Cornell University
Computer Systems Laboratory

Cross-Layer Workload Characterization of Meta-Tracing JIT VMs

Berkin Ilbeyi[1], Carl Friedrich Bolz-Tereick[2], and Christopher Batten[1]

[1] Cornell University, [2] Heinrich-Heine-Universität Düsseldorf

Dynamic languages are popular

Language Rank	Types	Spectrum Ranking
1. Python		100.0
2. C		99.7
3. Java		99.4
4. C++		97.2
5. C#		88.6
6. R		88.1
7. JavaScript		85.5
8. PHP		81.4
9. Go		76.1
10. Swift		75.3

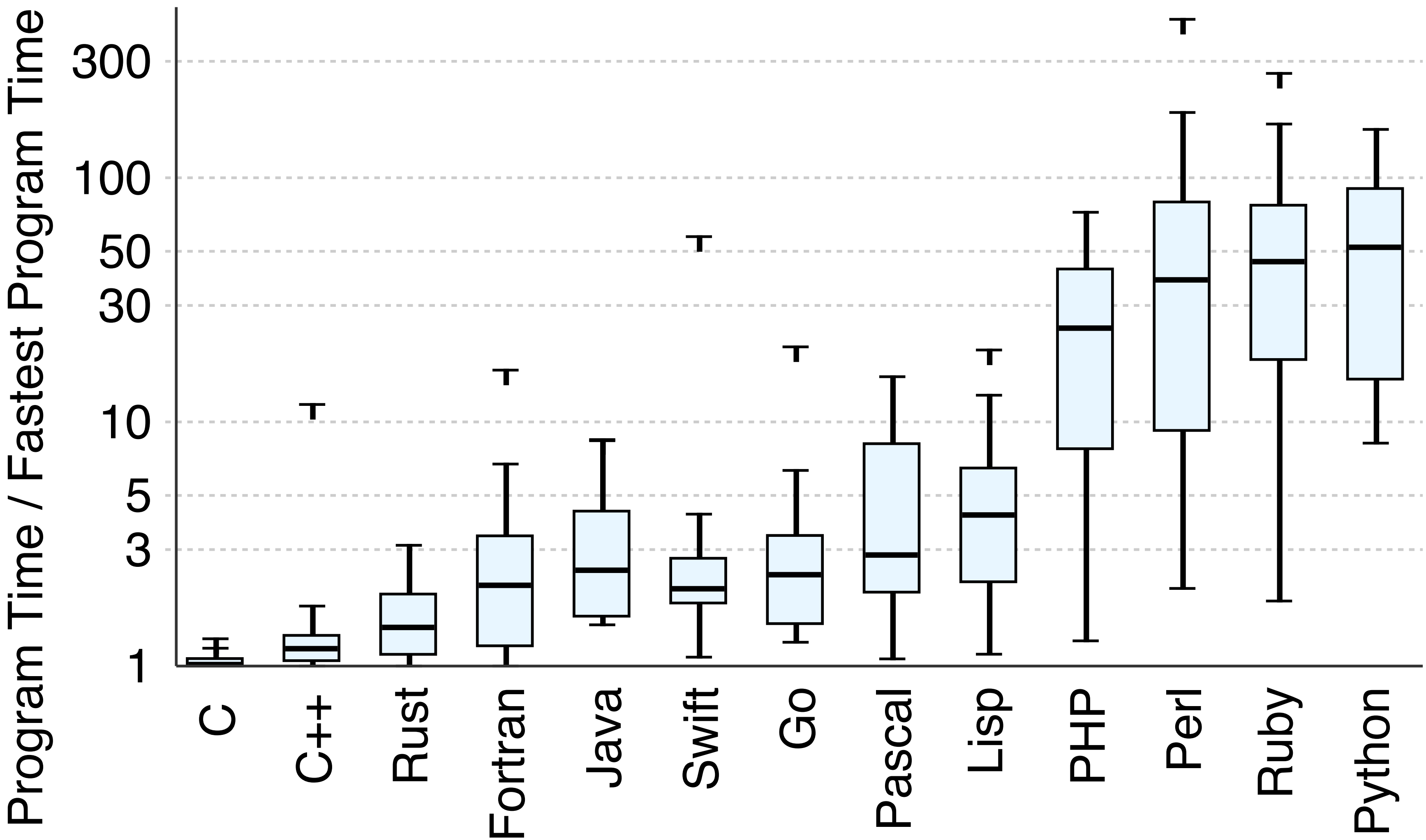
S. Cass. “The 2017 Top Programming Languages.” IEEE Spectrum.

Dynamic languages are popular

Language Rank	Types	Spectrum Ranking
1. Python	web, desktop	100.0
2. C	mobile, desktop, embedded	99.7
3. Java	web, mobile, desktop	99.4
4. C++	mobile, desktop, embedded	97.2
5. C#	web, mobile, desktop	88.6
6. R	desktop	88.1
7. JavaScript	web, mobile	85.5
8. PHP	web	81.4
9. Go	web, desktop	76.1
10. Swift	mobile, desktop	75.3

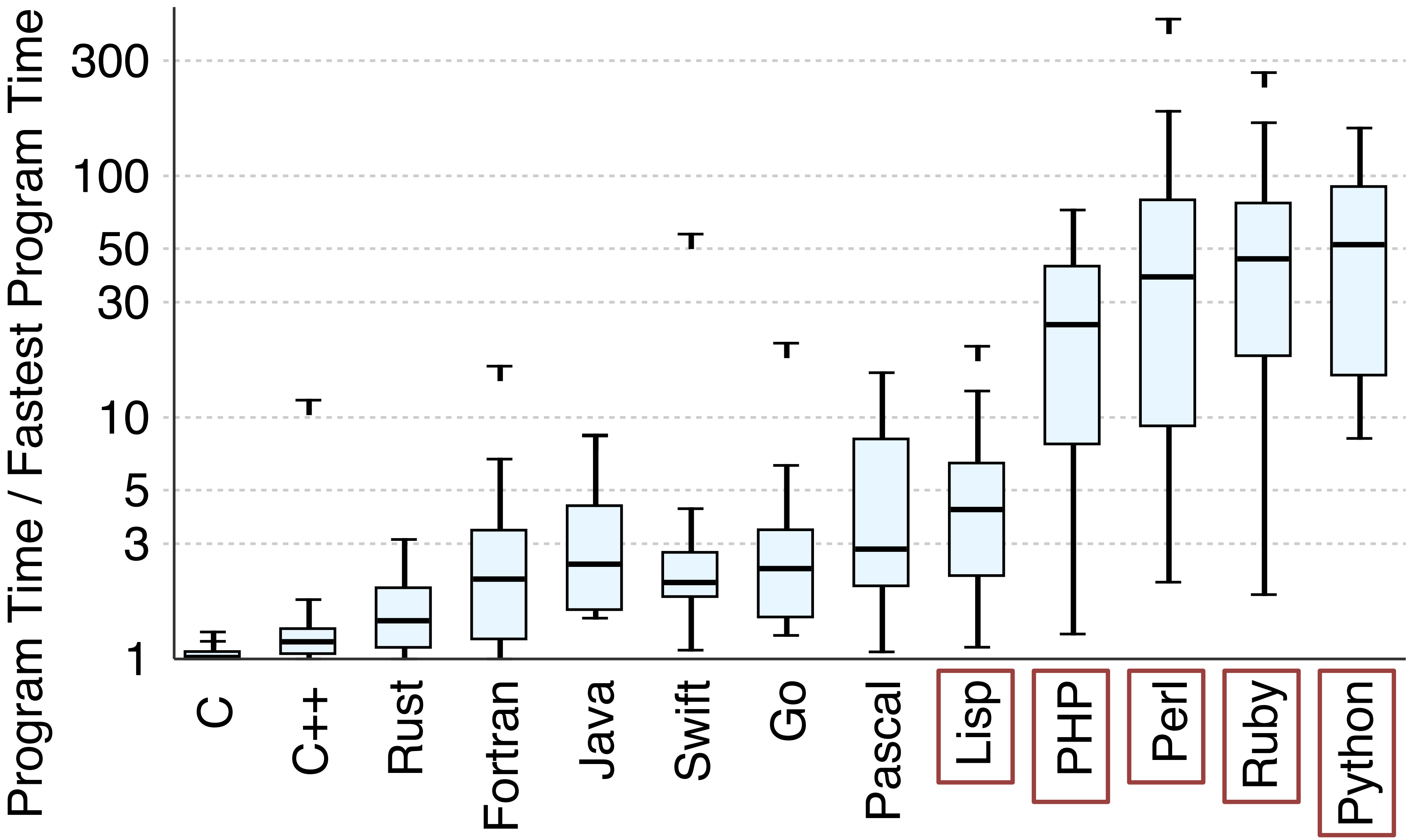
S. Cass. “The 2017 Top Programming Languages.” IEEE Spectrum.

Dynamic languages are slow



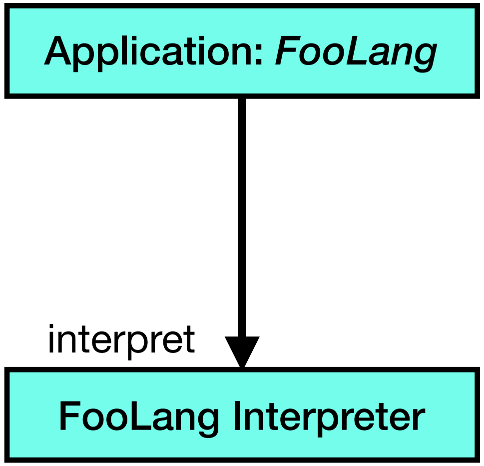
I. Guoy. “The Computer Languages Benchmarks Game.”

Dynamic languages are slow

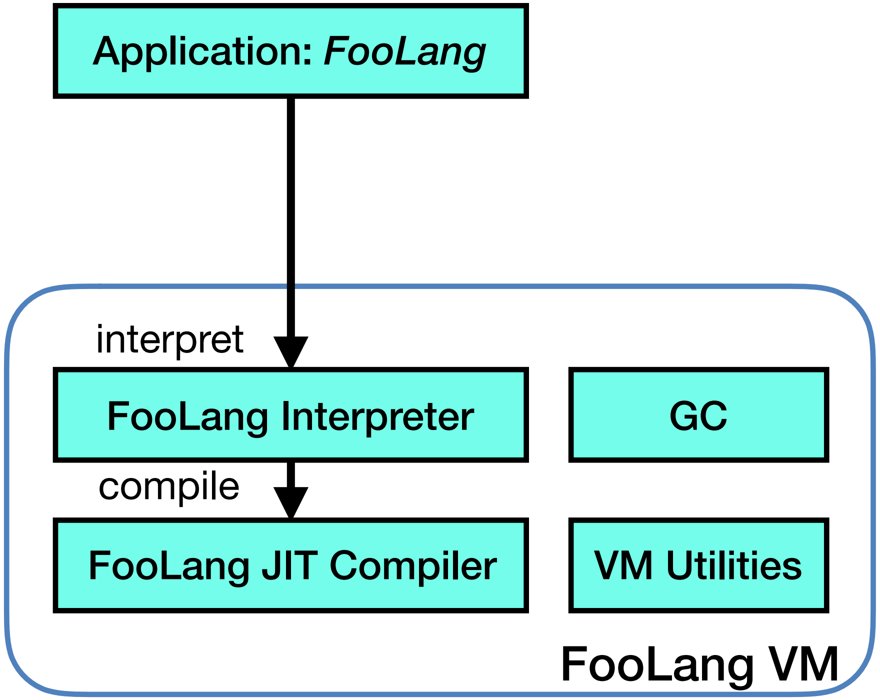


I. Guoy. "The Computer Languages Benchmarks Game."

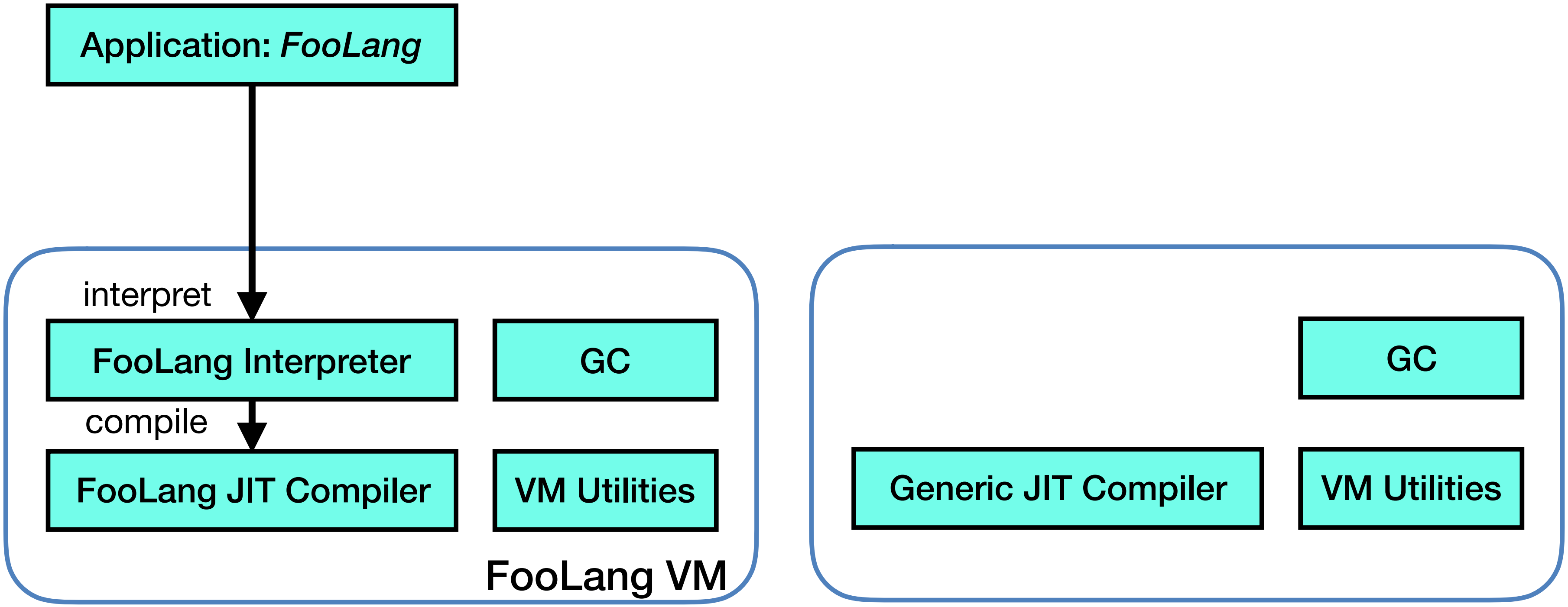
Just-in-time-compiling virtual machines



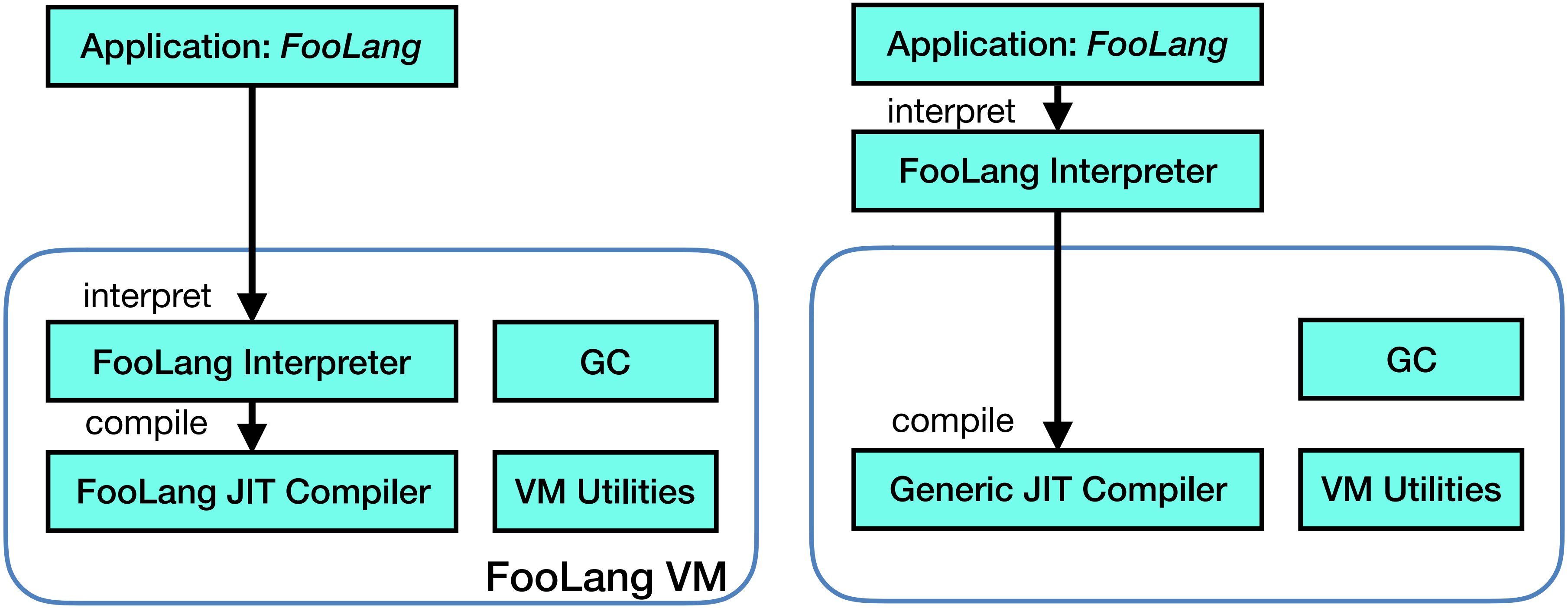
Just-in-time-compiling virtual machines



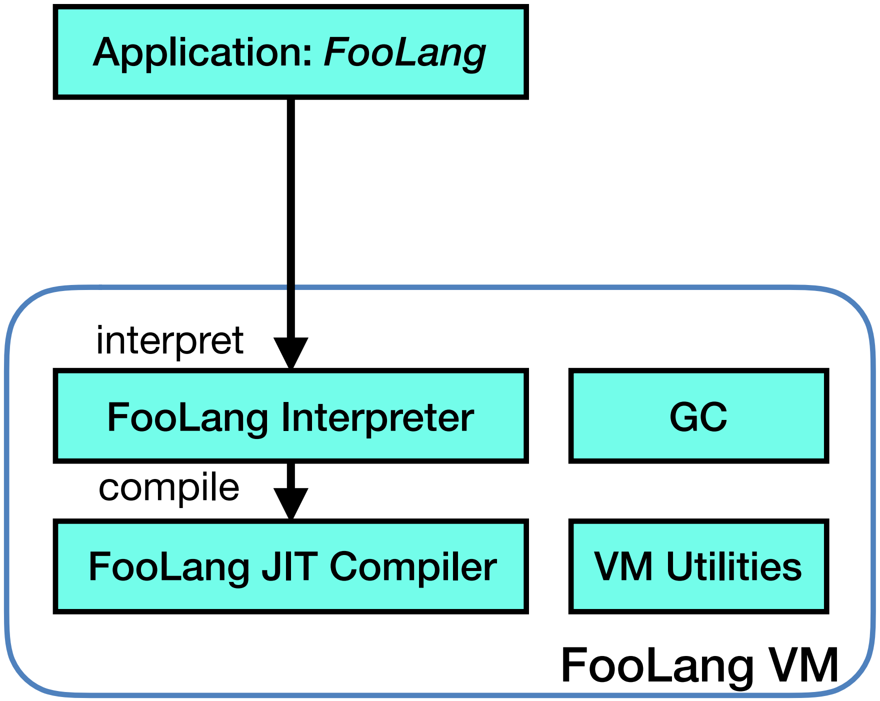
Just-in-time-compiling virtual machines

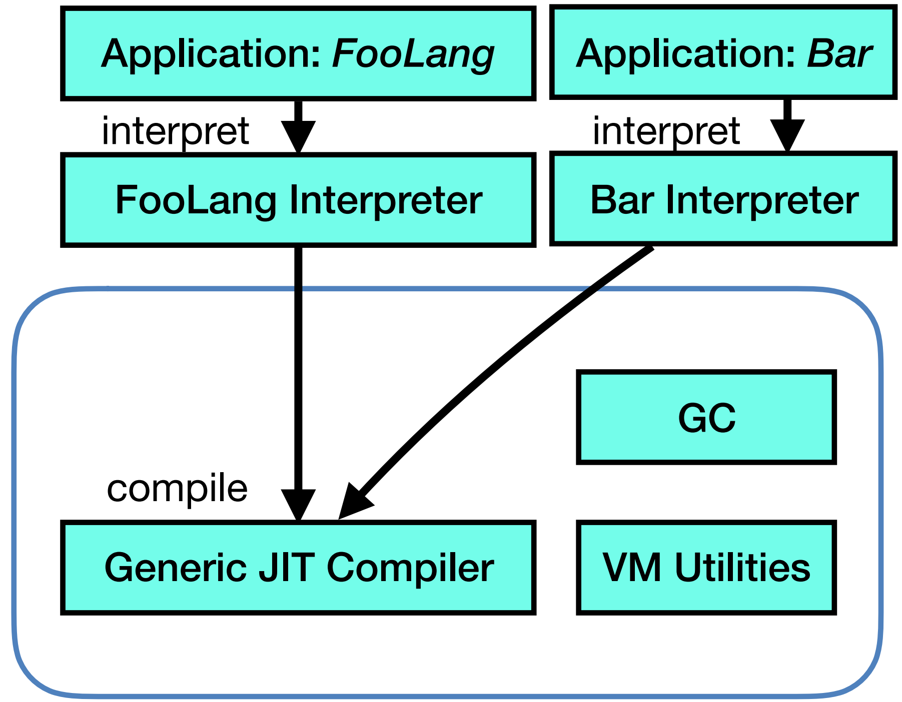


Just-in-time-compiling virtual machines

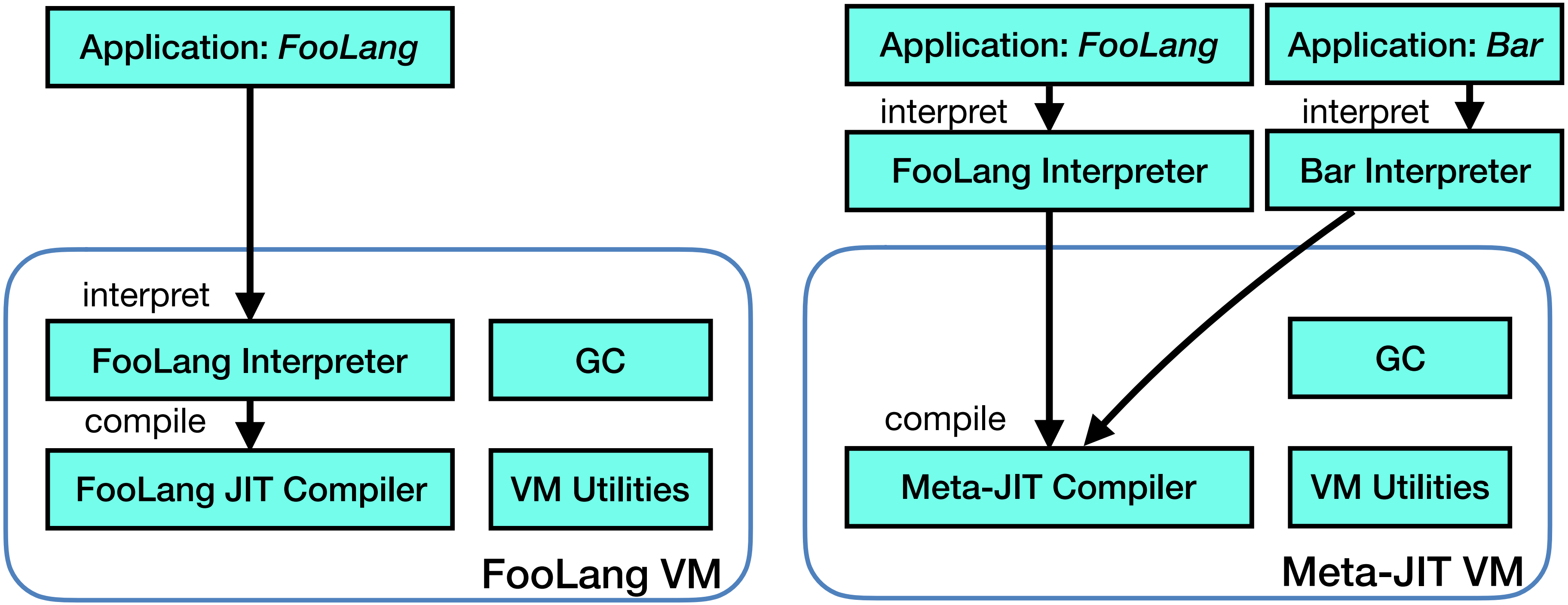


Just-in-time-compiling virtual machines





Just-in-time-compiling virtual machines



Meta-JIT approaches: meta-tracing and partial evaluation

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

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

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)
```

Meta-JIT approaches: meta-tracing and partial evaluation

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

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)
```

