cinder

WE DIDN’T START THE FIRE

Max Bernstein
Chief Potato

Meta

Agenda

01 Motivation

02 History of solving this @ Meta

a Skybison

b Cinder

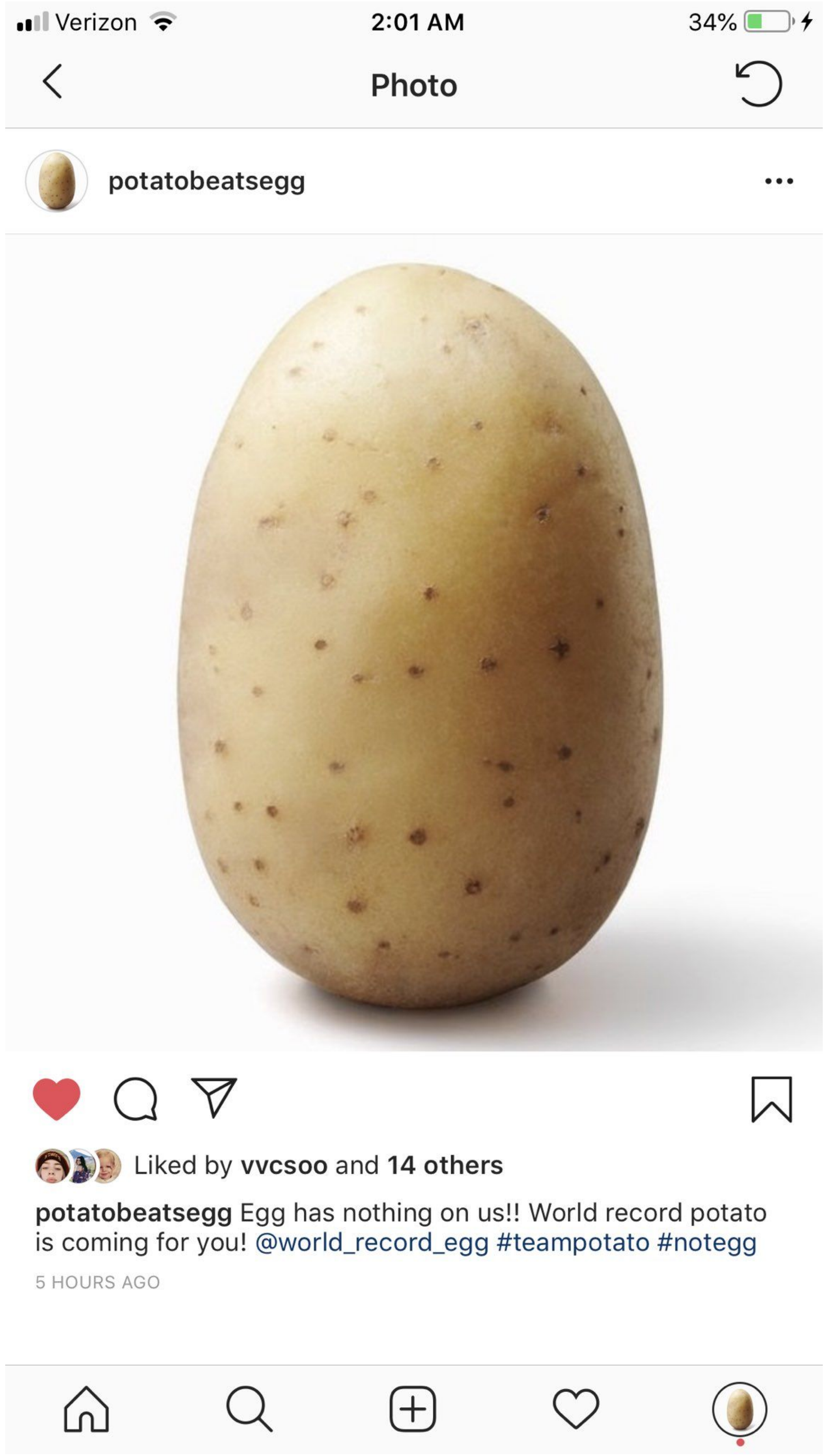
03 Big constraints we have

04 Cinder

Please ask me questions along the way

INSTAGRAM

- Bazillion machines
- Bazillion lines of code (mostly Python)
- Bazillion users posting and viewing stuff



Liked by **vvcsoo** and **14 others**

potatobeatsegg Egg has nothing on us!! World record potato is coming for you! @world_record_egg #teampotato #notegg

OUR SETUP

- One main process that boots the web server
- Forked single-threaded workers (over 20 per machine)
- Deploy frequently (every ~15 minutes)
- A large amount of native extensions (for perf and other things)

