: A Two-Level Simple Object Machine, April 2025. Version 1.0. doi:10.5281/zenodo.15286979.

22 Yusuke Izawa. Customized PyPy for Threaded Code Generation, April 2025. Version 1.0. doi:10.5281/zenodo.15287001.

23 Yusuke Izawa. RPython Extension that Enables a Monotonic Clock, April 2025. Version 1.0. doi:10.5281/zenodo.15286954.

24 Yusuke Izawa, Hidehiko Masuhara, and Carl Friedrich Bolz-Tereick. A Lightweight Method for Generating Multi-Tier JIT Compilation Virtual Machine in a Meta-Tracing JIT Compiler Framework (Extended Version). arXiv, 2025. doi:10.48550/arXiv.2504.17460.

25 Yusuke Izawa, Hidehiko Masuhara, Carl Friedrich Bolz-Tereick, and Youyou Cong. Threaded Code Generation with a Meta-Tracing JIT Compiler. *Journal of Object Technology*, 21(2):a1, 2022. doi:10.5381/jot.2022.21.2.a1.

26 Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™ Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1), 2008-05. doi:10.1145/1369396.1370017.

27 Stefan Marr. PySOM: A Simple Object Machime Smalltalk implemented in Python, 2013. URL: https://github.com/smarr/PySOM.

28 Stefan Marr. Som Benchmarks, 2013. URL: https://github.com/SOM-st/SOM/tree/9c04914f800dc3ccbfaa1dc8fcf78cc5714549a4/Examples/Benchmarks.

29 Stefan Marr. ReBench: Execute and Document Benchmarks Reproducibly, August 2018. Version 1.0. doi:10.5281/zenodo.1311762.

30 Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-language compiler benchmarking: Are we fast yet? In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, pages 120–131. Association for Computing Machinery, 2016. doi:10.1145/2989225.2989232.

31 Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. Graalsqueak: Toward a smalltalk-based tooling platform for polyglot programming. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2019, pages 14–26. ACM, 2019. doi:10.1145/3357390.3361024.

32 Michael Paleczny, Christopher Vick, and Cliff Click. The Java Hotspot™ Server Compiler. In *Proceedings of the 2001 Symposium on Java™ Virtual Machine Research and Technology Symposium - Volume 1*, JVM 2001, page 1. USENIX Association, 2001.

33 Tobias Pape, Tim Felgentreff, Robert Hirschfeld, Anton Gulenko, and Carl Friedrich Bolz. Language-independent storage strategies for tracing-jit-based virtual machines. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, pages 104—113, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2816707.2816716.

34 Fillip Pizlo. Speculation in JavaScriptCore, 2020. URL: https://webkit.org/blog/10308/speculation-in-javascriptcore/.

35 Aleksandar Prokopec. Multi-Tier Compilation in GraalVM, 2021. URL: https://medium.com/graalvm/multi-tier-compilation-in-graalvm-5fbc65f92402.

36 Armin Rigo and Samuele Pedroni. PyPy's Approach to Virtual Machine Construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA 2006, pages 944–953. Association for Computing Machinery, 2006. doi:10.1145/1176617.1176753.

37 Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013. doi:10.1145/2400682.2400689.

38 Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 662–676. Association for Computing Machinery, 2017. doi:10.1145/3062341.3062381.

39 Thomas Würthinger, Andreas Wöundefined, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS 2012, pages 73–82. Association for Computing Machinery, 2012. doi:10.1145/2384577.2384587.

40 Haoran Xu and Fredrik Kjolstad. Copy-and-Patch Compilation: A Fast Compilation Algorithm for High-Level Languages and Bytecode. *Proc. ACM Program. Lang.*, 5(OOPSLA), 2021. doi:10.1145/3485513.

A Construction and Validation of Synthesized Benchmark Programs

In this section, we describe the methodology to construct synthesized benchmark programs used in Section 6 and discuss the characteristic of them through JIT indicators through PyPy benchmark programs.

A.1 Construction of a Large-Workload Program for Evaluating Two-Level Compilation

The challenge in evaluating two-level compilation in 2SOM is how to test our methodology on a large-scale application. However, the code base of programs that 2SOM can run is smaller than that of Java or JavaScript VM, and there is no predefined benchmark set equivalent or similar to DaCapo [5] or Renaissance [**?**]. Therefore, we compose a program that replicates a large application with non-trivial workloads.

Given this situation, our options are (1) implementing a large-scale benchmark set comparable to DaCapo or Renaissance in the SOM language or (2) reproducing the workload of a large-scale application using SOM's existing benchmark set. While (1) would be the preferable approach, the SOM language lacks certain language features, which are not used for SOM's existing benchmark set but are needed for implementing a large-scale benchmark set, such as network communication and asynchronous processing, necessitating extensions to the language itself. In addition, implementing a benchmark set of this scale would require significant effort. For these reasons, we opt for approach (2).