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)
```

Meta-JIT approaches: meta-tracing and partial evaluation

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

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)
```

Meta-JIT approaches: meta-tracing and partial evaluation

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

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)
```

Bridge (float)

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

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)
```

Meta-JIT approaches: meta-tracing and partial evaluation

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

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)
```

Bridge (float)

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

JIT IR

Re-optimized JIT IR

```
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)
```

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?**

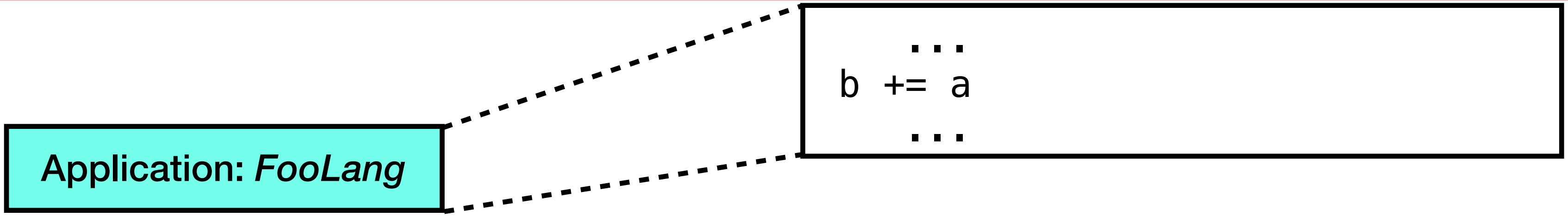
PyPy << C

- **Why are meta-tracing JITs for dynamic programming still slower than C?**

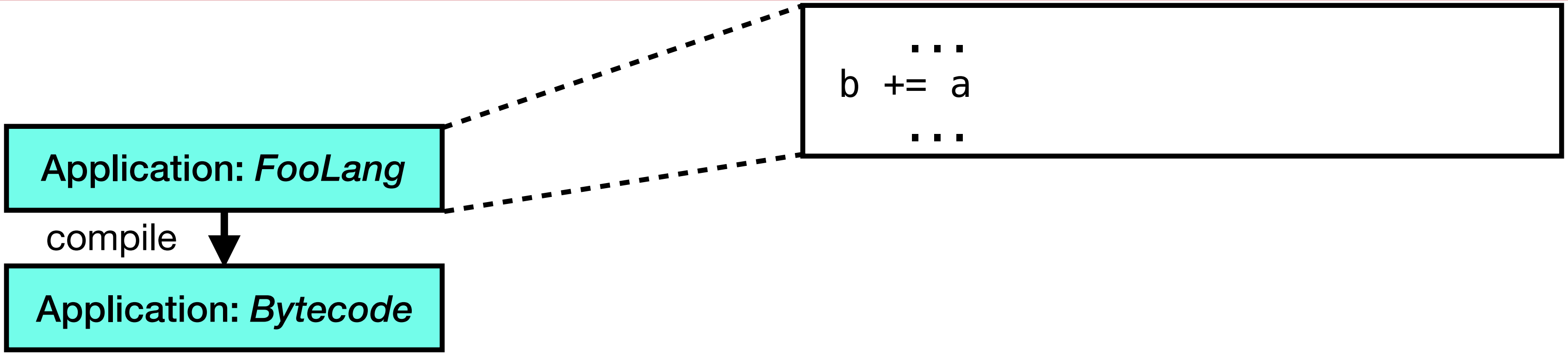
Python-based interpreter

Application: *FooLang*

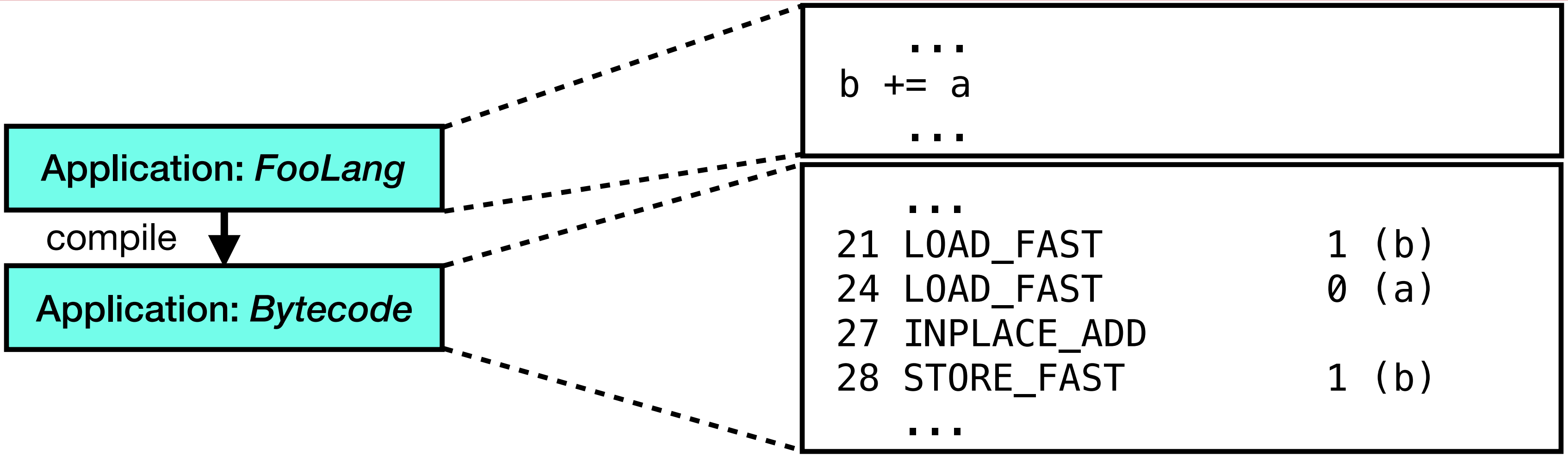
Python-based interpreter



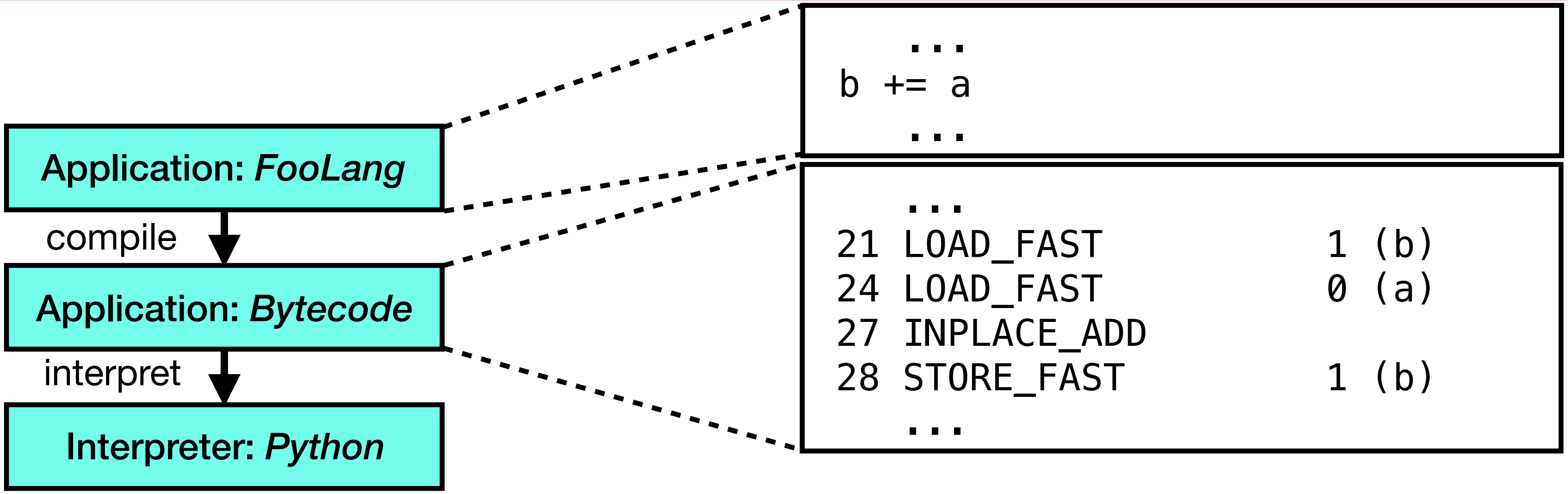
Python-based interpreter



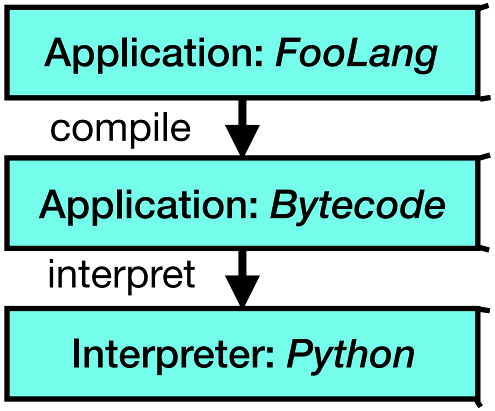
Python-based interpreter



Python-based interpreter



Python-based interpreter

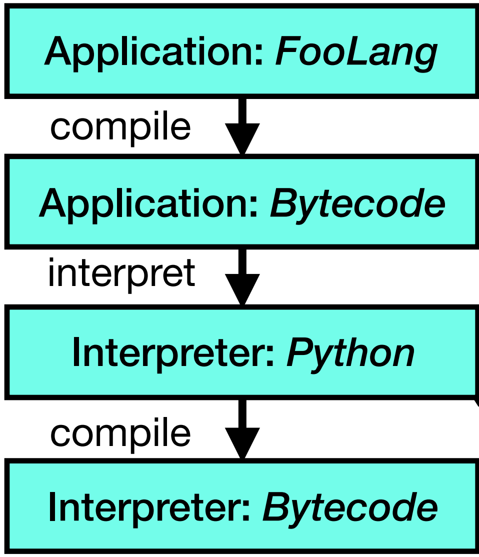



```
   ...
b += a
   ...
```

```
   ...
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

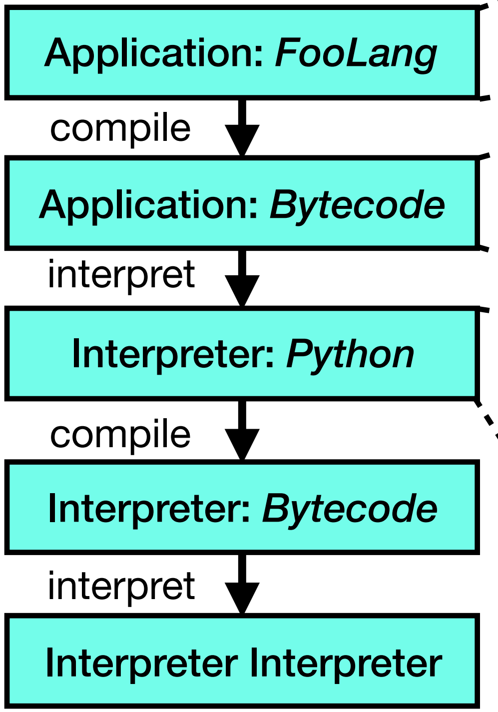



```
   ...
b += a
   ...
```

```
   ...
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

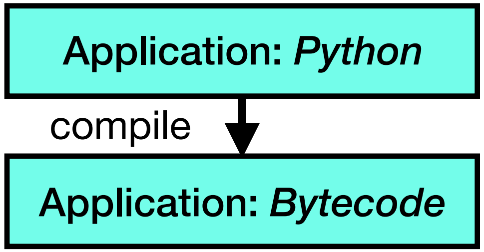



```
   ...
b += a
   ...
```

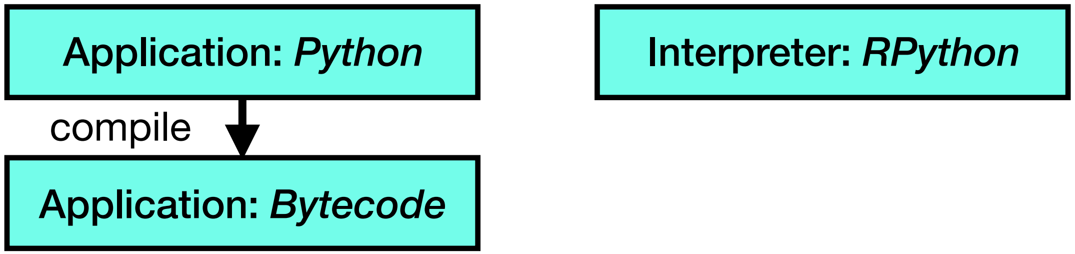
```
   ...
21 LOAD_FAST          1 (b)
24 LOAD_FAST          0 (a)
27 INPLACE_ADD
28 STORE_FAST         1 (b)
   ...
```

```
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])
  ...
```

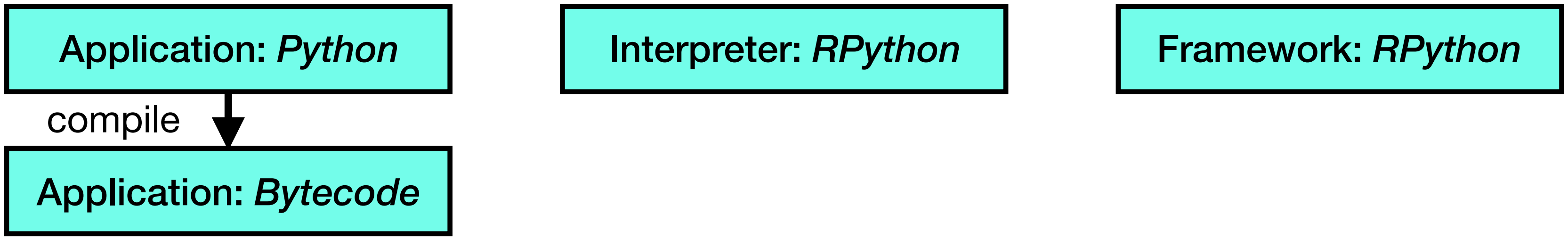
RPython Framework



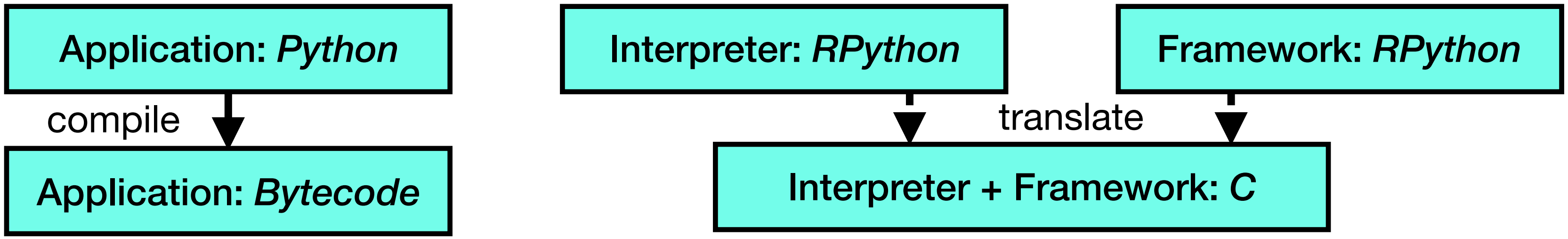
RPython Framework



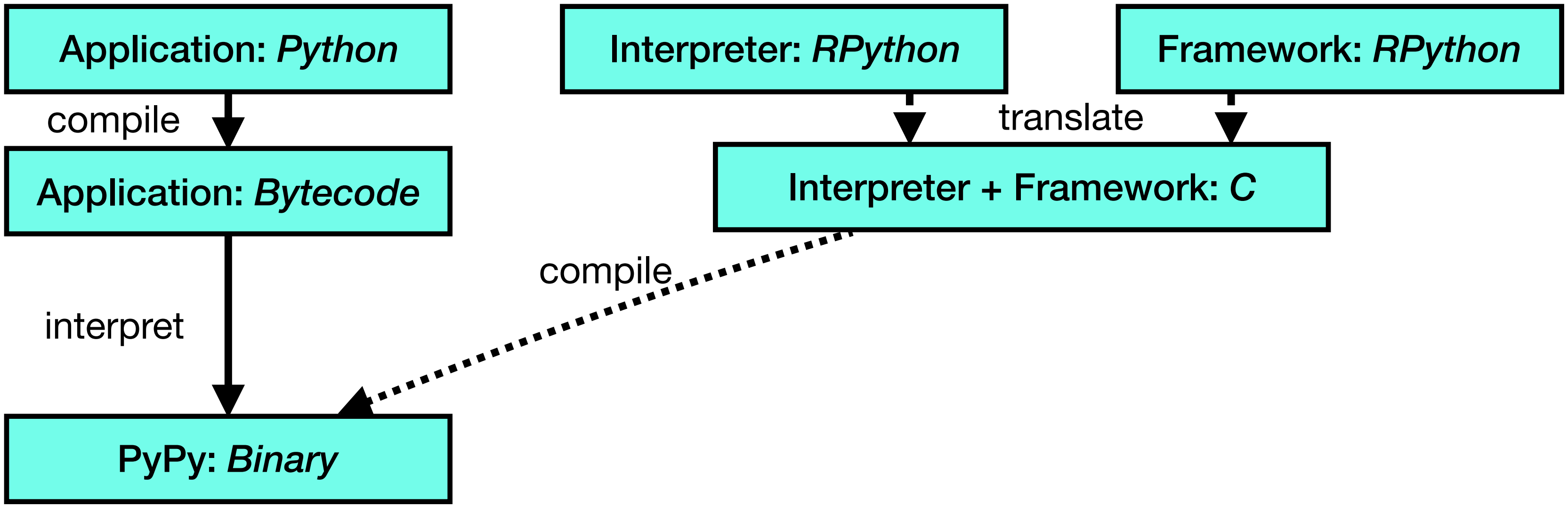
RPython Framework



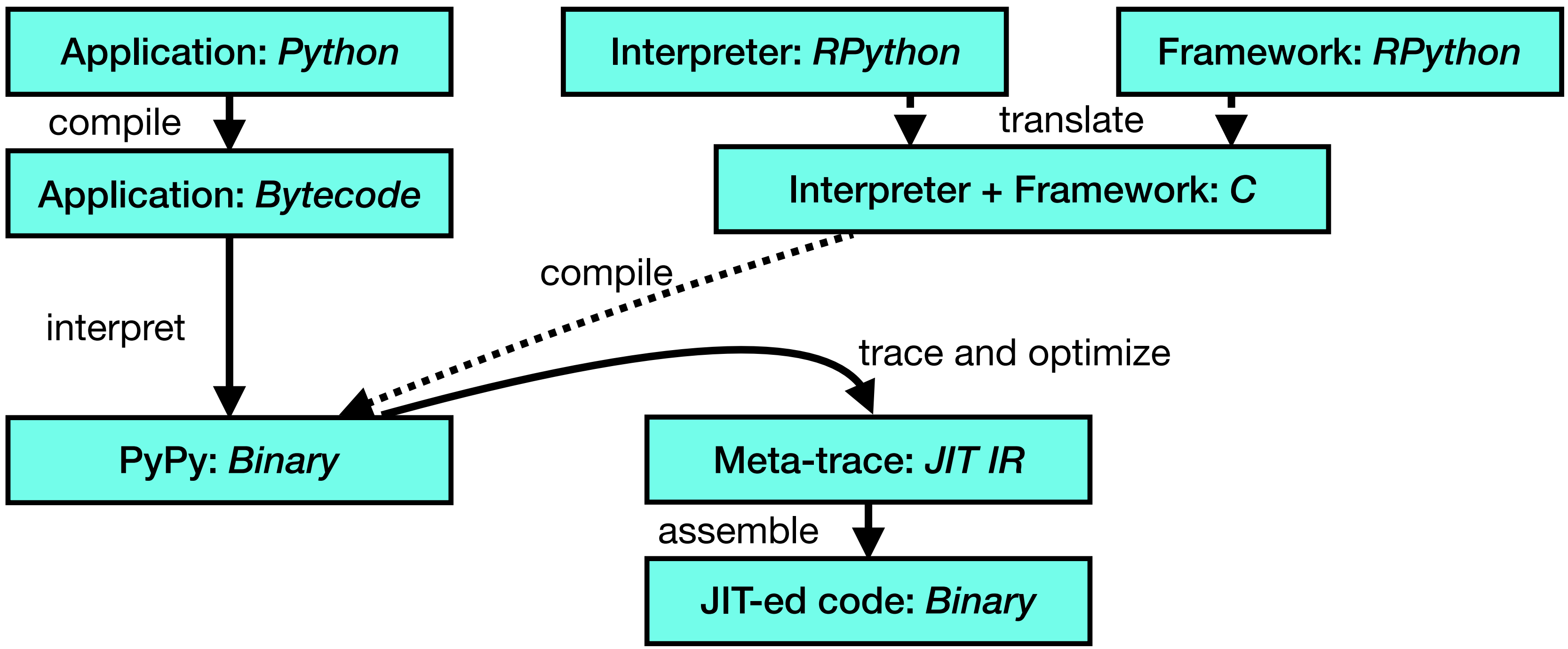
RPython Framework



RPython Framework



RPython Framework



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-interpreter

Meta-trace

```
  ...
```

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

Meta-trace

```
  ...
```

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

Meta-trace

```
  ...
```

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

Meta-trace

```
  ...
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])
  ...
```

Meta-interpreter

Meta-trace

```
  ...
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])
  ...
```

Meta-interpreter

Meta-trace

```
  ...
p1 = getarrayitem(p0, 1)
p2 = getarrayitem(p0, 0)
```

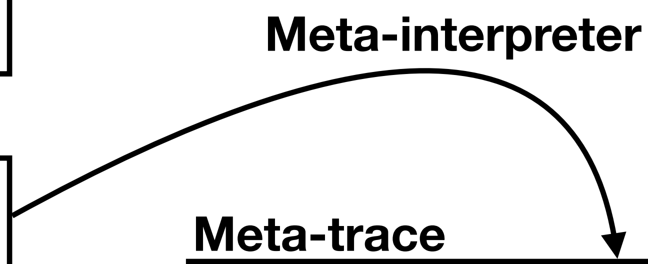
Meta-trace

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

```
  ...
p1 = getarrayitem(p0, 1)
p2 = getarrayitem(p0, 0)
```

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

Meta-trace

```
  ...
p1 = getarrayitem(p0, 1)
p2 = getarrayitem(p0, 0)
```

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

Meta-trace

```
  ...
p1 = getarrayitem(p0, 1)
p2 = getarrayitem(p0, 0)
guard_class(p1, int)
guard_class(p2, int)
```

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

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)
```

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

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()
  ...
```

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) ==
        type(v2) ==
      stack.push(v1 + v2)
    elif ...
  elif bc.type == LOAD_FAST:
    stack.push(local[bc.varnum])
  ...
```