IT IS THE YEAR 2017

- Instagram is one of the world's largest deployments of Python
- Server growth is looking exponential **(!!!)**

- You can only optimize application code so much
- I am an intern and then studying in Germany

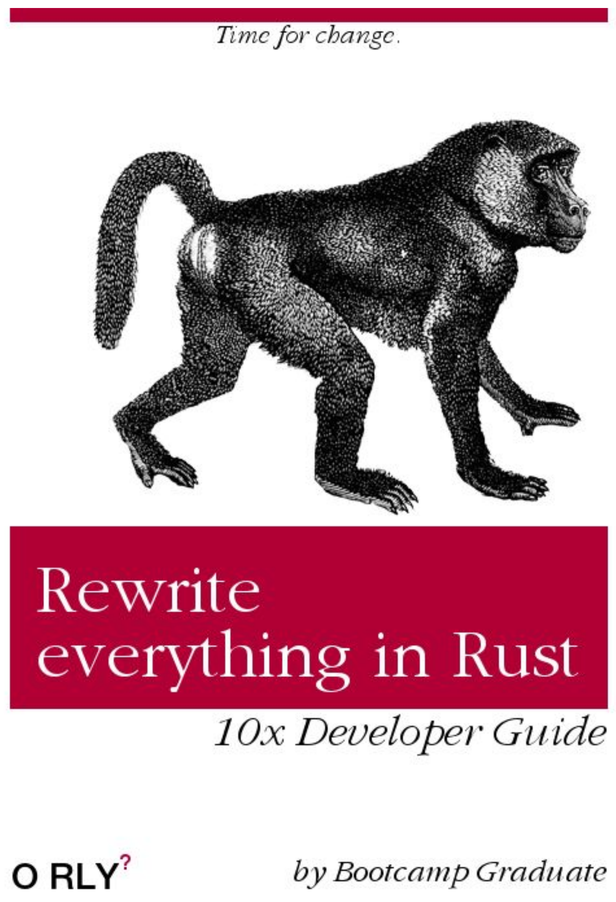
What can you do?

Building a compiler was not our first choice...

DELETE INSTAGRAM

REWRITE THE APP

- Stop-the-world *or* incrementally
- …but people *really* like Python
- …and there is so *so* much of it



REWRITE HOT CODE PATHS

- Native extensions in Rust/C/C++/Cython
- Compilation slows developers down
- Doesn’t integrate with our Python tooling

USE A FASTER PYTHON RUNTIME

- Everyone’s first question for us
- Modest performance improvements
- Needed to significantly modify application
- A lot of C extensions
- Multiprocessing/forked workers??

BUILD IT FROM SCRATCH: SKYBISON

- Skybison started ~2018
- New object model, caching, new interpreter, moving GC
- Going to take *some time* to get off the ground

In the meantime...

TEMPORARILY OPTIMIZE CPYTHON: CINDER

- Incremental wins to stave off capacity crunch
- Only needed until we have our bigger better solution
- Runtime code optimization
- Interpreter performance optimization (inline caching, etc)
- Simple code generation
- Can we optimize CPython enough to be sustainable long term?

CONSTRAINTS

- Need performance *now*
- C extensions all over the place, internal and external
- Keep startup time fast
- Keep memory low ([used to be memory bound]())
- Everything is async
- People love fast deploys

~~TEMPORARILY~~ OPTIMIZE CPYTHON: CINDER

- The C API claimed another victim. We wound down Skybison.
  - “How many C extensions are we going to have to modify to use the Limited API?”
  - HPy, we need you
- Easier to guarantee correctness if you build on CPython
- Cinder has C API, if not ABI, compatibility long and short term
- **Cinder is fast enough. Time to focus on Cinder.**

Presenting cinder

HIGH-LEVEL OVERVIEW

- Immortal objects
- Shadow code + cache invalidation machinery (forked interpreter)
- Just-in-time compiler
- Strict Modules
- Static Python
- Lazy imports
- Async optimizations

IMMORTAL OBJECTS

- Not refcounted
- Immortalize the heap pre-fork
- *Not* immutable
- Avoids refcounting-related copy-on-write
- ~5% gCPU

- Similar win in GC

SHADOW CODE

- Kicks in after N executions in the interpreter
- Inline caching in bytecode
- Bytecode quickening
- Need to invalidate caches when types change…
- Despite CoW, ~10% gCPU

