PYTHON PERFORMANCE AT SCALE

Making Python Faster at Instagram

PYTHON PERFORMANCE AT SCALE

1 Successful Improvements

2 Experimental Work

3 Results and What's Next

PYTHON AT INSTAGRAM

A super fast review!

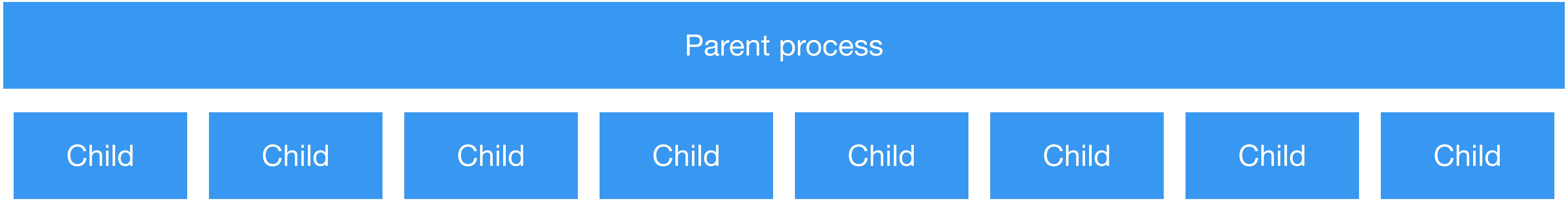
MONOLITHIC WEB APPLICATION

django

3.8

uWSGI

MONOLITHIC WEB APPLICATION



PROFILING

- Profiling data collected from production hosts
  - Linux perf sampling profiler
  - Tweaks for better profiling data (async call stacks)
  - Provides insight at Python and C level
- Metrics
  - RPS - Requests per Second under Load
  - Not stable over time, but good for short-term measurements of wins/losses

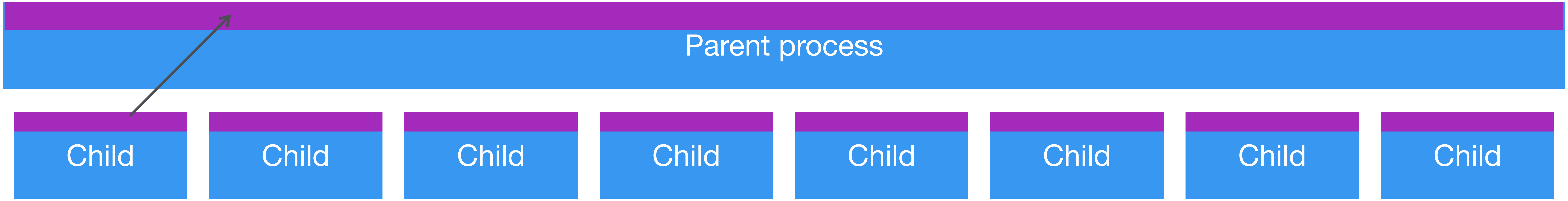
SUCCESSFUL IMPROVEMENTS

https://github.com/facebookincubator/cinder

IMMORTAL OBJECTS

Increasing shared memory

- Worker processes share read-only memory with parent process
- Becomes private to the worker process if the worker process writes to it



IMMORTAL OBJECTS

- Large source of writes is from reference counts to objects
- Uses a high-bit in ref count to mark objects as immortal
- Updates Py_INCREF/Py_DECREF to check for bit, and not update ref count
  - A significant amount of overhead, but the memory savings make it worth it in our workload
- Pre-fork the heap is collected and traversed
- All living objects are marked as immortal
- 5% win in production

ASYNC I/O

We do a lot of it!

- Send/Receive values without StopIteration
  - Creating exception objects was a major source of overhead
  - Simple benchmark is 1.6x times faster
- Upstreamed to Python 3.10
  - bpo-41756, bpo-42085
- 5% win in production

ASYNC I/O

We do a lot of it!