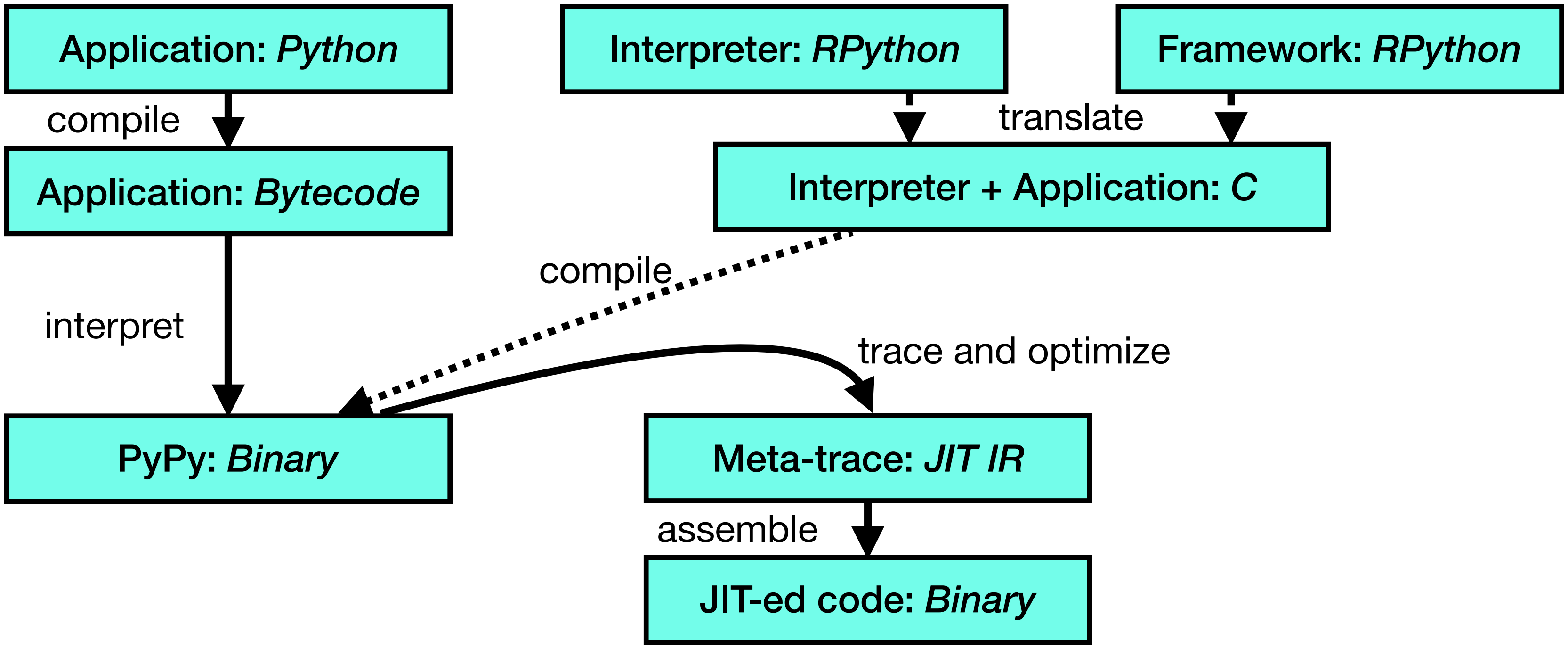
Meta-interpreter

Meta-trace

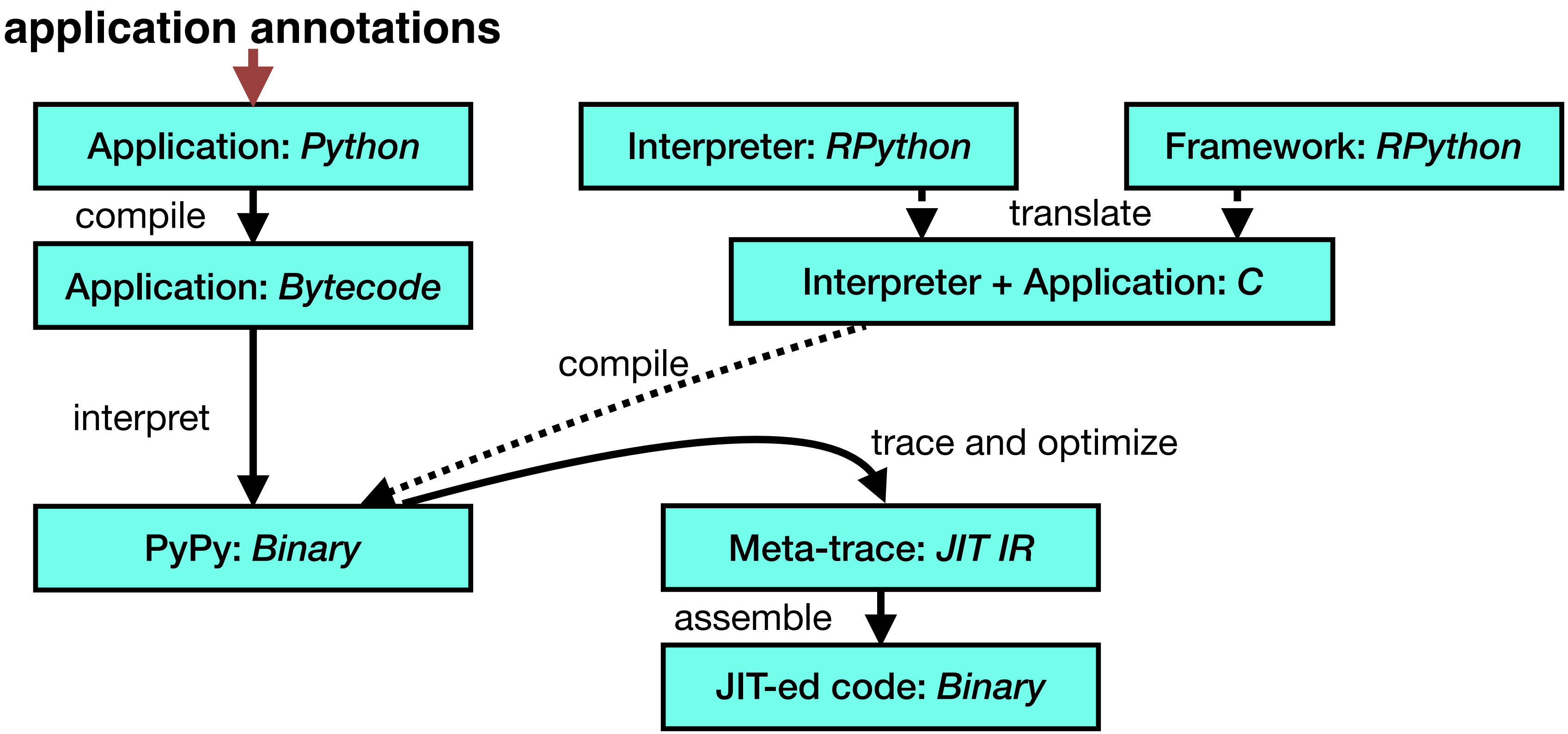
```
  ...
p1 = getarrayitem(p0, 1)
p2 = getarrayitem(p0, 0)
guard_class(p1, int)
i4 = getfield(p2, intval)
i5 = int_add_ovf(i3, i4)
guard_no_overflow()
  ...
```