DICT/TYPE/FUNCTION WATCHERS

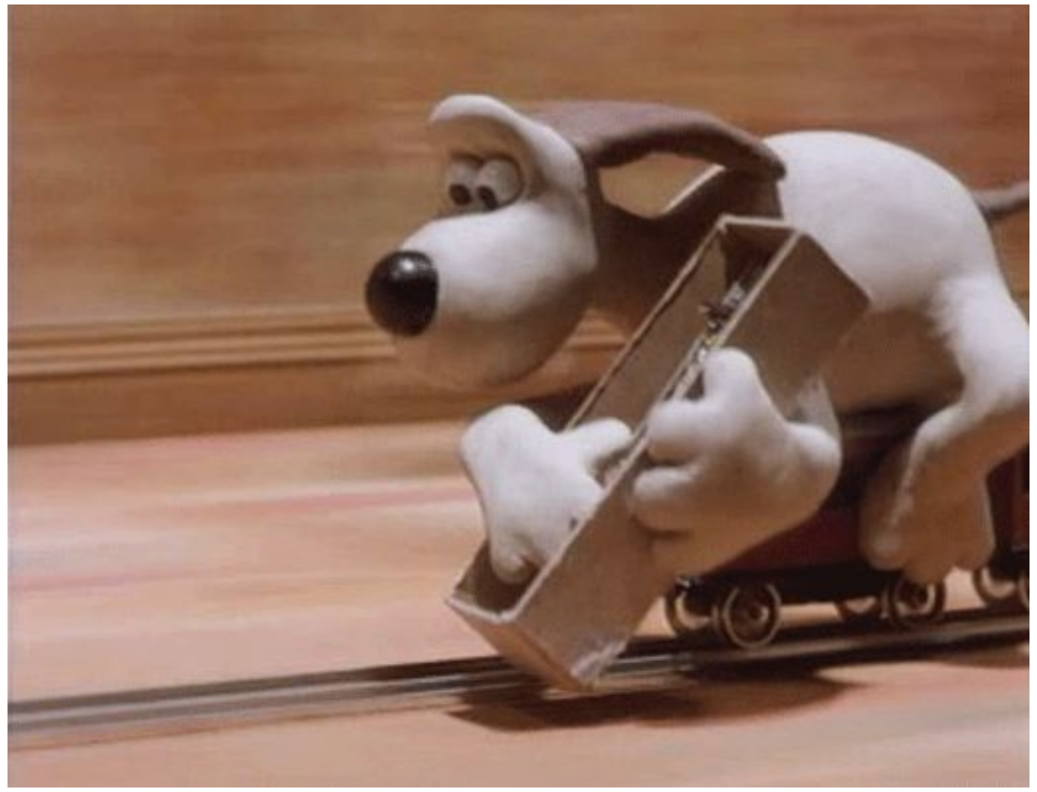
- Hooks for modifications to
  - types and supertypes
  - functions
  - globals
- Means we can skip checking in the common case

We will be using trycinder.com for demos!

METHOD JIT COMPILER

1. Bytecode
2. SSA HIR (type inference, inliner, type specialization, dead code elimination, ...)
3. SSA LIR
4. Register-allocated LIR
5. Assembly
6. ~10% gCPU

Can deopt (side-exit) into the interpreter



DEOPT (OSR)

- For exceptions, invalidated assumptions, …
- Reify stack and heap frames needed
- Call into the interpreter loop
- Until end of current compilation unit (including inlined frames)
- Very rare

```
def foo():
  return 3

def bar():
  return 4

def test():
  return foo() + bar()
```

```
(8) 0: LOAD_GLOBAL 0: foo
(8) 2: CALL_FUNCTION 0
(8) 4: LOAD_GLOBAL 1: bar
(8) 6: CALL_FUNCTION 0
(8) 8: BINARY_ADD 0
(8) 10: RETURN_VALUE 0
```

If only we could bind names statically…

```
(8) v5:OptObject = LoadGlobalCached<0; "foo">
(?) v6:MortalFunc[function:...] = GuardIs<0x7fe671d5ed30> v5
(8) v16:Object = LoadField<func_code@16, Object, borrowed> v6
(?) v17:MortalCode["foo"] = GuardIs<0x7fe671dbff30> v16
(8) v14:MortalLongExact[3] = LoadConst<MortalLongExact[3]>
(8) v8:OptObject = LoadGlobalCached<1; "bar">
(?) v9:MortalFunc[function:...] = GuardIs<0x7fe671d5ee50> v8
(8) v22:Object = LoadField<func_code@16, Object, borrowed> v9
(?) v23:MortalCode["bar"] = GuardIs<0x7fe671dbfe70> v22
(8) v20:MortalLongExact[4] = LoadConst<MortalLongExact[4]>
(8) v25:MortalLongExact[7] = LoadConst<MortalLongExact[7]>
(8) Return v25
```

