Interpreter Taming to Realize Multiple Compilations in a Meta-Tracing JIT Compiler Framework

Yusuke Izawa, Hidehiko Masuhara, and Carl Friedrich Bolz-Tereick

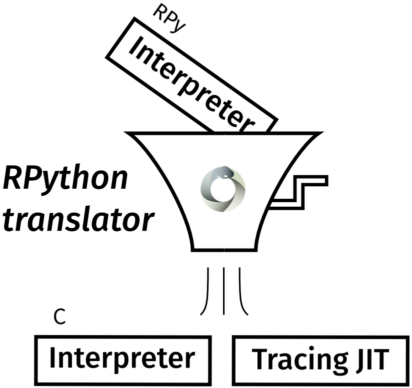
東京工業大学
Tokyo Institute of Technology

pypy

MoreVMs'23 workshop, March 13, 2023

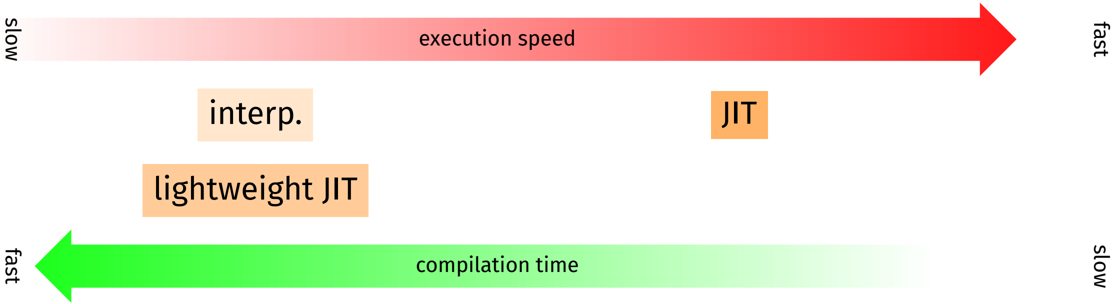
Background: RPython [Bol+09]

- A language implementation framework to develop a high-performance virtual machine (VM)
  - Generate a VM w/ tracing JIT compiler from an interpreter
- Used for generating several VMs such as PyPy [RP06], Pycket [Bau+15], and so forth



Background: Modern VMs Employ Multilevel Compilation

- Supported in modern VMs such as HotSpot$_{TM}$, V8, and so forth
- Balances code quality and compilation time by changing compilation levels

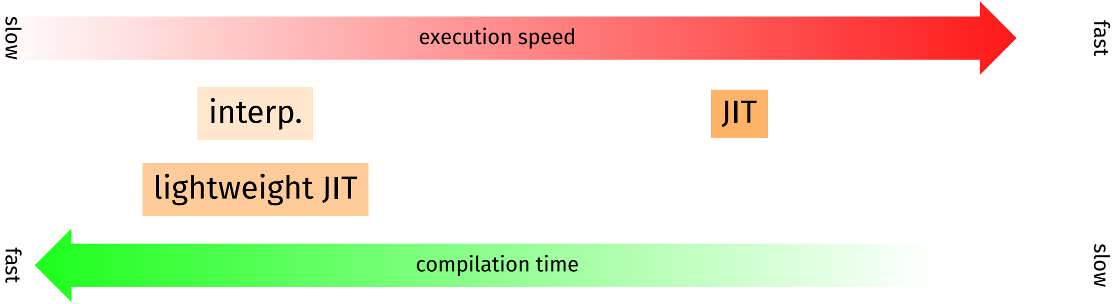


short-lived app.
e.g., batch-process

long-lived app.
e.g., server-side.

Background: Modern VMs Employ Multilevel Compilation

- Long-lived programs are applied to a JIT compiler
  - generating quality code but consuming compilation time

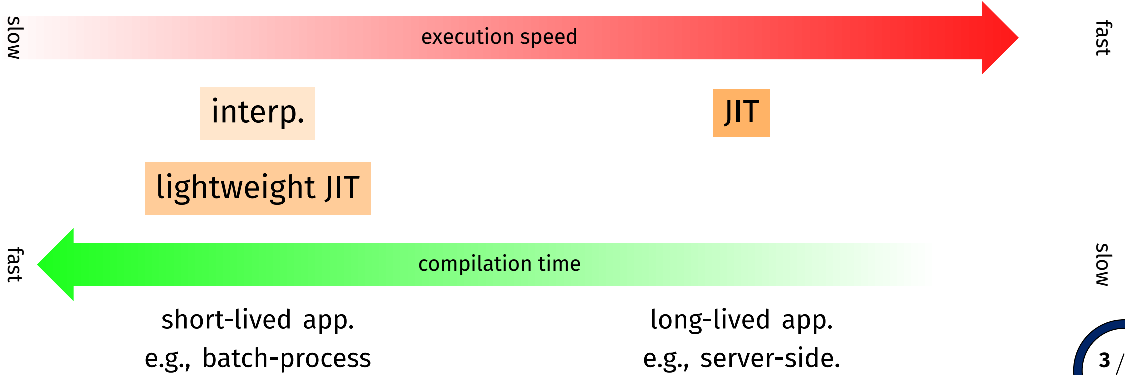


short-lived app.
e.g., batch-process

long-lived app.
e.g., server-side.

Background: Modern VMs Employ Multilevel Compilation

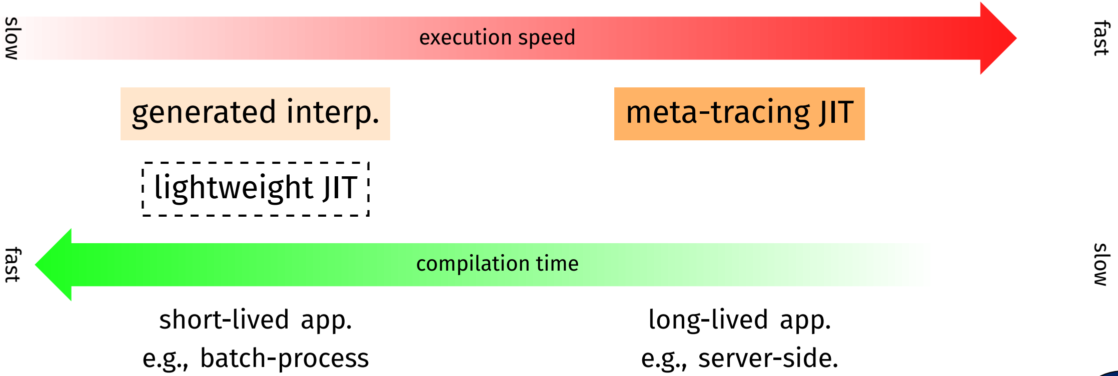
- Long-lived programs are applied to a JIT compiler
  - generating quality code but consuming compilation time
- Short-lived programs need to be run with a lightweight compiler
  - generating code quickly