Deoptimization back to interpreter on guard failure

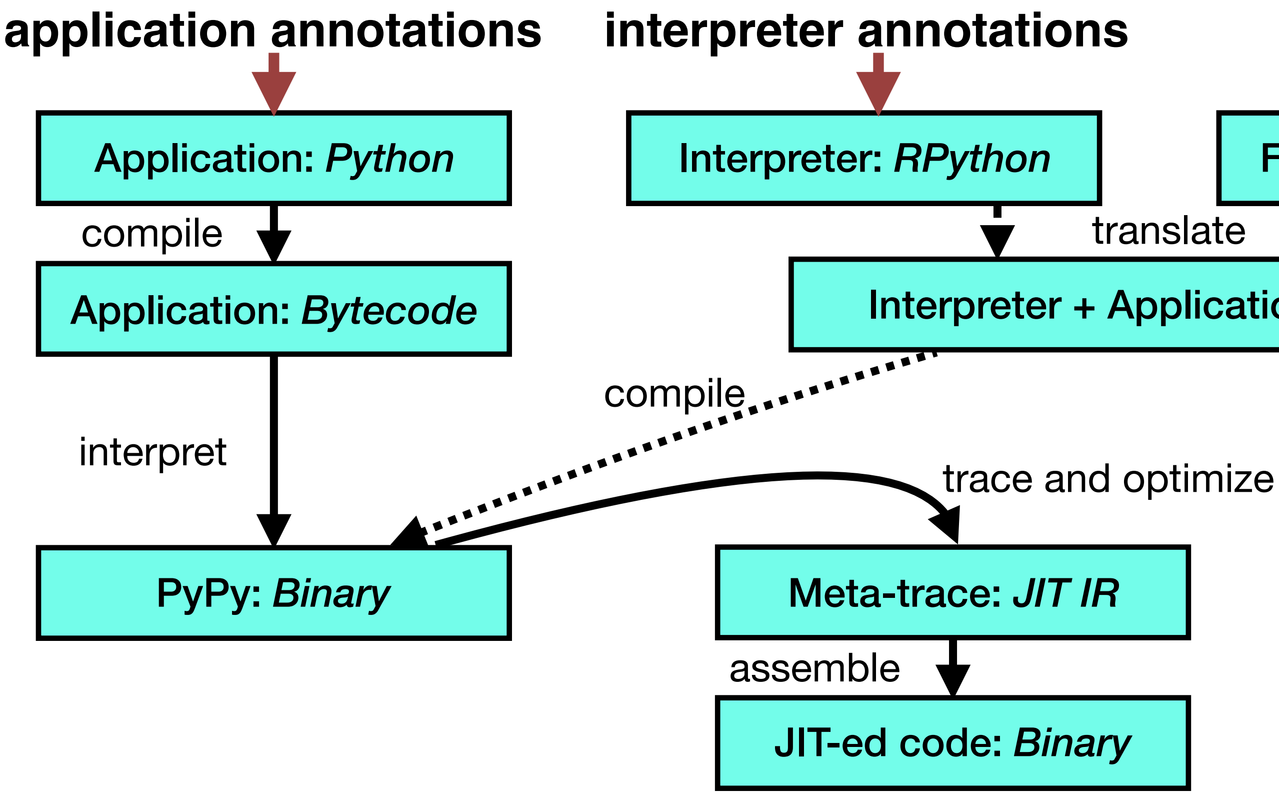
Cross-layer annotations



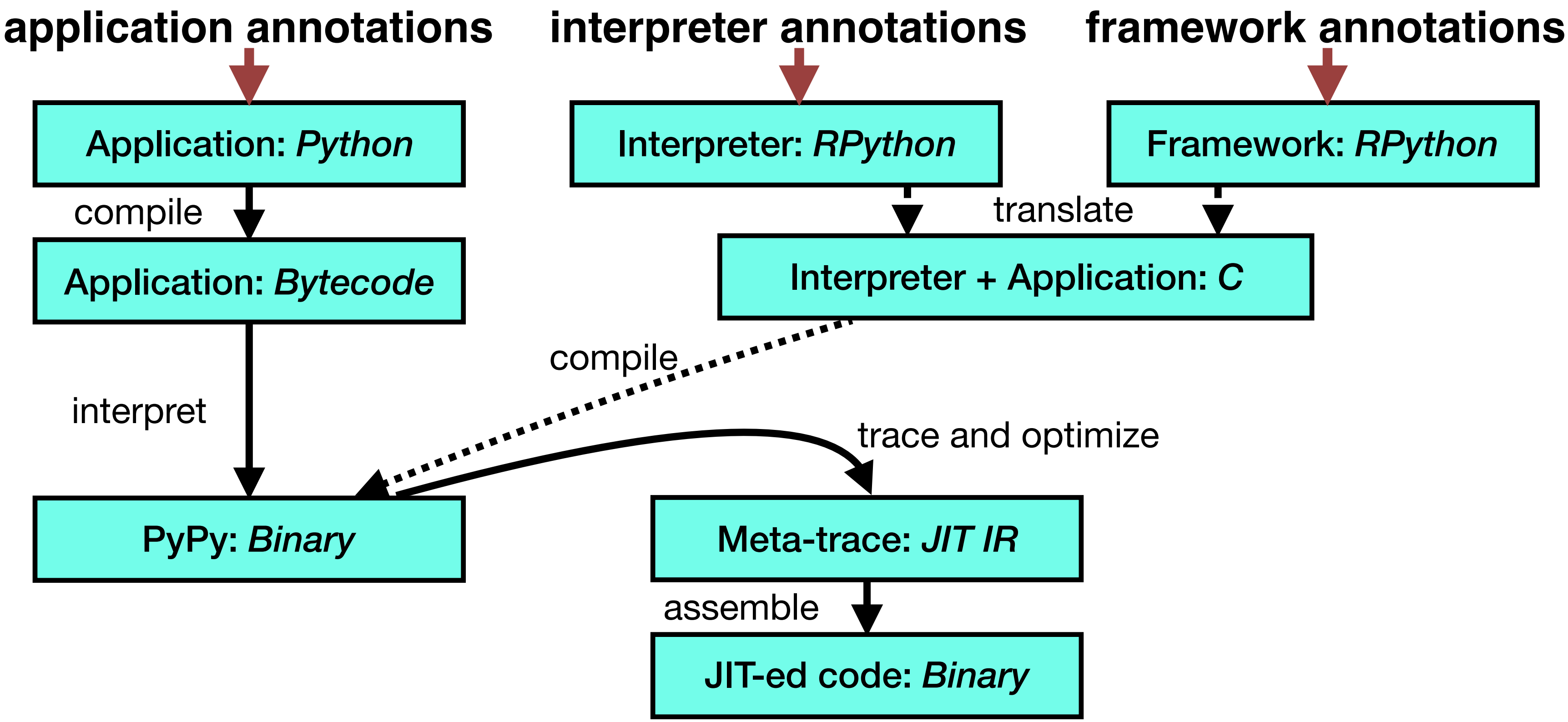
Cross-layer annotations



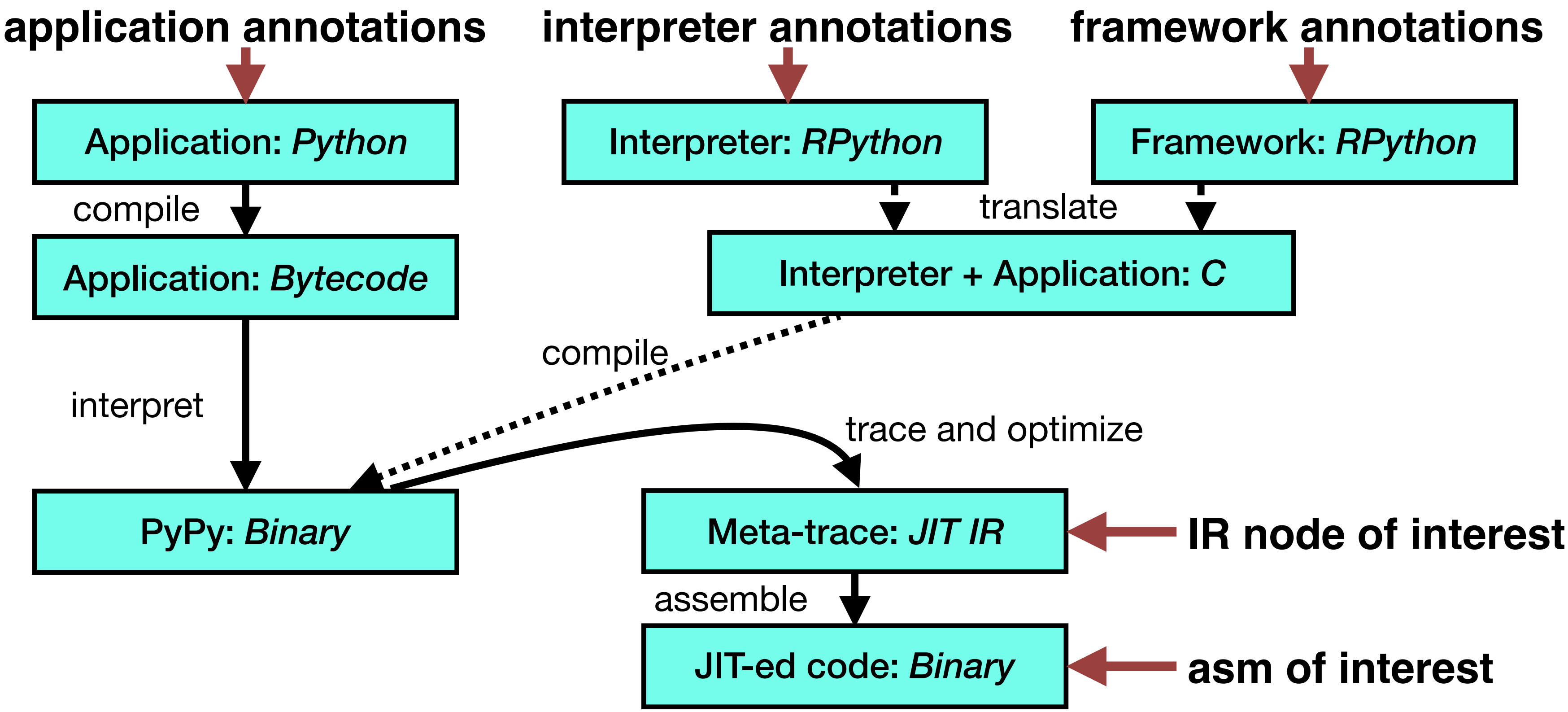
Cross-layer annotations



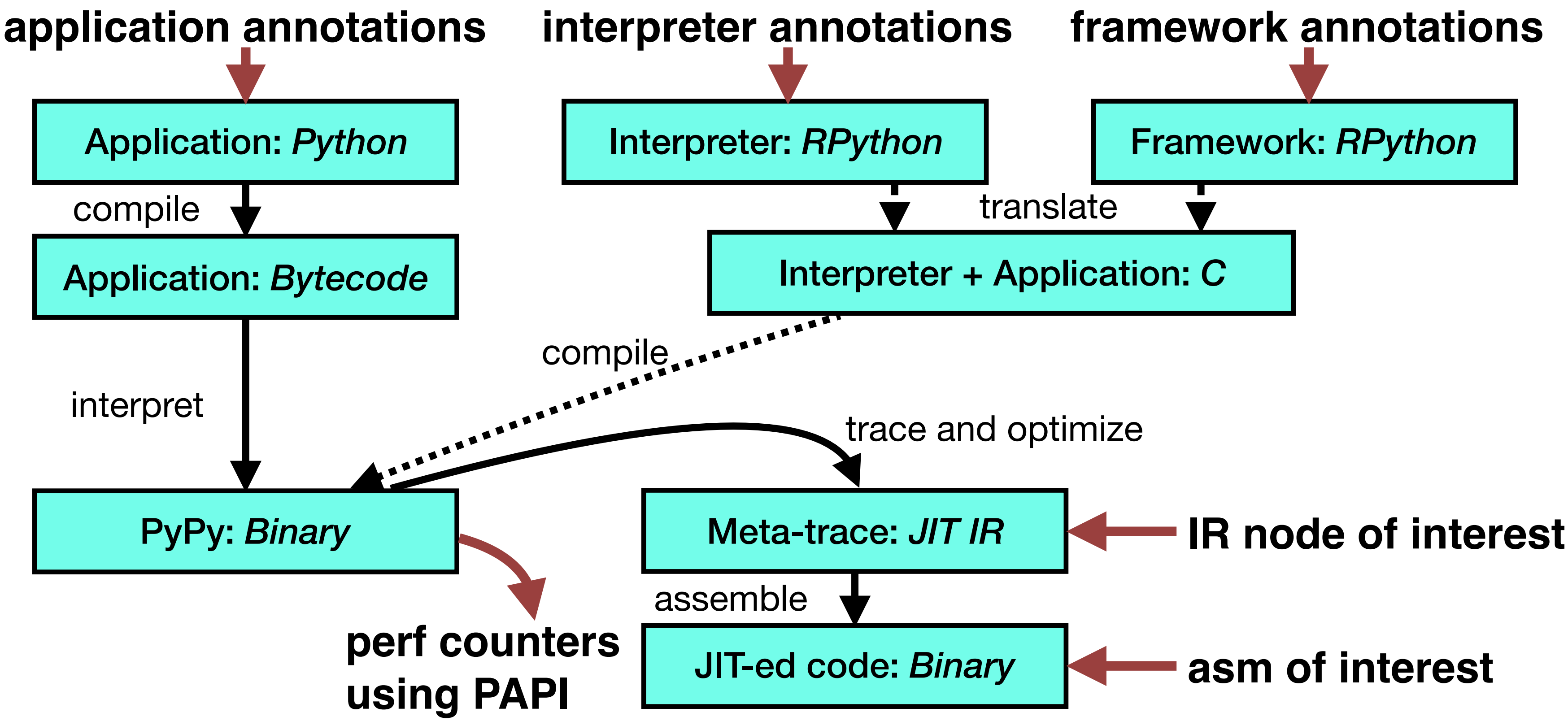
Cross-layer annotations



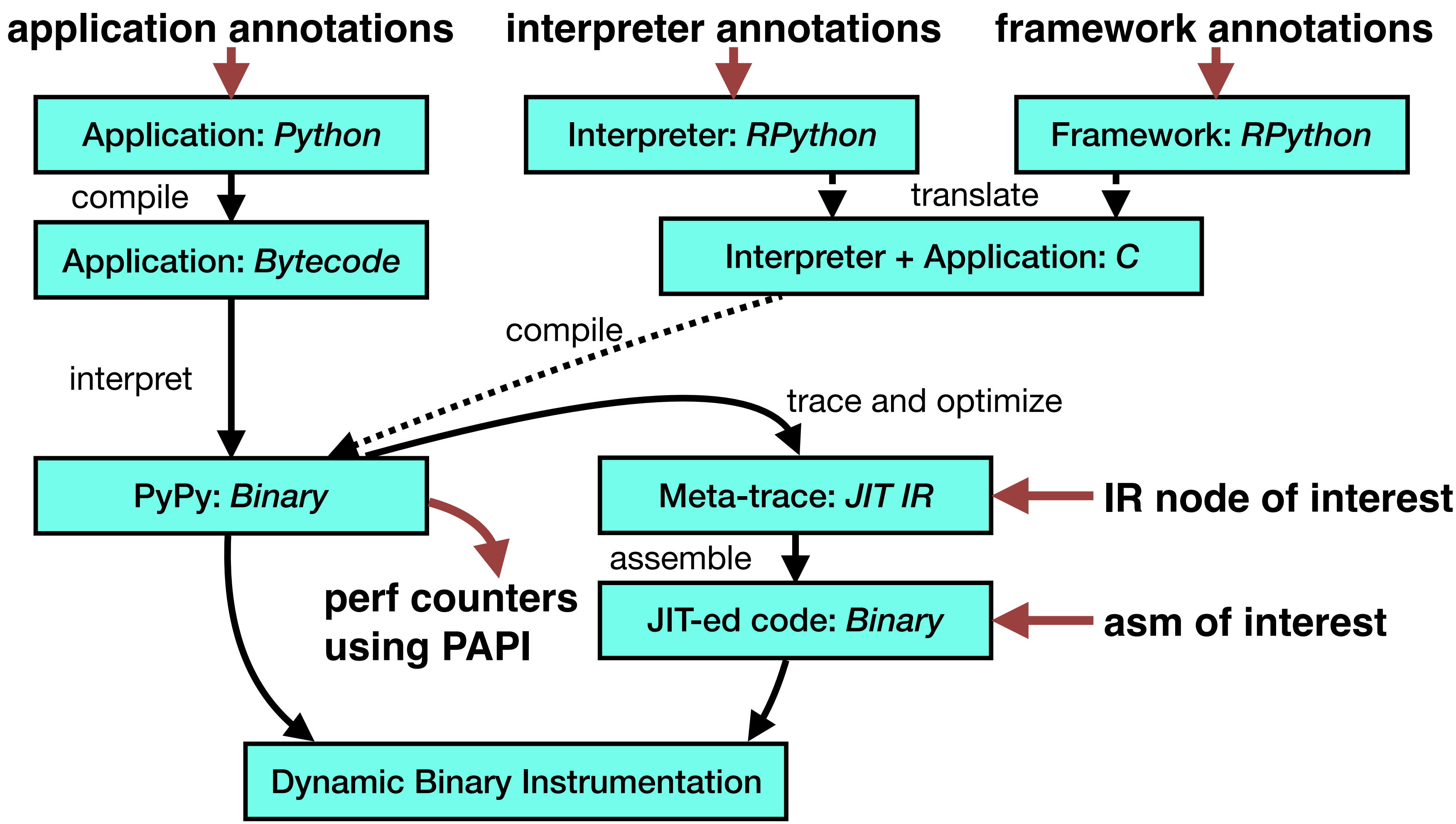
Cross-layer annotations



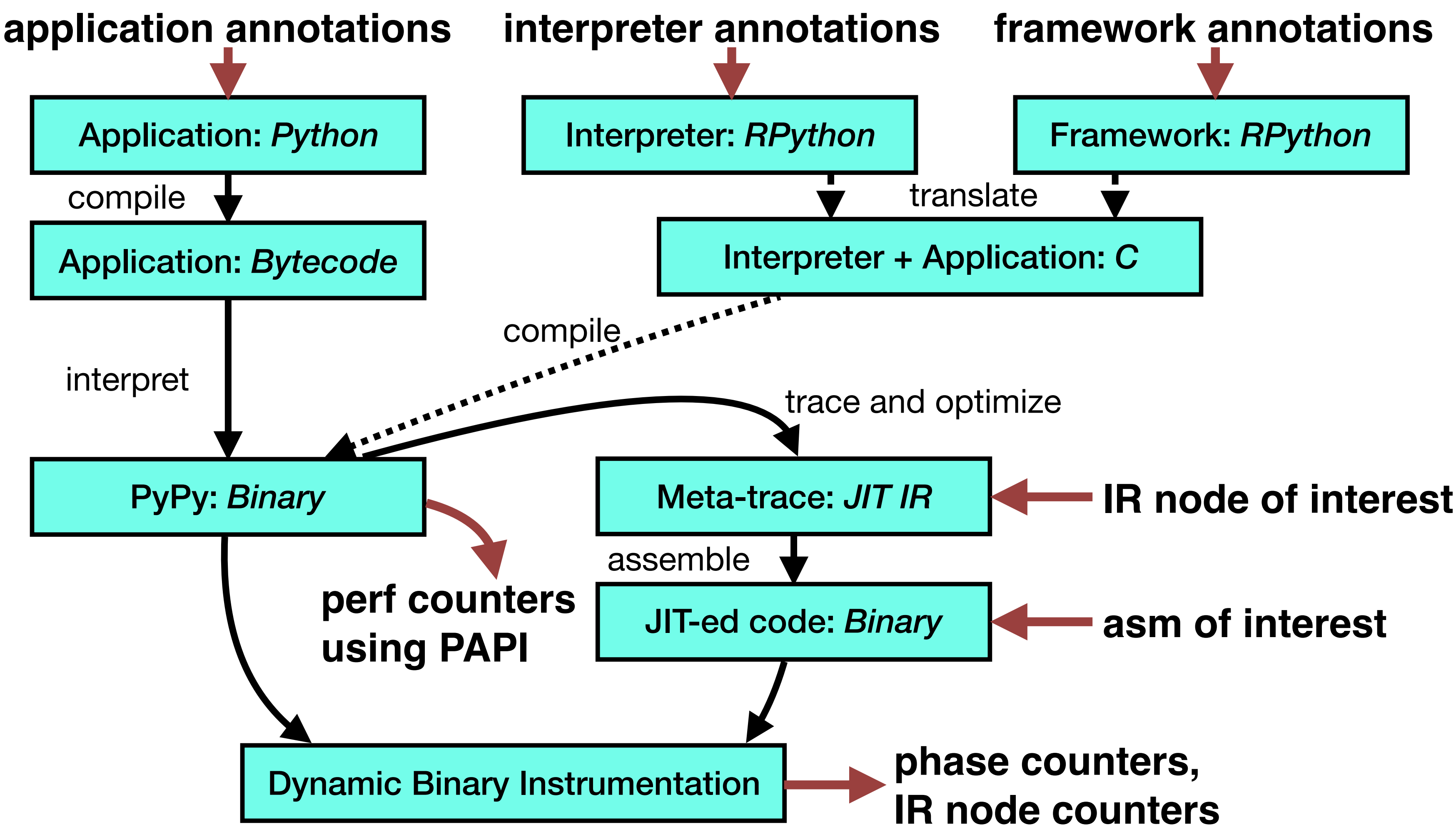
Cross-layer annotations



Cross-layer annotations



Cross-layer annotations



Cross-layer workload characterization of meta-tracing JIT VMs

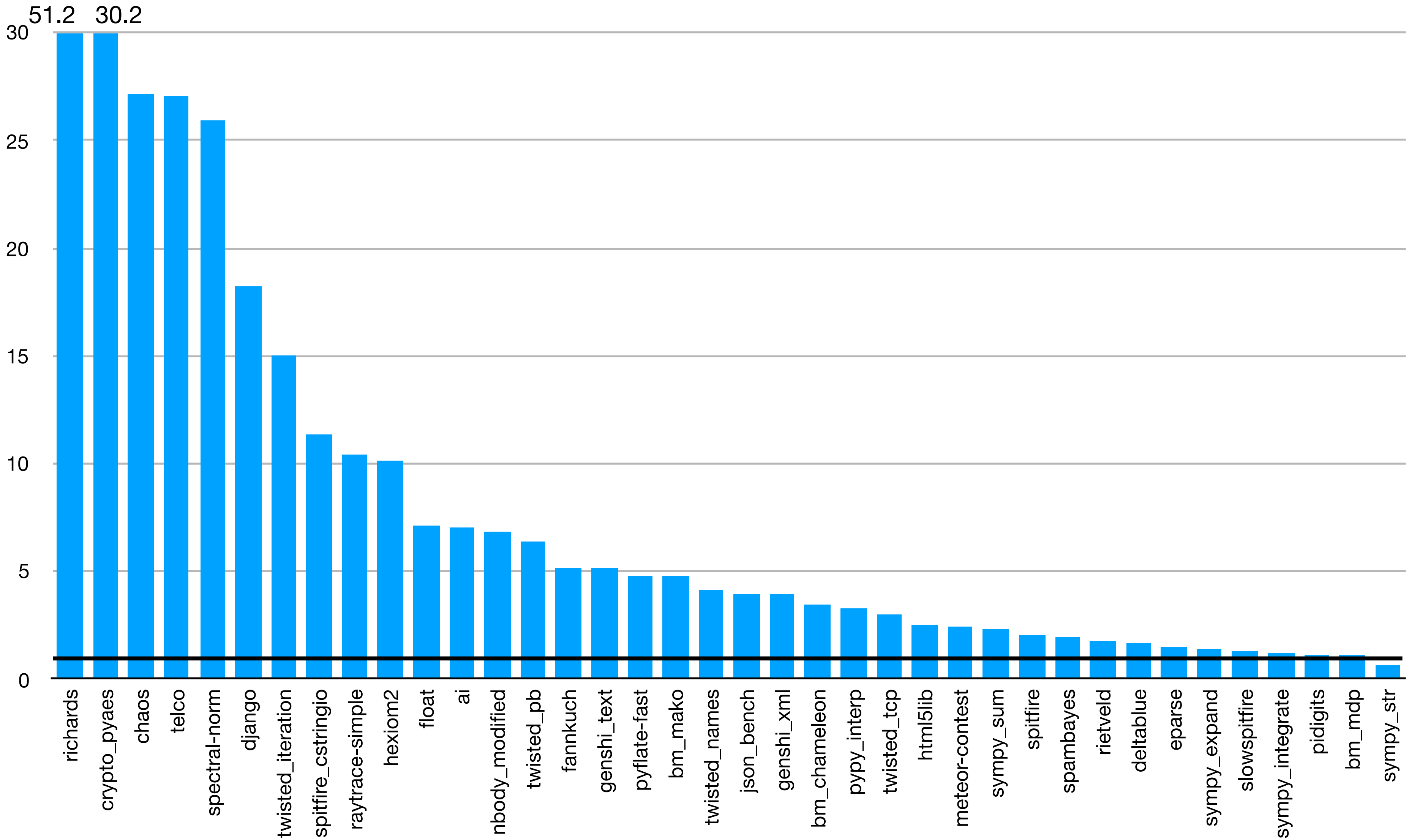
PyPy >> CPython

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

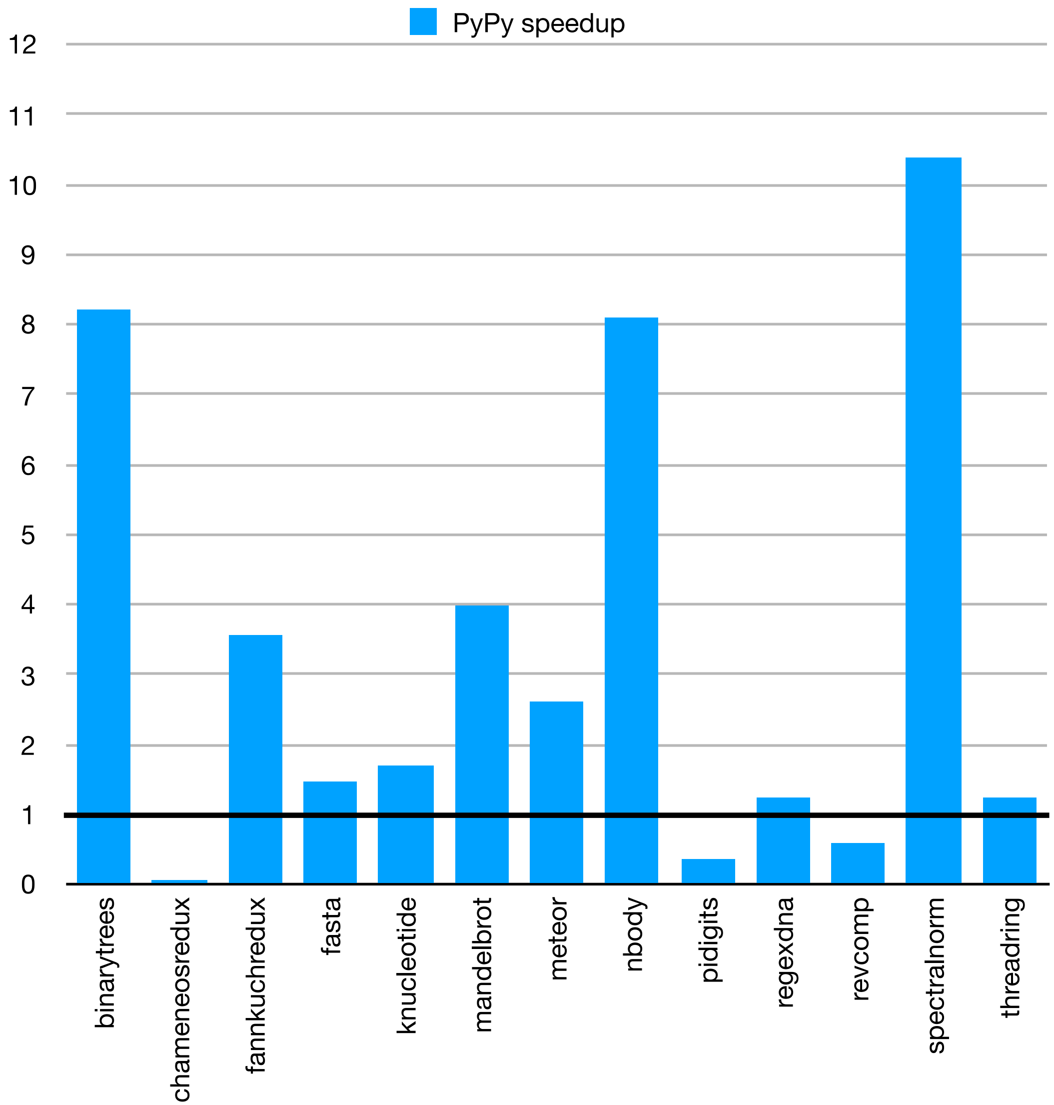
PyPy << C

- **Why are meta-tracing JITs for dynamic programming still slower than C?**

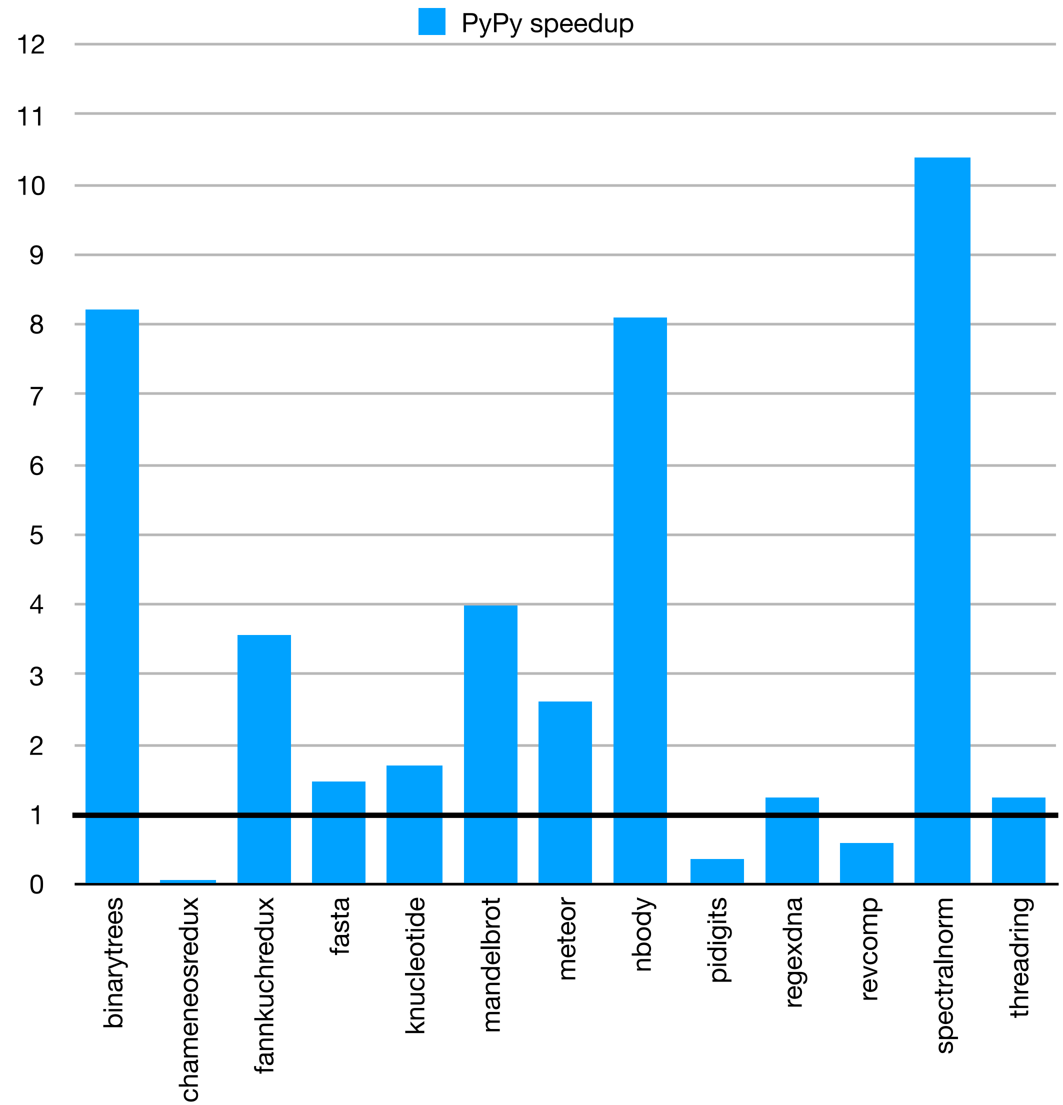
PyPy with meta-tracing JIT speedup over CPython: Meta-tracing JIT improves the performance significantly

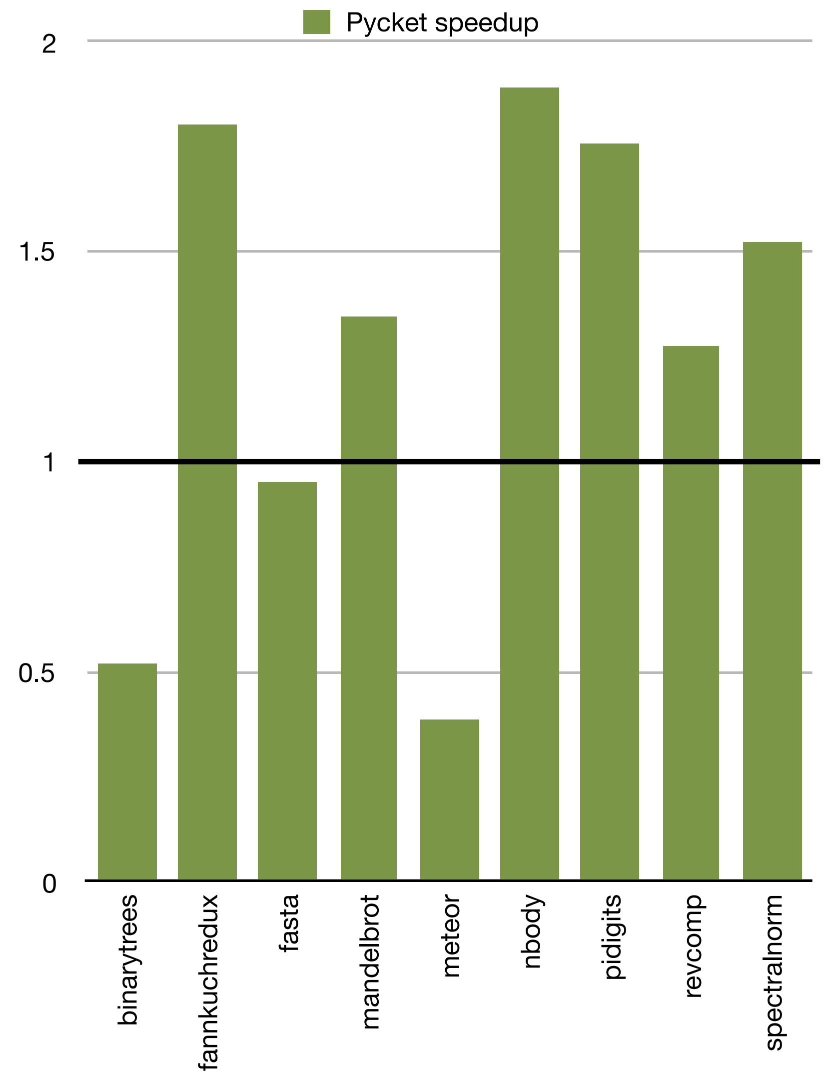


PyPy speedup over CPython and Pycket speedup over Racket: Meta-tracing JIT improves performance significantly across multiple languages

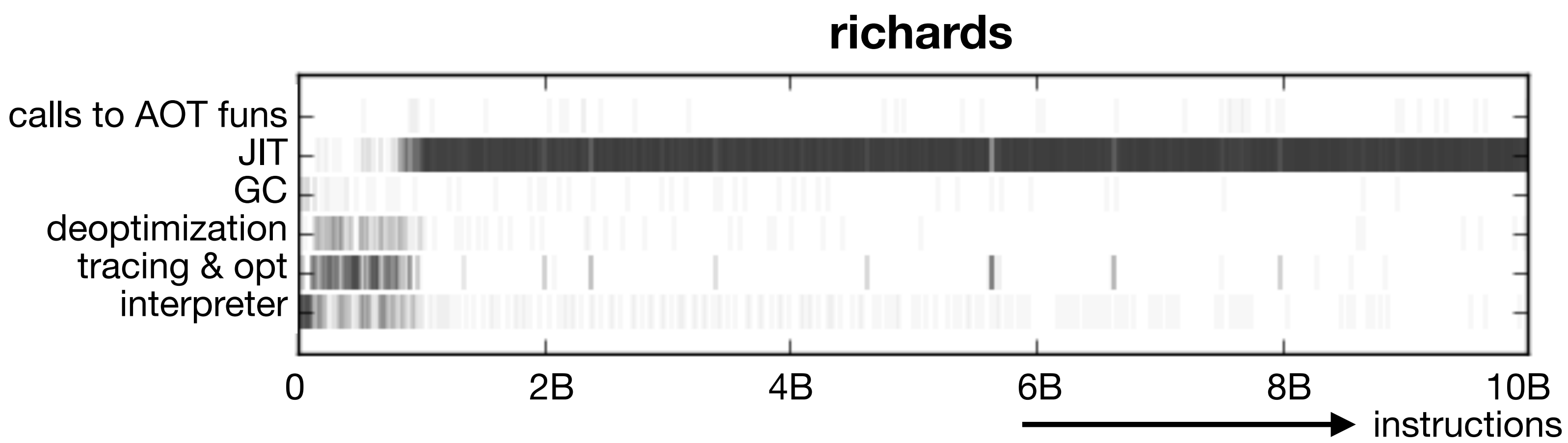


PyPy speedup over CPython and Pycket speedup over Racket: Meta-tracing JIT improves performance significantly across multiple languages