Initial LIR

```
(8) %4:Object = Move [0x12f6080]:Object
(8) Guard 3(0x3):64bit 0(0x0):Object %4:Object …(0x7fc1e6265d30):Object %4:Object
(8) %6:Object = Move [%4:Object + 0x10]:Object
(8) Guard 3(0x3):64bit 1(0x1):Object %6:Object …(0x7fc1e62c6f30):Object %6:Object %4:Object
(8) %8:Object = Move 16100224(0xf5ab80):Object
(8) %9:Object = Move [0x12f6100]:Object
(8) Guard 3(0x3):64bit 2(0x2):Object %9:Object …(0x7fc1e6265e50):Object %9:Object %8:Object
(8) %11:Object = Move [%9:Object + 0x10]:Object
(8) Guard 3(0x3):64bit 3(0x3):Object %11:Object …(0x7fc1e62c6e70):Object %11:Object %9:Object %8:Object
(8) %13:Object = Move 16100352(0xf5ac00):Object
(8) %14:Object = Move [%13:Object]:Object
(8) Inc %14:Object
(8) [%13:Object]:Object = Move %14:Object
```

Register-allocated LIR

```
(8) RAX:Object = Move [0x2200080]:Object
(8) RCX:Object = Move 140662143380784(0x7fee7516ed30):64bit
(8) Guard 3(0x3):64bit 0(0x0):Object RAX:Object RCX:Object RAX:Object
(8) RCX:Object = Move [RAX:Object + 0x10]:Object
(8) RDX:Object = Move 140662143778608(0x7fee751cff30):64bit
(8) Guard 3(0x3):64bit 1(0x1):Object RCX:Object RDX:Object RCX:Object RAX:Object
(8) RAX:Object = Move 16100224(0xf5ab80):Object
(8) RCX:Object = Move [0x2200100]:Object
(8) RDX:Object = Move 140662143381072(0x7fee7516ee50):64bit
(8) Guard 3(0x3):64bit 2(0x2):Object RCX:Object RDX:Object RCX:Object RAX:Object
(8) RDX:Object = Move [RCX:Object + 0x10]:Object
(8) RBX:Object = Move 140662143778416(0x7fee751cfe70):64bit
(8) Guard 3(0x3):64bit 3(0x3):Object RDX:Object RBX:Object RDX:Object RCX:Object RAX:Object
(8) RAX:Object = Move 16100352(0xf5ac00):Object
(8) RCX:Object = Move [RAX:Object]:Object
(8) Inc RCX:Object
(8) [RAX:Object]:Object = Move RCX:Object
```

Left to right HIR+asm

```
(8) v5:OptObject = LoadGlobalCached<0; "foo">
    0x7f071a2bfa27: mov rax,QWORD PTR ds:0x1929080

(8) v6:MortalFunc[function:...] = GuardIs<0x7f071a0a5d30> v5
    0x7f071a2bfa2f: movabs rcx,0x7f071a0a5d30
    0x7f071a2bfa39: cmp rax,rcx
    0x7f071a2bfa3c: jne 0x7f071a2bfad7

(8) v16:Object = LoadField<func_code@16, Object, borrowed> v6
    0x7f071a2bfa42: mov rcx,QWORD PTR [rax+0x10]

(8) v17:MortalCode["foo"] = GuardIs<0x7f071a106f30> v16
    0x7f071a2bfa46: movabs rdx,0x7f071a106f30
    0x7f071a2bfa50: cmp rcx,rdx
    0x7f071a2bfa53: jne 0x7f071a2bfade

(8) v14:MortalLongExact[3] = LoadConst<MortalLongExact[3]>
    0x7f071a2bfa59: mov rax,0xf5ab80

(8) v8:OptObject = LoadGlobalCached<1; "bar">
    0x7f071a2bfa60: mov rcx,QWORD PTR ds:0x1929100
```

```
(8) v9:MortalFunc[function:...] = GuardIs<0x7f071a0a5e50> v8
    0x7f071a2bfa68: movabs rdx,0x7f071a0a5e50
    0x7f071a2bfa72: cmp rcx,rdx
    0x7f071a2bfa75: jne 0x7f071a2bfae5

(8) v22:Object = LoadField<func_code@16, Object, borrowed> v9
    0x7f071a2bfa7b: mov rdx,QWORD PTR [rcx+0x10]

(8) v23:MortalCode["bar"] = GuardIs<0x7f071a106e70> v22
    0x7f071a2bfa7f: movabs rbx,0x7f071a106e70
    0x7f071a2bfa89: cmp rdx,rbx
    0x7f071a2bfa8c: jne 0x7f071a2bfaec

(8) v25:MortalLongExact[7] = LoadConst<MortalLongExact[7]>
    0x7f071a2bfa92: mov rax,0xf5ac00

(8) Incref v25
    0x7f071a2bfa99: mov rcx,QWORD PTR [rax]
    0x7f071a2bfa9c: inc rcx
    0x7f071a2bfa9f: mov QWORD PTR [rax],rcx
```

JIT PECULIARITIES

- Many forked workers => pre-fork compilation
- Static JIT list (will be dynamic later)
- So how do we warm up if we never run the code?