RPython is Not Yet Competitive w/ Language-Specific VMs

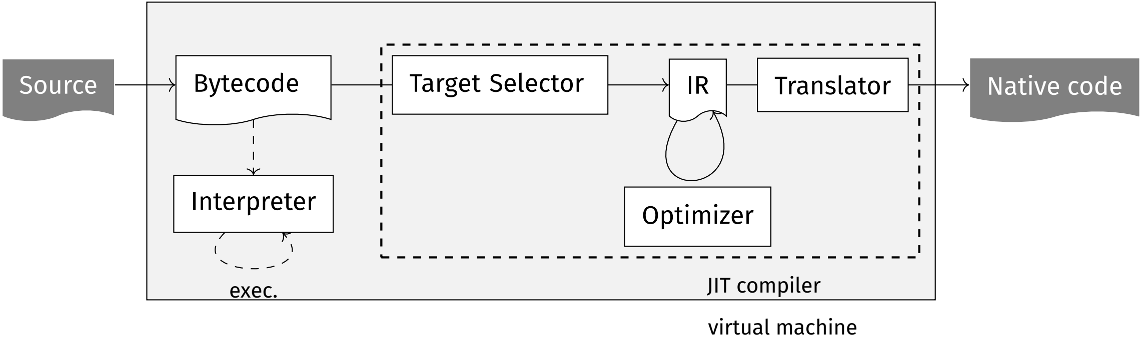
We support

- *In particular*: no support for multilevel compilation on RPython
  - *Dilemma*: hard to extend generated VMs from RPython



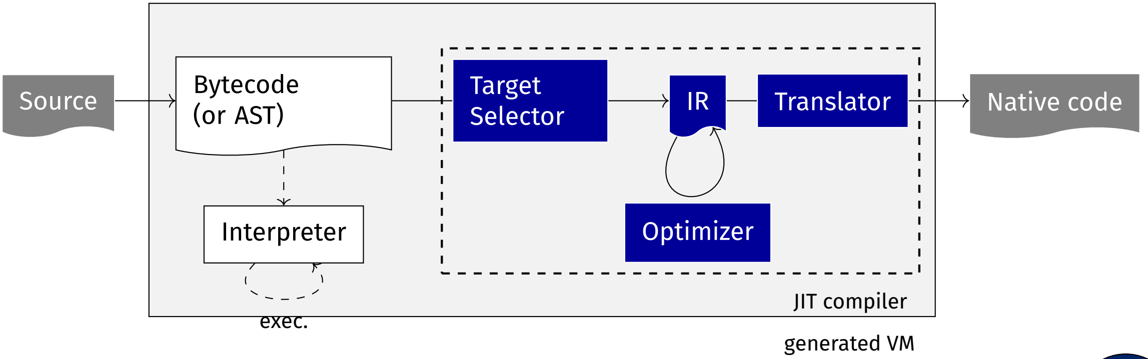
Dilemma: Hard to Extend Generated VMs (1)

- In a language-specific VM, all components are manageable



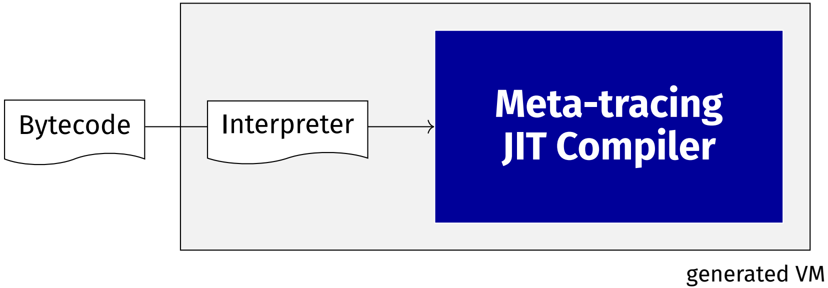
Dilemma: Hard to Extend Generated VMs (2)

- *Dilemma*: In RPython, only an interpreter (and bytecode compiler) can be managed
  - How to add lightweight compilation to RPython w/ lower effort?



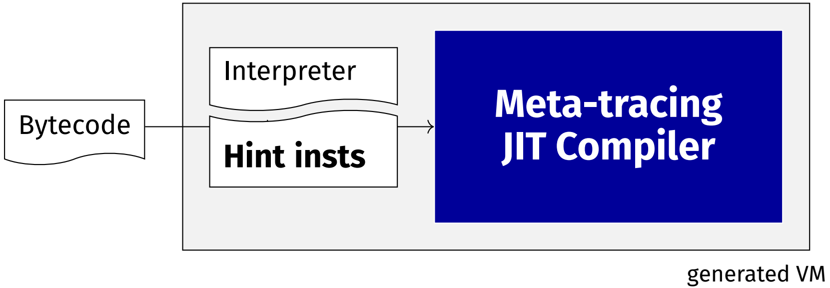
How to Add Lightweight Compilation to Meta-Tracing JIT w/ Lower Effort?: Hint instruction-based Approach

- *Approach*: control the behavior of meta-tracing JIT by inserting **hint instructions** into an interpreter, not creating compilers from scratch



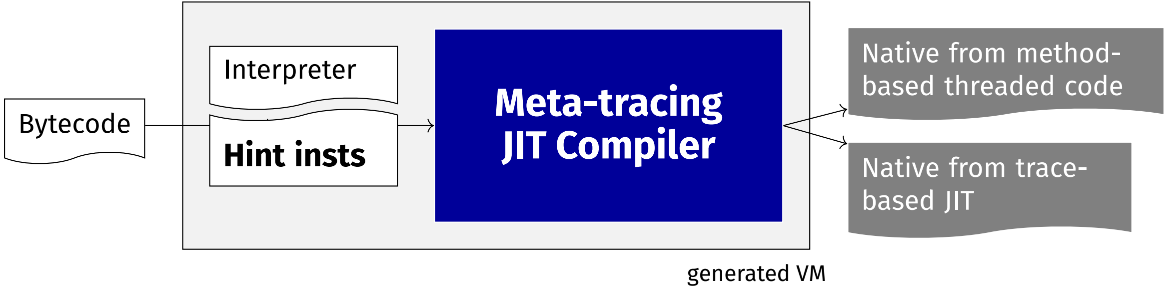
How to Add Lightweight Compilation to Meta-Tracing JIT w/ Lower Effort?: Hint instruction-based Approach

