Two projects to optimize Python

FOSDEM 2013, Bruxelles

Victor Stinner
<victor.stinner@gmail.com>

Distributed under CC BY-SA license: http://creativecommons.org/licenses/by-sa/3.0/

Agenda

- CPython bytecode is inefficient
- AST optimizer
- Register-based bytecode

Part I

CPython bytecode is inefficient

CPython is inefficient

- Python is very dynamic, cannot be easily optimized
- CPython 3.3 only support basic optimizations like replacing 1+1 with 2
- CPython bytecode is inefficient

Inefficient bytecode

Given a simple function:

```
def func():
    x = 33
    return x
```

Inefficient bytecode

- I get: (4 instructions)

```
LOAD_CONST 1 (33)
STORE_FAST 0 (x)
LOAD_FAST  0 (x)
RETURN_VALUE
```

- I expected: (2 instructions)

```
LOAD_CONST 1 (33)
RETURN_VALUE
```

- Or even: (1 instruction)

```
RETURN_CONST 0 (33)
```

How Python works

- Parse the source code
- Build an Abstract Syntax Tree (AST)
- Emit Bytebyte
- Evaluate bytecode

Let's optimize!

- Parse the source code
- Build an Abstract Syntax Tree (AST)
  → **astoptimizer**
- Emit Bytebyte
- Evaluate bytecode
  → **registervm**

Part II

AST optimizer

AST optimizer

- AST is high-level and contains a lot of information
- Rewrite AST to get faster code
- Disable dynamic features of Python to allow more optimizations
- Unpythonic optimizations are disabled by default

AST optimizations (1)

- Call builtin functions and methods:

```
len("abc") → 3
(32).bit_length() → 6
math.log(32) / math.log(2) → 5.0
```

- Evaluate str % args and print(arg1, arg2, ...)

```
"x=%s" % 5 → "x=5"
print(2.3) → print("2.3")
```

AST optimizations (2)

- Simplify expressions (2 instructions => 1):

```
not(x in y) → x not in y
```

- Optimize loops (Python 2 only):

```
while True: ... → while 1: ...

for x in range(10): ...
→ for x in xrange(10): ...
```

In Python 2, `True` requires a (slow) global lookup, the number 1 is a constant

AST optimizations (3)

- Replace list (build a runtime) with tuple (constant):

```
for x in [1, 2, 3]: ...
→ for x in (1, 2, 3): ...
```

- Replace list with set (Python 3 only):

```
if x in [1, 2, 3]: ...
→ if x in {1, 2, 3}: ...
```

In Python 3, {1, 2, 3} is converted to a constant frozenset (if used in a test)

AST optimizations (4)

- Evaluate operators:

```
"abcdef"[:3] → "abc"

def f(): return 2 if 4 < 5 else 3
→ def f(): return 2
```

- Remove dead code:

```
if 0: ...
→ pass
```

Used as a preprocessor

- `"if DEBUG"` and `"if os.name == 'nt'"` have a cost at runtime
- Tests can be removed at compile time:

```
cfg.add_constant('DEBUG', False)
cfg.add_constant('os.name',
                 os.name)
```

- Pythonic preprocessor: no need to modify your code, code works without the preprocessor

astoptimizer TODO list

- Constant folding: experimental support (buggy)
- Unroll (short) loops
- Function inlining (is it possible?)

Part III

Register-based bytecode

Stack-based bytecode

```
def func():
    x = 33
    return x + 1
```

```
LOAD_CONST 1 (33)   # stack: []
STORE_FAST 0 (x)    # stack: [33]
LOAD_FAST  0 (x)    # stack: []
LOAD_CONST 2 (1)    # stack: [33]
BINARY_ADD          # stack: [33, 1]
RETURN_VALUE        # stack: [34]
```

(6 instructions)

Register bytecode

```
def func():
    x = 33
    return x + 1
```

```
LOAD_CONST_REG 'x', 33 (const#1)
LOAD_CONST_REG R0, 1 (const#2)
BINARY_ADD_REG R0, 'x', R0
RETURN_VALUE_REG R0
```

(4 instructions)

registervm

- Rewrite instructions to use registers instead of the stack
- Use single assignment form (SSA)
- Build the control flow graph
- Apply different optimizations
- Register allocator
- Emit bytecode

registervm optim (1)

- Using registers allows more optimizations
- Move constants loads and globals loads (slow) out of loops:
```
return [str(item) for item in data]
```
- Constant folding:
```
x=1; y=x; return y
→ y=1; return y
```
- Remove duplicate load/store instructions: constants, names, globals, etc.

Merge duplicate loads

- Stack-based bytecode :

```
return (len("a"), len("a"))

LOAD_GLOBAL 'len' (name#0)
LOAD_CONST 'a' (const#1)
CALL_FUNCTION (1 positional)
LOAD_GLOBAL 'len' (name#0)
LOAD_CONST 'a' (const#1)
CALL_FUNCTION (1 positional)
BUILD_TUPLE 2
RETURN_VALUE
```

Merge duplicate loads

- Register-based bytecode :

```
return (len("a"), len("a"))

LOAD_GLOBAL_REG R0, 'len' (name#0)
LOAD_CONST_REG R1, 'a' (const#1)
CALL_FUNCTION_REG R2, R0, 1, R1
CALL_FUNCTION_REG R0, R0, 1, R1
CLEAR_REG R1
BUILD_TUPLE_REG R2, 2, R2, R0
RETURN_VALUE_REG R2
```

registervm optim (2)

- Convert binary operator to inplace operator:
  `x = x + y → x += y`
- Remove unreachable instructions (dead code)
- Remove useless jumps (relative jump + 0)

Pybench results

- BuiltinMethodLookup:
  fewer instructions: 390 => 22
  24 ms => 1 ms (24x faster)
- NormalInstanceAttribute:
  fewer instructions: 381 => 81
  40 ms => 21 ms (1.9x faster)
- StringPredicates:
  fewer instructions: 303 => 92
  42 ms => 24 ms (1.8x faster)

python™

Pybench results

- Pybench is a microbenchmark
- Don't expect such speedup on your applications
- registervm is still experimental and emits invalid code

Other projects

- PyPy and its amazing JIT
- Pymothoa, Numba: JIT (LLVM)
- WPython: "Wordcode-based" bytecode
- Hotpy 2
- Shedskin, Pythran, Nuitka: compile to C++

Questions?

https://bitbucket.org/haypo/astoptimizer

http://hg.python.org/sandbox/registervm

Contact:

victor.stinner@gmail.com

Distributed under CC BY-SA license: http://creativecommons.org/licenses/by-sa/3.0/

Thanks to David Malcom
for the LibreOffice template

http://dmalcolm.livejournal.com/