- Eager Evaluation
  - "await some_call()" will immediately run function
  - If call completes without blocking:
    - Coroutine creation is elided
    - A "wait handle" is returned
      - One singleton instance is used, as the handle is immediately consumed
  - Uses a new vectorcall flag to indicate a call is awaited
  - asyncio.gather also checks flag, and avoids task creation/scheduling overhead
  - 3% win in production

INLINE CACHING OF BYTE CODE

"shadow byte code"

- Hot methods get hidden copy of byte code ("shadow code") and caches
- Opcodes get replaced with more specific versions
- Over a 5% win in production

```
typedef struct _PyShadowCode {
    PyObject ***globals;
    Py_ssize_t globals_size;

    _ShadowCache l1_cache;

    _PyShadow_InstanceAttrEntry ***polymorphic_caches;
    Py_ssize_t polymorphic_caches_size;

    Py_ssize_t update_count;
    Py_ssize_t len;

    _Py_CODEUNIT code[];
} _PyShadowCode;
```

INLINE CACHING OF BYTE CODE

"shadow byte code"

LOAD_ATTR

- LOAD_ATTR_DICT_DESCR
- LOAD_ATTR_NO_DICT_DESCR
- LOAD_ATTR_DICT_NO_DESCR
- LOAD_ATTR_SPLIT_DICT
- LOAD_ATTR_SPLIT_DICT_DESCR
- LOAD_ATTR_TYPE
- LOAD_ATTR_MODULE
- LOAD_ATTR_SLOT
- LOAD_ATTR_POLYMORPHIC

