An Insight into CPython Compiler Design

Ramkumar Ramachandra

FOSS.IN/2009

03 December 2009

Outline

1. Short Discussion of the CPython Compiler
2. A Gentle Introduction to LLVM
3. Enter: Unladen Swallow
4. Conclusion

How Python is compiled

- Do the boring grammar parsing
- Compile the parse tree to bytecode
- Apply optimizations
- Interpret the bytecode

The various stages of compilation

- PyAST_FromNode() in Python/ast.c | Parse tree → AST
- PyAST_Compile() in compile.c | AST → CFG → Bytecode
- PyAST_Compile() calls PySymtable_Build() and compiler_mod() | AST → CFG
- assemble() | Post-order DFS | CFG → Bytecode

What the final bytecode looks like

```
a, b = 1, 0
if a or b:
    print "Hello", a
```

```
  1           0 LOAD_CONST               4 ((1, 0))
              3 UNPACK_SEQUENCE          2
              6 STORE_NAME               0 (a)
              9 STORE_NAME               1 (b)

  2          12 LOAD_NAME                0 (a)
             15 JUMP_IF_TRUE             7 (to 25)
             18 POP_TOP
             19 LOAD_NAME                1 (b)
             22 JUMP_IF_FALSE           13 (to 38)
        >>   25 POP_TOP

  3          26 LOAD_CONST               2 ('Hello')
             29 PRINT_ITEM
             30 LOAD_NAME                0 (a)
             33 PRINT_ITEM
             34 PRINT_NEWLINE
             35 JUMP_FORWARD             1 (to 39)
        >>   38 POP_TOP
        >>   39 LOAD_CONST               3 (None)
             42 RETURN_VALUE
```

Execute the bytecode

```c
PyObject *PyEval_EvalFrameEx(PyFrameObject *f, int throwflag) {
  PyObject *result;
  result = PyEval_EvalFrame(f);
  return result;
}
```

```c
PyObject *PyEval_EvalFrame(PyFrameObject *f)
{
  register PyObject **stack_pointer;  /* Next free slot */
  register unsigned char *next_instr;
  register int opcode;  /* Current opcode */
  register int oparg;   /* Current opcode argument, if any */
  PyObject *retval = NULL;       /* Return value */
  PyCodeObject *co;        /* Code object */
}
```

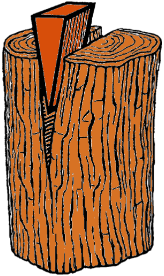
What is LLVM and why is it relevant?

- Compiler infrastructure
- Invents a new IR
- Replaces lower levels of GCC
- Provides static GCC-like compilation and JIT
- Python frontend possible

How Unladen Swallow started

- Objective: To speed up CPython
- Experiment with Psyco
- Temporarily use VMgen for eval loop
- Remove rarely used opcodes

Compile Python bytecode to LLVM IR



```
extern "C" _LlvmFunction *
_PyCode_ToLlvmIr(PyCodeObject *code)
{
  _LlvmFunction *wrapper = new _LlvmFunction();
  /* fbuilder functions in llvm_fbuilder.cc */
  wrapper->lf_function = fbuilder.function();
  return wrapper;
}
```

Changes to the eval loop

```
static int
mark_called_and_maybe_compile(PyCodeObject *co, PyFrameObject *f)
{
  co->co_hotness += 10;
  if (co->co_hotness > PY_HOTNESS_THRESHOLD) {
    if (co->co_llvm_function == NULL) {
      int target_optimization =
        std::max(Py_DEFAULT_JIT_OPT_LEVEL,
                 Py_OptimizeFlag);
      if (co->co_optimization < target_optimization) {
        // If the LLVM version of the function wasn't
        // created yet, setting the optimization level
        // will create it.
        r = _PyCode_ToOptimizedLlvmIr(co, target_optimization);
      }
    }
    if (co->co_native_function == NULL) {
      // Now try to JIT the IR function to machine code.
      co->co_native_function =
        _LlvmFunction_Jit(co->co_llvm_function);
    }
  }
  return 0;
}
```

Implement feedback-directed optimization

- Optimize native code, not bytecode
- Speed up builtin lookups/ inline simple builtins
- Don't compile cold branches
- Inline simple operators using type feedback

References

[1] Abelson, H., Sussman, G. J., and Sussman, J. *Structure and Interpretation of Computer Programs (SICP)*. The MIT Press, 1984.

[2] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, 2006.

[3] Aycock, J. *Compiling Little Languages in Python*.

[4] Cannon, B. *Design of the CPython Compiler*, 2005.

[5] Ertl, M. A., Gregg, D., Krall, A., and Paysan, B. vmgen — a generator of efficient virtual machine interpreters. *Software—Practice and Experience 32*, 3 (2002), 265–294.

[6] Montanaro, S. In *A Peephole Optimizer for Python* (1998).

[7] Wang, D. C., Appel, A. W., Korn, J. L., and Serra, C. S. In *The Zephyr Abstract Syntax Description Language* (1997).

Contact information

Ramkumar Ramachandra
artagnon@gmail.com
http://artagnon.com
Indian Institute of Technology, Kharagpur
Presentation source available on http://github.com/artagnon/foss.in