Beyond Bytecode: a Wordcode-based Python

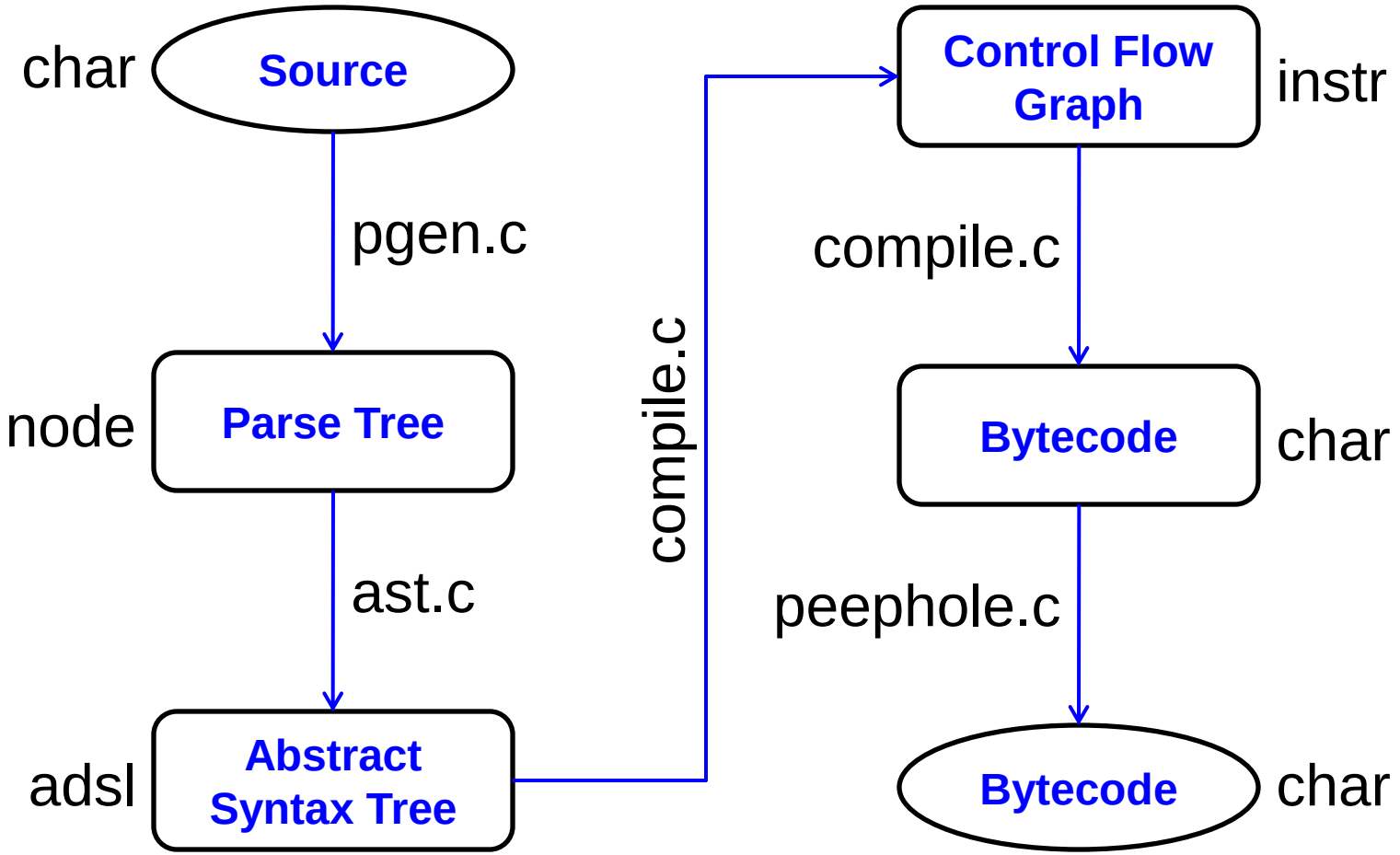
Cesare Di Mauro

A-Tono s.r.l.

PyCon Tre 2009 – Firenze (Florence)

May 9, 2009

From Python source to Bytecode



About Python bytecodes

Bytecode stream: an opcodes mixture.

- 1 byte (no parameter)
- 3 bytes (16 bits parameter value)
- 6 bytes (32 bits parameter value)

Byte order is little-endian (low byte first).

