Cross-Layer Workload Characterization of Meta-Tracing JIT VMs

Berkin Ilbeyi¹, Carl Friedrich Bolz-Tereick², and Christopher Batten¹

¹ Cornell University, ² Heinrich-Heine-Universität Düsseldorf
Dynamic languages are popular

<table>
<thead>
<tr>
<th>Language Rank</th>
<th>Types</th>
<th>Spectrum Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Python</td>
<td>![Icon]</td>
<td>100.0</td>
</tr>
<tr>
<td>2. C</td>
<td>![Icon]</td>
<td>99.7</td>
</tr>
<tr>
<td>3. Java</td>
<td>![Icon]</td>
<td>99.4</td>
</tr>
<tr>
<td>4. C++</td>
<td>![Icon]</td>
<td>97.2</td>
</tr>
<tr>
<td>5. C#</td>
<td>![Icon]</td>
<td>88.6</td>
</tr>
<tr>
<td>6. R</td>
<td>![Icon]</td>
<td>88.1</td>
</tr>
<tr>
<td>7. JavaScript</td>
<td>![Icon]</td>
<td>85.5</td>
</tr>
<tr>
<td>8. PHP</td>
<td>![Icon]</td>
<td>81.4</td>
</tr>
<tr>
<td>9. Go</td>
<td>![Icon]</td>
<td>76.1</td>
</tr>
<tr>
<td>10. Swift</td>
<td>![Icon]</td>
<td>75.3</td>
</tr>
</tbody>
</table>

Dynamic languages are popular

<table>
<thead>
<tr>
<th>Language Rank</th>
<th>Types</th>
<th>Spectrum Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Python</td>
<td>![Icons]</td>
<td>100.0</td>
</tr>
<tr>
<td>2. C</td>
<td>![Icons]</td>
<td>99.7</td>
</tr>
<tr>
<td>3. Java</td>
<td>![Icons]</td>
<td>99.4</td>
</tr>
<tr>
<td>4. C++</td>
<td>![Icons]</td>
<td>97.2</td>
</tr>
<tr>
<td>5. C#</td>
<td>![Icons]</td>
<td>88.6</td>
</tr>
<tr>
<td>6. R</td>
<td>![Icons]</td>
<td>88.1</td>
</tr>
<tr>
<td>7. JavaScript</td>
<td>![Icons]</td>
<td>85.5</td>
</tr>
<tr>
<td>8. PHP</td>
<td>![Icons]</td>
<td>81.4</td>
</tr>
<tr>
<td>9. Go</td>
<td>![Icons]</td>
<td>76.1</td>
</tr>
<tr>
<td>10. Swift</td>
<td>![Icons]</td>
<td>75.3</td>
</tr>
</tbody>
</table>

Dynamic languages are slow

Dynamic languages are slow

Just-in-time-compiling virtual machines

Application: *FooLang*

interpret

*FooLang* Interpreter
Just-in-time-compiling virtual machines

Application: **FooLang**

- **FooLang Interpreter**
- **FooLang JIT Compiler**
- **GC**
- **VM Utilities**

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
Just-in-time-compiling virtual machines

Application: FooLang

FooLang Interpreter

FooLang JIT Compiler

VM Utilities

GC

FooLang VM

Generic JIT Compiler

VM Utilities

GC

Motivation

• Meta-tracing
• PyPy >> CPython
• PyPy << C
Just-in-time-compiling virtual machines

Application: FooLang

- FooLang Interpreter
- FooLang JIT Compiler
- GC
- VM Utilities
- FooLang VM

Application: FooLang

- FooLang Interpreter
- Generic JIT Compiler
- GC
- VM Utilities

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
Just-in-time-compiling virtual machines

Application: *FooLang*

- **FooLang Interpreter**
- **FooLang JIT Compiler**
- **GC**
- **VM Utilities**

**FooLang VM**

Application: *FooLang*

- **FooLang Interpreter**
- **GC**
- **VM Utilities**

Application: *Bar*

- **Bar Interpreter**
- **GC**
- **VM Utilities**

Motivation

- Meta-tracing
- PyPy >> CPython
- PyPy << C
Just-in-time-compiling virtual machines

Motivation

- Meta-tracing
- PyPy >> CPython
- PyPy << C
Meta-JIT approaches: meta-tracing and partial evaluation

RPython Framework
Meta-tracing: meta-interpreter and tracing JIT

```
def max(a, b):
    if a > b:
        return a
    else:
        return b
```

Truffle/Graal Framework
Partial evaluation: partial evaluator and method JIT
Meta-JIT approaches: meta-tracing and partial evaluation

RPython Framework
Meta-tracing: meta-interpreter and tracing JIT

Linear JIT IR

```
guard_type(a, int)
guard_type(b, int)
c = int_gt(a, b)
guard_true(c)
return(a)
```

Truffle/Graal Framework
Partial evaluation: partial evaluator and method JIT

```
def max(a, b):
    if a > b:
        return a
    else:
        return b
```
Meta-JIT approaches: meta-tracing and partial evaluation