To replicate the workload of a large-scale application, we investigated the correlation between the number of method invocations and their rank. The rank refers to the position in a descending order of method invocation counts, where rank 1 indicates the most frequently invoked method, and the lowest rank corresponds to the method that was called only once. We measured the number of method invocations using the HPROF profiling tool[3] with interpreter execution for the DaCapo benchmark. Benchmarks with more than 400 method invocations were selected and plotted on logarithmic x- and y-axes in Figure 14. The y-axis represents the number of method calls, while the x-axis represents the rank of methods sorted in descending order based on their number of calls. As shown in Figure 14, regression analysis revealed a high correlation with an $R^2 > 0.98$. Therefore, we can conclude that the program in this experiment should be structured such that the correlation between method invocations and rank is high.

Based on the investigation, we design the experiment by combining all benchmark programs used in Section 6. The programs that comprise the experimental setup are termed subprograms. These subprograms are executed continuously in a single process. The number of internal iterations assigned to the subprograms is adjusted so that the R^2 value for the linear regression on the logarithmic scale of the number of method invocations and rank is as close to 0.98 as possible. Figure 15 illustrates the correlation in the same manner as Figure 14.

The experimental program consists of 20 subprograms. These subprograms are executed sequentially from top to bottom, with each subprogram assigned a predetermined number of iterations within the experimental program. To generate the set of 20 experiment programs, the execution order of the subprograms is randomly shuffled. Specifically, a base subprogram order is first determined. Then, by shuffling the execution order while keeping the workload unchanged, an alternative subprogram sequence is created. This process is repeated 19 more times to generate the remaining 19 experimental programs.

A.2 Validation of Our Synthesized Benchmark with PyPy's Real-World Benchmark Program

We validate the characteristics of our synthesized benchmark through PyPy's benchmark programs, which are hosted at Heptapod[4].

First, we measure the correlation between the number of method invocations and the rank of them as we measure the Dacapo benchmark. We selected the top eight PyPy benchmarks with the largest number of executed methods and conducted measurements on them.

The results are shown in Figure 16. The x-axis represents the method rank, and the y-axis represents the number of method invocations; both axes use a logarithmic scale. The black dots indicate the measured values, while the blue lines show the results of linear regression. The R^2 value for each plot is displayed in the top right corner. These results suggest that the benchmarks in the PyPy suite with a large number of method invocations exhibit a distribution of method call frequencies that closely resembles that of the Dacapo benchmarks.

Next, we compare our synthesized benchmark with the PyPy benchmark programs and discuss the characteristics of the synthesized workloads. Specifically, we use profiling results from the tracing JIT compiler of PyPy/RPython for comparison. For the PyPy benchmarks, we use PyPy's tracing JIT compiler, and for 2SOM, we consider only the tracing JIT compiler.

[3] `https://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html`
[4] `https://foss.heptapod.net/pypy/benchmarks`

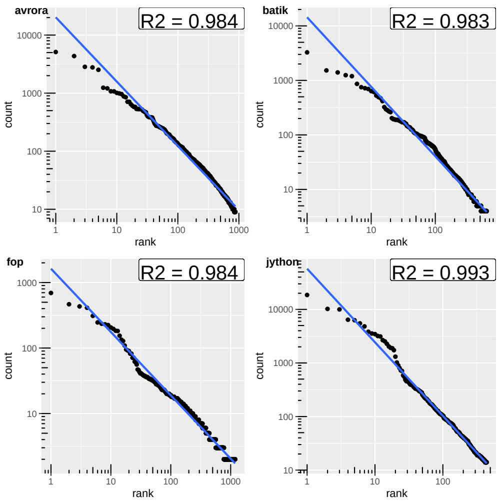


Figure 14 Correlation between the number of method invocations and rank for avrora, batik, fop, and jython in the DaCapo benchmark. x- and y-axes are logarithmic. The blue line is fitted by least-squares regression.

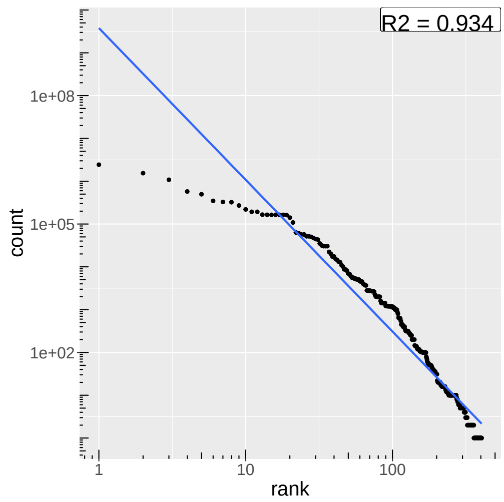


Figure 15 Correlation between the number of method invocations and rank for the program constructed for the experiment of multi-tier JIT compilation.

Using these setups, we run both the PyPy and synthesized benchmarks to measure the average trace length (in terms of IR-level instructions generated by the tracing JIT compiler) and examine its correlation with the number of invoked methods.

While Figure 16 focused on the top eight benchmarks with the most invoked methods, this analysis includes smaller benchmarks as well, in order to capture broader trends and investigate the characteristics of larger-scale benchmarks in the PyPy suite.