23	BINARY_ADD	100	1	2	LOAD_CONST	513	(‘spam’)
Single byte opcode		Multi byte opcode					

An example: Fibonacci's sequence

```
def fib(n):
    if n <= 1:
        return 1
    else:
        return fib(n - 2) + fib(n - 1)
```

```
2          0 LOAD_FAST                0 (n)
           3 LOAD_CONST               1 (1)
           6 COMPARE_OP               1 (<=)
           9 JUMP_IF_FALSE            5 (to 17)
          12 POP_TOP
3         13 LOAD_CONST               1 (1)
          16 RETURN_VALUE
     >>   17 POP_TOP
5         18 LOAD_GLOBAL              0 (fib)
          21 LOAD_FAST                0 (n)
          24 LOAD_CONST               2 (2)
          27 BINARY_SUBTRACT
          28 CALL_FUNCTION            1
          31 LOAD_GLOBAL              0 (fib)
          34 LOAD_FAST                0 (n)
          37 LOAD_CONST               1 (1)
          40 BINARY_SUBTRACT
          41 CALL_FUNCTION            1
          44 BINARY_ADD
          45 RETURN_VALUE
          46 LOAD_CONST               0 (None)
          49 RETURN_VALUE
```

With Python 2.6.1 we have:

- 22 opcodes / instructions
- 50 bytes space needed

A look at the VM (ceval.c) main loop

```
for (;;) {
    opcode = NEXTOP();
    if (HAS_ARG(opcode))
        oparg = NEXTARG();
    switch(opcode) {
        case BINARY_ADD:
            // Code here
    }
}
```

Branch misprediction

CPU pipeline flush & reload

A Wordcode-based Python

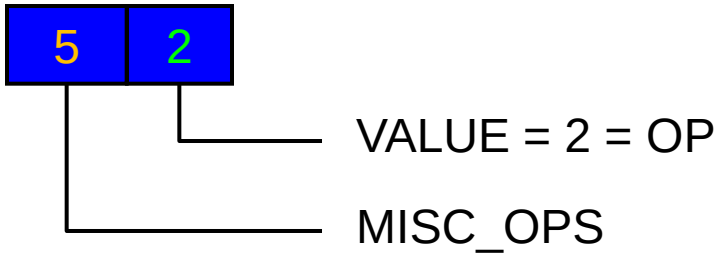
- Opcodes encoded in “words” (= 16 bits value, 2 bytes)
- 1, 2 or 3 words (2, 4, or 6 bytes) per instruction
- All opcodes bring a parameter
- Old instructions without parameter “grouped” into 6 special opcodes
- New instructions “families”
- Instructions with more than one parameter
- Little-endian (low byte first)

Wordcode structure

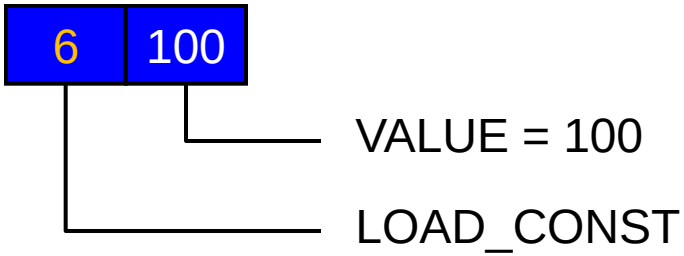
Word is split in 2 bytes:

- instruction kind (low byte)
- 8 bits parameter value (high byte)

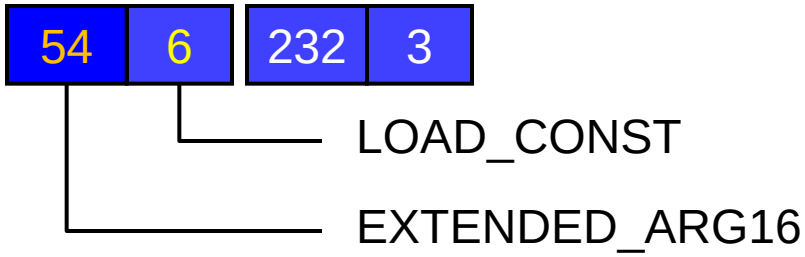
BINARY_ADD



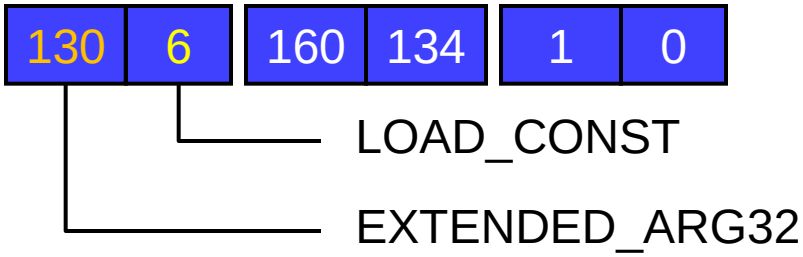
LOAD_CONST (100)