RPython Framework
Meta-tracing: meta-interpreter and tracing JIT

Linear JIT IR

```python
guard_type(a, int)
guard_type(b, int)
c = int_gt(a, b)
guard_true(c)
return(a)
```

Truffle/Graal Framework
Partial evaluation: partial evaluator and method JIT

```
def max(a, b):
    if a > b:
        return a
    else:
        return b
```

```
guard_type(a, int)
guard_type(b, int)
c = int_gt(a, b)
jump_if_false(c, L1)
return(a)
L1: return(b)
```
Meta-JIT approaches: meta-tracing and partial evaluation

RPython Framework
Meta-tracing: meta-interpreter and tracing JIT

Linear JIT IR

```python
guard_type(a, int)
guard_type(b, int)
c = int_gt(a, b)
guard_true(c)
return(a)
```

Bridge (a <= b)

```python
return(b)
```

Truffle/Graal Framework
Partial evaluation: partial evaluator and method JIT

JIT IR

```python
def max(a, b):
    if a > b:
        return a
    else:
        return b
```

```python
guard_type(a, int)
guard_type(b, int)
c = int_gt(a, b)
jump_if_false(c, L1)
return(a)
L1: return(b)
```
Meta-JIT approaches: meta-tracing and partial evaluation

**RPython Framework**
Meta-tracing: meta-interpreter and tracing JIT

```
def max(a, b):
    if a > b:
        return a
    else:
        return b
```

**Truffle/Graal Framework**
Partial evaluation: partial evaluator and method JIT

```
def max(a, b):
    if a > b:
        return a
    else:
        return b
```

---

**Linear JIT IR**
- `guard_type(a, int)`
- `guard_type(b, int)`
- `c = int_gt(a, b)`
- `guard_true(c)`
- `return(a)`

**Bridge (a <= b)**
- `return(b)`

**JIT IR**
- `guard_type(a, int)`
- `guard_type(b, int)`
- `c = int_gt(a, b)`
- `jump_if_false(c, L1)`
- `return(a)`
- `L1: return(b)`

**Bridge (float)**
- `guard_type(a, float)`
- `guard_type(b, float)`
- `c = float_gt(a, b)`
- `guard_true(c)`
- `return(a)`

---

**Motivation**

- Meta-tracing
- PyPy >> CPython
- PyPy << C
Meta-JIT approaches: meta-tracing and partial evaluation

**RPython Framework**
Meta-tracing: meta-interpreter and tracing JIT

```python
def max(a, b):
    if a > b:
        return a
    else:
        return b
```

**Truffle/Graal Framework**
Partial evaluation: partial evaluator and method JIT

```python
i = is_type(a, int)
jump_if_false(i, L2)
guard_type(b, int)
c = int_gt(a, b)
jump_if_false(c, L1)
return(a)
L1: return(b)
L2: guard_type(a, float)
guard_type(b, float)
c = float_gt(a, b)
jump_if_false(c, L3)
return(a)
L3: return(b)
```

Motivation
• Meta-tracing
• PyPy >> CPython
• PyPy << C
Cross-layer workload characterization of meta-tracing JIT VMs

PyPy >> CPython
Cross-layer workload characterization of meta-tracing JIT VMs

PyPy >> CPython

- How can meta-tracing JITs significantly improve the performance of multiple dynamic languages?
Cross-layer workload characterization of meta-tracing JIT VMs

PyPy >> CPython

- How can meta-tracing JITs significantly improve the performance of multiple dynamic languages?

PyPy << C
Cross-layer workload characterization of meta-tracing JIT VMs

PyPy >> CPython

- How can meta-tracing JITs significantly improve the performance of multiple dynamic languages?

PyPy << C

- Why are meta-tracing JITs for dynamic programming still slower than C?
Python-based interpreter

Application: FooLang
Python-based interpreter

Application: *FooLang*

```python
b += a
```

Motivation
- Meta-tracing
- PyPy >> CPython
- PyPy << C
Python-based interpreter

Application: FooLang

compile

Application: Bytecode

\[ b += a \]

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
Python-based interpreter

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
Python-based interpreter

Application: FooLang
compile

Application: Bytecode
interpret

Interpreter: Python

\[ b += a \]

```
...
  21 LOAD_FAST          1 (b)
  24 LOAD_FAST          0 (a)
  27 INPLACE_ADD
  28 STORE_FAST         1 (b)
...
```
Python-based interpreter

Application: **FooLang**

- compile

Application: **Bytecode**

- interpret

Interpreter: **Python**

```
... 
b += a
...
```

```
21 LOAD_FAST 1 (b)
24 LOAD_FAST 0 (a)
27 INPLACE_ADD
28 STORE_FAST 1 (b)
... 
```

while True:
    `bc = bcs[bci]`
    `bci += bc.length`
    if `bc.type == INPLACE_ADD`:
        `v1 = stack.pop()`
        `v2 = stack.pop()`
        if `(type(v1) == int and type(v2) == int)`:  
            `stack.push(v1 + v2)`
        `elif ...`
    `elif bc.type == LOAD_FAST`:
        `stack.push(local[bc.varnum])`
    ...
Python-based interpreter