- *Approach*: control the behavior of meta-tracing JIT by inserting **hint instructions** into an interpreter, not creating compilers from scratch
  - **Hint instruction**: a pseudo function that can influence the behavior of meta-tracing JIT



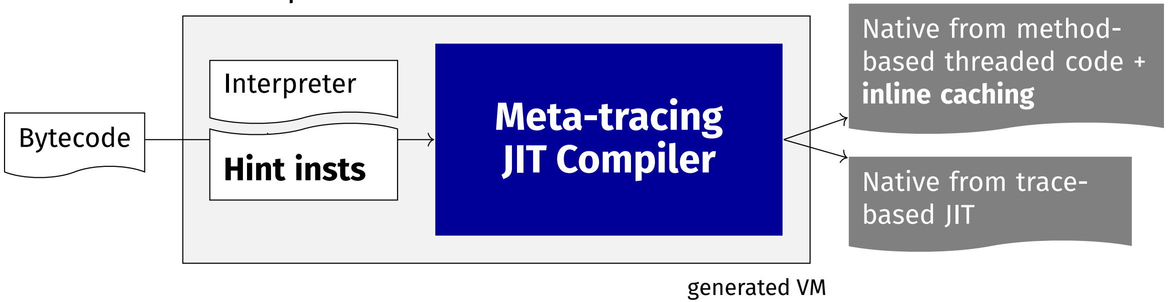
How to Add Lightweight Compilation to Meta-Tracing JIT w/ Lower Effort?: Hint instruction-based Approach

- *Previous work*: add threaded code generation to meta-tracing JIT [JOT ’22]



How to Add Lightweight Compilation to Meta-Tracing JIT w/ Lower Effort?: Hint instruction-based Approach

- *Previous work*: add threaded code generation to meta-tracing JIT [JOT '22]
- *This work*: realize inline caching [DS84] in threaded code generation and multilevel compilation