LOAD_CONST (1000)



LOAD_CONST (100000)



A look at the new VM main loop

```
for (;;) {
    NEXTOPCODE();
    switch(opcode) {
        case LOAD_CONST:
            // Code here
    }
}
```

```
opcode = NEXTOP();
if (HAS_ARG(opcode))
    oparg = NEXTARG();
```

```
#define NEXTOPCODE() \
    oparg = *next_instr++; \
    opcode = oparg & 0xff; \
    oparg >>= 8
```

Special opcodes examples

- UNARY_OPS UNARY_NEGATIVE, UNARY_NOT, GET_ITER
- BINARY_OPS BINARY_POWER, INPLACE_OR, CMP_EQ
- TERNARY_OPS SLICE_3, BUILD_SLICE_3, BUILD_CLASS
- STACK_OPS POP_TOP, ROT_TWO, DUP_TOP_THREE
- STACK_ERR_OPS STORE_SLICE_0, STORE_MAP, PRINT_ITEM
- MISC_OPS BINARY_ADD, RAISE_0, RETURN_VALUE

Instructions converted into special opcodes:

- DUP_TOPX
- COMPARE_OP
- RAISE_VARARGS
- BUILD_SLICE

Difficult opcode prediction

```
case GET_ITER:
    v = TOP();
    x = PyObject_GetIter(v);
    Py_DECREF(v);
    if (x != NULL) {
        SET_TOP(x);
        PREDICT(FOR_ITER);
        continue;
    }
    STACKADJ(-1);
    break;

PREDICTED_WITH_ARG(FOR_ITER);
case FOR_ITER:
    // CODE HERE
```

~~case GET_ITER:~~

PyObject_GetIter

~~PREDICTED_WITH_ARG(FOR_ITER);~~

```
case FOR_ITER:
    // CODE HERE
```

Complex peephole optimizer

```
case LOAD_CONST:
  cumlc = lastlc + 1;
  j = GETARG(codestr, i);
  if (codestr[i+3] != JUMP_IF_FALSE
    || codestr[i+6] != POP_TOP ||
    ISBASICBLOCK(blocks,i,7)  ||
    PyObject_IsTrue(
      PyList_GET_ITEM(consts, j)))
    continue;
  memset(codestr+i, NOP, 7);
```

```
case EXT16(LOAD_CONST):
  GETWORD(codestr + i + 1, oparg);
  handle_load_const(codestr,
    codelen, blocks, consts,
    i, oparg, 1);
  break;
/* Check for 8 bit args */
default:
  opcode = EXTRACTOP(rawopcode);
  oparg = EXTRACTARG(rawopcode);
  switch (opcode) {
    case LOAD_CONST:
      handle_load_const(codestr,
        codelen, blocks, consts,
        i, oparg, 0);
  }
```

Word endianess (and align) matters

```
#ifdef WORDS_BIGENDIAN
#define NEXTOPCODE() \
  oparg = *next_instr++; \
  opcode = oparg >> 8; \
  oparg &= 0xff
#else
#define NEXTOPCODE() \
  oparg = *next_instr++; \
  opcode = oparg & 0xff; \
  oparg >>= 8
```

```
unsigned short *
```

```
typedef struct {
    Py_ssize_t ob_refcnt;
    struct _typeobject *ob_type;
    Py_ssize_t ob_size;
    long ob_shash;
    int ob_sstate;
    char ob_sval[1];
} PyStringObject;
```

32 bits aligned

An example: Fibonacci's sequence

```
def fib(n):
    if n <= 1:
        return 1
    else:
        return fib(n - 2) + fib(n - 1)
```

```
2          0 FAST_BINOP_CONST    n <= 1
           2 JUMP_IF_FALSE       1 (to 4)
3          3 RETURN_CONST        1 (1)
5     >>   4 LOAD_GLOBAL         0 (fib)
           5 FAST_BINOP_CONST    n - 2
           7 CALL_FUNCTION       1
           8 LOAD_GLOBAL         0 (fib)
           9 FAST_BINOP_CONST    n - 1
          11 CALL_FUNCTION       1
          12 BINARY_ADD
          13 RETURN_VALUE
          14 RETURN_CONST        0 (None)
```