Application: *FooLang*

Application: *Bytecode*

Interpreter: *Python*

```
... 
   b += a 
...
```

```
21 LOAD_FAST 1 (b)
24 LOAD_FAST 0 (a)
27 INPLACE_ADD
28 STORE_FAST 1 (b)
... 
```

```
while True:
    bc = bcs[bci]
    bci += bc.length
    if bc.type == INPLACE_ADD:
        v1 = stack.pop()
        v2 = stack.pop()
        if (type(v1) == int and
            type(v2) == int):
            stack.push(v1 + v2)
        elif ...
    elif bc.type == LOAD_FAST:
        stack.push(local[bc.varnum])
... 
```
Python-based interpreter

Application: FooLang
compile

Application: Bytecode
interpret

Interpreter: Python
compile

Interpreter: Bytecode
interpret

Interpreter Interpreter

... b += a ...

while True:
    bc = bcs[bci]
    bci += bc.length
    if bc.type == INPLACE_ADD:
        v1 = stack.pop()
        v2 = stack.pop()
        if (type(v1) == int and
            type(v2) == int):
            stack.push(v1 + v2)
        elif ...
        elif bc.type == LOAD_FAST:
            stack.push(local[bc.varnum])
    ...

21 LOAD_FAST 1 (b)
24 LOAD_FAST 0 (a)
27 INPLACE_ADD
28 STORE_FAST 1 (b)
RPython Framework

Application: *Python*

compile

Application: *Bytecode*
RPython Framework

Application: Python

compile

Application: Bytecode

Interpreter: RPython

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
RPython Framework

- Application: Python
- Interpreter: RPython
- Framework: RPython

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
RPython Framework

Application: Python

compile

Application: Bytecode

Interpreter: RPython

translate

Interpreter + Framework: C

Framework: RPython

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
RPython Framework

Application: Python

Application: Bytecode

PyPy: Binary

Interpreter: RPython

Framework: RPython

Interpreter + Framework: C

compile

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
RPython Framework

Application: Python
compile
Application: Bytecode
interpret
PyPy: Binary

Interpreter: RPython
translate
Interpreter + Framework: C
compile
Meta-trace: JIT IR
trace and optimize
assemble
JIT-ed code: Binary

Motivation
Meta-tracing
PyPy >> CPython
PyPy << C
Meta-trace

Application bytecode

```
... 21 LOAD_FAST 1 (b)  
24 LOAD_FAST 0 (a)  
27 INPLACE_ADD  
28 STORE_FAST 1 (b)  
...```

Interpreter

```
while True:  
    bc = bcs[bci]  
    bci += bc.length  
    if bc.type == INPLACE_ADD:  
        v1 = stack.pop()  
        v2 = stack.pop()  
        if (type(v1) == int and  
            type(v2) == int):  
            stack.push(v1 + v2)  
        elif ...  
    elif bc.type == LOAD_FAST:  
        stack.push(local[bc.varnum])  
...```
Meta-trace

Application bytecode

while True:
    bc = bcs[bci]
    bci += bc.length
    if bc.type == INPLACE_ADD:
        v1 = stack.pop()
        v2 = stack.pop()
        if (type(v1) == int and type(v2) == int):
            stack.push(v1 + v2)
        else:
            ...  
    elif bc.type == LOAD_FAST:
        stack.push(local[bc.varnum])
    ...  

Interpreter
Meta-trace

Application bytecode

```
... 
21 LOAD_FAST    1 (b) 
24 LOAD_FAST    0 (a) 
27 INPLACE_ADD  
28 STORE_FAST   1 (b)  
... 
```

Interpreter

```
while True:
    bc = bcs[bci]
    bci += bc.length
    if bc.type == INPLACE_ADD:
        v1 = stack.pop()
        v2 = stack.pop()
        if (type(v1) == int and
            type(v2) == int):
            stack.push(v1 + v2)
        elif ...
        elif bc.type == LOAD_FAST:
            stack.push(local[bc.varnum])
    ...
```
Meta-trace

Application bytecode

...  
21  LOAD_FAST  1 (b)  
24  LOAD_FAST  0 (a)  
27  INPLACE_ADD  
28  STORE_FAST  1 (b)  
...  