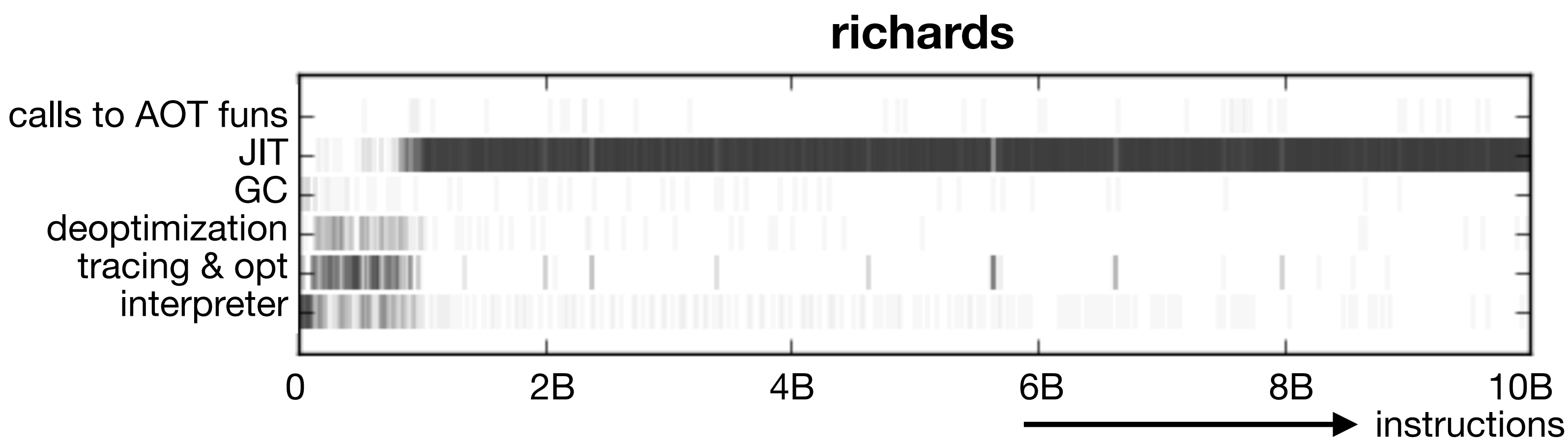


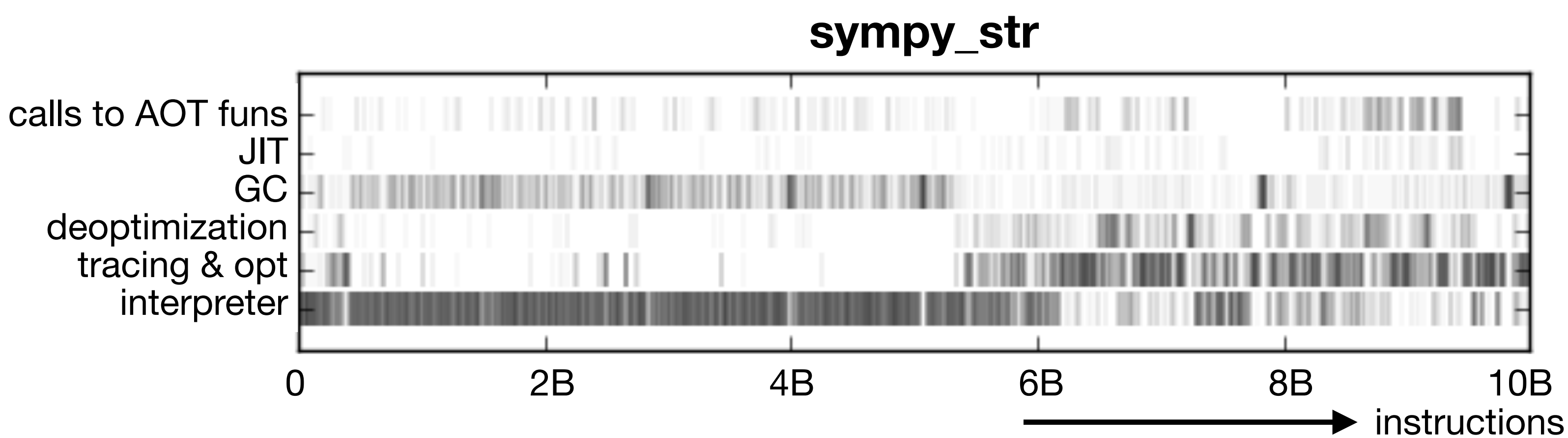


Meta-tracing JIT VM phases

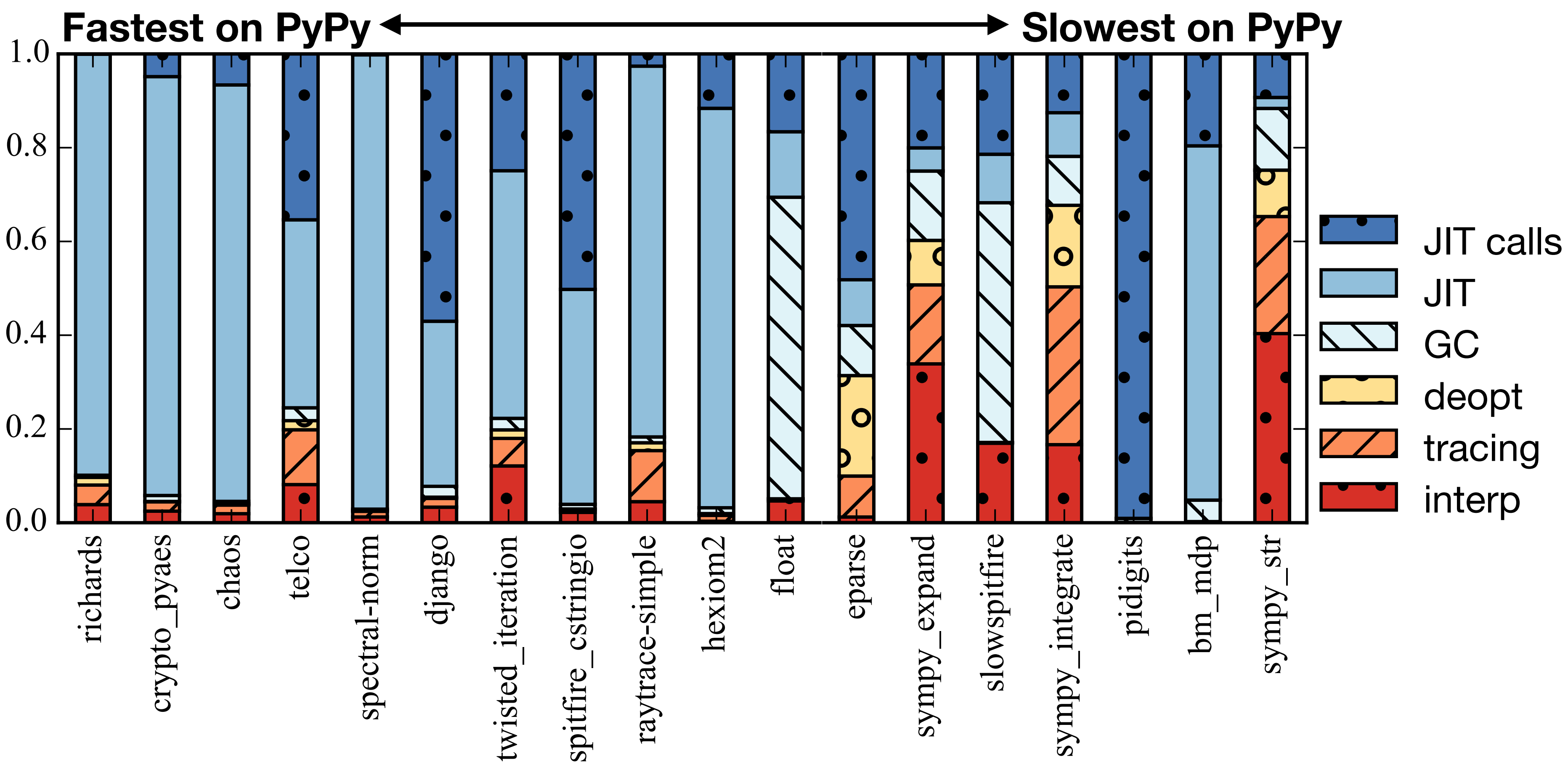


Meta-tracing JIT VM phases

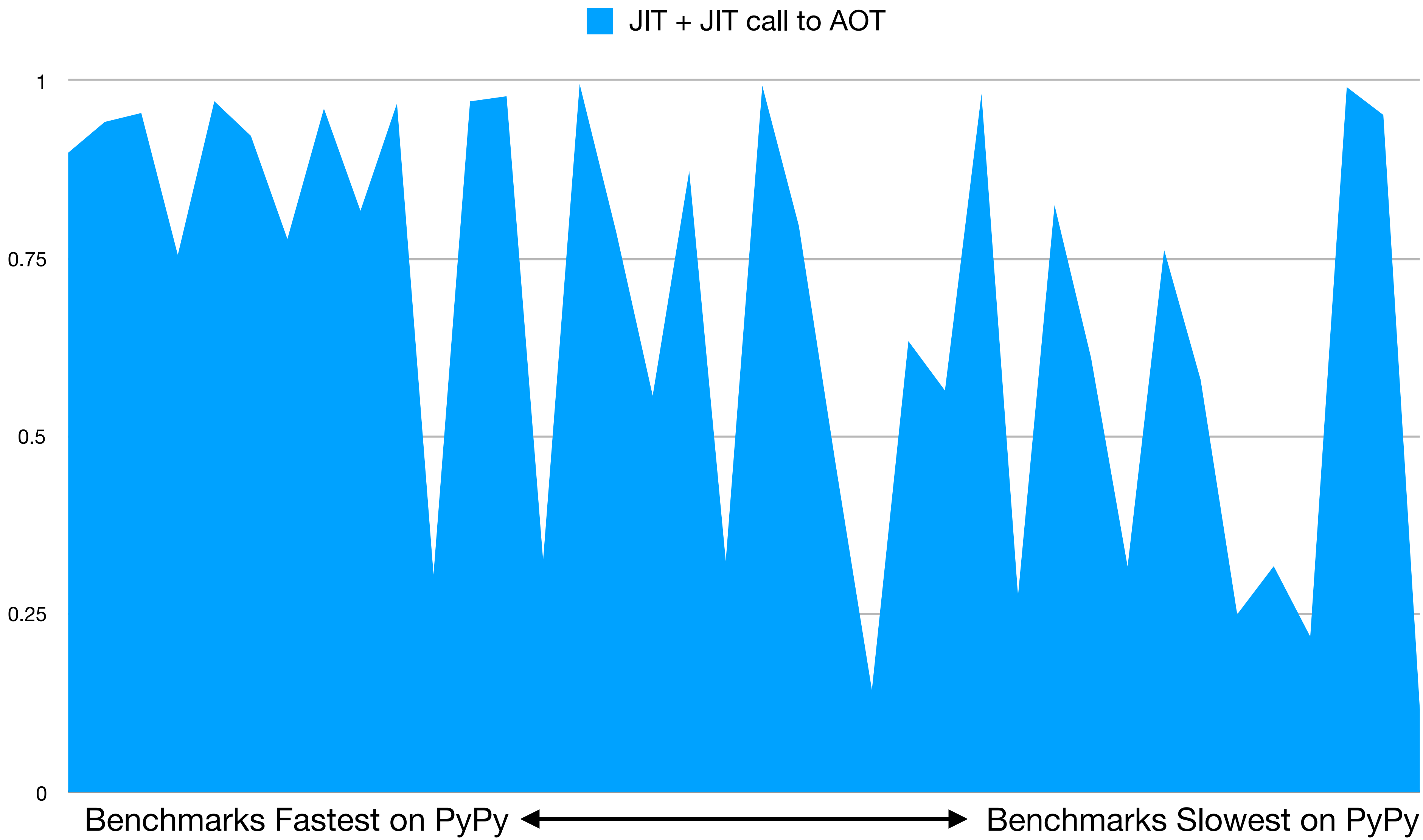




Meta-tracing JIT VM phases



The JIT phase:
The fastest benchmarks tend to execute JIT-compiled code the most



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-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

```
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

```
  ...
guard_gt(i0, 0)
i3 = getarrayitem(p1, 0)
setarrayitem(p2, 0, i3)
```

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