Proposal: Multilevel RPython

Interpreter w/
hint instructions

User writes

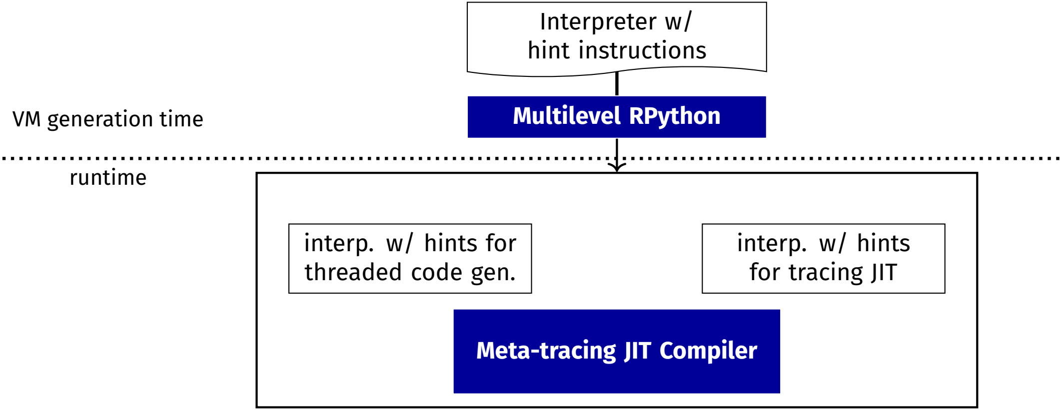
VM generation time

Multilevel RPython

runtime

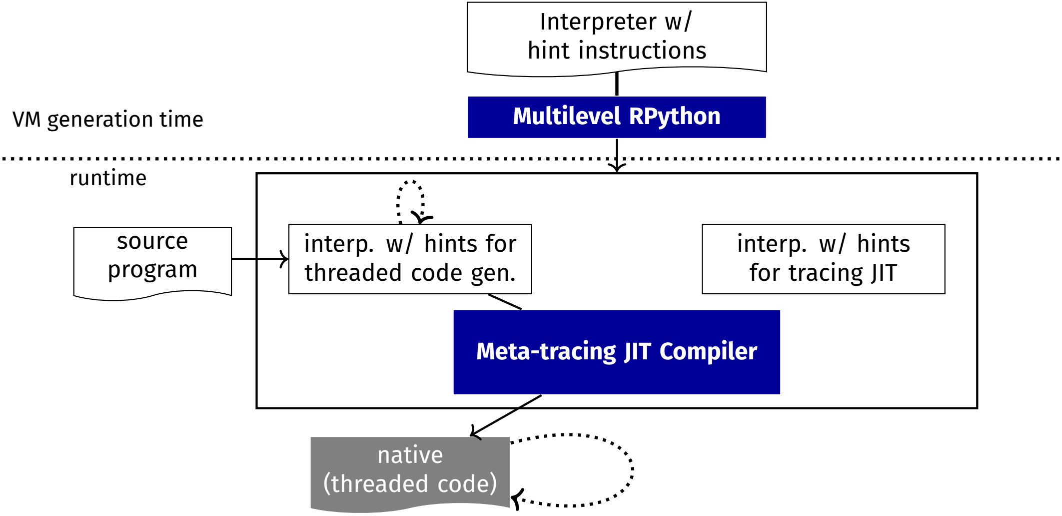
Proposal: Multilevel RPython

- Each interpreter represents each compilation level



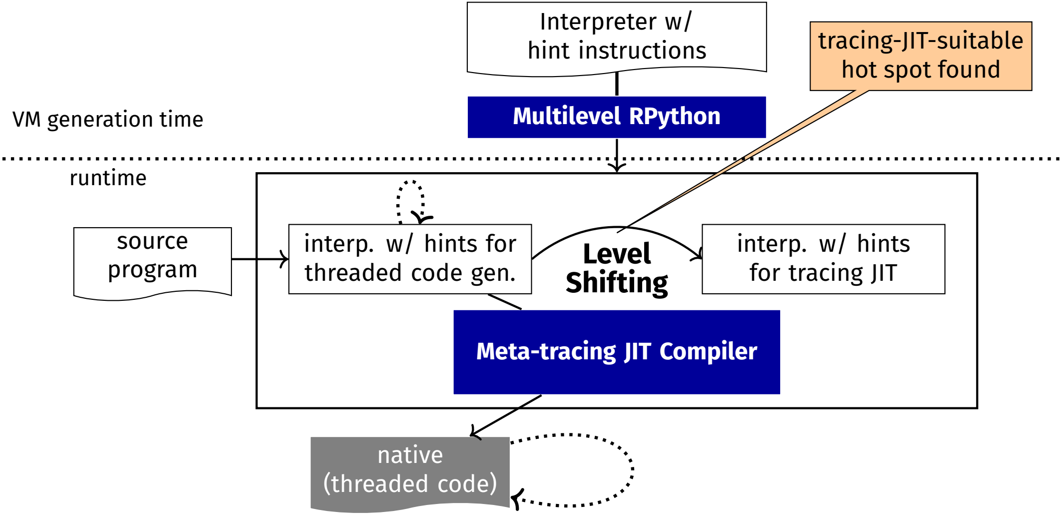
Proposal: Multilevel RPython

- Each interpreter represents each compilation level



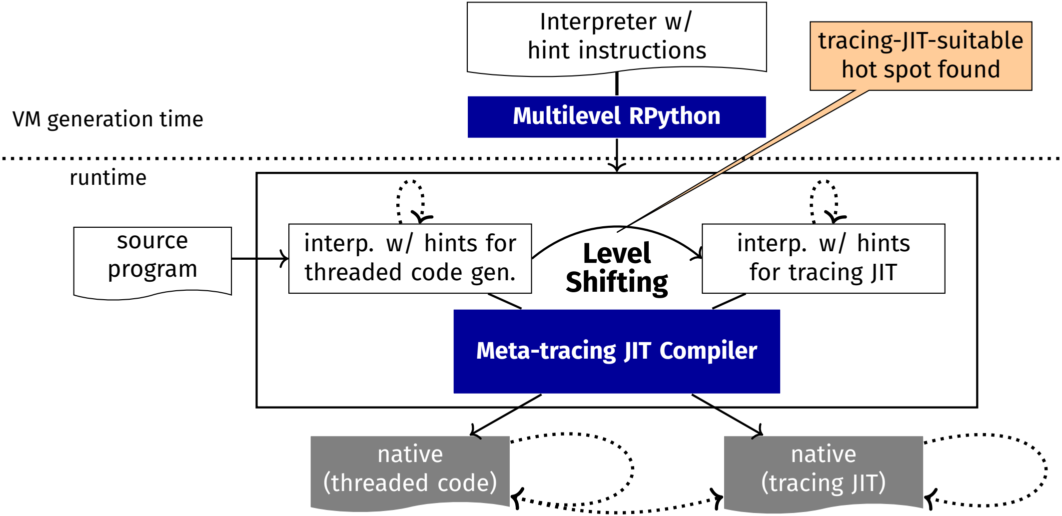
Proposal: Multilevel RPython

- Each interpreter represents each compilation level



Proposal: Multilevel RPython

- Each interpreter represents each compilation level



Level Shifting Between Threaded Code Gen. and Tracing JIT

Interpreter Shifter

```
def interp(..)
  while True:
    try:
      result = interp_threaded(pc)
    catch ContinueInTracing as e:
      pc = e.pc

    try:
      result = interp_tracing(pc)
    catch ContinueInThreaded as e:
      pc = e.pc
```

entry

level up

level down

Interpreter for threaded code generation

```
def interp_threaded(pc):
  threaded_driver.can_enter_jit(..)
  while True:
  instr = bytecode[pc++]
  if instr == JUMP_BACKWARD:
    if bytecode.counts[pc] > THRESHOLD:
      raise ContinueInTracing(pc)
    bytecode.couts[pc] += 1
    pc = bytecode[pc++]
  elif ..
```

Profiles the exec.

```
if exceeds threshold:
    start lightweight
    compilation
```

Interpreter for tracing JIT compilation

```
def interp_tracing(pc):
  while True:
  instr = bytecode[pc++]
  if instr == JUMP_BACKWARD:
    if bytecode.counts[pc] ≤ THRESHOLD:
      raise ContinueInThreaded(pc)
    bytecode.couts[pc] += 1
    pc = bytecode[pc++]
    tracing_driver.can_enter_jit(..)
  elif ..
```

Profiles the exec.

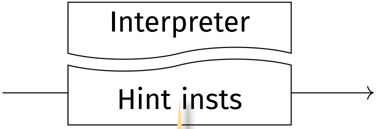
```
if exceeds threshold:
    start heavyweight
    compilation
```

Brief Recap: Threaded Code Generation [JOT '22]

- Generate threaded code [Bel73] by inserting hint instructions into an interpreter
  - 👍 small code size and short compilation time
  - ⚠ method calls in threaded code was slow (next)

Bytecode

```
L0:
  INC
  JUMP_IF L1
  INC
  JUMP L0
L1:
  CALL "g"
  RET
```



Shallow Tracing and Traversal Stack

Trace styled w/ threaded code

```
L0:
  call(INC)
  guard_false(.., L1)
  call(INC)
  jump(L0)
L1:
  call(CALL("g"))
  finish(..)
```

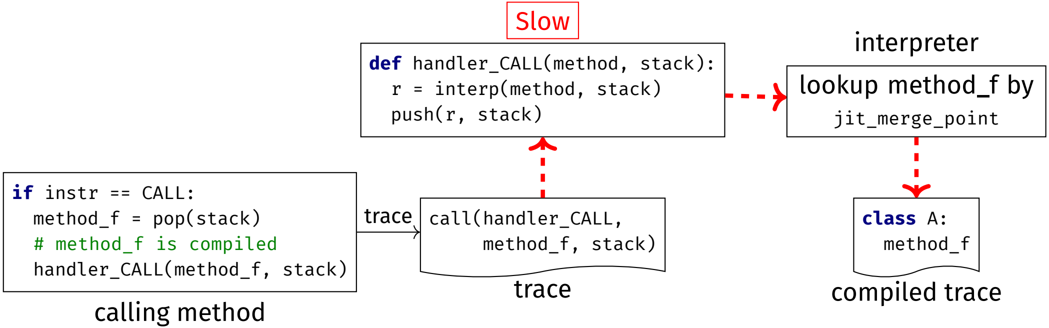
Method-based

Asm output

```
L0:
  call INC
  jnz  L1
  call INC
  jmp  L0
L1:
  call CALL
  ret
```