Interpreter

while True:  
    bc = bcs[bci]  
    bci += bc.length  
    if bc.type == INPLACE_ADD:  
        v1 = stack.pop()  
        v2 = stack.pop()  
        if (type(v1) == int and  
            type(v2) == int):  
            stack.push(v1 + v2)  
    elif ...  
elif bc.type == LOAD_FAST:  
    stack.push(local[bc.varnum])  
...
Meta-trace

Application bytecode

```
...  
21 LOAD_FAST 1 (b)  
24 LOAD_FAST 0 (a)  
27 INPLACE_ADD  
28 STORE_FAST 1 (b)  
...  
```

Interpreter

```
while True:
    bc = bcs[bci]
    bci += bc.length
    if bc.type == INPLACE_ADD:
        v1 = stack.pop()
        v2 = stack.pop()
        if (type(v1) == int and
            type(v2) == int):
            stack.push(v1 + v2)
        elif ...
    elif bc.type == LOAD_FAST:
        stack.push(local[bc.varnum])
    ...
```
Meta-trace

Application bytecode

```python
... 21 LOAD_FAST 1 (b)
24 LOAD_FAST 0 (a)
27 INPLACE_ADD
28 STORE_FAST 1 (b)
...```

Interpreter

```python
while True:
    bc = bcs[bci]
    bci += bc.length
    if bc.type == INPLACE_ADD:
        v1 = stack.pop()
        v2 = stack.pop()
        if (type(v1) == int and
            type(v2) == int):
            stack.push(v1 + v2)
        elif ...
        elif bc.type == LOAD_FAST:
            stack.push(local[bc.varnum])
    ...
```

Meta-interpreter

```python
... p1 = getarrayitem(p0, 1)
...```
Meta-trace

Application bytecode

```
... 21 LOAD_FAST 1 (b)
24 LOAD_FAST 0 (a)
27 INPLACE_ADD
28 STORE_FAST 1 (b)
...
```

Interpreter

```
while True:
    bc = bcs[bci]
    bci += bc.length
    if bc.type == INPLACE_ADD:
        v1 = stack.pop()
        v2 = stack.pop()
        if (type(v1) == int and
            type(v2) == int):
            stack.push(v1 + v2)
        elif ...
    elif bc.type == LOAD_FAST:
        stack.push(local[bc.varnum])
    ...
```
while True:
    bc = bcs[bci]
    bci += bc.length
    if bc.type == INPLACE_ADD:
        v1 = stack.pop()
        v2 = stack.pop()
        if (type(v1) == int and
            type(v2) == int):
            stack.push(v1 + v2)
        elif ...
        elif bc.type == LOAD_FAST:
            stack.push(local[bc.varnum])
    ...

Meta-code

while True:
    bc = bcs[bci]
    bci += bc.length
    if bc.type == INPLACE_ADD:
        v1 = stack.pop()
        v2 = stack.pop()
        if (type(v1) == int and
            type(v2) == int):
            stack.push(v1 + v2)
        elif ...
        elif bc.type == LOAD_FAST:
            stack.push(local[bc.varnum])
    ...

Meta-interpreter
while True:
    bc = bcs[bci]
    bci += bc.length
    if bc.type == INPLACE_ADD:
        v1 = stack.pop()
        v2 = stack.pop()
        if (type(v1) == int and type(v2) == int):
            stack.push(v1 + v2)
        elif ...
    elif bc.type == LOAD_FAST:
        stack.push(local[bc.varnum])
    ...

Meta-trace

...
Meta-trace

Application bytecode

```python
... 21 LOAD_FAST 1 (b) 24 LOAD_FAST 0 (a) 27 INPLACE_ADD 28 STORE_FAST 1 (b)
...```

Interpreter

```python
while True:
    bc = bcs[bci]
    bci += bc.length
    if bc.type == INPLACE_ADD:
        v1 = stack.pop()
        v2 = stack.pop()
        if (type(v1) == int and type(v2) == int):
            stack.push(v1 + v2)
        elif ...
    elif bc.type == LOAD_FAST:
        stack.push(local[bc.varnum])
...```

Meta-interpreter

```
... p1 = getarrayitem(p0, 1) p2 = getarrayitem(p0, 0) guard_class(p1, int) guard_class(p2, int)
...```
while True:
    bc = bcs[bci]
    bci += bc.length
    if bc.type == INPLACE_ADD:
        v1 = stack.pop()
        v2 = stack.pop()
        if (type(v1) == int and type(v2) == int):
            stack.push(v1 + v2)
    elif ...
    elif bc.type == LOAD_FAST:
        stack.push(local[bc.varnum])
    ...

...
Application bytecode

...  
21 LOAD_FAST 1 (b)  
24 LOAD_FAST 0 (a)  
27 INPLACE_ADD  
28 STORE_FAST 1 (b)  
...  

Interpreter

while True:
    bc = bcs[bci]  
    bci += bc.length  
    if bc.type == INPLACE_ADD:
        v1 = stack.pop()  
        v2 = stack.pop()  
        if (type(v1) == int and  
            type(v2) == int):
            stack.push(v1 + v2)  
        elif ...  
    elif bc.type == LOAD_FAST:
        stack.push(local[bc.varnum])  
...  

Meta-interpreter

Meta-trace

...  
  p1 = getarrayitem(p0, 1)  
  p2 = getarrayitem(p0, 0)  
  guard_class(p1, int)  
  guard_class(p2, int)  
  i3 = getfield(p1, intval)  
  i4 = getfield(p2, intval)  
  i5 = int_add_ovf(i3, i4)  
  guard_no_overflow()  
...  

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
Meta-trace

Application bytecode

