MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Runtime Feedback in a Meta-Tracing JIT for Efficient Dynamic Languages

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, Armin Rigo

Maaz Hussain
Arnav Reddy
Julian Whittaker
Group 24

Motivation: Slow Dynamic Languages

- Language Overhead
  - Dynamic Dispatch, Type Checking, Complex Object Models
- JIT Compiler
  - Complicated & Error-prone

Background: Traditional Tracing JITs

- Traditional tracing JITs
  - Traces hot loops in **User-code**
  - **4 stages**
    - Profile
    - Trace
    - Optimize
    - Execute

Meta-Tracing JITs (PyPy)

- Meta-Tracing JIT (PyPy)
  - Traces **Interpreter-code** through user loops
  - Language Agnostic
  - Applies the 4 stages of a traditional tracing JIT on the interpreted language

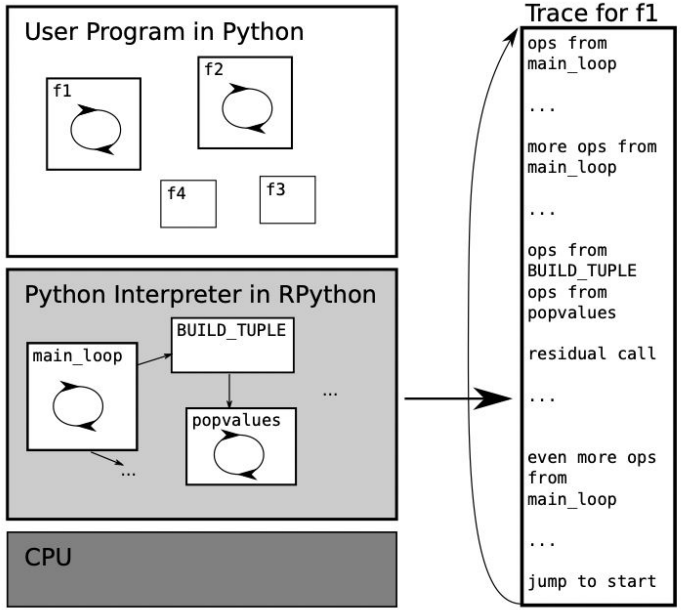


Figure 1. The levels involved in tracing

Simple Object Model (Slow)

```
class Instance(object):                                  16
    def __init__(self, cls):                             17
        self.cls = cls                                   18
        self.attributes = {}                             19

    def getfield(self, name):                            21
        return self.attributes.get(name, None)           22

    def write_attribute(self, name, value):              24
        self.attributes[name] = value                    25

    def getattr(self, name):                             27
        result = self.getfield(name)                     28
        if result is None:                               29
            result = self.cls.find_method(name)          30
            if result is None:                           31
                raise AttributeError                     32
        return result                                    33
```

Figure 2. Original Version of a Simple Object Model

```
# inst_1.getattr("a")                                    27
attributes_1 = inst_1.attributes                         22
result_1 = dict.get(attributes_1, "a")                   22
guard(result_1 is not None)                              29

# inst_1.getattr("b")                                    27
attributes_2 = inst_1.attributes                         22
v_1 = dict.get(attributes_2, "b", None)                  22
guard(v_1 is None)                                       29
cls_1 = inst_1.cls                                       30
methods_1 = cls.methods                                  10
result_2 = dict.get(methods_1, "b", None)                10
guard(result_2 is not None)                              31
v_2 = result_1 + result_2                                -1
```

Figure 3. Trace Through the Object Model

Key Hint 1: promote()

- Hint to tracer that variable is constant
  - Guards
  - Unlocks **constant folding**
- Used in practice for variable **types**

`def f1(x, y):`
`  z = x * 2 + 1`
`  return z + y`
	`def f1(x, y):`
	`  promote(x)`
	`  z = x * 2 + 1`
	`  return z + y`

$v_1 = x_1 * 2$ $z_1 = v_1 + 1$ $v_2 = z_1 + y_1$ **return**(v_2)	guard$(x_1 == 4)$ $v_1 = x_1 * 2$ $z_1 = v_1 + 1$ $v_2 = z_1 + y_1$ **return**(v_2)
	guard$(x_1 == 4)$ $v_2 = 9 + y_1$ **return**(v_2)

Key Hint 2: @elidable

- Manipulate objects, not primitives
- Hint to tracer that function is pure
  - Unlocks **constant folding**
- Immutable data structures

```
@elidable
getIndex(key):
    ...

promote(map)
index = map.getIndex("foo")
return storage[index]
```

Application Walkthrough (Maps)

- Slow Dictionary accesses
- Intuition: Many instances share **layout** (attribute keys)
- Split Instance Dict
  - **Map** (Immutable)
  - storage