HOW DO WE GET TYPE INFORMATION?

- Normally: multi-stage JIT with run-time profiling
- But time spent compiling in workers has opportunity cost
- And type annotations are not so helpful…

TYPE HINTS?

```
def add(x: int, y: int) -> int:
    return x + y


class C(int):
    def __add__(self, other):
        print("Haha no")
        return 7

print(add(C(1), C(2)))
```

TYPE HINTS?

```
def character_at(left: str, right: int):
    return left[right]

character_at(1, 2)  # type: ignore
```

```
sequoia% python3 -m mypy lies.py
Success: no issues found in 1 source file
sequoia%
```

TYPE PROFILES

- Profile types in the interpreter (~5 machines, all the time)
- Ship histograms to a DB
- Process data into binary blob to ship to prod hosts
- Read type profile at boot
- For monomorphic profiles, *GuardType*
- For polymorphic profiles, *CondBranchCheckType* and polymorphic compiled code

STRICT MODULES

- No **top-level** side effects visible outside the module
- Comes with its own module loader
- Abstract interpreter of Python code
- Eventual goal: full transitive closure of strict modules

Auto slotification! Errors at load-time!

```
import __strict__

class C:
  def __init__(self):
      self.myattr = None

a = C()
a.my_attr = 42

# AttributeError: 'C' object has no attribute 'my_attr'
```

Read-only fields on types

```
class C:
    def f(self):
        return 42

a = C()
a.f = lambda: "I'm a special snowflake"

# AttributeError: 'C' object attribute 'f' is read-only
```

Limited to strict modules

```
import __strict__
from nonstrict_module import something

x = something()

# UnknownValueCallException: Call of unknown value 'something()' is prohibited at module level.
```

STATIC PYTHON

- New compiler and type checker with its own type system
- Use PEP 484 type hints that we already have for correctness
- **Replace C extension code**
- Verify types and names at bytecode compile time
- Generate specialized bytecode
- Can run in interpreter (boxed) or JIT (unboxed)
- *The code is managed!*
- ~10% gCPU

Remember that example from earlier?

```
import __static__

def foo():
  return 3

def bar():
  return 4

def test():
  return foo() + bar()
```

Tighter name binding!

```
(9) v17:MortalLongExact[7] = LoadConst<MortalLongExact[7]>
    0x7fdd974d28b3: mov rax,0xf5ac00

(9) Incref v17
    0x7fdd974d28ba: mov rcx,QWORD PTR [rax]
    0x7fdd974d28bd: inc rcx
    0x7fdd974d28c0: mov QWORD PTR [rax],rcx
```

Normal-looking Python code

```python
import __static__

class C:
    def __init__(self) -> None:
        self.a: int = 1

def test(instance: C) -> int:
    return instance.a
```

CPython bytecode

```
(7) 0: LOAD_FAST 0: instance
(7) 2: LOAD_ATTR 0: a
(7) 4: RETURN_VALUE 0
```

CPython interpreter machinery

```
PyObject *
_PyObject_GenericGetAttrWithDict(PyObject *obj, PyObject *name,
                                 PyObject *dict, int suppress)
{
     // …
    if (!PyUnicode_Check(name)){
        PyErr_Format(PyExc_TypeError,
                     "attribute name must be string, not '%.200s'",
                     Py_TYPE(name)->tp_name);
        return NULL;
    }
    Py_INCREF(name);
    // …
    descr = _PyType_Lookup(tp, name);
    f = NULL;
    if (descr != NULL) {
      // …
    }
    if (dict == NULL) {
      // …
    }
    if (dict != NULL) {
      // …
    }
    if (f != NULL) {
       // …
    }
    // …
}
```

```
for (Py_ssize_t i = 0; i < n; i++) {
    PyObject *base = PyTuple_GET_ITEM(mro, i);
    PyObject *dict = _PyType_CAST(base)->tp_dict;
    assert(dict && PyDict_Check(dict));
    res = _PyDict_GetItem_KnownHash(dict, name, hash);
    if (res != NULL) {
        break;
    }
    if (PyErr_Occurred()) {
        *error = -1;
        goto done;
    }
}
```