```
...  
21 LOAD_FAST 1 (b)  
24 LOAD_FAST 0 (a)  
27 INPLACE_ADD     
28 STORE_FAST 1 (b) 
...  
```

Interpreter

```
while True:
    bc = bcs[bci]
    bci += bc.length
    if bc.type == INPLACE_ADD:
        v1 = stack.pop()
        v2 = stack.pop()
        if (type(v1) == int and type(v2) == int):
            stack.push(v1 + v2)
        elif ...
        elif bc.type == LOAD_FAST:
            stack.push(local[bc.varnum])
    ...  
```

Meta-interpreter

```
...  
p1 = getarrayitem(p0, 1)  
p2 = getarrayitem(p0, 0)  
guard_class(p1, int)  
i3 = getfield(p1, intval)  
i4 = getfield(p2, intval)  
i5 = int_add_ovf(i3, i4)  
guard_no_overflow()  
...  
```

Deoptimization back to interpreter on guard failure
Cross-layer annotations

Application: Python
compile
Application: Bytecode
interpret
PyPy: Binary

Interpreter: RPython
translate
Interpreter + Application: C
compile
trace and optimize
Meta-trace: JIT IR
assemble
JIT-ed code: Binary

Framework: RPython
Cross-layer annotations

Application: Python

Interpreter: RPython

Framework: RPython

Application: Bytecode

Interpreter + Application: C

Meta-trace: JIT IR

JIT-ed code: Binary

PyPy: Binary

compile

interpret

translate

compile

trace and optimize

assemble

application annotations
Cross-layer annotations

**Application annotations**
- Application: *Python*
  - compile
  - interpret
- Application: *Bytecode*

**Interpreter annotations**
- Interpreter: *RPython*
  - translate
- Interpreter + Application: *C*
  - compile
  - trace and optimize
- Meta-trace: *JIT IR*
  - assemble
  - JIT-ed code: *Binary*
Cross-layer annotations

- **Application annotations**
  - Application: *Python*
  - Application: *Bytecode*
  - PyPy: *Binary*

- **Interpreter annotations**
  - Interpreter: *RPython*
  - Interpreter + Application: *C*
  - Meta-trace: *JIT IR*
  - JIT-ed code: *Binary*

- **Framework annotations**
  - Framework: *RPython*
Cross-layer annotations

application annotations

Application: *Python*

compile

Application: *Bytecode*

interpret

PyPy: *Binary*

interpreter annotations

Interpreter: *RPython*

translate

Interpreter + Application: *C*

compile

Meta-trace: *JIT IR*

trace and optimize

JIT-ed code: *Binary*

assemble

IR node of interest

asm of interest

framework annotations

Framework: *RPython*
Cross-layer annotations

application annotations

Application: Python

compile

Application: Bytecode

interpret

PyPy: Binary

interpreter annotations

Interpreter: RPython

translate

Framework: RPython

PyPy >> CPython • PyPy << C

PyPy << C

framework annotations

Interpreter + Application: C

Meta-trace: JIT IR

trace and optimize

IR node of interest

perf counters

using PAPI

assemble

JIT-ed code: Binary

asm of interest

Meta-tracing

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
Cross-layer annotations

**Application annotations**
- Application: Python
  - compile
  - Application: Bytecode
  - interpret
- PyPy: Binary
  - perf counters using PAPI
  - Dynamic Binary Instrumentation

**Interpreter annotations**
- Interpreter: RPython
  - translate
  - Meta-trace: JIT IR
  - assemble
  - JIT-ed code: Binary
  - trace and optimize

**Framework annotations**
- Framework: RPython

**Motivation**
- Meta-tracing
- PyPy >> CPython
- PyPy << C
Cross-layer annotations

- Application: Python
  - compile
  - interpret
- Application: Bytecode
- PyPy: Binary
  - perf counters using PAPI
  - IR node of interest
  - phase counters, IR node counters

- Interpreter: RPython
  - translate
- Interpreter + Application: C
  - compile
  - trace and optimize
- Meta-trace: JIT IR
  - assemble
  - asm of interest

- Framework: RPython
  - Dynamic Binary Instrumentation

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
Cross-layer workload characterization of meta-tracing JIT VMs

PyPy >> CPython
- How can meta-tracing JITs significantly improve the performance of multiple dynamic languages?

PyPy << C
- Why are meta-tracing JITs for dynamic programming still slower than C?
PyPy with meta-tracing JIT speedup over CPython: Meta-tracing JIT improves the performance significantly
PyPy speedup over CPython and Pycket speedup over Racket: Meta-tracing JIT improves performance significantly across multiple languages
PyPy speedup over CPython and Pycket speedup over Racket: Meta-tracing JIT improves performance significantly across multiple languages.

- PyPy speedup
- Pycket speedup

**Motivation**
- Meta-tracing
- PyPy >> CPython
- PyPy << C
Meta-tracing JIT VM phases

calls to AOT funs
JIT
GC
deoottimization
tracing & opt
interpreter

0                      2B                   4B                    6B                    8B                 10B
instructions

richards

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
Meta-tracing JIT VM phases

**richards**

- Calls to AOT funs
- JIT
- GC
- Deoptimization
- Tracing & opt
- Interpreter

**sympy_str**

- Calls to AOT funs
- JIT
- GC
- Deoptimization
- Tracing & opt
- Interpreter
# Meta-tracing JIT VM phases

| Motivation • Meta-tracing • PyPy >> CPython • PyPy << C |

## Fastest on PyPy

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>richards</td>
<td>1.00</td>
</tr>
<tr>
<td>crypto_pyaes</td>
<td>0.98</td>
</tr>
<tr>
<td>chaos</td>
<td>0.97</td>
</tr>
<tr>
<td>telco</td>
<td>0.96</td>
</tr>
<tr>
<td>spectral_norm</td>
<td>0.94</td>
</tr>
<tr>
<td>django</td>
<td>0.93</td>
</tr>
<tr>
<td>twisted_iteration</td>
<td>0.92</td>
</tr>
<tr>
<td>spifire_cstringio</td>
<td>0.91</td>
</tr>
<tr>
<td>raytrace-simple</td>
<td>0.90</td>
</tr>
<tr>
<td>hexiom2</td>
<td>0.89</td>
</tr>
<tr>
<td>float</td>
<td>0.88</td>
</tr>
<tr>
<td>eparse</td>
<td>0.87</td>
</tr>
<tr>
<td>sympy_expand</td>
<td>0.86</td>
</tr>
<tr>
<td>slowspifire</td>
<td>0.85</td>
</tr>
<tr>
<td>sympy_integrate</td>
<td>0.84</td>
</tr>
</tbody>
</table>

## Slowest on PyPy

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>pidigs</td>
<td>0.00</td>
</tr>
<tr>
<td>bm_mdp</td>
<td>0.00</td>
</tr>
<tr>
<td>sympy_str</td>
<td>0.00</td>
</tr>
</tbody>
</table>

## Phase Breakdown

- **JIT calls**
- **JIT**
- **GC**
- **deopt**
- **tracing**
- **interp**
The JIT phase:
The fastest benchmarks tend to execute JIT-compiled code the most

- **Motivation**
  - Meta-tracing

- **Comparison**
  - PyPy >> CPython
  - PyPy << C
Meta-tracing inlines all loops and can hurt performance

Interpreter

