Pyjion

Brett Cannon & Dino Viehland, Microsoft (Azure Data Science Tools)

https://github.com/microsoft/pyjion

What are we trying to do?

Introduce a JIT API to CPython

… we hope ☺

3 overall goals

1. Introduce a C API to CPython for “plugging in” a JIT
    1. Needs to allow for full backwards-compatibility, else not useful
    2. All of this is dependent on python-dev accepting the C API proposal
2. Develop a proof-of-concept JIT for CPython using the CoreCLR JIT
    1. Needs to be faster for some workloads to show benefit
    2. Needs to be backwards-compatible (enough) to work with extension modules
3. Create a C++ framework for CPython JITs to build off of
    1. Abstracts out the common stuff when it comes to working with CPython’s bytecode
    2. Entirely optional and just a nicety for other (potential) JIT authors

Why?

Because faster is always nicer

… especially when it’s already compatible with your stuff.

How does this compare to … ?

PyPy

- Toolchain to generate a JIT
  - Includes an implementation for Python
- Currently considered the fastest implementation of Python
- Does not work with all C extension modules
  - CFFI
  - Partial C API support

Pyston

- Alpha-quality VM from Dropbox that uses 3 execution tiers
  - AST (which is really a CFG)
  - Baseline JIT
  - LLVM JIT
- Re-uses large portions of CPython to keep compatibility
  - Works w/ extension modules Dropbox cares about

How does this compare to … ?

Numba

- Numeric-specific JIT sponsored by Continuum Analytics
- You decorate any functions or methods you wish to pass to the LLVM JIT
- Supports GPUs

Psyco & Unladen Swallow

- Both projects tried to add a JIT to CPython
- Pysco was retired and helped lead to PyPy
- Unladen Swallow was shut down after a year of fighting with bugs in LLVM
  - Was sponsored by Google

How?

High-level overview

- JIT at the code object level
  - All executed code in Python is from a code object, even modules
  - Think of each local scope as representing a code object
- We translate Python bytecode to equivalent MSIL
  - Python's bytecode is very CISC and type-agnostic, so a single opcode generates a lot of IR
  - Both MSIL and Python bytecode is stack-based (although Python has two stacks)
- Uses an abstract interpreter to gather details on the code
  - Used to infer types from both type literals and syntactic operations on inferred types
  - Basic escape analysis to know when float and integer literals can be treated natively before needing to be boxed at the Python level
  - Plans to add more features

High-level overview

- Use CPython's C API to maintain compatibility
  - Emit IR to directly call the C API as necessary
  - Has allowed for faster bootstrapping by avoiding the need to translate all operations into MSIL
  - Long-term this is not an optimal solution in all cases as emitting JIT code should (theoretically) lead to better performance than calling into C code

Changes to CPython's C API

- `InterpreterState->eval_frame`
  - Function pointer with the same call signature as `PyEval_EvalFrameEx()`
  - Current `PyEval_EvalFrameEx()` gets renamed to `PyEval_EvalFrameDefault()`
  - `PyEval_EvalFrameEx()` ends up calling `interp->eval_frame()`
- `PyCodeObject->co_extra`
  - Scratch space for frame evaluation function
  - Simply a `PyObject*` so memory management is simple

Pyjion's use of the code object scratch space

- `j_run_count`
  - How many times the code object has been executed
- `j_failed`
  - Flag signaling to not bother trying to JIT compile the code object
- `j_evalfunc`
  - Trampoline to either trace types or execute JIT-compiled code
- `j_evalstate`
  - Opaque pointer to JIT-compiled code
- `j_specialization_threshold`
  - Execution count threshold to take any type tracing results into account