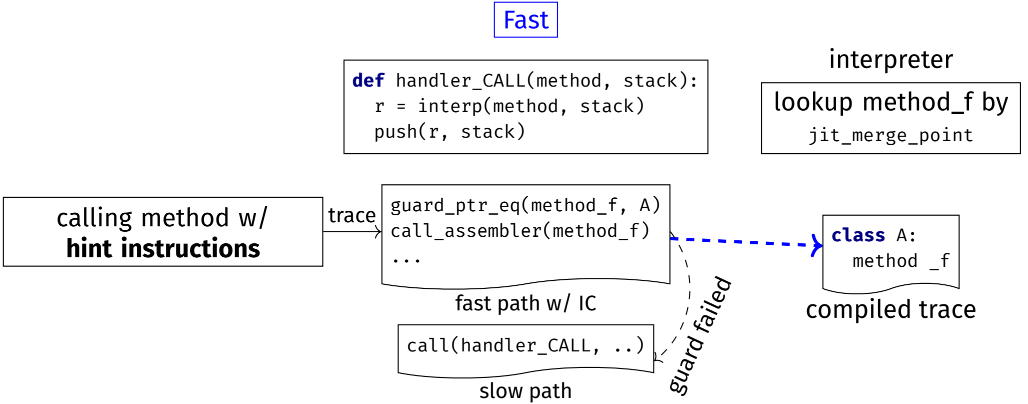
Technical Problem: Method Calls in Threaded Code Was Slow

- Every time go into the system method lookup routine of `jit_merge_point`



Solution: Inline Caching w/ Hint Instructions

- Convert into a direct call by inline caching w/ hint instructions



Inline Caching w/ Hint Instructions

- *Technique*: tame interpreter w/ hint instructions
  - record the runtime type of a `method` during interpretation
  - verify the runtime type of a `method`
  - call `call_assembler` at `CALL` when the verification passed

```
def handler_CALL(stack):
  method = pop(stack)
  if we_are_interpreted():
    record_tbl[pc] = method.type
```

```
...
guard_ptr_eq(method_f, A)
r = call_assembler(method, stack)
setitem(r, stack)
...
```

fast path

```
call(handler_CALL, ..)
...
```

slow path

Inline Caching w/ Hint Instructions

- *Technique*: tame interpreter w/ hint instructions
  - record the runtime type of a `method` during interpretation
  - verify the runtime type of a `method`
  - call `call_assembler` at `CALL` when the verification passed

```
...
guard_ptr_eq(method_f, A)
r = call_assembler(method, stack)
setitem(r, stack)
...
```

fast path

```
while True:
  instr = bytecode[pc++]
  if instr == CALL:
      method_f = pop(stack)
      if check_typ(method_f, pc):
```

```
call(handler_CALL, ..)
...
```

slow path

Inline Caching w/ Hint Instructions

- *Technique*: tame interpreter w/ hint instructions
  - record the runtime type of a `method` during interpretation
  - verify the runtime type of a `method`
  - call `call assembler` at `CALL` when the verification passed

```
...
guard_ptr_eq(method_f, A)
r = call_assembler(method, stack)
setitem(r, stack)
...
```

fast path

```
if check_typ(method_f, pc):
  r = call_assembler(method_f, stack, ..)
  push(r)
else:
  handler_CALL(stack, pc, ..)
```

```
call(handler_CALL, ..)
...
```

slow path

Inline Caching w/ Hint Instructions

- *Technique*: tame interpreter w/ hint instructions
  - record the runtime type of a `method` during interpretation
  - verify the runtime type of a `method`
  - call `call assembler` at `CALL` when the verification passed

```
def handler_CALL(stack):
  method = pop(stack)
  if we_are_interpreted():
    record_tbl[pc] = method.type
while True:
  instr = bytecode[pc++]
  if instr == CALL:
      method_f = pop(stack)
      if check_typ(method_f, pc):
        r = call_assembler(method_f, stack, ..)
        push(r)
      else:
        handler_CALL(stack, pc, ..)
```

```
...
guard_ptr_eq(method_f, A)
r = call_assembler(method, stack)
setitem(r, stack)
...
```

fast path

```
call(handler_CALL, ..)
...
```

slow path

Implementation

- Implemented on PySOM
  - PySOM: A subset of Smalltalk implementation by RPython
    - 14000 LOC in RPython
- 10 out of 72 instructions are instrumented to do threaded code generation
  - `jump_on_false`, `jump_backward`, `return_local`, ...
- Total LOC:
  - PySOM: about 450 LOC addition
  - RPython: about 600 LOC addition

Evaluation and Experiment: Overview

Micro-Benchmark Evaluation

- Evaluate the cost and benefit of two JITs: threaded code generation (+ inline caching) and tracing JIT

Multilevel Experiment in a Simulated Real-World Workload

- Evaluate the performance of multilevel compilation in RPython against single level compilation
  - Multilevel: threaded code + tracing JIT
  - Single level: tracing JIT

Micro-benchmark Evaluation: What is Evaluated?

Evaluate the cost and benefit of two different JITs

threaded code generation and tracing JIT

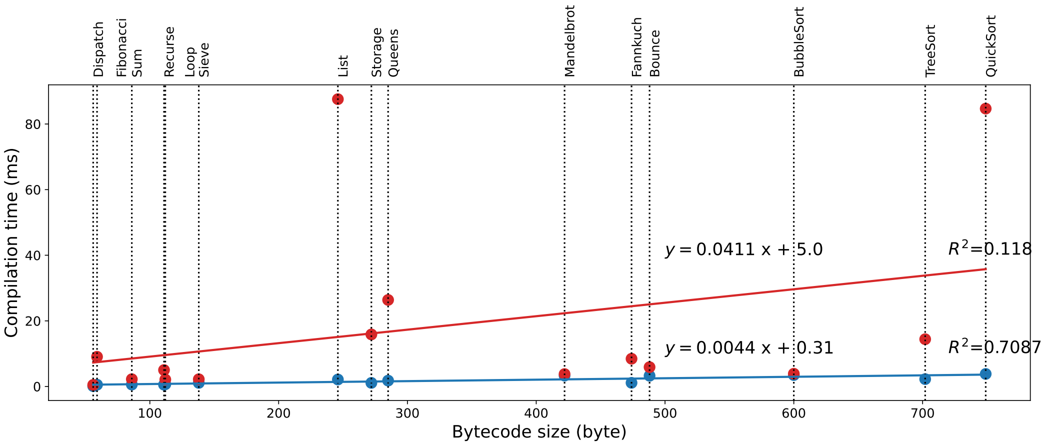
- Cost: compilation time
  - Correlation between bytecode size and compile time
- Benefit: peak performance at steady state
  - Comparison with interpreter execution

Targets

- Targets: PySOM original + Are We Fast Yet? [MDM16] micro benchmark
- Methodology: Ran 2000 times in one set for each program, iterated 30 sets