STORE_ATTR

- STORE_ATTR_DICT
- STORE_ATTR_SPLIT_DICT
- STORE_ATTR_DESCR
- STORE_ATTR_SLOT

LOAD_GLOBAL

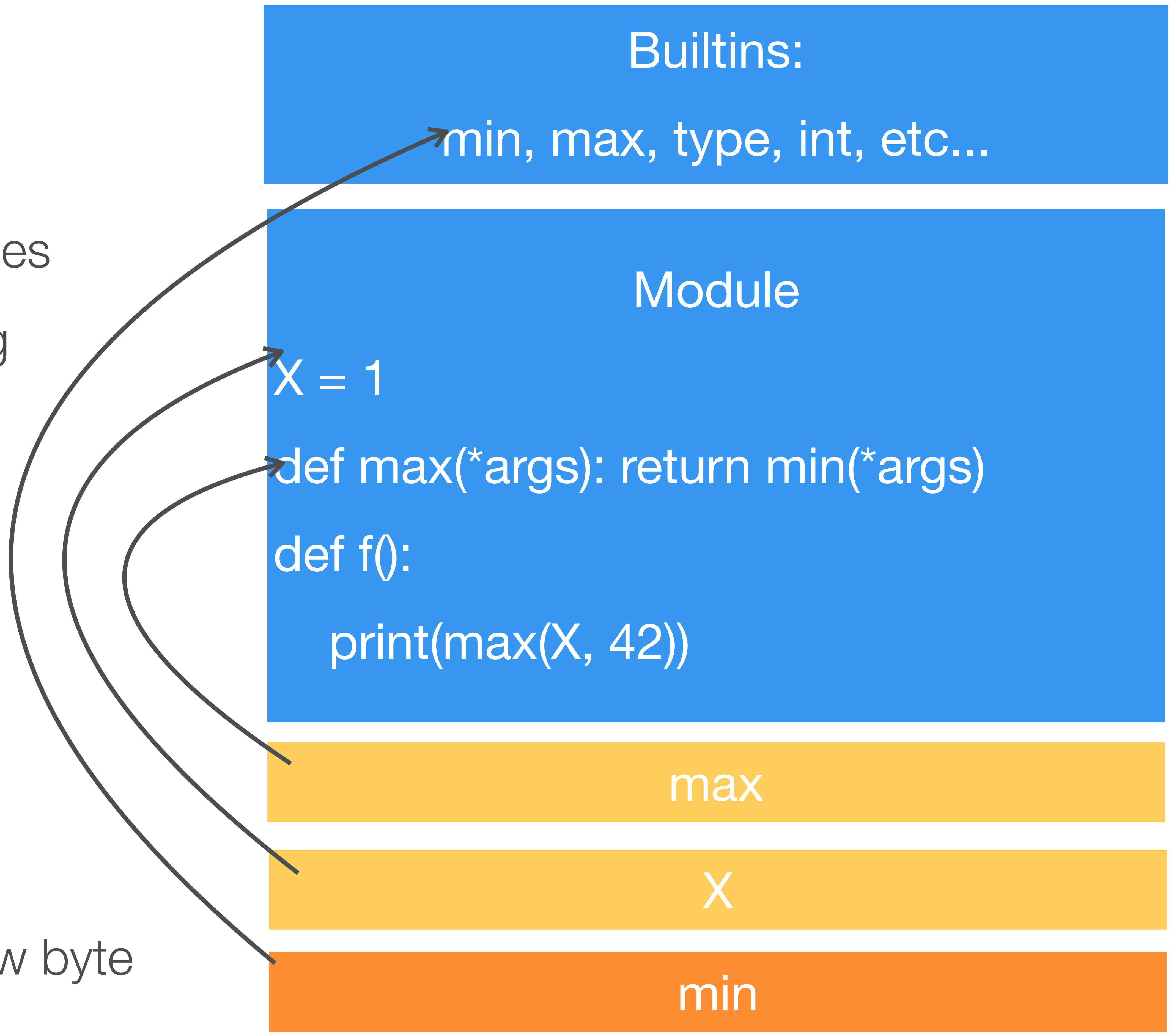
- LOAD_GLOBAL_CACHED

BINARY_SUBSCR

- BINARY_SUBSCR_LIST
- BINARY_SUBSCR_TUPLE
- BINARY_SUBSCR_DICT
- BINARY_SUBSCR_DICT_STR
- BINARY_SUBSCR_TUPLE_CONST_INT

DICTIONARY WATCHERS

- Provides updates to globals, builtins when modified
  - Re-uses existing version tag to mark watched dictionaries
  - dictionaries marked with low bit in dictionary version tag
  - dictionary versions bumped by 2



- Led to an additional 5% win when integrated with shadow byte code

TARGETED OPTIMIZATIONS

- Fixed __builtins__ (1%)
  - Technically a CPython implementation detail

TARGETED OPTIMIZATIONS

- PyType_Lookup
  - bpo-43452
  - Up to 1.19x faster on nbody, minimum 1.03x improvement across dozens of benchmarks
  - No measurable difference in production

TARGETED OPTIMIZATIONS

- ThreadState lookup avoidance
- Prefetching (~1%)
  - Frame creation

BUILD SYSTEM IMPROVEMENTS

- Profile Guided Optimizations (PGO) + (Thin)LTO
- Binary Optimization and Layout Tool (BOLT) - 4%
  - Currently training against production hosts
- Huge Pages - ~3%
  - Helps reduce iTLB misses

EXPERIMENTAL CHANGES

JIT, Static Python, Pyro

JIT

- Custom method at a time JIT
- Nearly full coverage for all opcodes
    - Unsupported opcodes are rare, or not used in methods (e.g. IMPORT_STAR)

JIT

Front End

- Front end lowers to HIR
  - SSA
  - Ref count insertion
  - Other optimization passes

```
def f(self):
    self.x = 1
```

```
fun __main__:f {
  bb 0 {
    v0 = LoadArg<0; "self">
    v1 = LoadConst<LongExact[1]>
    v0 = CheckVar<0; "self"> v0
    v2 = StoreAttr<0; "x"> v0 v1
    v3 = LoadConst<NoneType>
    Return v3
  }
}
```

```
fun __main__:f {
  bb 0 {
    v4:Object = LoadArg<0; "self">
    v5:LongExact[1] = LoadConst<LongExact[1]>
    v7:NoneType = StoreAttr<0; "x"> v4 v5
    v8:NoneType = LoadConst<NoneType>
    Incref v8
    Return v8
  }
}
```

JIT

Back End

- Backend lowers to LIR
  - Register allocation
  - Targeted optimizations while lowering:
    - Direct dispatch to known functions
- asmjit used for x64 code generation

```
def f(self):
    self.x = 1
```

```
BB %0 - succs: %3
       %1:Object = Bind R10:Object
       %2:Object = Bind R11:Object
# v4:Object = LoadArg<0; "self">
       %4:Object = Bind RDI:Object
# v5:LongExact[1] = LoadConst<LongExact[1]>
       %5:Object = Move 0x7f65cce1f1a0:Object
# v7:NoneType = StoreAttr<0; "x"> v4 v5
       %6:Object = Call ...
# v8:NoneType = LoadConst<NoneType>
       %8:Object = Move 0x7f65ccdef900:Object
# Incref v8
       %9:Object = Move [%8:Object]:Object
       BitTest %9:Object, 60(0x3c):Object
       BranchB
# Return v8
       Return %8:Object
```

STATIC PYTHON

Provides similar performance gains to MyPyC or Cython, but with a normal Python programming experience, and no extra compile steps.