```
  ...
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

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

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

Meta-interpreter

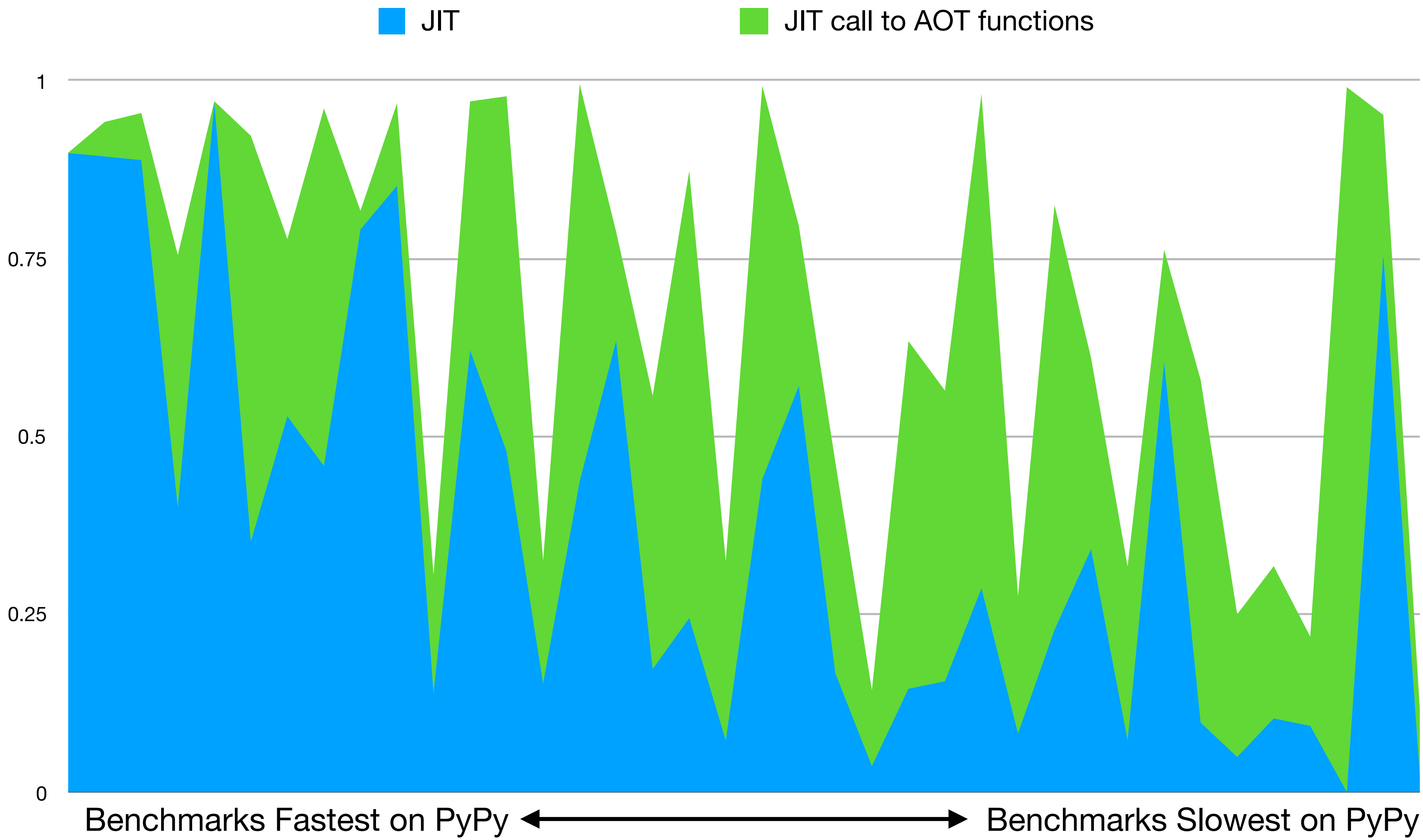
Meta-trace

```
  ...
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

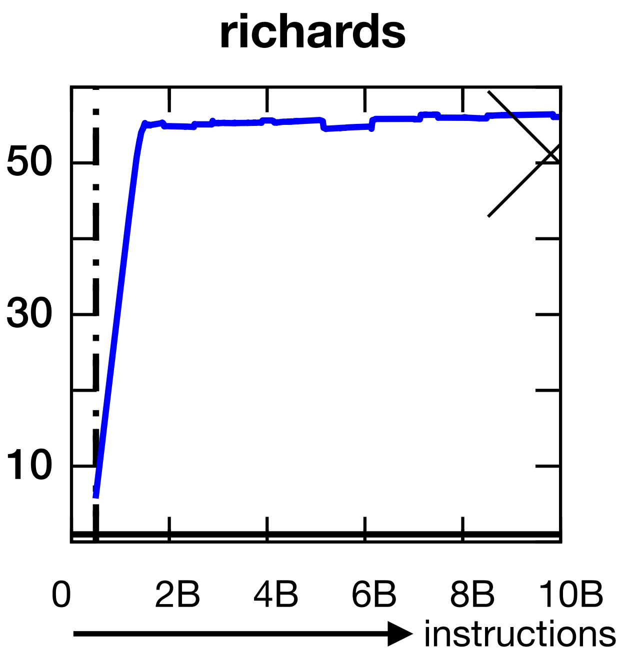
Benchmark	%	Source	Function
ai	19.4	interpreter	`setobject.get_storage_from_list`
bm_chameleon	17.9	RPython types	`rordereddict.ll_call_lookup_function`
bm_mako	26.1	RPython lib	`runicode.unicode_encode_ucs1_helper`
json_bench	18.5	PyPy module	`_pypyjson.raw_encode_basestring_ascii`
nbody_modified	44.6	external lib	`pow`

JIT calls to AOT-compiled functions: AOT-compiled functions can improve performance by avoiding long traces



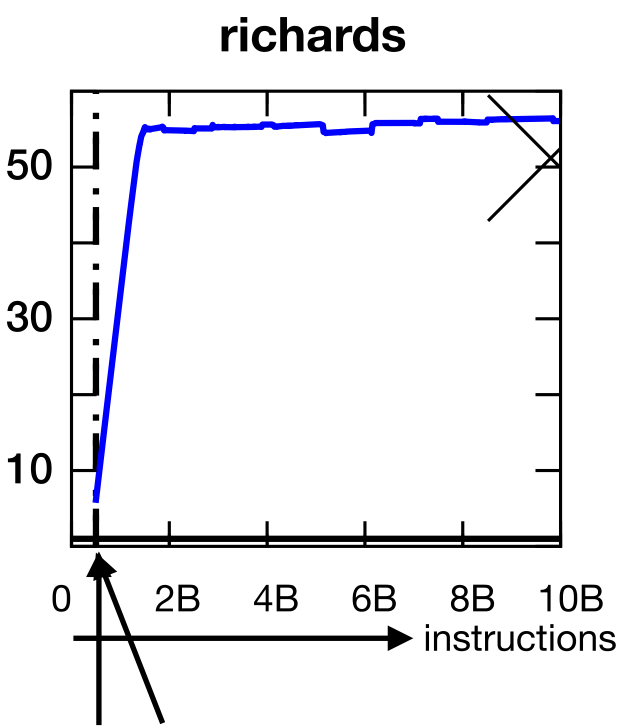
PyPy bytecode execution rate compared to CPython: Benchmarks that perform the best also warm up the fastest

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

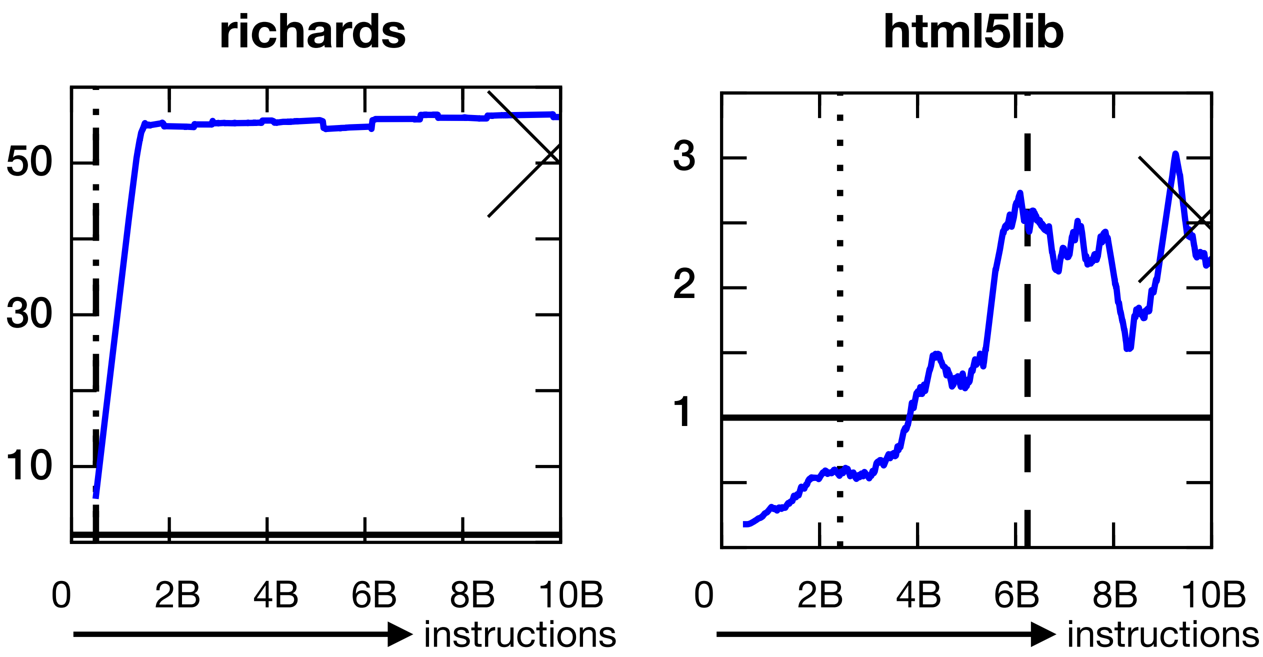
Breakeven point: the performance of the two VMs at this point is equal



PyPy w/o JIT breakeven point
CPython breakeven point

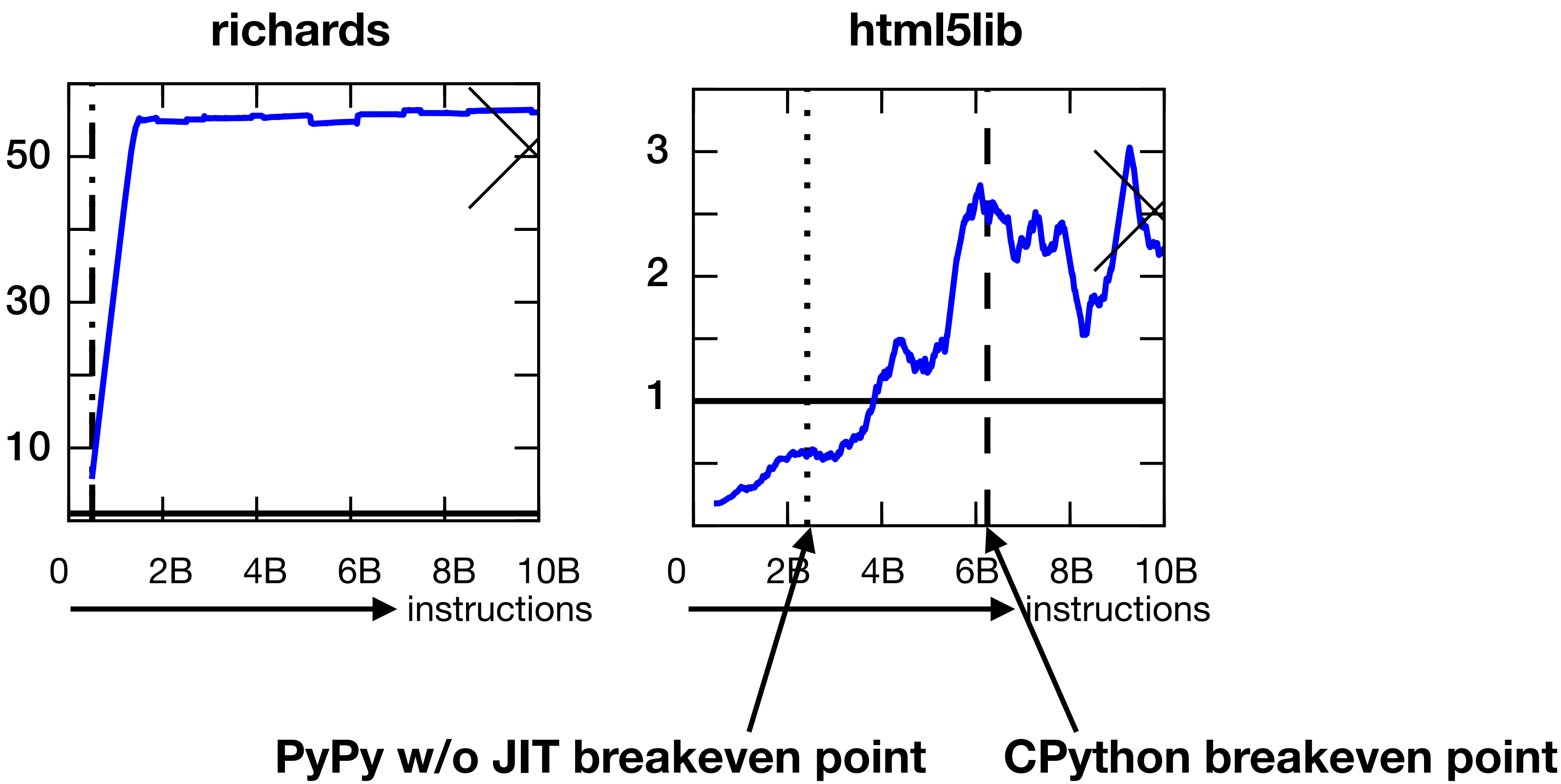
PyPy bytecode execution rate compared to CPython: Benchmarks that perform the best also warm up the fastest

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



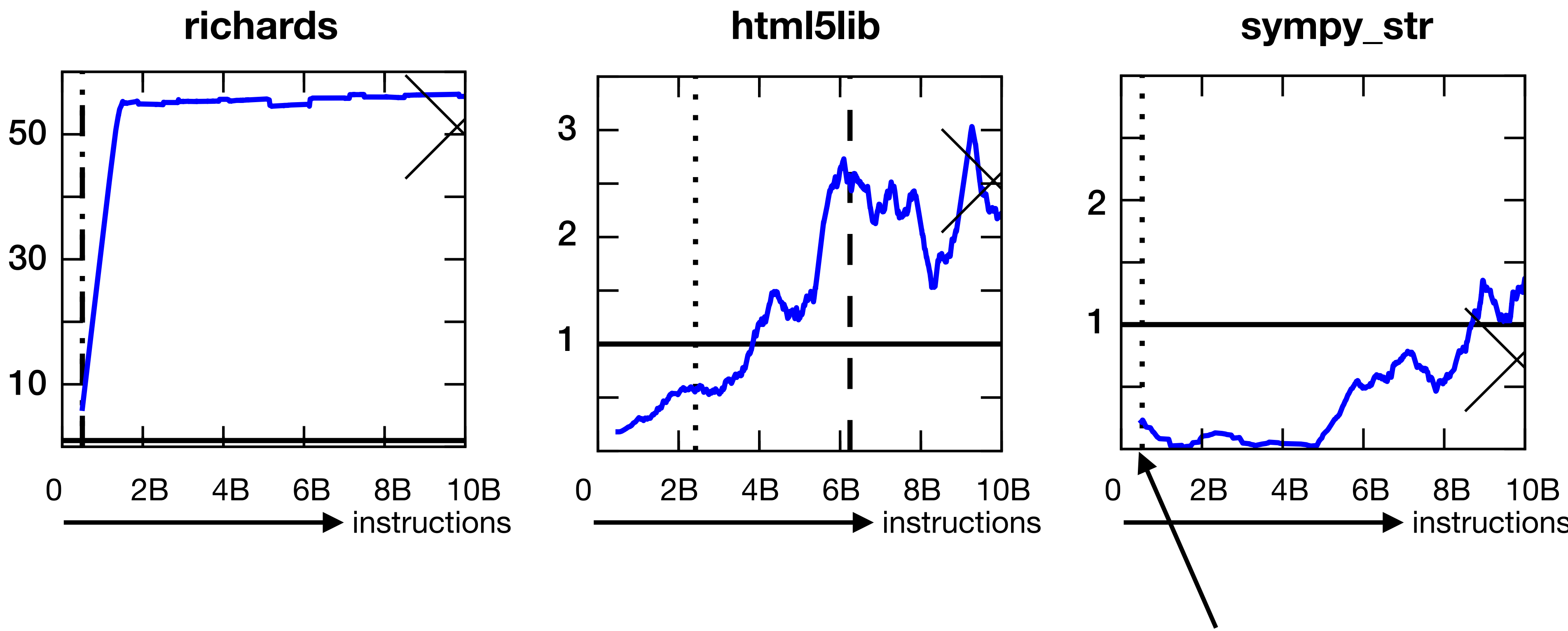
PyPy bytecode execution rate compared to CPython: Benchmarks that perform the best also warm up the fastest

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