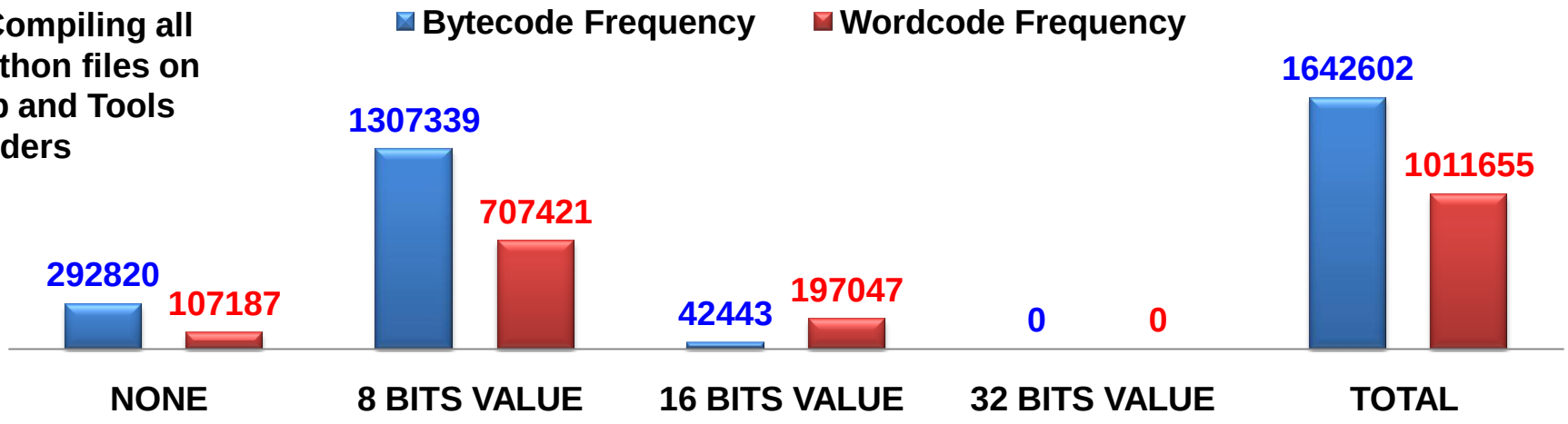
With WPython 2.6.1 we have:

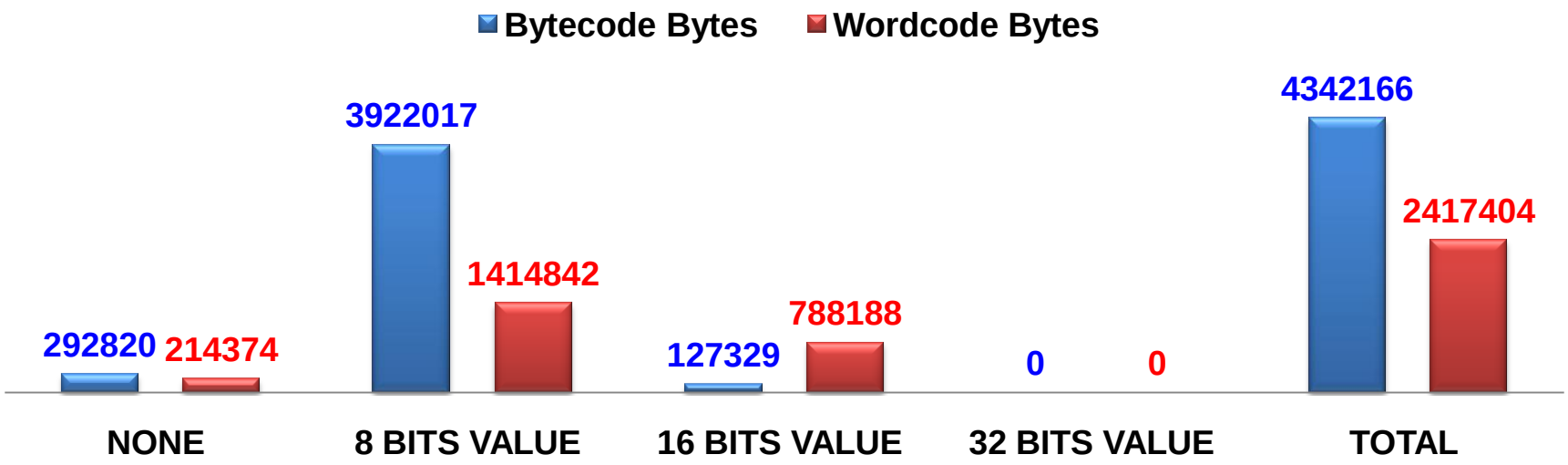
- 12 opcodes / instructions
- 30 bytes space needed