Source Loader

Loads files marked with import __static__, supports cross module compilation

Byte Codes

Opcodes like INVOKE_FUNCTION, LOAD_FIELD, which work on metadata

PEP 484 Annotations

Normal annotations are leveraged, several new types like int64 are defined.

Interop

Type safety is enforced at boundaries of untyped Python, and elided within static Python

Static Compiler

Uses normal Python AST module, written in Python, based upon updated Python 2.x "compiler" package

STATIC PYTHON

```
from __future__ import annotations
import __static__
from __static__ import int64

from typing import Final, Optional
MUL: Final[int] = 1

class C:
  def __init__(self, next: Optional[C] = None):
    self.next = next
    if next is not None:
      self.len: int64 = next.len * int64(MUL)
```

Indicator that static loader should be used

Type annotation used for primitive type

Final constants can be inlined by the compiler

Arguments are type checked:

CHECK_ARGS ((0, ('__main__', 'C'),
1, ('__main__', 'C', '?')))

Fields are transformed to typed slots:

__slots__ = ('len', 'next')
__slot_types__ = {"len": ('__static__', 'int64'),
"next": ('__main__', 'C', '?')}

Field stores are generated:
STORE_FIELD ('__main__', 'C', 'next')

Primitive math is generated:

PRIMITIVE_LOAD_CONST 1

PRIMITIVE_BINARY_OP 2 (multiply)

PYRO

- Experimental, from-scratch runtime, reusing standard library
- Differences from CPython:
  - Compacting GC
  - Tagged Pointers
  - Hidden Classes
- C-API emulated for PEP-384 subset
- Open questions:
  - Adapting to PEP-384 at scale
  - Performance of API emulation

WHAT'S NEXT

Upstreaming, Results

WHAT'S NEXT?