PyPy 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

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?**

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

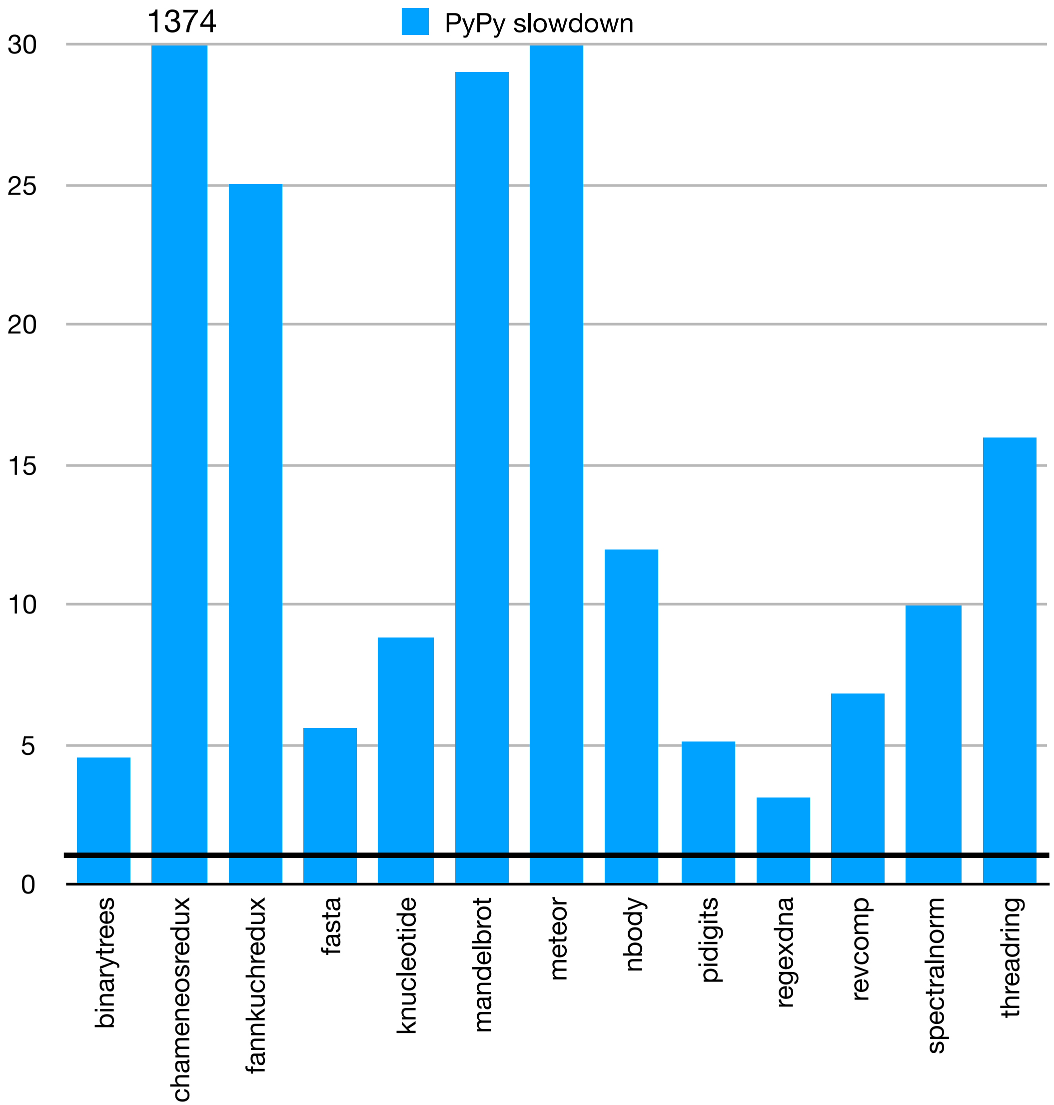
PyPy >> CPython

- **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

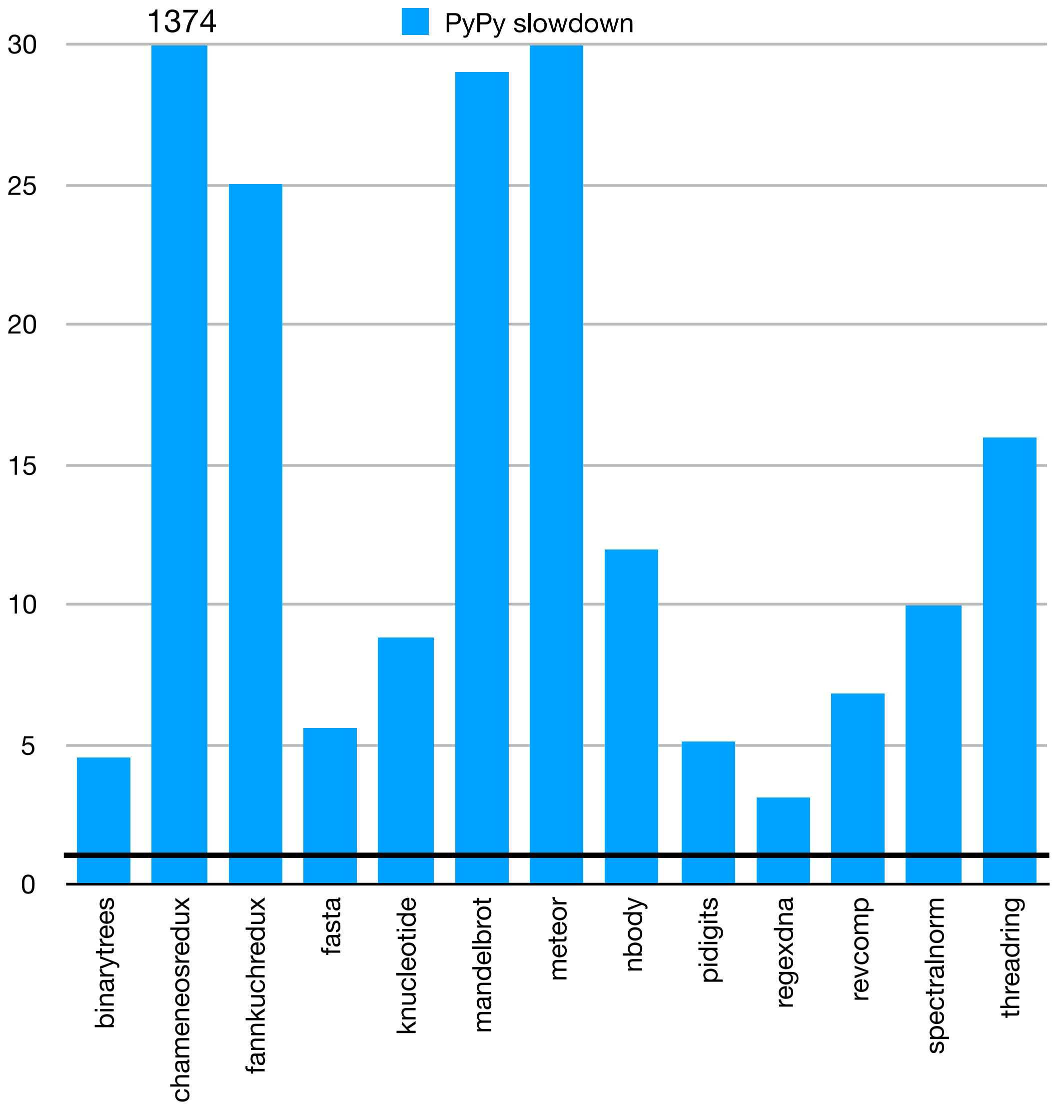
- **Why are meta-tracing JITs for dynamic programming still slower than C?**

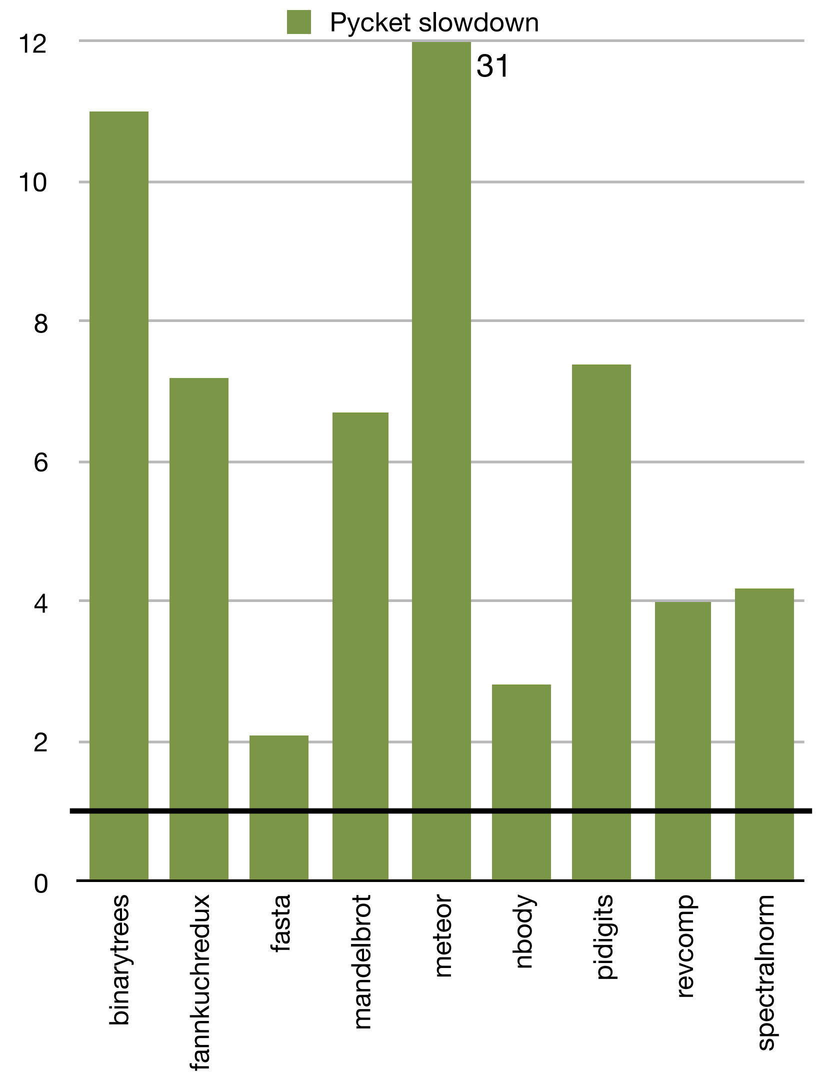
PyPy and Pycket slowdown over C/C++:
Meta-tracing JIT has a big performance gap between static languages



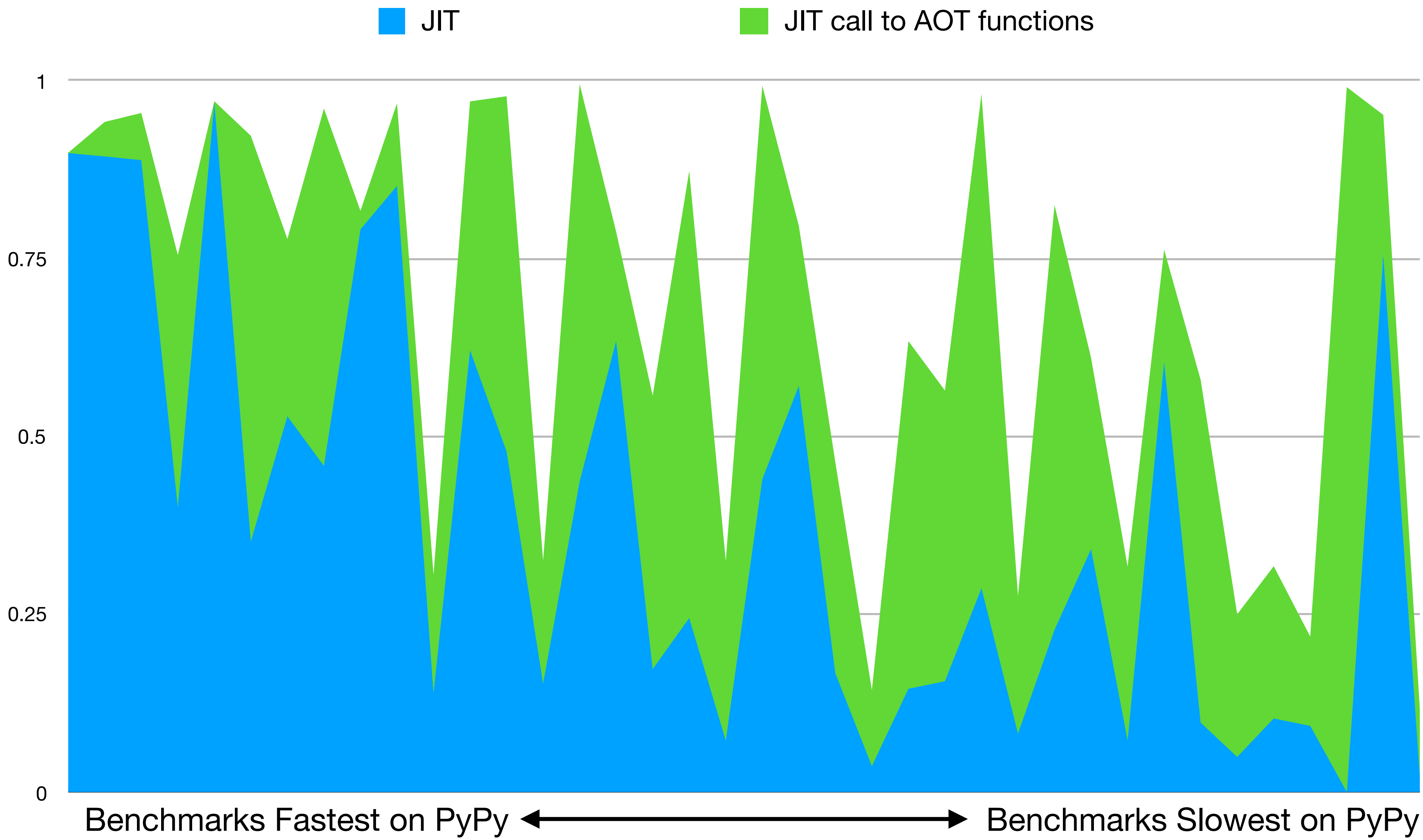
31

PyPy and Pycket slowdown over C/C++:
Meta-tracing JIT has a big performance gap between static languages

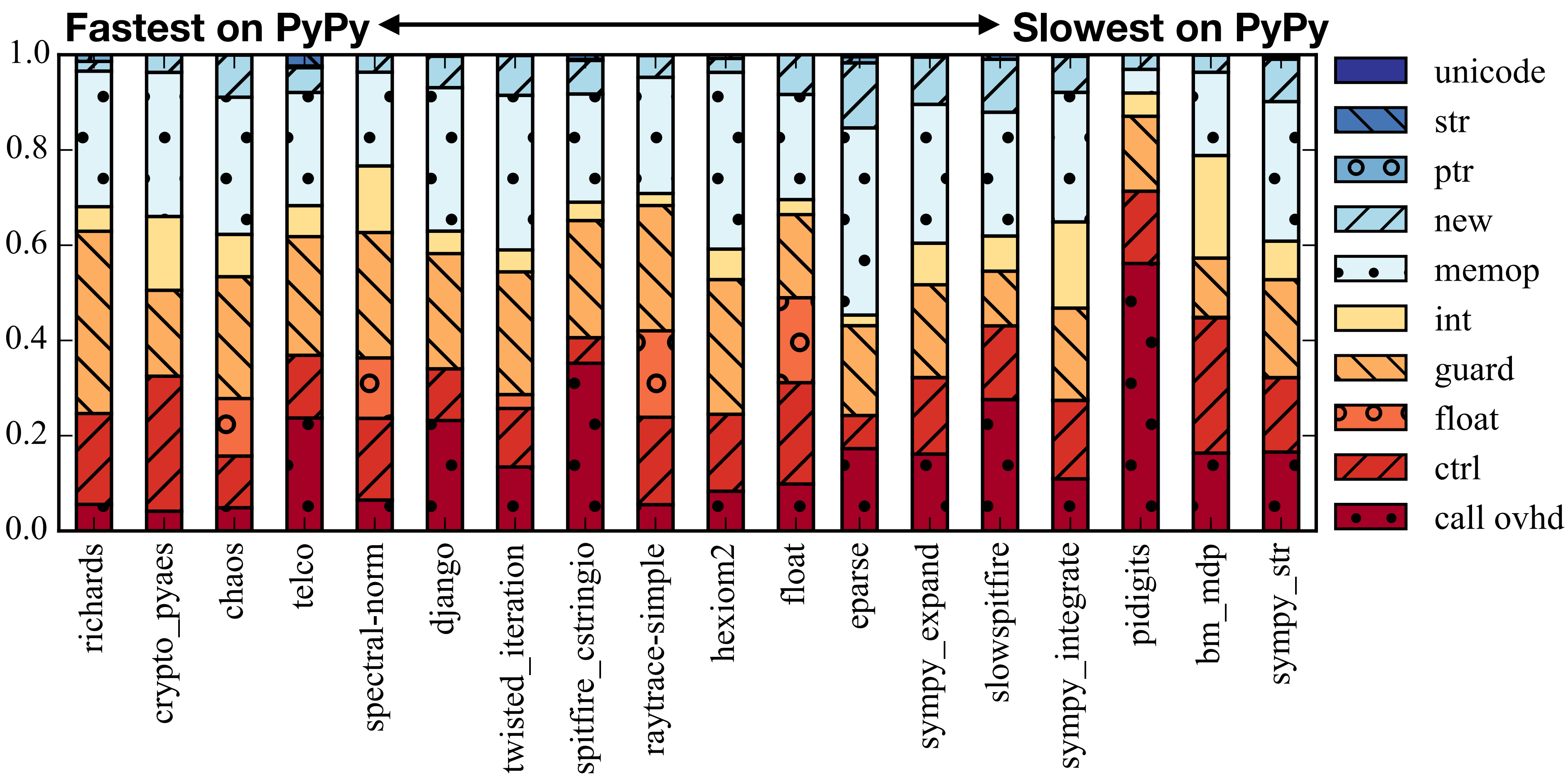




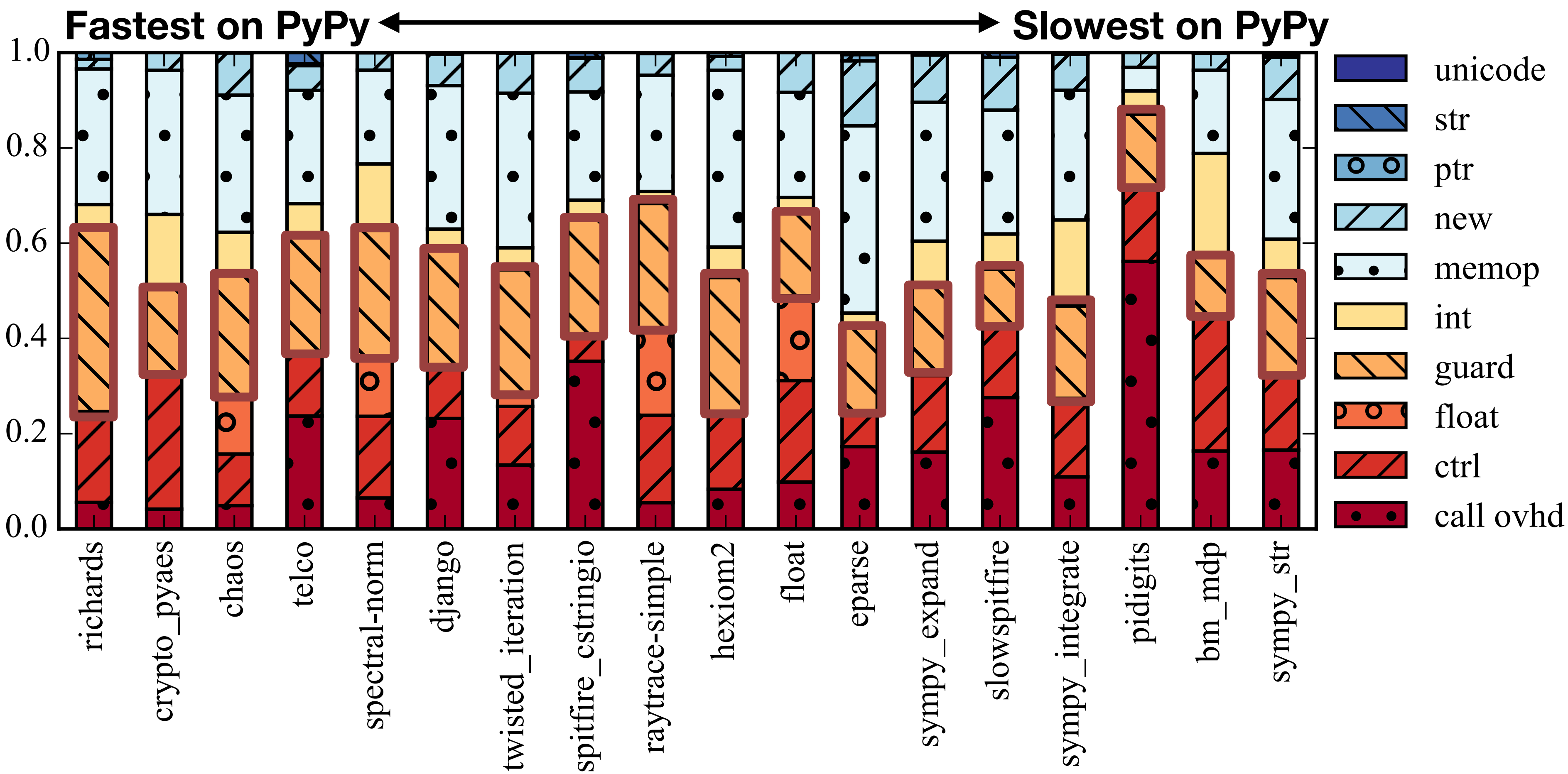
Meta-tracing JIT phases



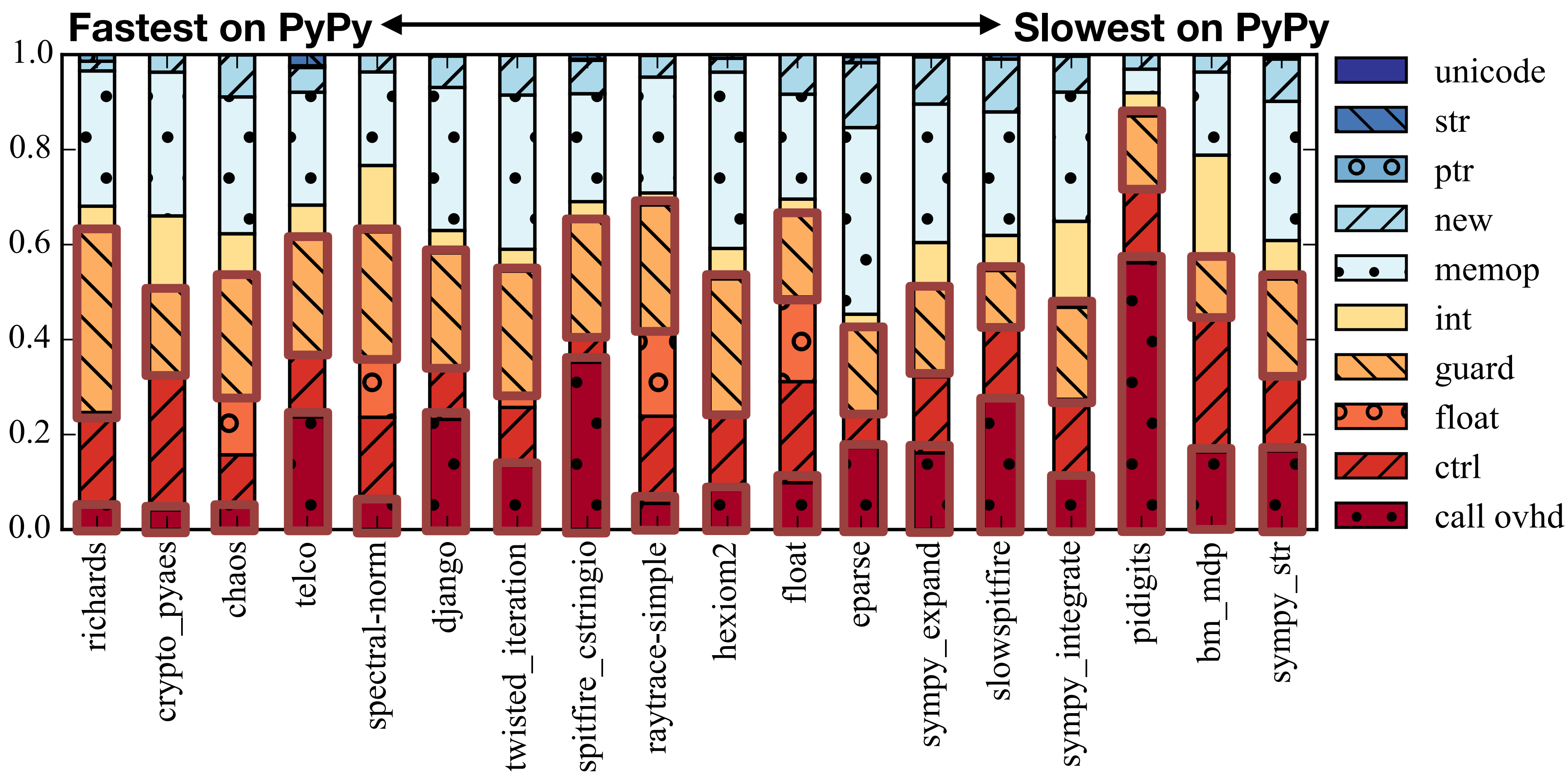
Meta-tracing JIT IR node breakdown: Likely a big part of JIT compiled code is overhead



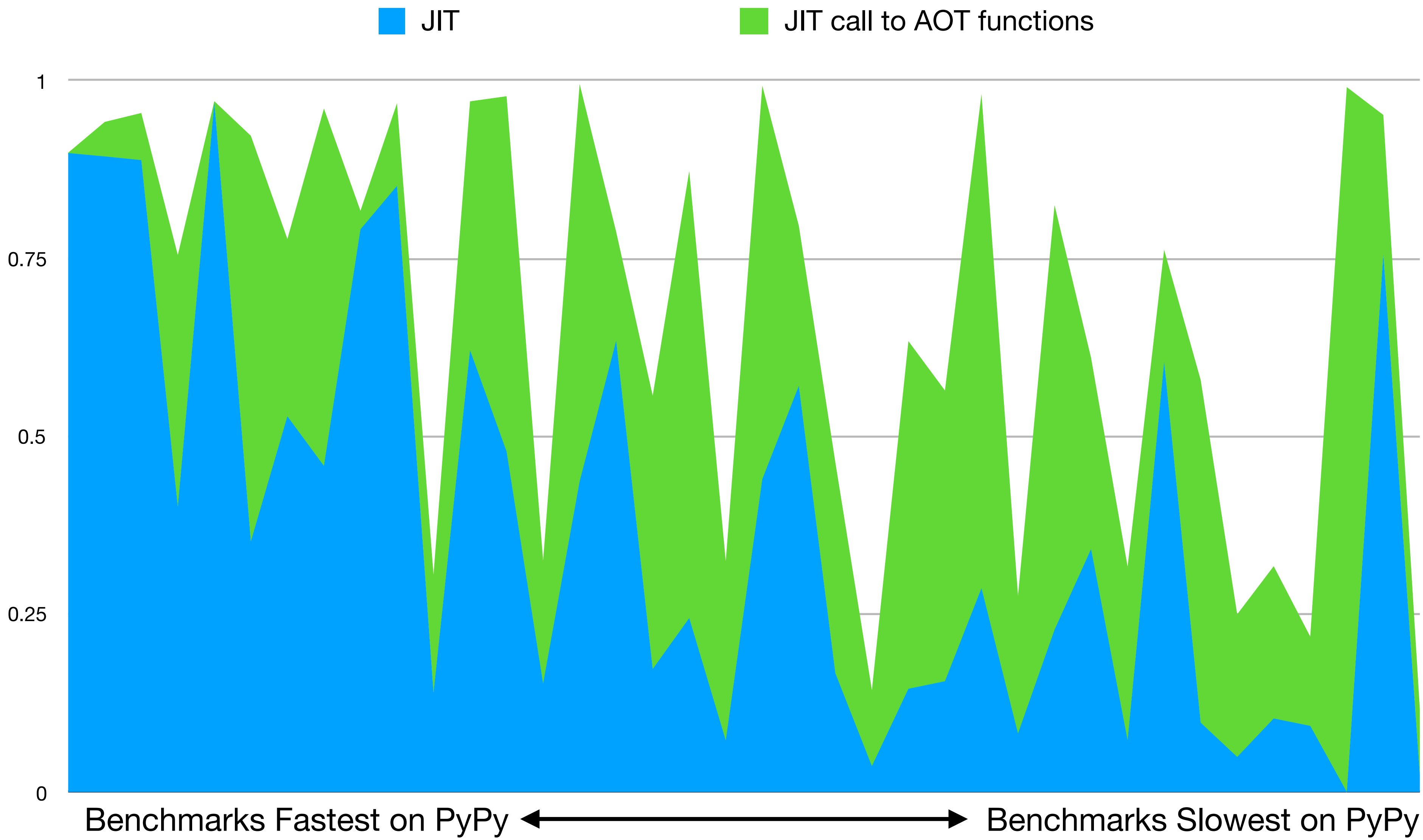
Meta-tracing JIT IR node breakdown:
Likely a big part of JIT compiled code is overhead



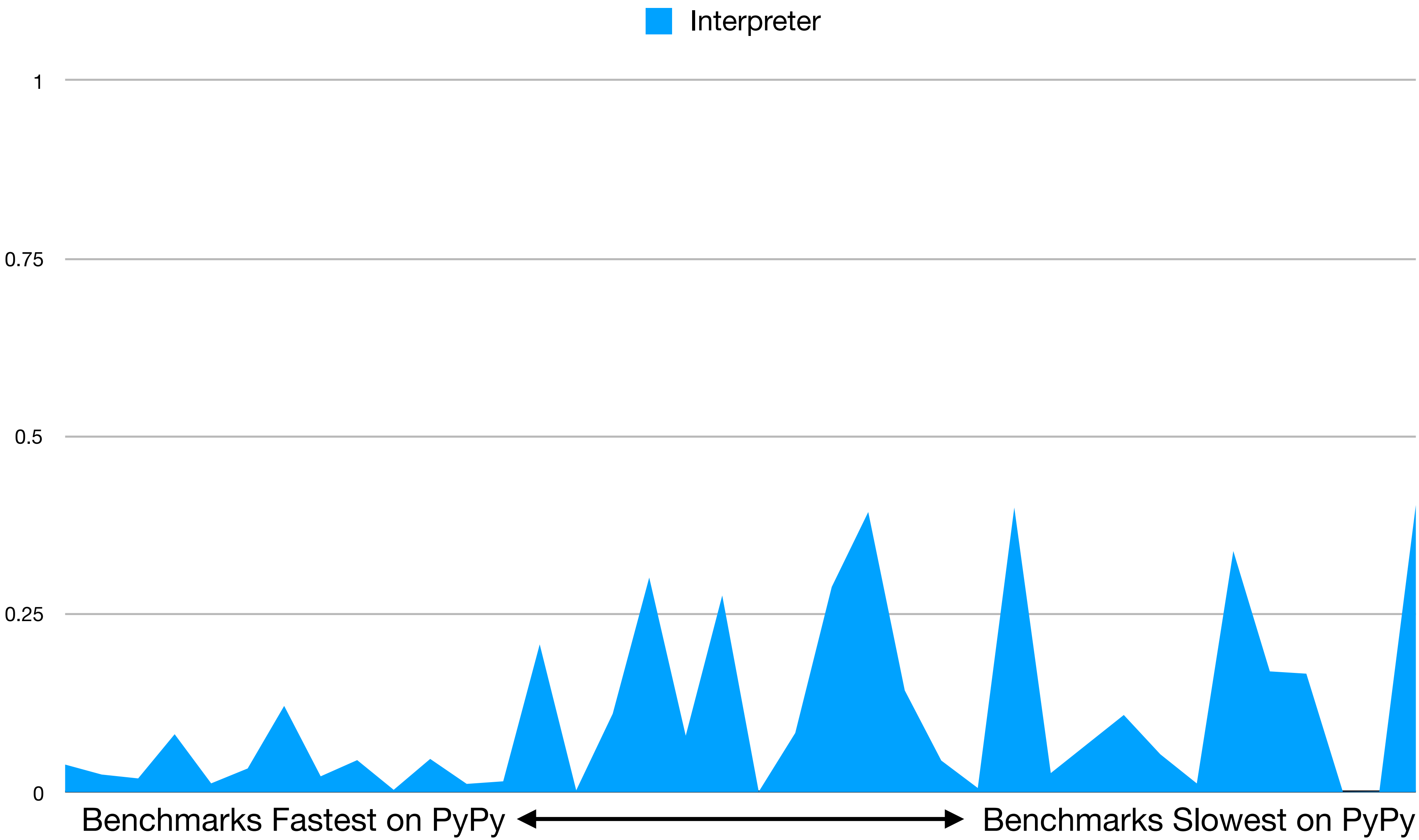
Meta-tracing JIT IR node breakdown: Likely a big part of JIT compiled code is overhead



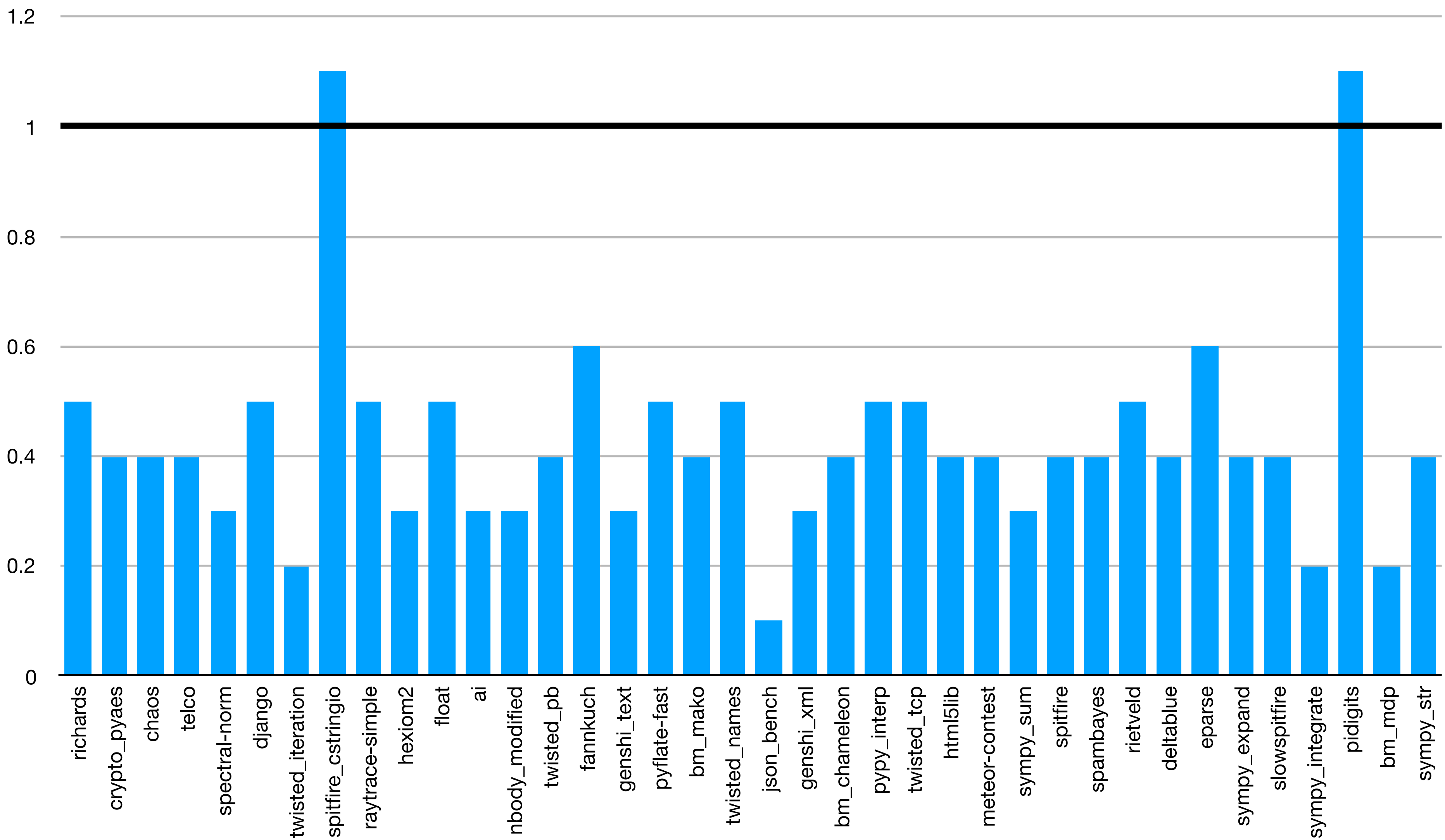
Meta-tracing JIT phases



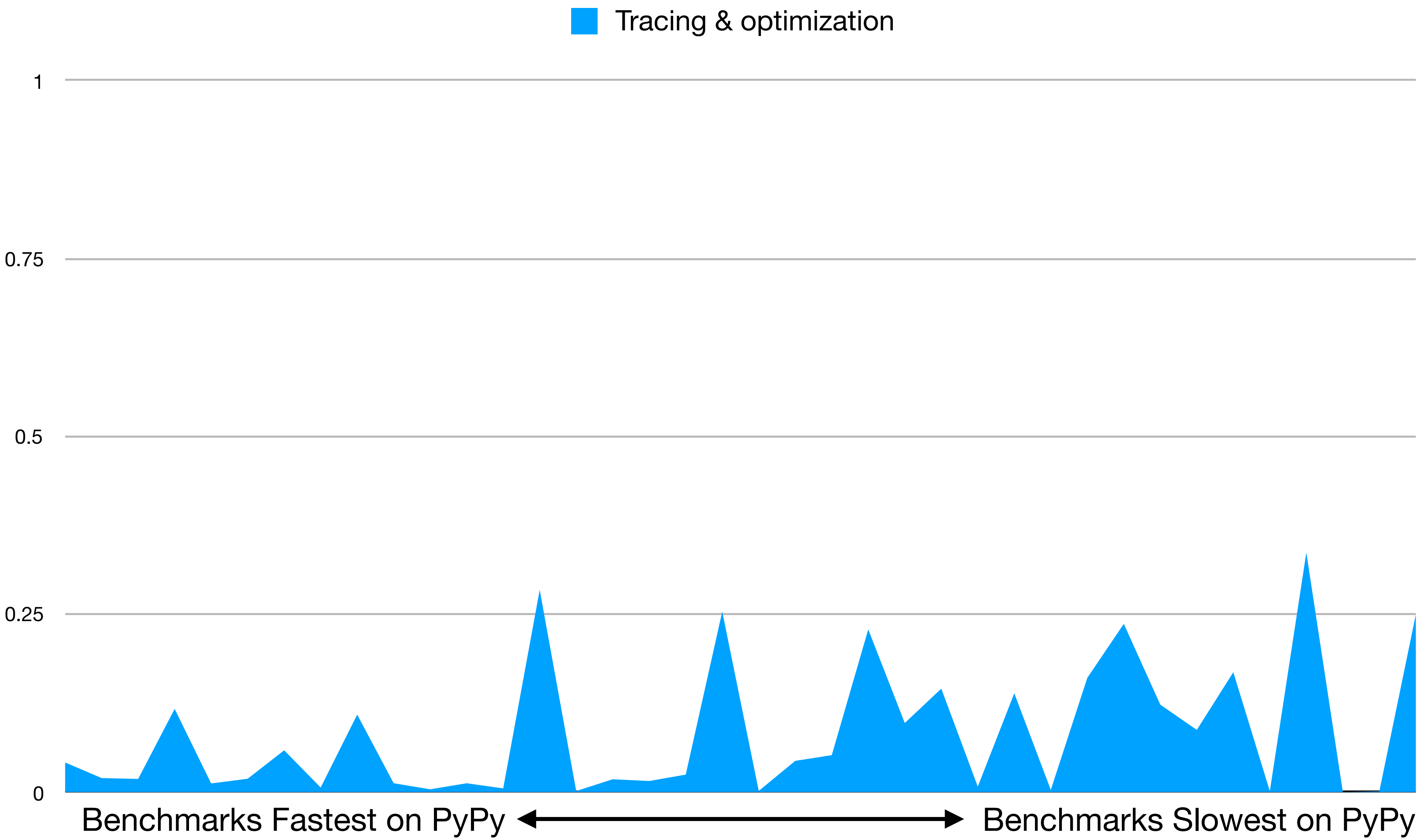
Interpreter phase



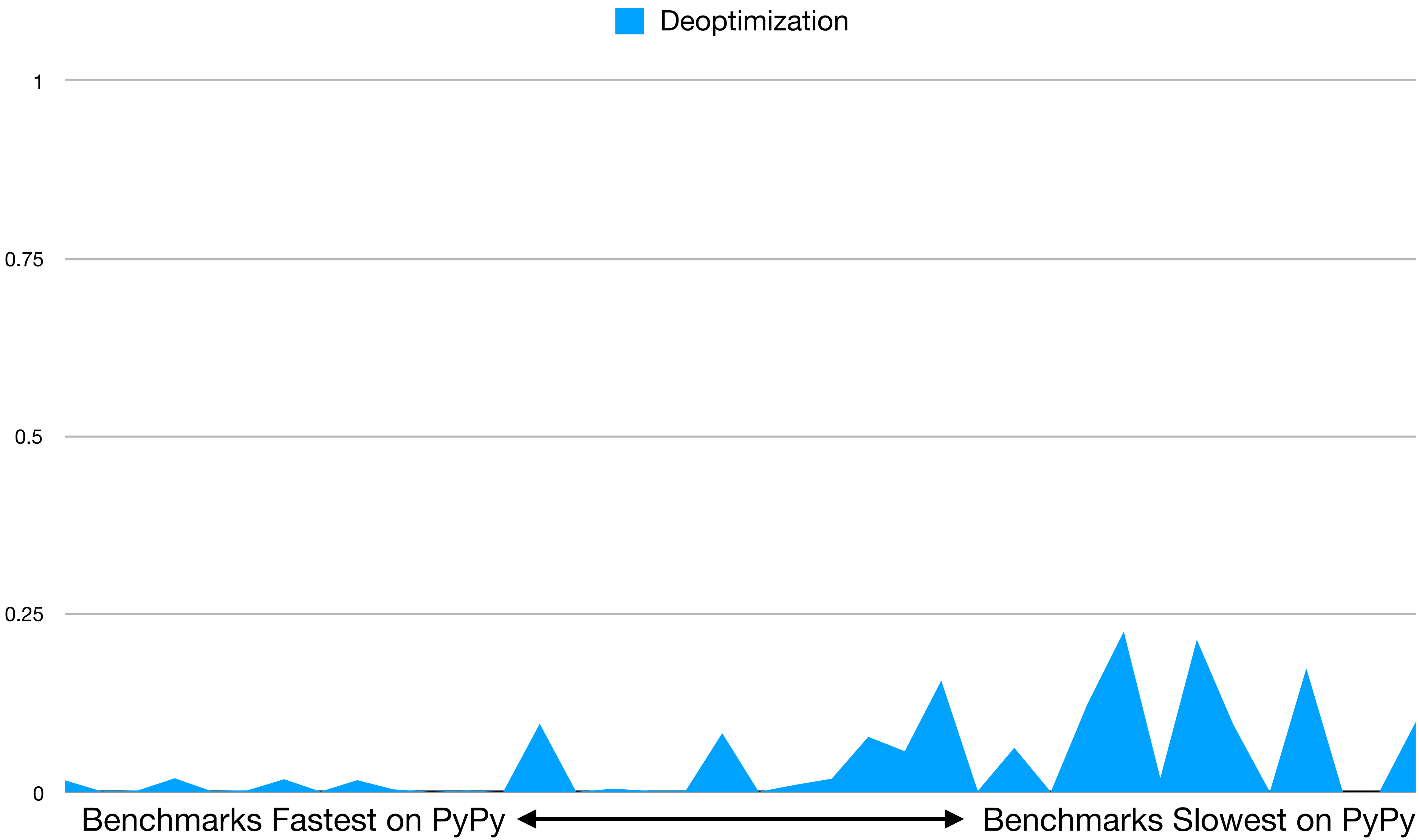
PyPy *without* meta-tracing JIT speedup over CPython: RPython-to-C translation has overheads



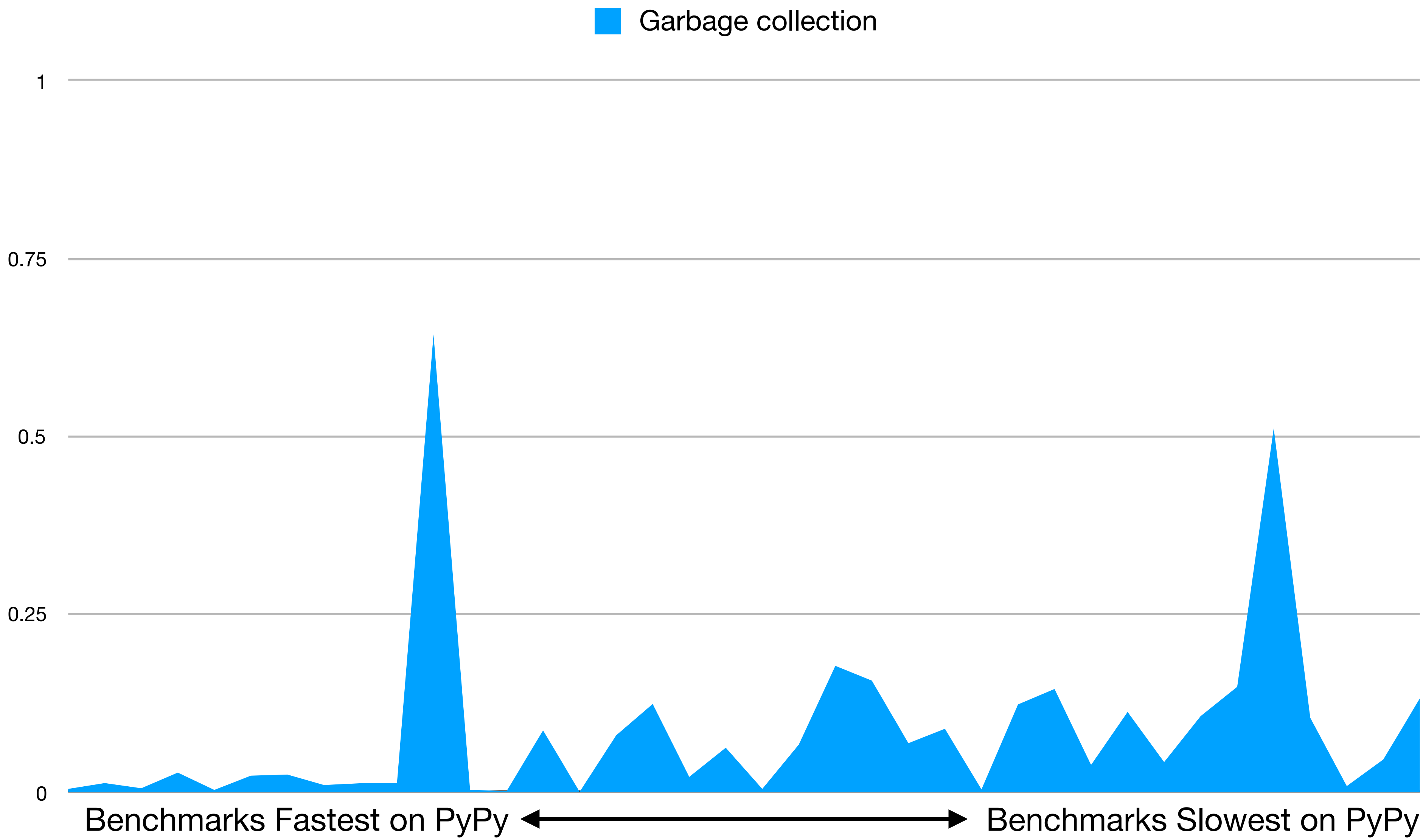