12 / 22 = -45% instructions, 30 / 50 = -40% space (bytes)

Opcodes summary*

* Compiling all Python files on Lib and Tools folders





JUMPs enhancements

```
def f(x):
  return 1 if x else 0
```

```
2        0 LOAD_FAST       0 (x)
         3 JUMP_IF_FALSE   5 (to 11)
         6 POP_TOP
         7 LOAD_CONST      1 (1)
        10 RETURN_VALUE
   >>   11 POP_TOP
        12 LOAD_CONST      2 (0)
        15 RETURN_VALUE
```

```
2        0 LOAD_FAST       0 (x)
         1 JUMP_IF_FALSE   2 (to 4)
         2 LOAD_CONST      1 (1)
         3 RETURN_VALUE
   >>    4 LOAD_CONST      2 (0)
         5 RETURN_VALUE
```

Always pops

```
def f(x, y):
  return x and y
```

Pops if not condition

```
2        0 LOAD_FAST       0 (x)
         3 JUMP_IF_FALSE   4 (to 10)
         6 POP_TOP
         7 LOAD_FAST       1 (y)
   >>   10 RETURN_VALUE
```

```
2        0 LOAD_FAST                0 (x)
         1 JUMP_IF_FALSE_ELSE_POP   1 (to 3)
         2 LOAD_FAST                1 (y)
   >>    3 RETURN_VALUE
```

FUNCTIONs enhancements

```
2    0 LOAD_GLOBAL     0 (g)
     3 LOAD_GLOBAL     1 (len)
     6 LOAD_FAST       1 (x)
     9 CALL_FUNCTION   1
    12 CALL_FUNCTION   1
    15 POP_TOP
3   16 LOAD_FAST       0 (self)
    19 LOAD_ATTR       2 (clear)
    22 CALL_FUNCTION   0
    25 POP_TOP
4   26 LOAD_FAST       0 (self)
    29 LOAD_ATTR       3 (get)
    32 CALL_FUNCTION   0
    35 STORE_FAST      1 (x)
5   38 LOAD_GLOBAL     4 (h)
    41 CALL_FUNCTION   0
    44 POP_TOP
    45 LOAD_CONST      0 (None)
    48 RETURN_VALUE
```

```
def f(self, x):
  g(len(x))
  self.clear()
  x = self.get()
  h()
```

```
2    0 LOAD_GLOBAL                0 (g)
     1 LOAD_GLOB_FAST_CALL_FUNC   len; x; CALL 1
     3 CALL_PROCEDURE             1
3    4 FAST_ATTR_CALL_PROC        self.clear()
4    6 FAST_ATTR_CALL_FUNC        self.get()
     8 STORE_FAST                 1 (x)
5    9 LOAD_GLOBAL                4 (h)
    10 CALL_PROC_RETURN_CONST     0; RETURN None
```

A register-based VM for Python?

NO! Python is too complex

An “hybrid” stack-register VM is simpler:

- add new opcodes on VM (ceval.c + opcode.h)
- based on peephole optimizer (peephole.c)
- more compact code
- less stack usage
- less reference counting

Cons:

- too complex cases need stack
- supports only locals (and consts); little support for attributes
- requires peephole optimizer

Introducing “MOVE” instructions

- FAST <- FAST | CONST | GLOBAL | FAST.ATTR
- FAST.ATTR <- FAST | CONST | FAST.ATTR

```
def f(self, x):
  a = x
  b = 1
  c = len
  d = self.point
  self.x = x
  self.y = 'spam'
  self.z = self.point
```

```
2   0 MOVE_FAST_FAST            x -> a
3   2 MOVE_CONST_FAST           1 -> b
4   4 MOVE_GLOBAL_FAST          len -> c
5   6 MOVE_FAST_ATTR_FAST       self.point -> d
6   8 MOVE_FAST_FAST_ATTR       x -> self.x
7  10 MOVE_CONST_FAST_ATTR      'spam' -> self.y
8  12 MOVE_FAST_ATTR_FAST_ATTR  self.point -> self.z
   14 RETURN_CONST              0 (None)
```

“Register” binary instructions

FAST <- (FAST | CONST) BINARY_OP (FAST | CONST)