1. More Upstreaming

---

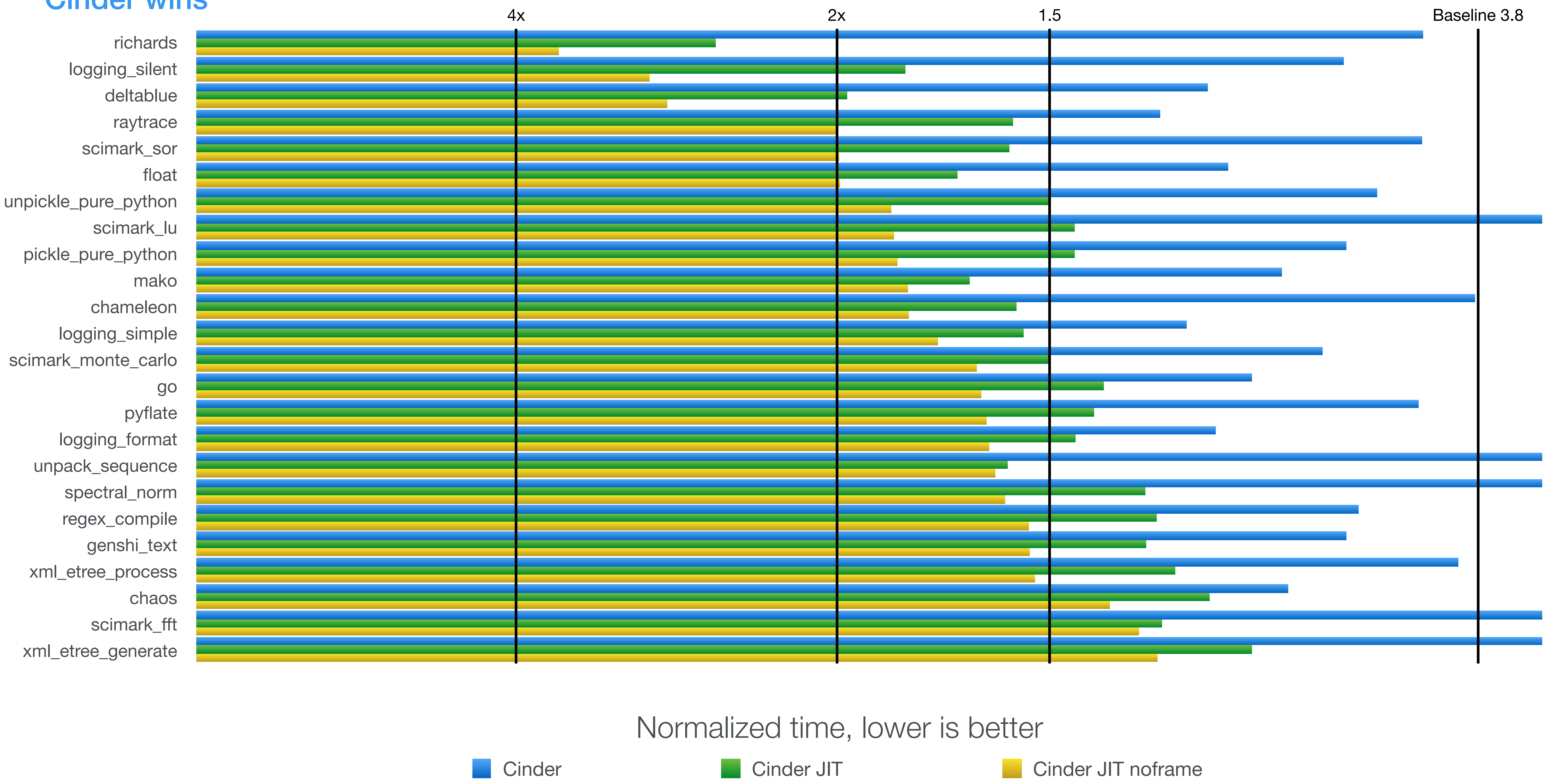
2. Our Results

RESULTS

- Production improvements: 20-30%
- Harder to measure as changes are incremental over time