The results are shown in Figure 17. Blue dots represent PyPy benchmarks, while orange dots represent the synthesized benchmarks from 2SOM. For the top eight benchmarks by number of methods, their names are annotated in the figure.

From the results, we observe that the synthesized benchmarks exhibit a relatively constant number of hotspots, lacking the distribution observed in PyPy benchmarks. Nevertheless, they do not deviate significantly from the larger-scale PyPy benchmarks. For instance, `telco`, `bm_mdp`, `sqlitesynth`, and pyflate-fast are located near the synthesized benchmarks. On the other hand, benchmarks with more than approximately 1,000 methods-—such as `bm_html5lib`, `bm_krakatau`, and `bm_django`-—are positioned far from the synthesized ones.

Thus, while the synthesized benchmarks may not capture the full diversity of workloads, their overall scale is comparable to that of the larger PyPy benchmarks.

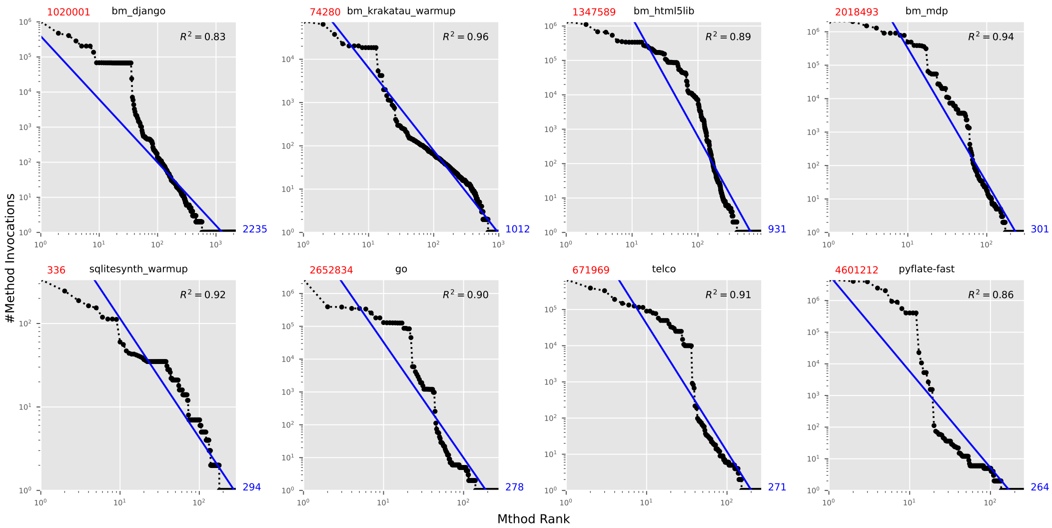


Figure 16 Correlation between the number of method invocations and the rank of methods.

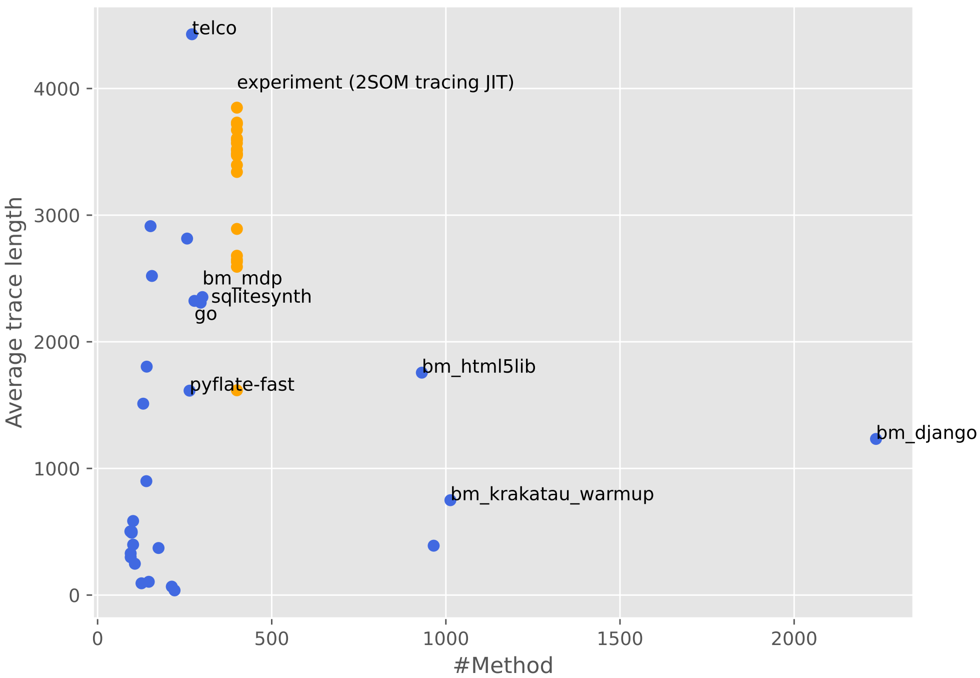


Figure 17 Correlation between average trace lengths and number of invoked methods in PyPy and synthesized benchmarks.