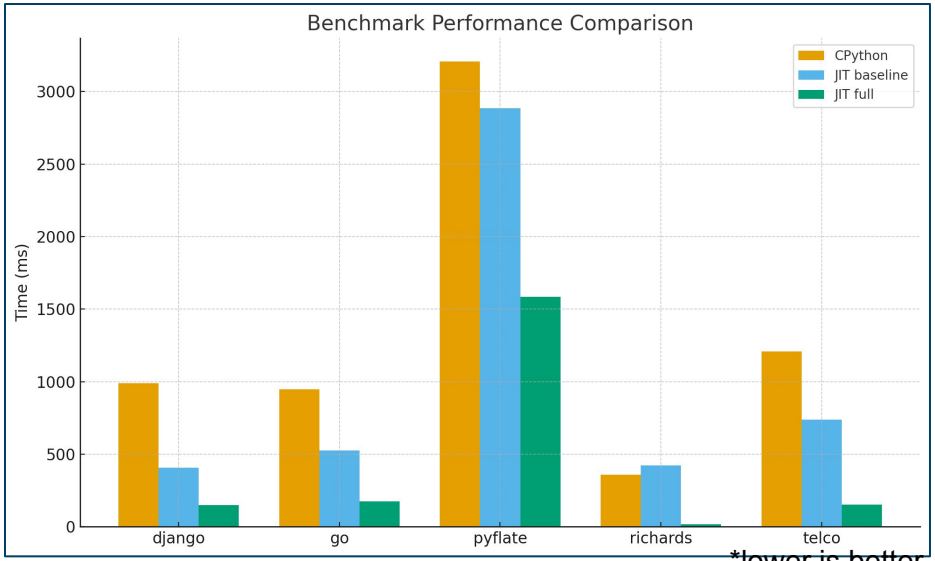
```
# inst1.getattr("a")
map1 = inst1.map
guard(map1 == 0xb74af4a8)
index1 = Map.getindex(map1, "a")
guard(index1 != -1)
storage1 = inst1.storage
result1 = storage1[index1]

# inst1.getattr("b")
map2 = inst1.map
guard(map2 == 0xb74af4a8)
index2 = Map.getindex(map2, "b")
guard(index2 == -1)
cls1 = inst1.cls
methods1 = cls1.methods
result2 = dict.get(methods1, "b", None)
guard(result2 is not None)
v2 = result1 + result2
```

Performance Results

	CPython	JIT baseline	JIT full
django[ms]	988.67 ± 0.49 6.62 ×	405.62 ± 4.80 2.72 ×	149.31 ± 1.37 1.00 ×
go[ms]	947.43 ± 1.30 5.44 ×	525.53 ± 7.67 3.01 ×	174.32 ± 7.78 1.00 ×
pyflate[ms]	3209.20 ± 3.65 2.02 ×	2884.26 ± 21.11 1.82 ×	1585.48 ± 5.22 1.00 ×
richards[ms]	357.79 ± 1.32 20.00 ×	421.87 ± 0.48 23.58 ×	17.89 ± 1.15 1.00 ×
telco[ms]	1209.67 ± 2.20 7.88 ×	738.18 ± 3.29 4.81 ×	153.48 ± 1.86 1.00 ×

Figure 9. Benchmark Results



*lower is better

CPython is the default bytecode-based Python Interpreter. JIT baseline is is the PyPy interpreter without maps or type version enabled. JIT full is PyPy running with its full suite of optimizations.
All benchmarks are written in a subset of Python (RPython)

Results Discussion

- 1.8x to almost 20x speed ups
- Better performance than widely used CPython
  - In OOP contexts
- Better than baseline JIT
  - Hints allow for optimizations that baseline JIT couldn't effectively do on its own
  - Agnostic backend allows for cross-language usage
- Interpreter level changes -> abstracted to end developer

Limitations & Tradeoffs

- promote(): Code Bloat
- @elidable: Correctness
- Expensive writes to class definitions (versioning)
- Significant interpreter changes
- Object-Oriented targeted speedups

Impact

pypy/module/_lsprof/interp_lsprof.py

```
@jit.elidable_promote()
def _get_or_make_builtin_entry(self, w_func, w_type
```

- Ideas still used in modern PyPy
- Extended to other languages like Ruby [2]
- Influenced later runtime optimization systems like Truffle/GraalVM [3]

References

[1] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. 2011. Runtime feedback in a meta-tracing JIT for efficient dynamic languages. In Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS '11). Association for Computing Machinery, New York, NY, USA, Article 9, 1–8. https://doi.org/10.1145/2069172.2069181

[2] Maxime Chevalier-Boisvert, Noah Gibbs, Jean Boussier, Si Xing (Alan) Wu, Aaron Patterson, Kevin Newton, and John Hawthorn. 2021. YJIT: a basic block versioning JIT compiler for CRuby. In Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL 2021). Association for Computing Machinery, New York, NY, USA, 25–32. https://doi.org/10.1145/3486606.3486781

[3] Christian Wimmer and Thomas Würthinger. 2012. Truffle: a self-optimizing runtime system. In Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity (SPLASH '12). Association for Computing Machinery, New York, NY, USA, 13–14. https://doi.org/10.1145/2384716.2384723

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Q & A