How to get most out of your PyPy?

Maciej Fijałkowski, Alex Gaynor, Armin Rigo

PyCon 2012

7 March 2012

First rule of optimization?

- if it’s not correct, it doesn’t matter

First rule of optimization?

- if it’s not correct, it doesn’t matter

Second rule of optimization?

- if it's not faster, you're wasting time
- but if you iterate fast, you can afford wasting time

Second rule of optimization?

- if it's not faster, you're wasting time
- but if you iterate fast, you can afford wasting time

Second rule of optimization?

- if it’s not faster, you’re wasting time
- but if you iterate fast, you can afford wasting time

Third rule of optimization?

- measure twice, cut once

Third rule of optimization?

- measure twice, cut once

(C)Python performance tricks

- `map()` instead of list comprehensions
- `def f(int=int):`, make globals local
- `append = my_list.append`, grab bound methods outside loop
- avoiding function calls
- don't write Python

(C)Python performance tricks

- `map()` instead of list comprehensions
- `def f(int=int):`, make globals local
- `append = my_list.append`, grab bound methods outside loop
- avoiding function calls
- don't write Python

Forget these

- PyPy has totally different performance characterists
- which we're going to learn about now
- you cannot speak about operations in isolation (more later)

Why PyPy?

- performance
- memory
- sandbox

Why not PyPy (yet)?

- embedded python interpreter
- embedded systems
- not x86-based systems
- extensions, extensions, extensions

Performance

- the main thing we'll concentrate on today
- PyPy is an interpreter + a JIT
- compiling Python to assembler via magic (we'll talk about it later)
- very different performance characteristics from CPython

Performance sweetspots

- every VM has its sweetspot
- we try hard to make it wider and wider

CPython's sweetspot

- moving computations to C, example:

  `map(operator.attrgetter("a"), my_list)`

PyPy's sweetpot

- **simple** python
- if I can't understand it, JIT won't either

How PyPy runs your program, involved parts

- a simple bytecode compiler (just like CPython)
- an interpreter loop written in RPython
- a JIT written in RPython
- an assembler backend

Bytecode interpreter

- executing one bytecode at a time
- add opcode for example
- .... goes on and on
- example

Tracing JIT

- once the loop gets hot, it starts tracing (1039 runs, or 1619 function calls)
- generates operations following how the interpreter executes them
- optimizes chunks of operations
- compiles to assembler (x86, ppc or arm)

PyPy's specific features

- JIT complete by design, as long as the interpreter is correct
- only **one** language description, in a high level language
- decent tools for inspecting the generated code

The PyPy cake

- Your Python code
- PyPy interpreter (RPython)
- High-level flow graphs
- Low-level flow graphs
  - JIT
    - knows about exceptions and GC
  - C representation
    - reduces flow graphs to remove exceptions and GC

Performance characteristics - runtime

- runtime the same or a bit slower as CPython
- examples of runtime:
  - `list.sort`
  - `long + long`
  - `set & set`
  - `unicode.join`
  - ...

Performance characteristics - JIT

- important - JIT never considers operations in isolation
- JIT always works on a loop or a function
- JIT heuristically optimized for what we believe is common Python
- often much faster than CPython once warm

Heuristics

- what to specialize on (assuming stuff is constant)
- data structures
- relative cost of operations

Heuristic example - dicts vs objects

- dicts - an unknown set of keys, potentially large
- objects - a relatively stable, constant set of keys (but not enforced)
- performance example

Specialized lists

- lists are specialized for type - `int`, `float`, `str`, `unicode` and `range()`.
- appending a new type to an existing list makes you iterate over the entire list and rewrite everything.

Simpler is Faster

- some examples
- simple is good
- python is vast
- if we've never seen a use of some piece of stdlib, chances are it'll be suboptimal on pypy
- no really, simple is good

Things we could improve

- frame access is slow
- list comprehension vs generator expression
- profiling & tracing hooks
- all works but could be optimized more

JitViewer

- `bitbucket.org/pypy/jitviewer`
- `mkvirtualenv -p <path to pypy>`
- `python setup.py develop`

The overview

- usually three pieces per loop
- prologue and two loop iterations (loop invariants in the first bit)
- they contain guards
- guards can be compiled to more code (bridges) that jump back to the loop or somewhere else
- functions are inlined
- sometimes completely twisted flow