Static Python bytecode

```
(7) 0: CHECK_ARGS 1: (0, ('explorer_lib', 'C'))
(8) 2: LOAD_FAST 0: instance
(8) 4: LOAD_FIELD 2: ('explorer_lib', 'C', 'a')
(8) 6: RETURN_VALUE 0
```

JIT-compiled Static Python code

```
(8) v4:OptObject = LoadField<a@16, OptObject, borrowed> v2
    0x7fd1bb93d911: mov rax,QWORD PTR [rdi+0x10]

(8) v5:Object = CheckField<"a"> v4
    0x7fd1bb93d915: test rax,rax
    0x7fd1bb93d918: je 0x7fd1bb93d96a

(8) Incref v5
    0x7fd1bb93d91e: mov rcx,QWORD PTR [rax]
    0x7fd1bb93d921: bt rcx,0x3c
    0x7fd1bb93d926: jb 0x7fd1bb93d932
    0x7fd1bb93d92c: inc rcx
    0x7fd1bb93d92f: mov QWORD PTR [rax],rcx
```

Unboxed primitive types!

```
import __static__
from __static__ import int64

class Point:
  def __init__(self, x: int64, y: int64) -> None:
      self.x: int64 = x
      self.y: int64 = y

def test(point: Point) -> int64:
  return point.x + point.y
```

Stored unboxed and inline in the object!

Static Python bytecode

```
(9) 0: CHECK_ARGS 1: (0, ('explorer_lib', 'Point'))
(10) 2: LOAD_FAST 0: point
(10) 4: LOAD_FIELD 2: ('explorer_lib', 'Point', 'x')
(10) 6: LOAD_FAST 0: point
(10) 8: LOAD_FIELD 3: ('explorer_lib', 'Point', 'y')
(10) 10: PRIMITIVE_BINARY_OP 0
(10) 12: RETURN_PRIMITIVE 7
```

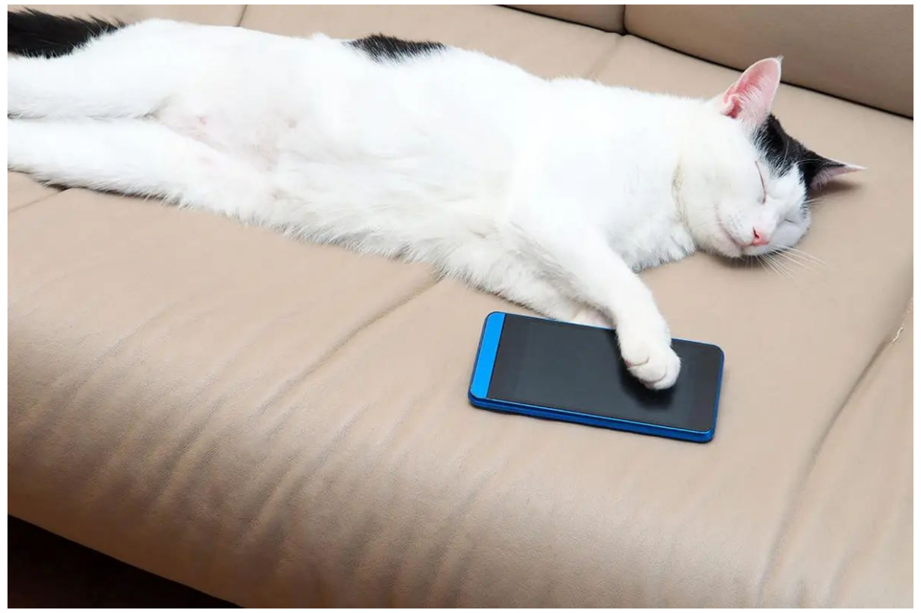
JIT-compiled Static Python code

```
(10) v6:CInt64 = LoadField<x@24, CInt64, borrowed> v4
     0x7ff62be24931: mov rax,QWORD PTR [rdi+0x18]

(10) v8:CInt64 = LoadField<y@16, CInt64, borrowed> v4
     0x7ff62be24935: mov rcx,QWORD PTR [rdi+0x10]

(10) v9:CInt64 = IntBinaryOp<Add> v6 v8
     0x7ff62be24939: add rax,rcx
```