Bumps in the road

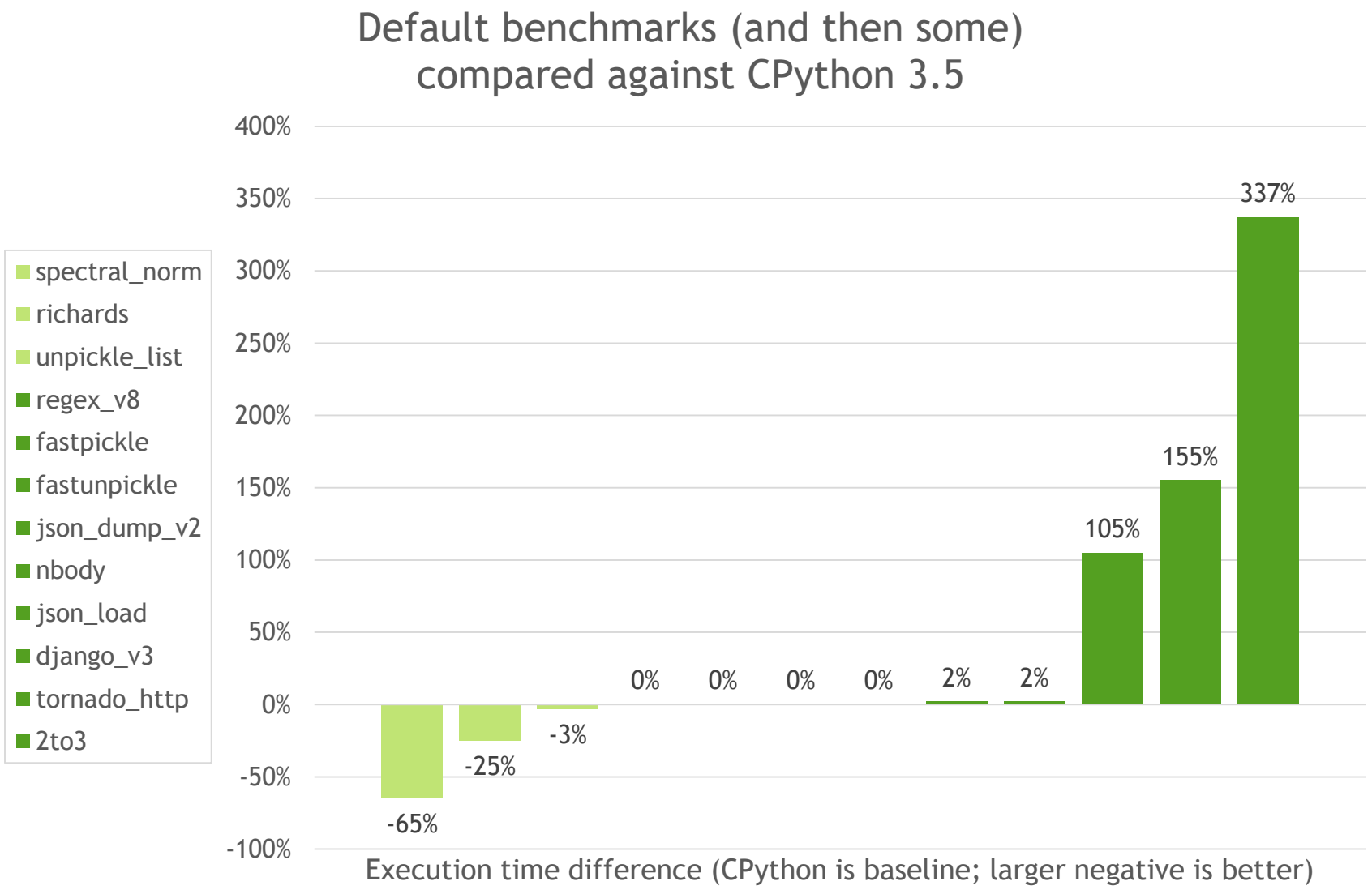
- CPython has two stacks while CoreCLR JIT has one
  - CPython has one for execution, other for exception handling
  - Makes it tricky to have to store things locally in JIT that would normally have gone on the second stack in CPython
- CPython has a few opcodes that result in a non-constant number of items on the stack
  - CoreCLR JIT **forbids** having anything left on the stack when you exit a frame
  - Exception handling opcodes can vary what is left on the stack based on arguments
    - Curse you, `END_FINALLY`!
- Iteration opcodes leave something on the stack after every iteration
  - Another issue thanks to the CoreCLR JIT forbidding leaving anything on the stack
    - Cure you, `FOR_ITER`/`GET_ITER`!

Bumps in the road

- Error checking everywhere since we have to account for potentially raised exceptions at any point Python code is executed
- Tough to balance cost of compiling versus execution cost
  - Really small functions are not necessarily worth the overhead of the JIT compilation plus any overhead in execution

Performance

Because people like numbers, no matter how alpha your code is.


Default benchmarks (and then some) compared against CPython 3.5
spectral_norm
richards
unpickle_list
regex_v8
fastpickle
fastunpickle
json_dump_v2
nbody
json_load
django_v3
tornado_http
2to3
400%
350%
300%
250%
200%
150%
100%
50%
0%
-50%
-100%
-65%
-25%
-3%
0%
0%
0%
0%
2%
2%
105%
155%
337%
Execution time difference (CPython is baseline; larger negative is better)

A more general performance picture

- Out of 41 benchmarks, the average performance showed …
  - 14 benchmarks are slower
  - 12 are statistically the same
  - 15 are faster

Future optimization possibilities

- We currently do very few type optimizations
  - E.g. only optimize ints and floats in a specific order
- Python 3.6 should open new possibilities
  - New opcodes open up possibility of optimizing more things
  - Dict versioning would allow for watching namespaces and caching objects
    - Caching might come into CPython itself which we could leverage

When?

- PEP for changes in CPython is out for review
  - This is the most critical aspect of the whole project!
  - Without this then Pyjion will forever be a modified CPython interpreter and that isn't sustainable
- Pyjion is compatible enough today
  - Basically you can't see all local variables when debugging, but compatible otherwise
  - Not tested w/ other projects yet, though
- C++ framework for JITs is still just an idea
  - Designed the base C++ classes for this, but it's still evolving and we haven't worried about locking anything down

Q&A

https://github.com/Microsoft/Pyjion

We're hiring: pythonjobs@microsoft.com