BINARY_OP = add, power, multiply, divide, true divide, floor divide, modulo, subtract, array subscription, left shift, right shift, binary and, binary xor, binary or

```
def f(a, x, y):
  z = x + y
  w = a[z]
  z = 1 + x
  w = x + 1
  x &= y
```

```
2   0 FAST_ADD_FAST_TO_FAST      x + y -> z
3   2 FAST_SUBSCR_FAST_TO_FAST   a [] z -> w
4   4 CONST_ADD_FAST_TO_FAST     1 + x -> z
5   6 FAST_ADD_CONST_TO_FAST     x + 1 -> w
6   8 FAST_INPLACE_BINOP_FAST    x &= y -> x
   10 RETURN_CONST               0 (None)
```

“Stack <-> register” instructions

FAST <- UNARY_OP (FAST | TOP)
FAST <- TOP BINARY_OP (FAST | CONST)
FAST <- SECOND BINARY_OP TOP
TOP <- (FAST | CONST | TOP) BINARY_OP (FAST | CONST)
TOP <- UNARY_OP FAST

```
def f(x, y):
  z = -x
  w = x * y + 2
  a = -x * -y
```

```
2   0 FAST_UNOP_TO_FAST   - x -> z
3   2 FAST_BINOP_FAST     x * y
    4 CONST_ADD_TO_FAST   + 2 -> w
4   6 FAST_UNOP           - x
    8 FAST_UNOP           - y
   10 BINOP_TO_FAST       * -> a
   12 RETURN_CONST        0 (None)
```

New costant folding code

Moved from peephole.c to ast.c and compile.c:

- more pervasive
- more efficient
- supports tuples, lists, and dictionaries (even “deep” ones)
- supports partially constant tuples, lists, funcs def & call

```
def f():
  return 1 + 2 * 3
```

With Python 2.6.1:

```
2    0 LOAD_CONST      1 (1)
     3 LOAD_CONST      4 (6)
     6 BINARY_ADD
     7 RETURN_VALUE

>>> f.func_code.co_consts
(None, 1, 2, 3, 6)
```

With WPython 2.6.1:

```
2    0 RETURN_CONST    1 (7)

>>> f.func_code.co_consts
(None, 7)
```

Deep costant tuples and lists

```
def f():
  return 'x', [1, 2], 'y'
```

With Python 2.6.1:

```
2   0 LOAD_CONST    1 ('x')
    3 LOAD_CONST    2 (1)
    6 LOAD_CONST    3 (2)
    9 BUILD_LIST    2
   12 LOAD_CONST    4 ('y')
   15 BUILD_TUPLE   3
   18 RETURN_VALUE

>>> f.func_code.co_consts
(None, 'x', 1, 2, 'y')
```

With WPython 2.6.1:

```
2   0 LOAD_CONST       1 (('x', [1, 2], 'y'))
    1 TUPLE_DEEP_COPY
    2 RETURN_VALUE

>>> f.func_code.co_consts
(None, ('x', [1, 2], 'y'))
```

Costant dictionaries

```
def f(x):
  return {'a' : 1,'b' : 2, 'c' : 3}[x]
```

With Python 2.6.1:

```
2   0 BUILD_MAP       3
    3 LOAD_CONST      1 (1)
    6 LOAD_CONST      2 ('a')
    9 STORE_MAP
3  10 LOAD_CONST      3 (2)
   13 LOAD_CONST      4 ('b')
   16 STORE_MAP
   17 LOAD_CONST      5 (3)
   20 LOAD_CONST      6 ('c')
   23 STORE_MAP
   24 LOAD_FAST       0 (x)
   27 BINARY_SUBSCR
   28 RETURN_VALUE
```

With WPython 2.6.1:

```
2   0 LOAD_CONST      1 ({'a': 1, 'c': 3, 'b': 2})
    1 DICT_DEEP_COPY
3   2 FAST_BINOP      [] x
    4 RETURN_VALUE
```

```
>>> f.func_code.co_consts
(None, {'a': 1, 'c': 3, 'b': 2})
```

```
>>> f.func_code.co_consts
(None, 1, 'a', 2, 'b', 3, 'c')
```

Constant parameters on calls

```
def f(i):
  g('a', 'b', w = 1, x = i, y = 2, z = 3)
```