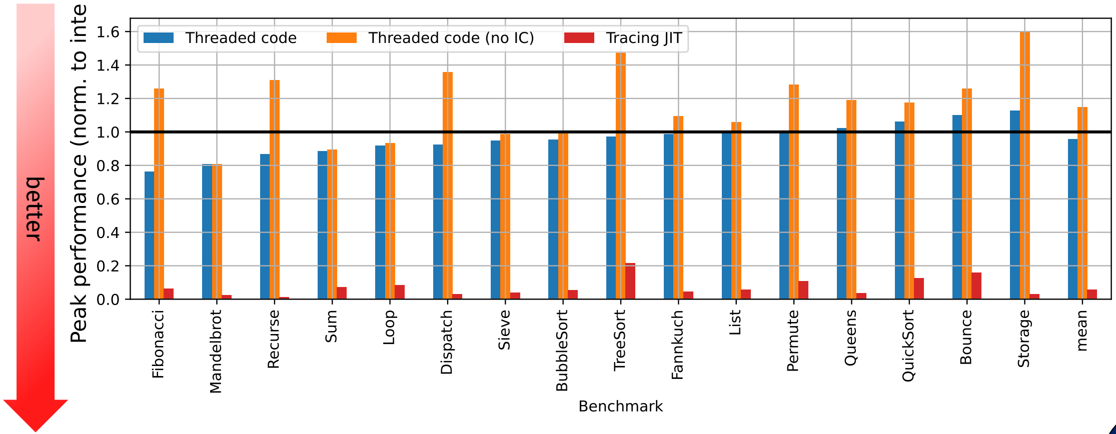
Compilation Time and Bytecode Sizes

- Threaded code: compilation time is proportional to bytecode size
- Tracing JIT: unstable



Peak Performance at Steady State

- *Overall*: threaded code was 4 % faster than interpreter, 94 % slower than tracing JIT
  - Inline caching improved threaded code approx. 20 %

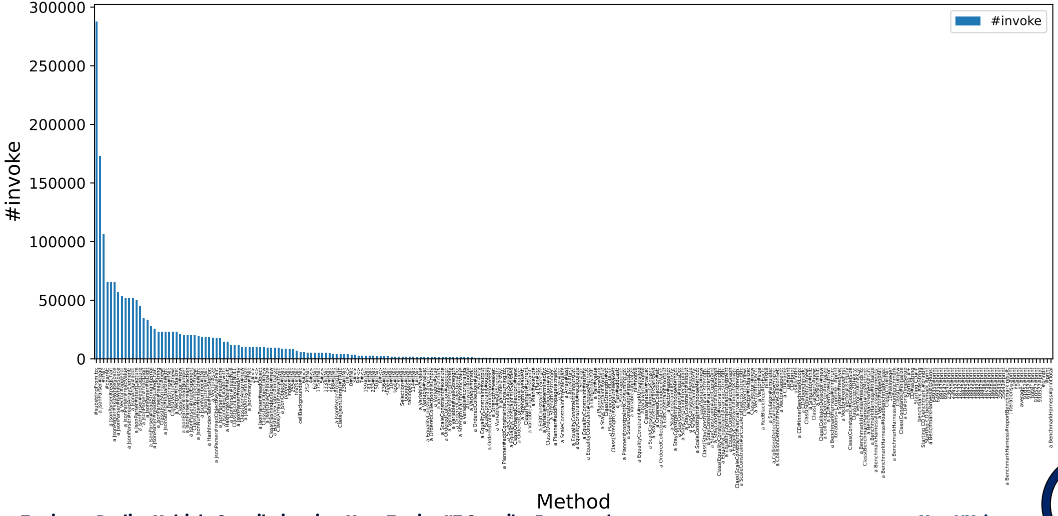


Multilevel JIT Experiment: Is Adding Threaded Code Generation Beneficial for RPython?

- Experimentation in real-world applications is currently difficult
  - difficult to access to the enterprise app, current implementation size
- Simulated a real-world workload w/ large benchmarks
  - Richards + Json + CD + DeltaBlue

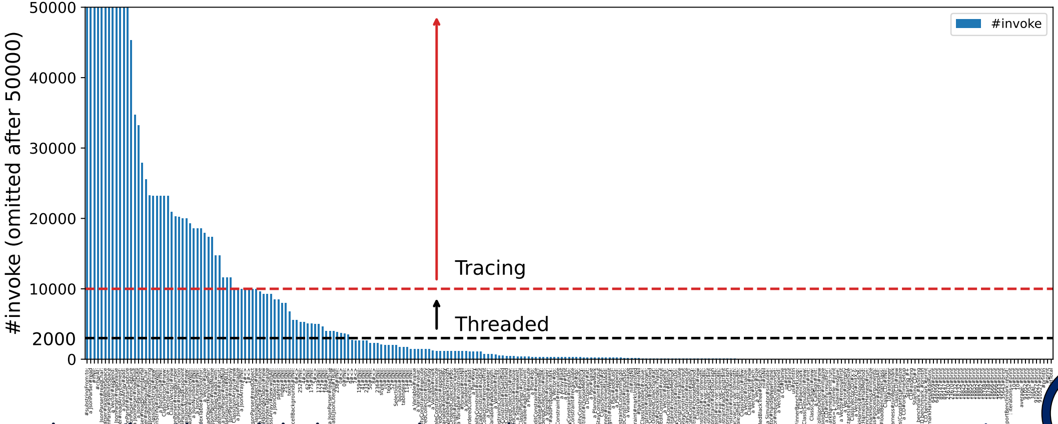
#Invocation: Simulated Larger Application

- Simulated Application: Richards + Json + CD + DeltaBlue



#Invocation: Threshold for Simulated Larger App

- Determined by the following heuristics:
  - about 30% of the methods are compiled by threaded code
  - about 20% are by tracing JIT
- Based on the result of DaCapo benchmark [Bla+06]



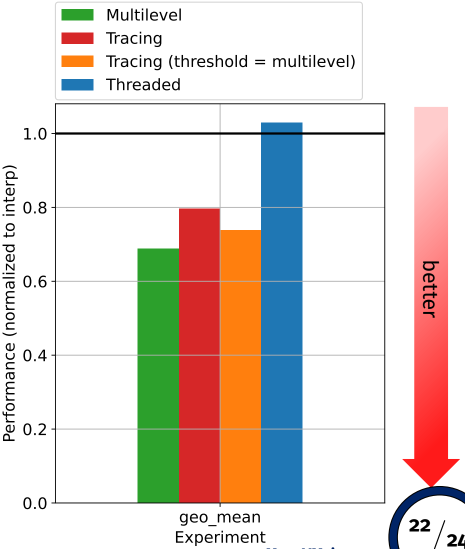
Performance: Simulated Larger Application