```python
while True:
    ...
    memcpy(d, s, n)
    ...

def memcpy(dest, src, n):
    i = 0
    while i < n:
        dest[i] = src[i]
        i += 1
```
Meta-tracing inlines all loops and can hurt performance

Interpreter

```
while True:
    ...
    memcpy(d, s, n)
    ...

def memcpy(dest, src, n):
    i = 0
    while i < n:
        dest[i] = src[i]
        i += 1
```

Meta-interpreter

Meta-trace
Meta-tracing inlines all loops and can hurt performance

**Interpreter**

```python
while True:
    ...
    memcpy(d, s, n)
    ...

def memcpy(dest, src, n):
    i = 0
    while i < n:
        dest[i] = src[i]
        i += 1
```

**Meta-interpreter**

**Meta-trace**

```python
...  
guard_gt(i0, 0)
  i3 = getarrayitem(p1, 0)
  setarrayitem(p2, 0, i3)
```
Meta-tracing inlines all loops and can hurt performance

Interpreter

```python
while True:
    ...
    memcpy(d, s, n)
    ...

def memcpy(dest, src, n):
    i = 0
    while i < n:
        dest[i] = src[i]
        i += 1
```

Meta-interpreter

```
... guard_gt(i0, 0)
  i3 = getarrayitem(p1, 0)
  setarrayitem(p2, 0, i3)
  guard_gt(i0, 1)
  i4 = getarrayitem(p1, 1)
  setarrayitem(p2, 1, i4)
```
Meta-tracing inlines all loops and can hurt performance

**Interpreter**

```python
while True:
    ...
    memcpy(d, s, n)
    ...

def memcpy(dest, src, n):
    i = 0
    while i < n:
        dest[i] = src[i]
        i += 1
```

**Meta-interpreter**

**Meta-trace**

```python
... guard_gt(i0, 0)
i3 = getarrayitem(p1, 0)
setarrayitem(p2, 0, i3)
guard_gt(i0, 1)
i4 = getarrayitem(p1, 1)
setarrayitem(p2, 1, i4)
guard_gt(i0, 2)
i5 = getarrayitem(p1, 2)
setarrayitem(p2, 2, i5)
...```
Examples of significant AOT-compiled functions