With Python 2.6.1:

```
2   0 LOAD_GLOBAL    0 (g)
    3 LOAD_CONST     1 ('a')
    6 LOAD_CONST     2 ('b')
    9 LOAD_CONST     3 ('w')
   12 LOAD_CONST     4 (1)
   15 LOAD_CONST     5 ('x')
   18 LOAD_FAST      0 (i)
   21 LOAD_CONST     6 ('y')
   24 LOAD_CONST     7 (2)
   27 LOAD_CONST     8 ('z')
   30 LOAD_CONST     9 (3)
   33 CALL_FUNCTION  1026
   36 POP_TOP
   37 LOAD_CONST     0 (None)
   40 RETURN_VALUE
```

```
>>> f.func_code.co_consts
(None, 'a', 'b', 'w', 1, 'x', 'y', 2, 'z', 3)
```

With WPython 2.6.1:

```
2   0 LOAD_GLOBAL    0 (g)
    1 LOAD_CONSTS    1 (('a', 'b', 'w', 1, 'x'))
    2 LOAD_FAST      0 (i)
    3 LOAD_CONSTS    2 (('y', 2, 'z', 3))
    4 CALL_PROC_RETURN_CONST  66; RETURN None
```

```
>>> f.func_code.co_consts
(None, ('a', 'b', 'w', 1, 'x'), ('y', 2, 'z', 3))
```

Optimized loops: no setup/exit!

With Python 2.6.1:

```
2          0 LOAD_CONST       1 (1)
           3 STORE_FAST       1 (i)
3          6 SETUP_LOOP      28 (to 37)
  >>       9 LOAD_FAST        1 (i)
          12 LOAD_FAST        0 (n)
          15 COMPARE_OP       1 (<=)
          18 JUMP_IF_FALSE   14 (to 35)
          21 POP_TOP
4         22 LOAD_FAST        1 (i)
          25 LOAD_CONST       1 (1)
          28 INPLACE_ADD
          29 STORE_FAST       1 (i)
          32 JUMP_ABSOLUTE    9
  >>      35 POP_TOP
          36 POP_BLOCK
  >>      37 LOAD_CONST       0 (None)
          40 RETURN_VALUE
```

```
def loop(n):
  i = 1
  while i <= n:
    i += 1
```

"Virtual" instruction: suppressed if no break or continue found

With WPython 2.6.1:

```
2          0 MOVE_CONST_FAST          1 -> i
3 >>       2 FAST_BINOP_FAST          i <= n
           4 JUMP_IF_FALSE            3 (to 8)
4          5 FAST_INPLACE_ADD_CONST   i += 1
           7 JUMP_ABSOLUTE            2
  >>       8 RETURN_CONST             0 (None)
```

Suppressed if no break or continue

“Slimmer” comprehensions

With Python 2.6.1:

```
>>   34 FOR_ITER          38 (to 75)
     37 STORE_FAST         4 (y)
     40 LOAD_FAST          4 (y)
     43 LOAD_CONST         1 (0)
     46 COMPARE_OP         4 (>)
     49 JUMP_IF_FALSE     15 (to 67)
     52 POP_TOP
     53 LOAD_FAST          2 (_[1])
     63 LIST_APPEND
     64 JUMP_ABSOLUTE     34
>>   67 POP_TOP
     68 JUMP_ABSOLUTE     34
     71 JUMP_ABSOLUTE     11
>>   74 POP_TOP
>>   75 JUMP_ABSOLUTE     11
>>   78 DELETE_FAST        2 (_[1])
     81 RETURN_VALUE
```

```
def f(a, b):
  return [x * y \
          for x in a if x > 0 \
          for y in b if y > 0]
```