Result

- Multilevel is the fastest
  - about 14 % faster than tracing
  - about 5 % faster than tracing (w/ same threshold to multilevel)

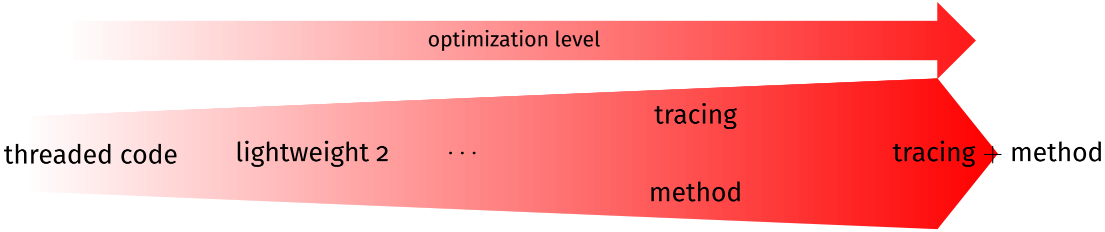
Policy	compile time	elapsed time	comp / elapse
Multilevel	66ms	90ms	0.73
Tracing	73ms	104ms	0.70
Tracing (same threshold)	62ms	94ms	0.65
Threaded	32ms	135ms	0.23
Interpreter	0ms	131ms	0

Policy	Loop threshold	Function threshold
Multilevel	1539	2039
Tracing	1039	1639



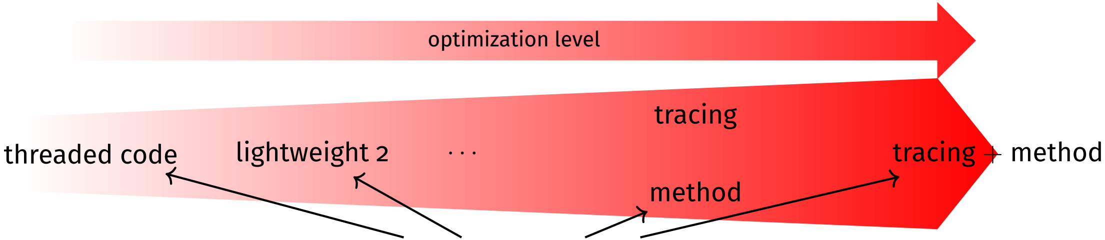
Adaptive Compilation in RPython

Perform multilevel compilation with “one interpreter” and “one engine”



Adaptive Compilation in RPython

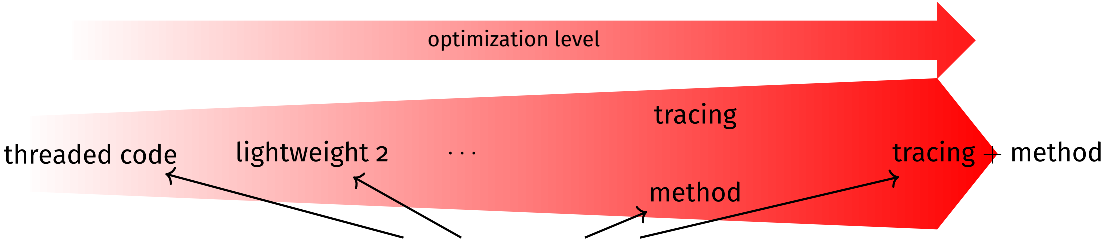
Perform multilevel compilation with "one interpreter" and "one engine"



one generic interp. → common interp + a bit different definitions

Adaptive Compilation in RPython

Perform multilevel compilation with “one interpreter” and “one engine”



one generic interp. → common interp + a bit different definitions

perform on **one engine** = RPython

Conclusion

- Showed it is possible to add a new behavior in a meta-tracing JIT compiler framework by using hint instructions
  - threaded code generation [JOT '22]
  - inline caching [This talk]
- Multilevel compilation showed 14% better overall performance in the application that simulated a real-word workload than tracing-JIT-only compilation

How to Reduce the RPython's Compilation Time

Initial trace is so long on tracing recursive calls

```
pc = 0; bytecode = [..]; stack = [..]
while True:
  instr = bytecode[pc++]
  if instr == INT:
    n = ord(bytecode[pc++])
    push(stack, n)
  elif instr == LT:
    y, x = pop(stack), pop(stack)
    if x < y: push(stack, True)
    else: push(stack, False)
  elif instr == JUMP_IF:
    if not top(stack):
    pc = bytecode[pc++]
  elif instr == JUMP_BACK:
    pc = bytecode[pc++]
  elif instr == CALL:
    target = bytecode[pc++]
    r = interp(stack, target)
    push(r)
  elif ..
```

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

How to Reduce the RPython's Compilation Time

Initial trace is so long on tracing recursive calls

```
pc = 0; bytecode = [..]; stack = [..]
while True:
  instr = bytecode[pc++]
  if instr == INT:
    n = ord(bytecode[pc++])
    push(stack, n)
  elif instr == LT:
    y, x = pop(stack), pop(stack)
    if x < y: push(stack, True)
    else: push(stack, False)
  elif instr == JUMP_IF:
    if not top(stack):
    pc = bytecode[pc++]
  elif instr == JUMP_BACK:
    pc = bytecode[pc++]
  elif instr == CALL:
    target = bytecode[pc++]
    r = interp(stack, target)
    push(r)
  elif ..
```

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

```
v4 = list_read(v3, v0)
v5 = add(v0, 1)
guard_eq(v4, LT)
v6 = add(v5, 1)
v7 = list_pop(v1)
v8 = list_pop(v1)
guard_type(v7, int)
guard_type(v8, int)
guard_not_less_than(v7, v8)
list_append(v1, False)
v10 = list_read(v3, v0)
guard_eq(v10, DUP)
...
guard_eq(v18, SUB)
...
guard_ed(v26, CALL)
... (inlined) ...
guard_eq(v34, DUP)
...
auard_eq(v42, SUB)
...
... (stop inlining) ...
...
guard_eq(v58, CALL)
v1092 = call("sum", ..)
...
finish(v2000)
```

How to Reduce the RPython's Compilation Time

RPython consumes time on ..