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>%</th>
<th>Source</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ai</td>
<td>19.4</td>
<td>interpreter</td>
<td>setobject.get_storage_from_list</td>
</tr>
<tr>
<td>bm_chameleon</td>
<td>17.9</td>
<td>RPython types</td>
<td>rordereddict.ll_call_lookup_function</td>
</tr>
<tr>
<td>bm_mako</td>
<td>26.1</td>
<td>RPython lib</td>
<td>runicode.unicode_encode_ucs1_helper</td>
</tr>
<tr>
<td>json_bench</td>
<td>18.5</td>
<td>PyPy module</td>
<td>_pypyjson.raw_encode_basestring_ascii</td>
</tr>
<tr>
<td>nbody_modified</td>
<td>44.6</td>
<td>external lib</td>
<td>pow</td>
</tr>
</tbody>
</table>

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
JIT calls to AOT-compiled functions:
AOT-compiled functions can improve performance by avoiding long traces

- **Motivation**
  - Meta-tracing
  - PyPy >> CPython
  - PyPy << C
PyPy bytecode execution rate compared to CPython:
Benchmarks that perform the best also warm up the fastest
PyPy bytecode execution rate compared to CPython:
Benchmarks that perform the best also warm up the fastest
PyPy bytecode execution rate compared to CPython:
Benchmarks that perform the best also warm up the fastest

Breakeven point: the performance of the two VMs at this point is equal

-richards-

PyPy w/o JIT breakeven point
CPython breakeven point
PyPy bytecode execution rate compared to CPython:
Benchmarks that perform the best also warm up the fastest

Breakeven point: the performance of the two VMs at this point is equal

richards

html5lib
PyPy bytecode execution rate compared to CPython: Benchmarks that perform the best also warm up the fastest

Breakeven point: the performance of the two VMs at this point is equal

PyPy w/o JIT breakeven point

CPython breakeven point

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
PyPy bytecode execution rate compared to CPython: Benchmarks that perform the best also warm up the fastest

Breakeven point: the performance of the two VMs at this point is equal

PyPy w/o JIT breakeven point

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
PyPy >> CPython

- How can meta-tracing JITs significantly improve the performance of multiple dynamic languages?

PyPy << C

- Why are meta-tracing JITs for dynamic programming still slower than C?
Cross-layer workload characterization of meta-tracing JIT VMs

**PyPy >> CPython**

- How can meta-tracing JITs significantly improve the performance of multiple dynamic languages?
  - *Meta-tracing JIT compilation significantly improves the performance*

**PyPy << C**

- Why are meta-tracing JITs for dynamic programming still slower than C?
Cross-layer workload characterization of meta-tracing JIT VMs

PyPy >> CPython

- How can meta-tracing JITs significantly improve the performance of multiple dynamic languages?
  - *Meta-tracing JIT compilation significantly improves the performance*
  - *AOT-compiled functions are good to break pathological traces*

PyPy << C

- Why are meta-tracing JITs for dynamic programming still slower than C?
Cross-layer workload characterization of meta-tracing JIT VMs

PyPy >> CPython

- How can meta-tracing JITs significantly improve the performance of multiple dynamic languages?
  - *Meta-tracing JIT compilation significantly improves the performance*
  - *AOT-compiled functions are good to break pathological traces*
  - *Easier-to-JIT programs perform the best and warm up the fastest*

PyPy << C

- Why are meta-tracing JITs for dynamic programming still slower than C?
PyPy and Pycket slowdown over C/C++:
Meta-tracing JIT has a big performance gap between static languages
PyPy and Pycket slowdown over C/C++:
Meta-tracing JIT has a big performance gap between static languages
Meta-tracing JIT phases

- Benchmarks Fastest on PyPy
- Benchmarks Slowest on PyPy

JIT
JIT call to AOT functions
Meta-tracing JIT IR node breakdown:
Likely a big part of JIT compiled code is overhead

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
Meta-tracing JIT IR node breakdown:
Likely a big part of JIT compiled code is overhead

Fastest on PyPy ➔ Slowest on PyPy

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
Meta-tracing JIT IR node breakdown:
Likely a big part of JIT compiled code is overhead
Meta-tracing JIT phases

- Motivation
- Meta-tracing
- PyPy >> CPython
- PyPy << C

Benchmarks Fastest on PyPy

Benchmarks Slowest on PyPy

JIT

JIT call to AOT functions
Interpreter phase

- Benchmarks Fastest on PyPy
- Benchmarks Slowest on PyPy

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
PyPy *without* meta-tracing JIT speedup over CPython: RPython-to-C translation has overheads
Tracing and optimization phase

- Benchmarks Fastest on PyPy
- Benchmarks Slowest on PyPy

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
Deoptimization phase

- Deoptimization
Garbage collection phase

- Benchmarks Fastest on PyPy
- Benchmarks Slowest on PyPy

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C

Cornell University Computer Systems Laboratory
Meta-tracing JIT VM overheads:
Overheads are diverse and can add up to significant portion of execution
Iron law of processor performance:
Does meta-tracing VM code execute poorly in addition to more instructions?