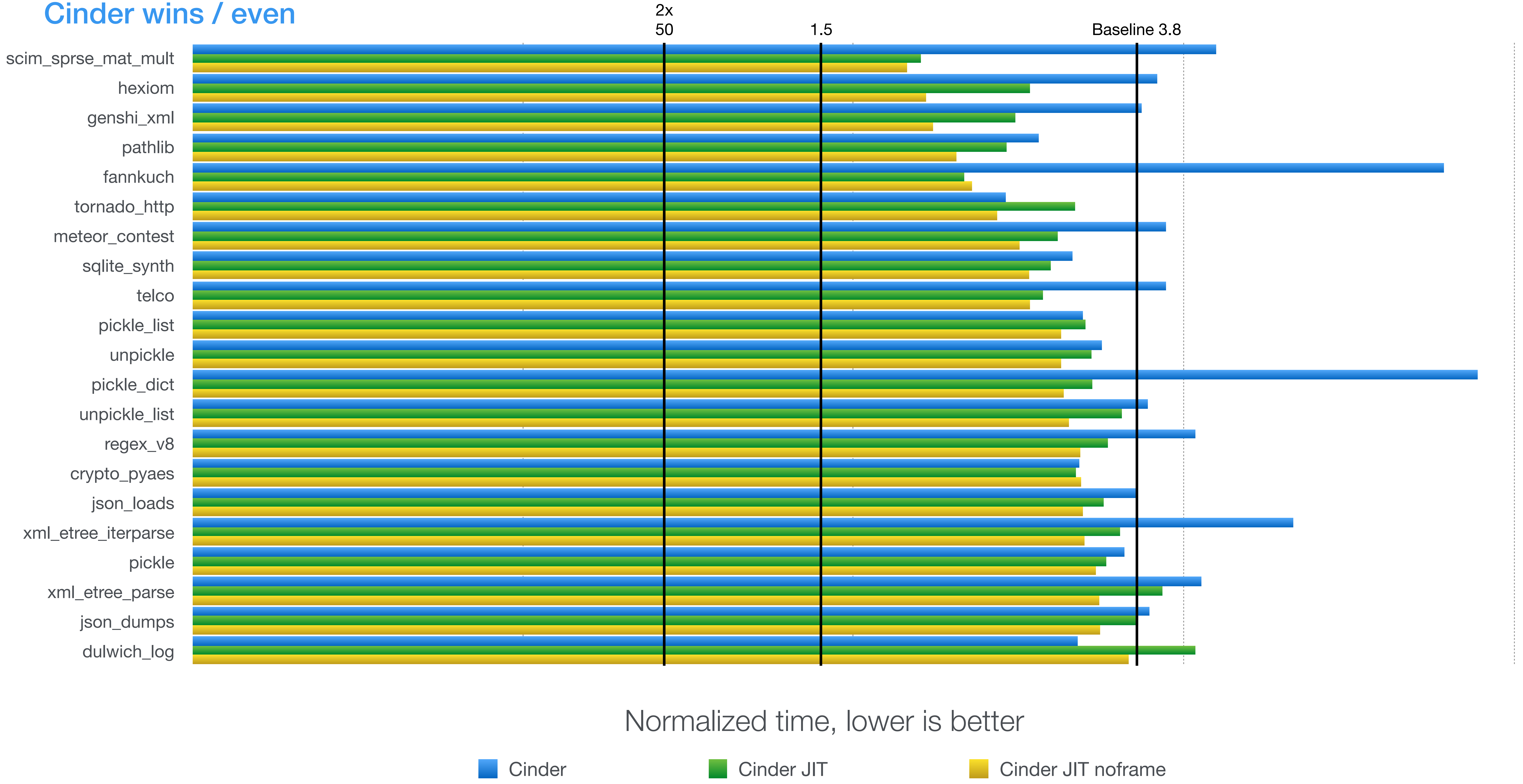
BENCHMARKS

Cinder wins



BENCHMARKS

Cinder wins / even

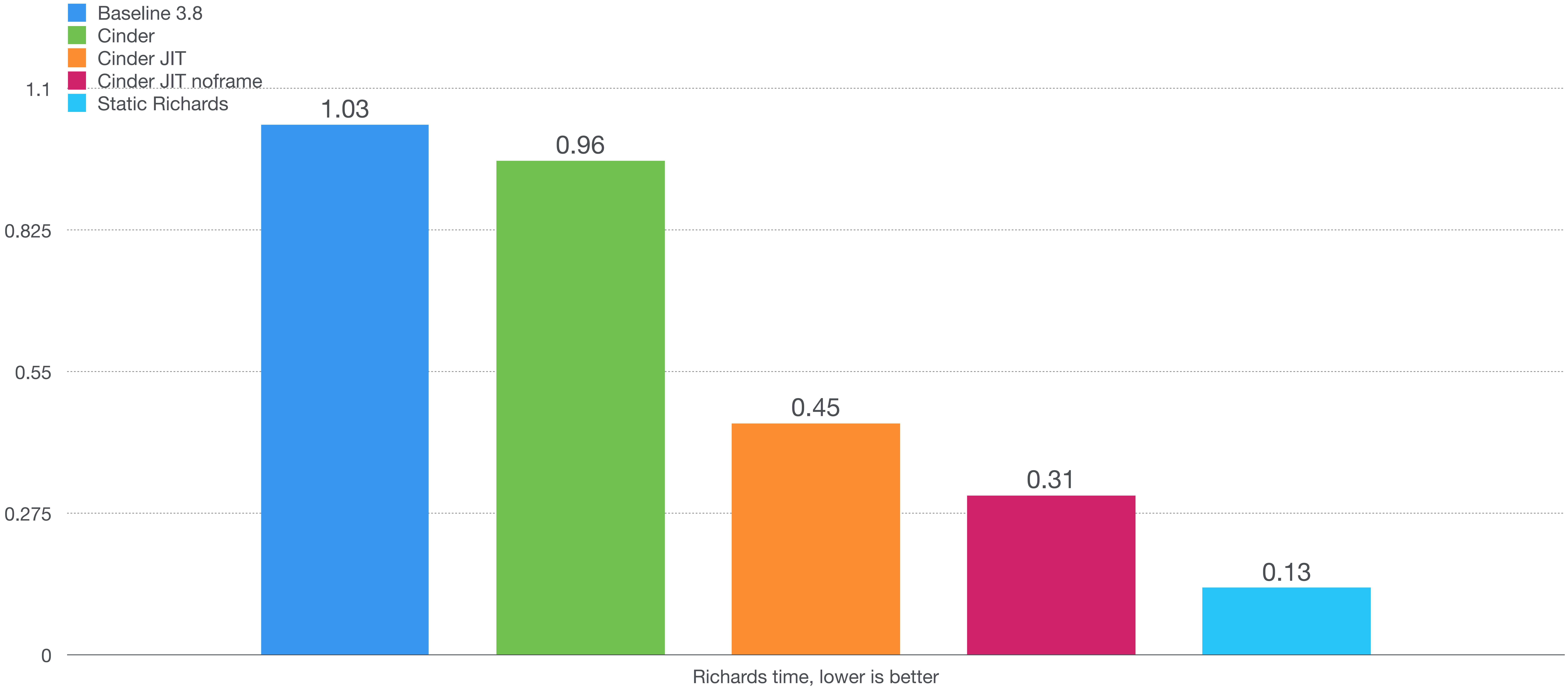


BENCHMARKS

Cinder losses

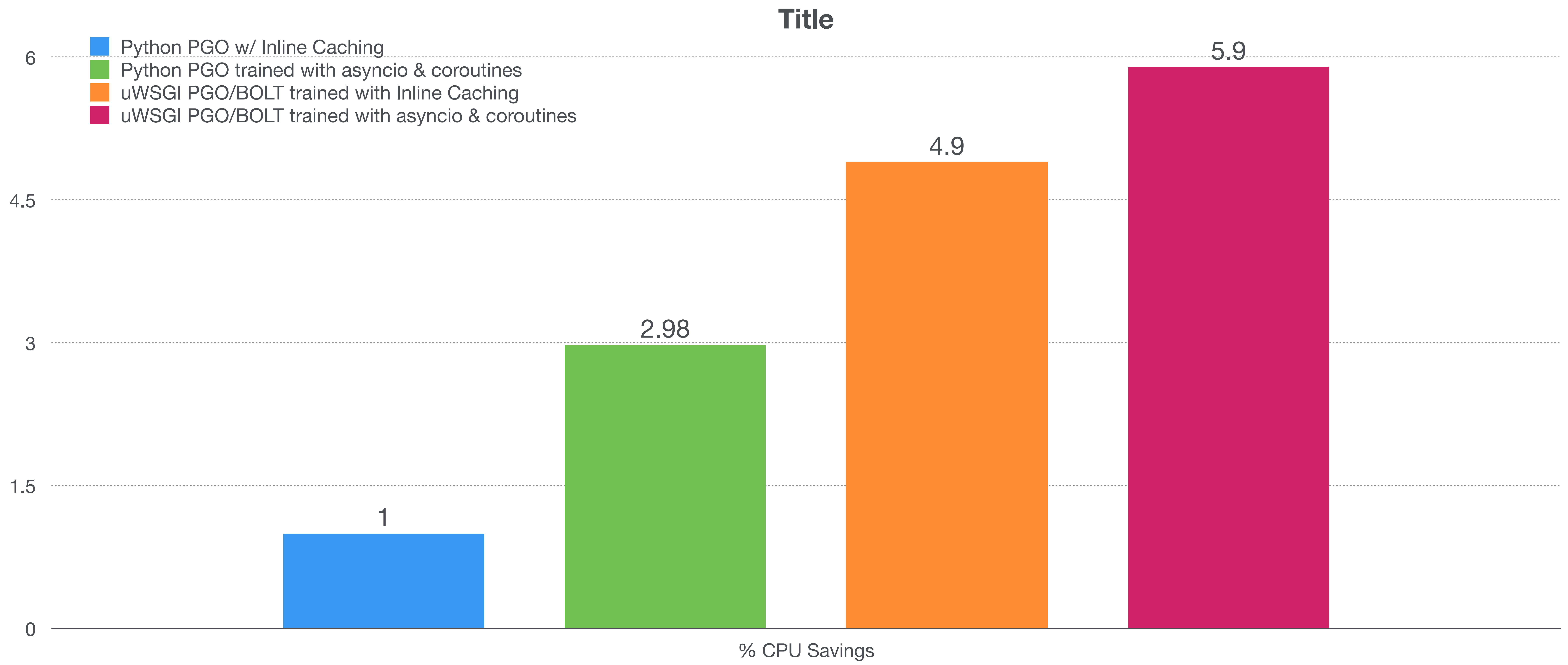
BENCHMARK IN DEPTH

Deep dive - Richards



BUILD CHANGES

Tweaking the build process for massive wins




BOLDLY YOU. STRONGER US.

https://github.com/facebookincubator/cinder

And we're hiring!