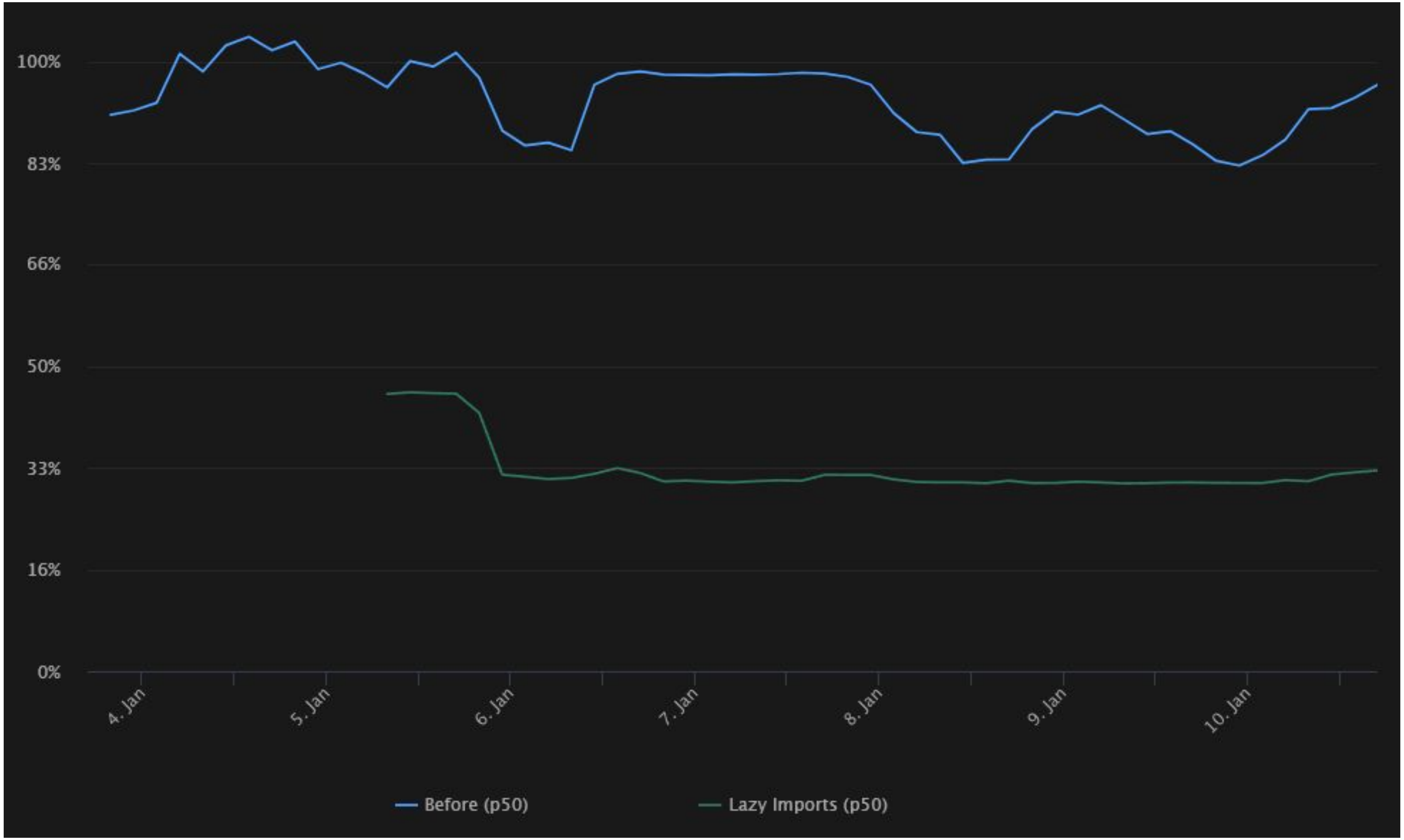
Zero-initialized! No need to check if set.



LAZY IMPORTS

- Inside the runtime and transparent to user code
- Stub out modules, objects, functions, etc
- Import module when imported objects are first referenced
  - Both managed and C extension access of names
- Improve developer experience in *startup time*
- Works pretty well with Strict Modules to minimize import gotchas
- Working on upstreaming this (PEP 690!)

Cutting p50 load times in half... many minutes saved



ASYNC OPTIMIZATIONS

- Exception-free returns for coroutines
  - bpo-41756 — ~5% gCPU
  - bpo-42085 — 1.5% gCPU
- Await-aware calls — 2% gCPU
  - Eagerly evaluate up to completion (no allocation!), or up to its first suspension
- Method table dispatch for asyncio components — ~1% gCPU

```
async def first_callee():
    return 3

async def second_callee():
    return 4

async def do_something_important():
    result = await first_callee()
    other_result = await second_callee()
    return result + other_result

import asyncio
print(asyncio.run(do_something_important()))
```

No yield! Why wait?