\[
\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycle}}{\text{Instructions}} \times \frac{\text{Time}}{\text{Cycle}}
\]
Comparing meta-tracing JIT IPC to C/C++:
Meta-tracing has a similar IPC for most benchmarks

Motivation • Meta-tracing • PyPy >> CPython • PyPy << C
Comparing meta-tracing JIT IPC to C/C++:
Meta-tracing has a similar IPC for most benchmarks
IPC measurements can be accurately matched against VM phases.
Microarchitectural characterization by the VM phase:
Meta-tracing-JIT-compiled code has a similar IPC, fewer branches and mispredictions.
Microarchitectural characterization by the VM phase: Meta-tracing-JIT-compiled code has a similar IPC, fewer branches and mispredictions.
Microarchitectural characterization by the VM phase: Meta-tracing-JIT-compiled code has a similar IPC, fewer branches and mispredictions.
Cross-layer workload characterization of meta-tracing JIT VMs

**PyPy >> CPython**

- How can meta-tracing JITs significantly improve the performance of multiple dynamic languages?
  - *Meta-tracing JIT compilation significantly improves the performance*
  - *AOT-compiled functions are good to break pathological traces*
  - *Easier-to-JIT programs perform the best and warm up the fastest*

**PyPy << C**

- Why are meta-tracing JITs for dynamic programming still slower than C?
Cross-layer workload characterization of meta-tracing JIT VMs

PyPy $\gg$ CPython

- How can meta-tracing JITs significantly improve the performance of multiple dynamic languages?
  - *Meta-tracing JIT compilation significantly improves the performance*
  - *AOT-compiled functions are good to break pathological traces*
  - *Easier-to-JIT programs perform the best and warm up the fastest*

PyPy $\ll$ C

- Why are meta-tracing JITs for dynamic programming still slower than C?
  - *Meta-tracing JIT has an order of magnitude performance gap vs. C/C++*
Cross-layer workload characterization of meta-tracing JIT VMs

PyPy >> CPython

- How can meta-tracing JITs significantly improve the performance of multiple dynamic languages?
  - *Meta-tracing JIT compilation significantly improves the performance*
  - *AOT-compiled functions are good to break pathological traces*
  - *Easier-to-JIT programs perform the best and warm up the fastest*

PyPy << C

- Why are meta-tracing JITs for dynamic programming still slower than C?
  - *Meta-tracing JIT has an order of magnitude performance gap vs. C/C++*
  - *A big part of meta-tracing-JIT-compiled code is likely overhead*
Cross-layer workload characterization of meta-tracing JIT VMs

**PyPy >> CPython**

- How can meta-tracing JITs significantly improve the performance of multiple dynamic languages?
  - *Meta-tracing JIT compilation significantly improves the performance*
  - *AOT-compiled functions are good to break pathological traces*
  - *Easier-to-JIT programs perform the best and warm up the fastest*

**PyPy << C**

- Why are meta-tracing JITs for dynamic programming still slower than C?
  - *Meta-tracing JIT has an order of magnitude performance gap vs. C/C++*
  - *A big part of meta-tracing-JIT-compiled code is likely overhead*
  - *The meta-tracing JIT VM has a number of other diverse overheads*
Cross-layer workload characterization of meta-tracing JIT VMs

PyPy >> CPython

- How can meta-tracing JITs significantly improve the performance of multiple dynamic languages?
  - Meta-tracing JIT compilation significantly improves the performance
  - AOT-compiled functions are good to break pathological traces
  - Easier-to-JIT programs perform the best and warm up the fastest

PyPy << C

- Why are meta-tracing JITs for dynamic programming still slower than C?
  - Meta-tracing JIT has an order of magnitude performance gap vs. C/C++
  - A big part of meta-tracing-JIT-compiled code is likely overhead
  - The meta-tracing JIT VM has a number of other diverse overheads
  - The problem is more instructions, not instructions that execute poorly
Cross-layer workload characterization of meta-tracing JIT VMs

**PyPy >> CPython**

- How can meta-tracing JITs significantly improve the performance of multiple dynamic languages?
  - *Meta-tracing JIT compilation significantly improves the performance*
  - *AOT-compiled functions are good to break pathological traces*
  - *Easier-to-JIT programs perform the best and warm up the fastest*

**PyPy << C**

- Why are meta-tracing JITs for dynamic programming still slower than C?
  - *Meta-tracing JIT has an order of magnitude performance gap vs. C/C++*
  - *A big part of meta-tracing-JIT-compiled code is likely overhead*
  - *The meta-tracing JIT VM has a number of other diverse overheads*
  - *The problem is more instructions, not instructions that execute poorly*
  - *There is no silver bullet in addressing the performance gap*
Cross-Layer Workload Characterization of Meta-Tracing JIT VMs

Berkin Ilbeyi¹, Carl Friedrich Bolz-Tereick², and Christopher Batten¹

¹ Cornell University, ² Heinrich-Heine-Universität Düsseldorf