Tracing and optimization phase



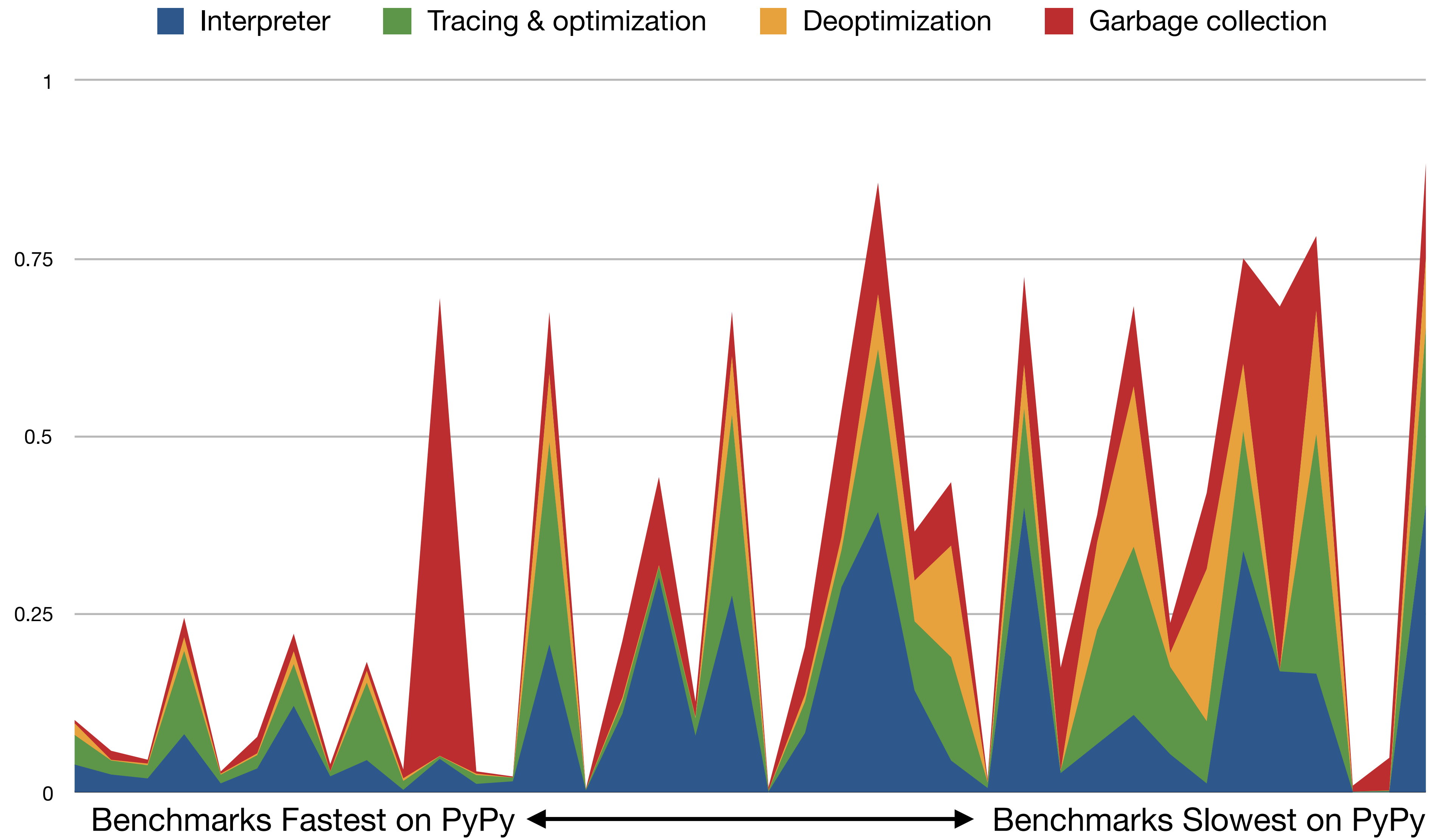
Deoptimization phase



Garbage collection phase



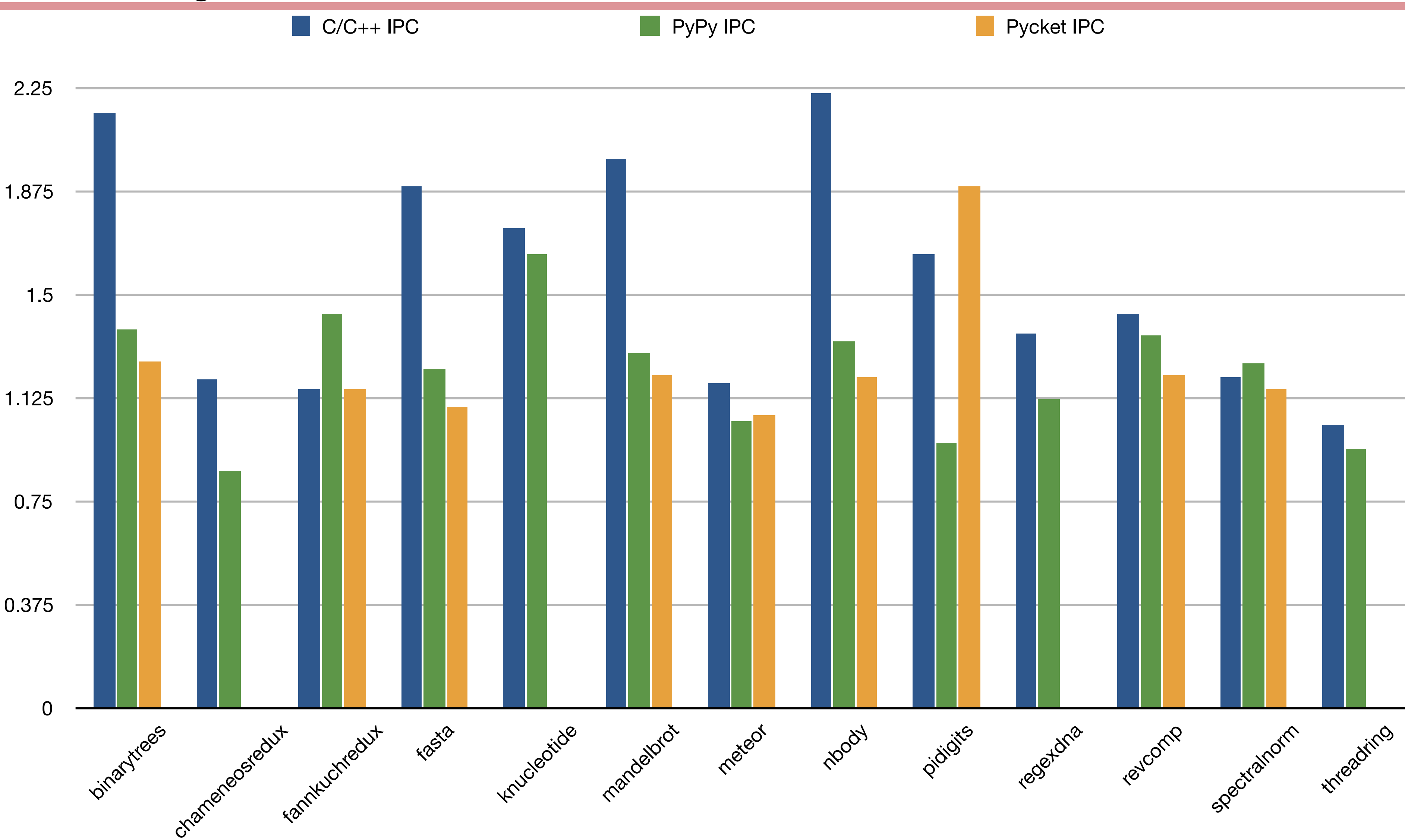
Meta-tracing JIT VM overheads:
Overheads are diverse and can add up to significant portion of execution



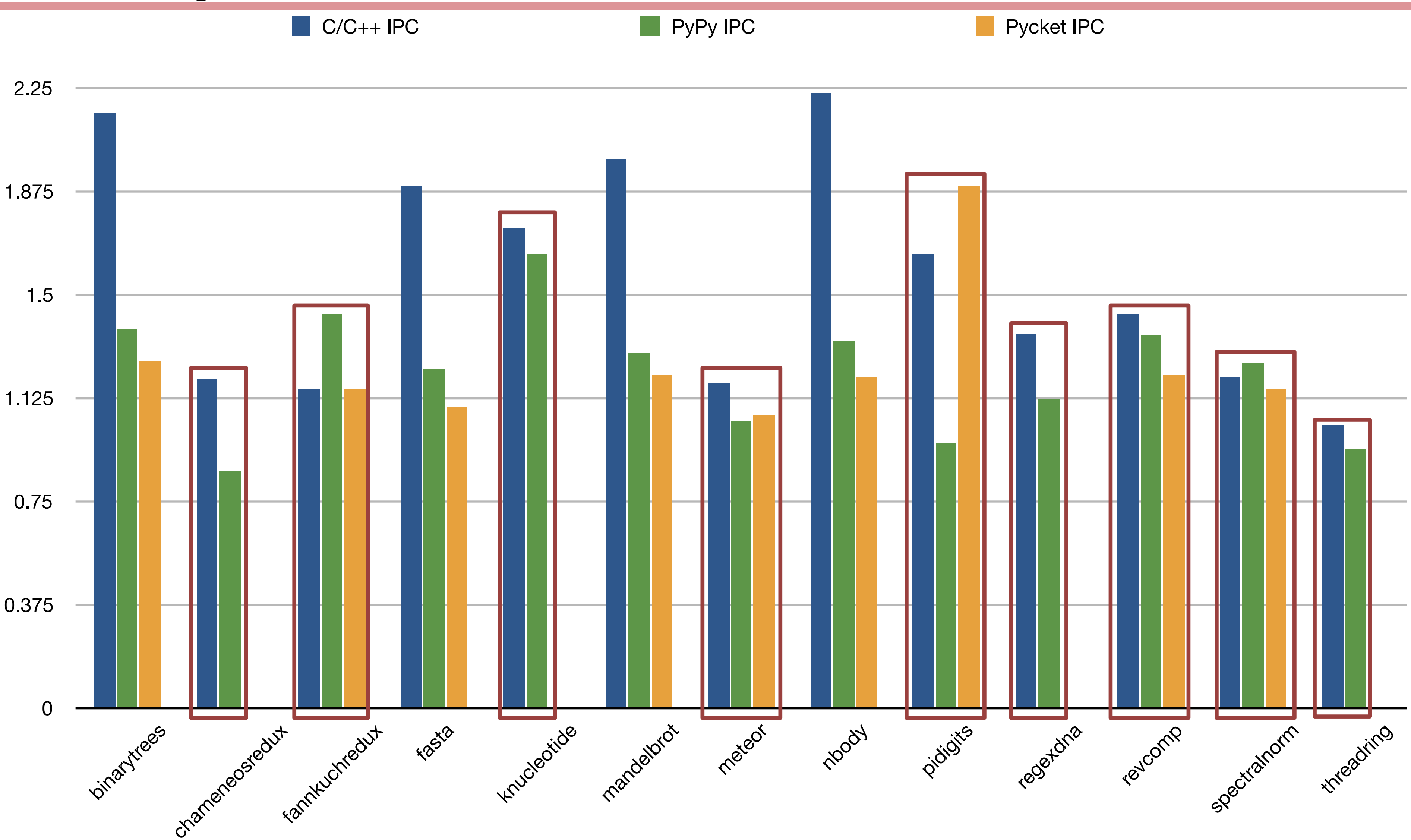
Iron law of processor performance:
Does meta-tracing VM code execute poorly in addition to more instructions?

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycle}}{\text{Instructions}} \times \frac{\text{Time}}{\text{Cycle}}$$

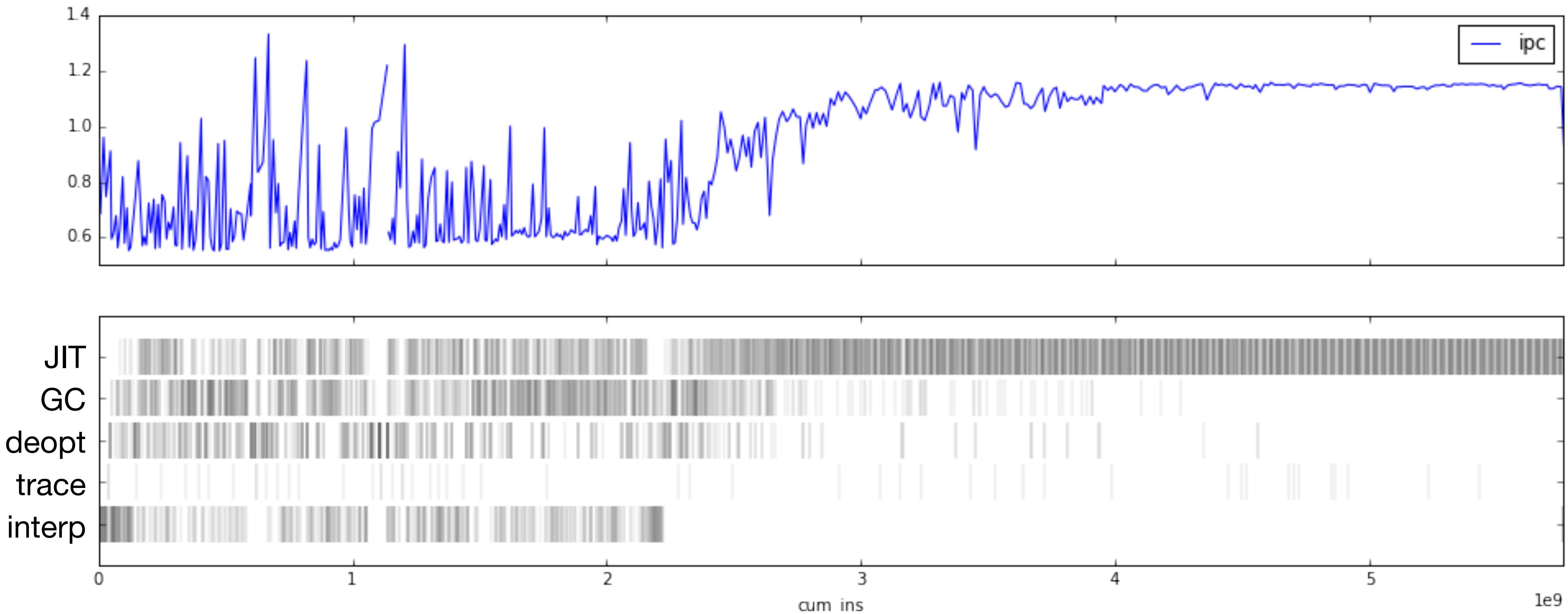
Comparing meta-tracing JIT IPC to C/C++: Meta-tracing has a similar IPC for most benchmarks



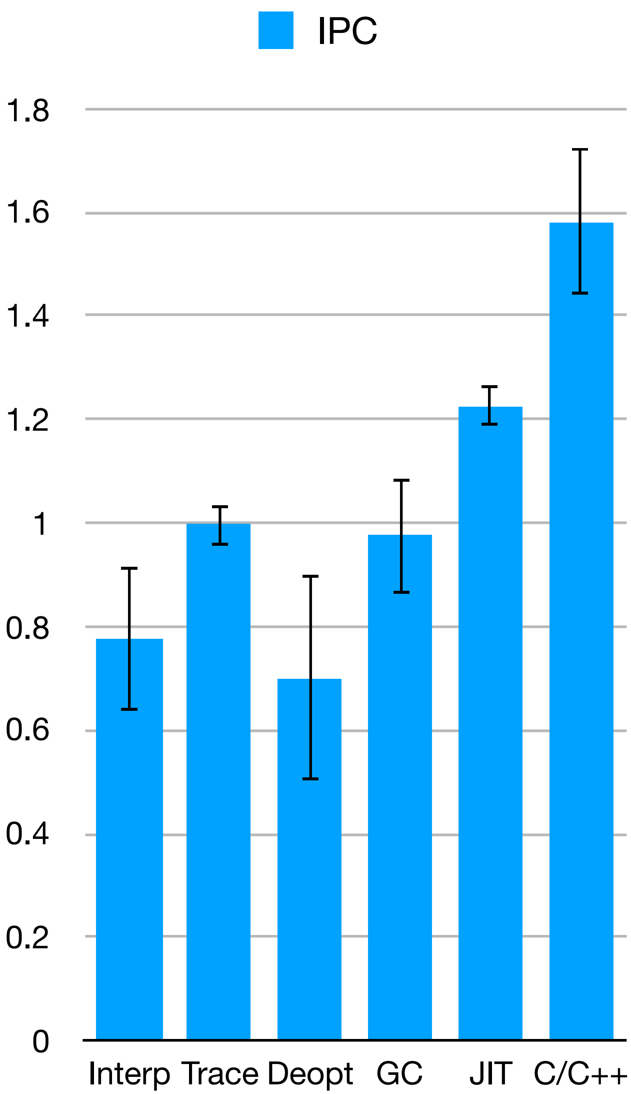
Comparing meta-tracing JIT IPC to C/C++: Meta-tracing has a similar IPC for most benchmarks



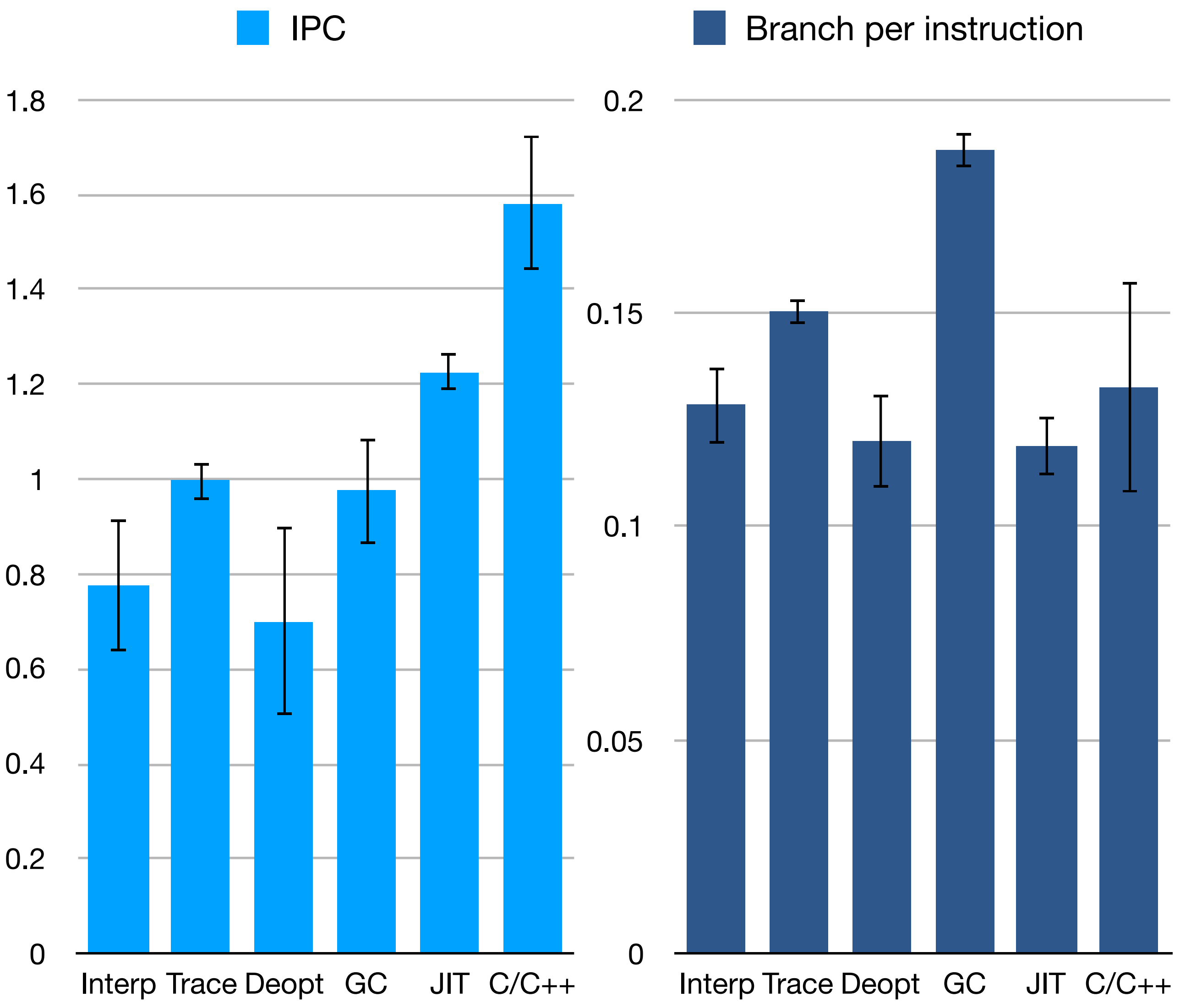
IPC measurements can be accurately matched against VM phases



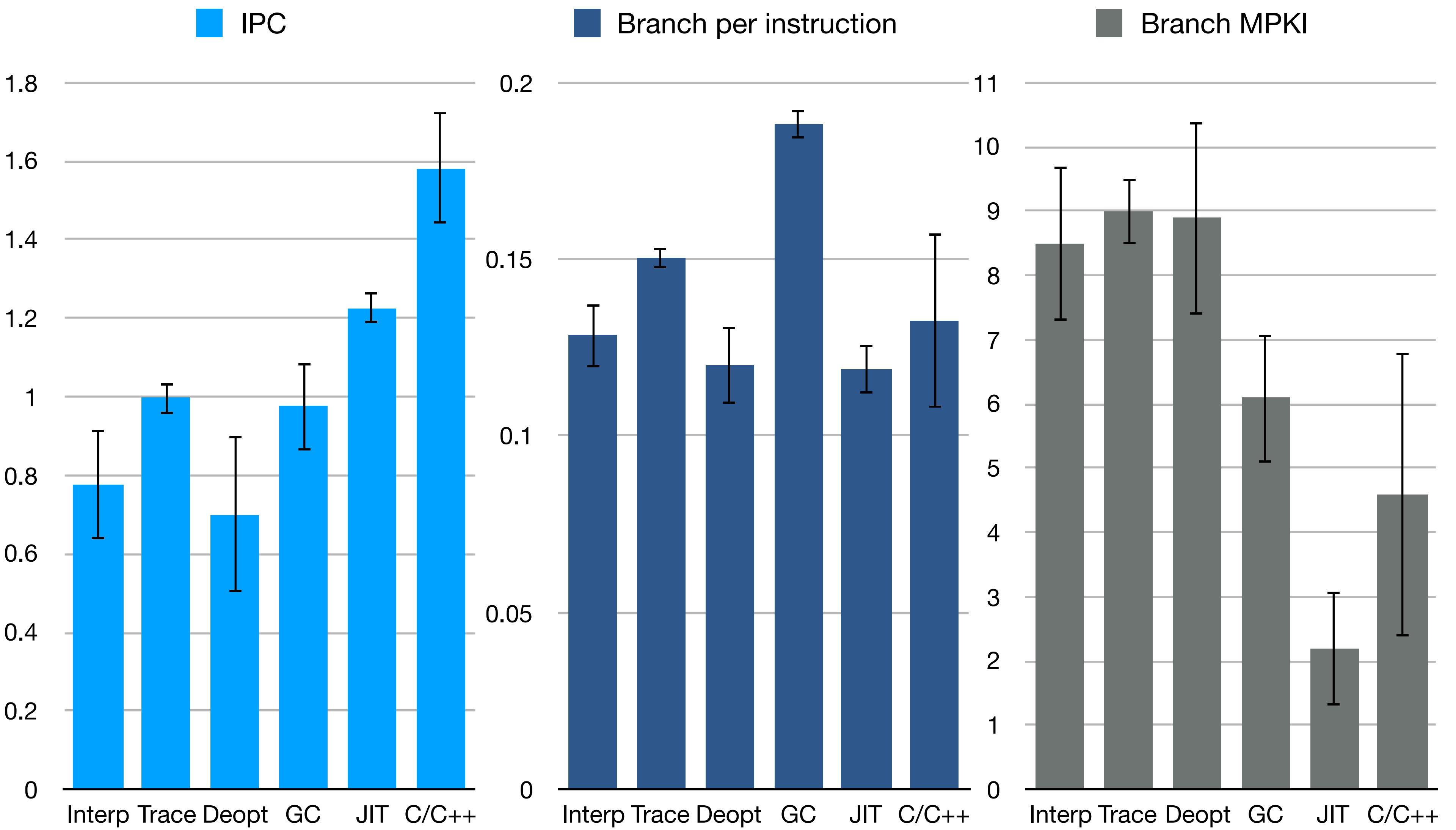
Microarchitectural characterization by the VM phase:
Meta-tracing-JIT-compiled code has a similar IPC, fewer branches and mispredictions



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 >> 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++***

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***

Cornell University
Computer Systems Laboratory

CSL

Cross-Layer Workload Characterization of Meta-Tracing JIT VMs

Berkin Ilbeyi[1], Carl Friedrich Bolz-Tereick[2], and Christopher Batten[1]

[1] Cornell University, [2] Heinrich-Heine-Universität Düsseldorf