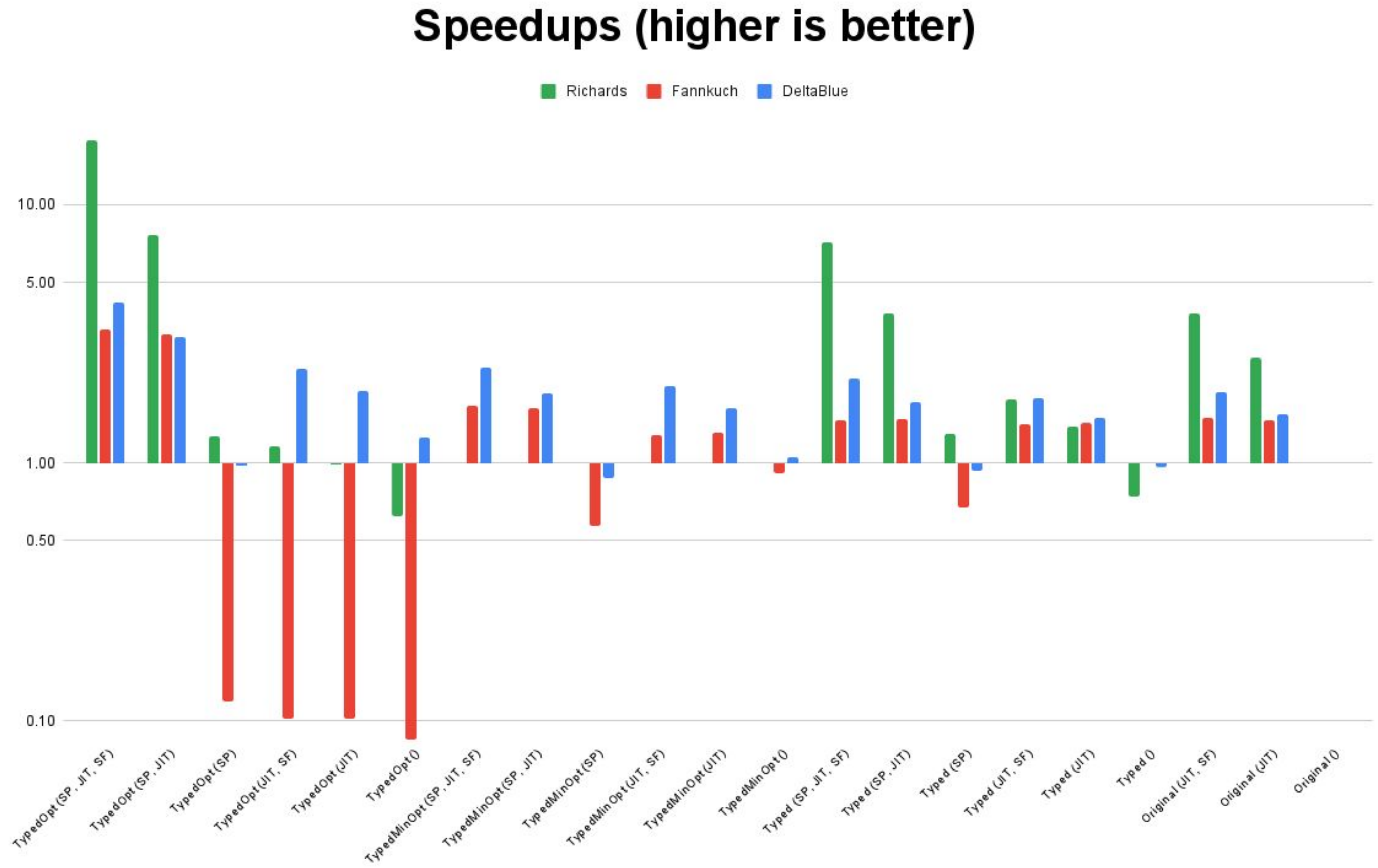
RECAP

- Immortal objects
- Shadow code + cache invalidation machinery
- Just-in-time compiler
- Strict Modules
- Static Python
- Lazy imports
- Async optimizations

IMPACT

- We have brought Python down to ~30% of gCPU in the app
- We’ve seen ~45% gCPU improvement for our production application
- Application developers opt into stricter typing when it provides reliability and performance benefits
    - Often replacing native extensions…!
- Working with CPython folks to upstream the relevant hooks
    - Helps Pyjion, Pyston, etc as well

Bonus: microbenchmarks



CONCLUSION

- Cinder was instrumental in keeping the lights on
- It *is* possible to optimize large Python applications
- Sometimes building a compiler is the right path

trycinder.com

github.com/facebookincubator/cinder

Meta