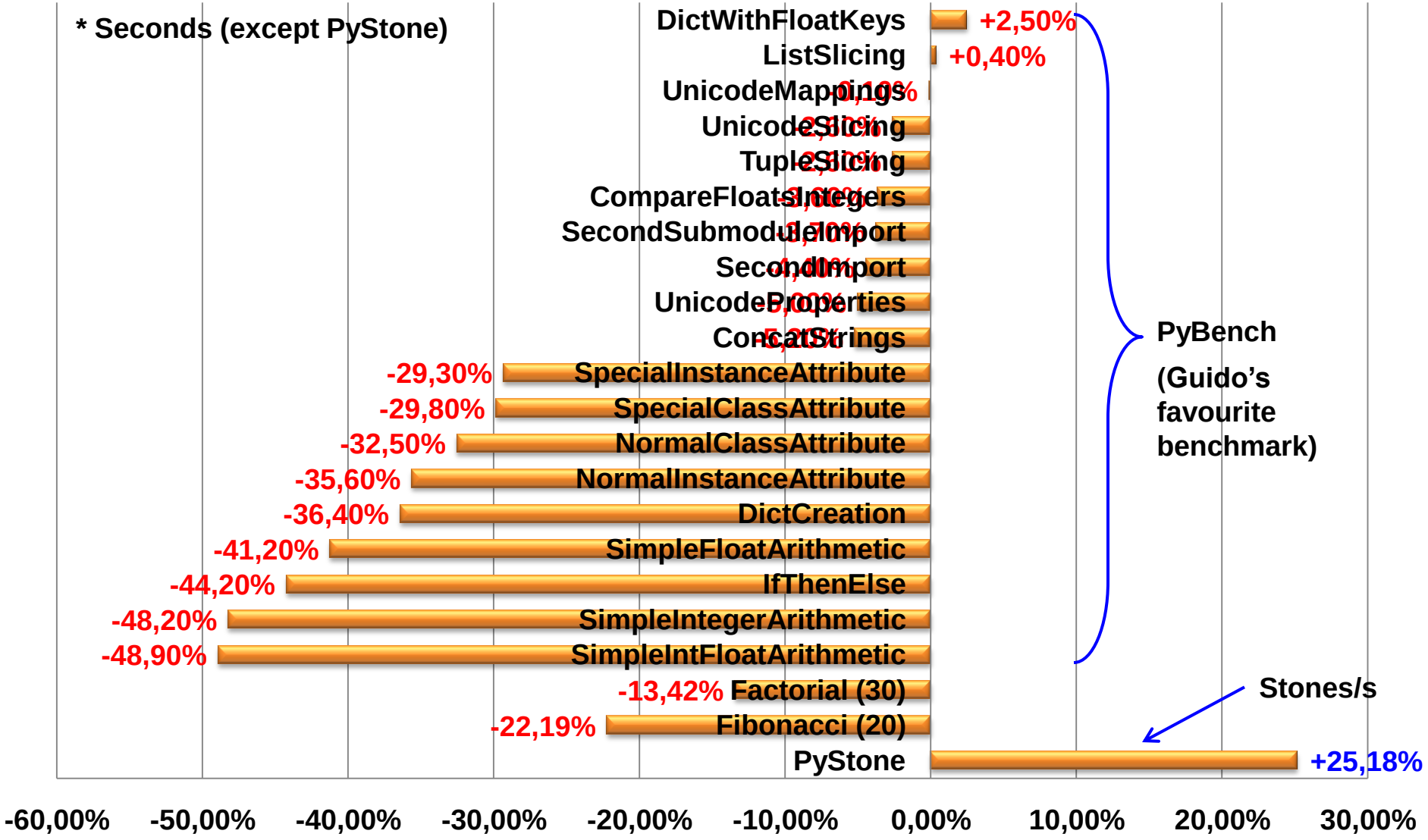
With WPython 2.6.1:

```
>>   12 FOR_ITER           9 (to 22)
     13 STORE_FAST         4 (y)
     14 FAST_BINOP_CONST   y > 0
     16 JUMP_IF_FALSE      4 (to 21)
     17 LOAD_FAST          2 (_[1])
     18 FAST_BINOP_FAST    x * y
     20 LIST_APPEND_LOOP  12
>>   21 JUMP_ABSOLUTE     12
>>   22 JUMP_ABSOLUTE      5
>>   23 DELETE_FAST        2 (_[1])
     24 RETURN_VALUE
```

Better peephole optimizer

- NOT also applyed to JUMP_IF_TRUE
- Aggressive unreachable code removing
- More lookheads on conditional jumps
- Static buffers allocation
- Buffers sharing
- No tuple -> list -> tuple conversion for constants
- Recognize new opcodes patterns
- Refactored code

Last but not least… SPEED!*



Open issues

- Pure python compiler needs updates (pyassem/codegen)
- Documentation untouched (Doc/library/dis.rst)
- Adding normal opcodes makes test_zipfile.py crazy!
- CodeObject output PyCF_ONLY_AST (test_compile.py)
- Disabled 2 tests (test_trace.py)
- Doctypes with absolute paths (test_syntax.py)
- Must add many tests to test_peephole.py
- String concats need optimizations
- Tested only on Windows, x86 CPUs (little-endian), 32 bits

Conclusions

- A new CPython “CISCy” 2.6.1 implementation presented
- Words used for opcodes (instead of bytes)
- Hybrid stack/register solution
- Space saved
- Faster on average
- VM main loop code refactored
- Room for more optimizations