- inlining a user program's function call

Initial trace

```
v4 = list_read(v3, v0)
v5 = add(v0, 1)
guard_eq(v4, LT)
v6 = add(v5, 1)
v7 = list_pop(v1)
v8 = list_pop(v1)
guard_type(v7, int)
guard_type(v8, int)
guard_not_less_than(v7, v8)
list_append(v1, False)
v10 = list_read(v3, v0)
guard_eq(v10, DUP)
...
guard_eq(v18, SUB)
...
guard_ed(v26, CALL)
... (inlined) ...
guard_eq(v34, DUP)
...
guard_eq(v42, SUB)
...
... (stop inlining) ...
...
guard_eq(v58, CALL)
v1092 = call("sum", ..)
...
finish(v2000)
```

How to Reduce the RPython's Compilation Time

RPython consumes time on ..

- inlining a user program's function call
- optimizing the initial trace

Optimized trace (constants and lists are folded)

```
v6 = list_pop(v1)
guard_type(v6, int)
guard_not_less_than(v6, 1)
v16 = dict_get(v2, "n")
guard_type(v16, int)
v21 = int_sub(v16, 1)
... (inlined) ...
finish(v64)
```

How to Reduce the RPython's Compilation Time

RPython consumes time on ..

- inlining a user program's function call
- optimizing the initial trace
- recompiling after guard failure (if failed many times)

Optimized trace (constants and lists are folded)

```
v6 = list_pop(v1)
guard_type(v6, int)
guard_not_less_than(v6, 1)
v16 = dict_get(v2, "n")
guard_type(v16, int)
v21 = int_sub(v16, 1)
... (inlined) ...
finish(v64)
```

How to Reduce the RPython's Compilation Time

Issue: how to reduce compilation time in RPython

- Optimization / Code gen.
- Tracing
  - RPython traces recursions
- Recompilation at guard failing

How to Reduce the RPython's Compilation Time

Issue: how to reduce compilation time in RPython

Optimization / Code gen.

↑

Lightweight compilation

Tracing

RPython traces recursions

↑

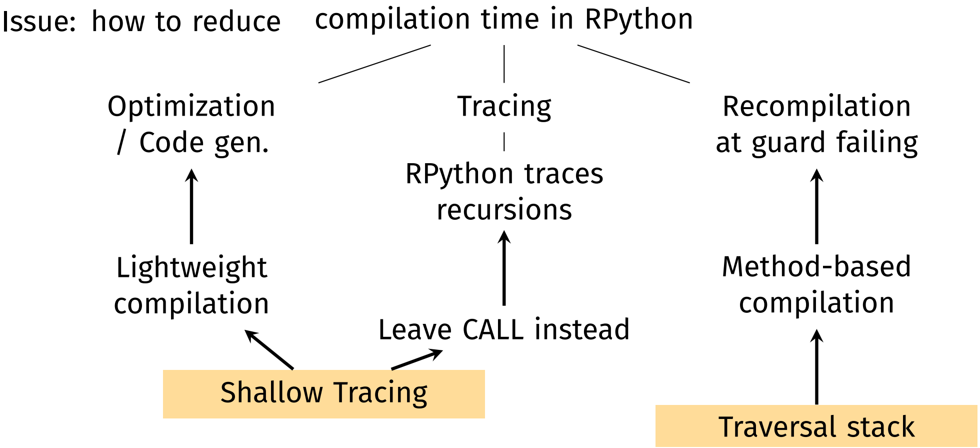
Leave CALL instead

Recompilation at guard failing

↑

Method-based compilation

How to Reduce the RPython's Compilation Time



Shallow Tracing Leaves CALLs and Doesn't Exec. Bodies

- `@dont_look_inside`: leaves a call to the decorated fun at tracing
  - but executes the body at tracing
- `if dummy: return`: skips executing the body during tracing
  - `dummy` turns into `False` after tracing

```
while True:
  instr = bytecode[pc++]
  if instr == INC:
    handler_INC(stack)
  elif instr == CALL:
    handler_CALL(stack)
  elif ...
    ...
    ...
```

```
@dont_look_inside
def handler_INC(stack,dummy=True):
  if dummy: return
  x = stack[sp--]
  z = add(x, 1)
  stack[sp++] = z

@dont_look_inside
def handler_CALL(stack,dummy=True):
  if dummy: return
  r = interp(stack, ..)
  push(r, stack)
```

```
# INC
call(handler_INC, ..)
# CALL
call(handler_CALL, ..)
...
```

Shallow Tracing is Implemented as Decorator

- `@dont_look_inside`: leaves a call to the decorated fun at tracing
  - but executes the body at tracing
- `if dummy: return`: skips executing the body during tracing
  - `dummy` turns into `False` after tracing

```
while True:
  instr = bytecode[pc++]
  if instr == INC:
    handler_INC(stack)
  elif instr == CALL:
    handler_CALL(stack)
  elif ...
    ...
    ...
```

```
@enable_shallow_tracing
def handler_INC(stack):

    x = stack[sp--]
    z = add(x, 1)
    stack[sp++] = z

@enable_shallow_tracing
def handler_CALL(stack):

    r = interp(stack, ..)
    push(r)
```

```
# INC
call(handler_INC, ..)
# CALL
call(handler_CALL, ..)
...
```

Realize Next Level by One Step "Deeper" Shallow Tracing

- Fold stack manipulations in shallow tracing

```
while True:
  instr = bytecode[pc++]
  if instr == INC:
    handler_INC(stack)
  elif instr == CALL:
    handler_CALL(stack)
  elif ...
    ...
    ...
```

```
def handler_INC(stack):

  x = stack[sp--]
  z = add(x, 1)
  stack[sp++] = z

@enable_shallow_tracing
def add(x, y):

  return x + y
```

```
L0:
  x1 = stack[sp--]
  z2 = add(x1, 1)
  stack[sp++] = z2
  guard_true(..)
  x3 = stack[sp--]
  z4 = add(x3, 1)
  stack[sp++] = z4
  jump(L0)
L1:
  ...
  finish(..)
```

Realize Next Level by One Step "Deeper" Shallow Tracing

- Fold stack manipulations in shallow tracing

```
while True:
  instr = bytecode[pc++]
  if instr == INC:
    handler_INC(stack)
  elif instr == CALL:
    handler_CALL(stack)
  elif ...
    ...
    ...
```

```
def handler_INC(stack):

  x = stack[sp--]
  z = add(x, 1)
  stack[sp++] = z

@enable_shallow_tracing
def add(x, y):

  return x + y
```

```
L0:
  x1 = stack[sp--]
  z2 = add(x1, 1)
  stack[sp++] = z2
  guard_true(..)
  x3 = stack[sp--]
  z4 = add(z2, 1)
  stack[sp++] = z4
  jump(L0)
L1:
  ...
  finish(..)
```