Converting Python Virtual Machine Code to C

John Aycock
Department of Computer Science
University of Victoria
Victoria, B.C., Canada
aycock@csc.uvic.ca

Abstract

The optimization of a Python program has a limit point, beyond which a programmer must resort to C code in order to get more speed. Not all programmers are willing or able to take this step. `211` is an experimental program which automatically converts Python virtual machine code into C. In this paper I discuss `211`, its results, and suggest changes to Python's internals which should yield better results and faster programs.

1 Introduction

My first nontrivial Python program was a 1400-line program that built and manipulated large graphs. While I found Python made for very fast development, the result was somewhat disappointing — medium-sized inputs could take days to run, making large inputs out of the question.

I applied the usual optimization techniques [3]. Using Python's profiler, I was able to identify "hot spots" in the program where the majority of time was being spent; changing that code to use more efficient data structures and cache previously-computed information sped the program up substantially. Manually performing transformations such as moving invariant code out of loops helped to a lesser degree. The result of this optimization was a Python program which now took several hours for inputs which previously took it several days. A good improvement, but still not fast enough.

At this point, I hit an optimization barrier. The usual Python lore dictated that I should recode the hot spots into C, something time constraints prevented. In addition, I wanted to avoid the maintenance and portability issues presented by hybrid source code. What I wanted was a tool to automatically convert my most frequently-executed bits of Python code into C.

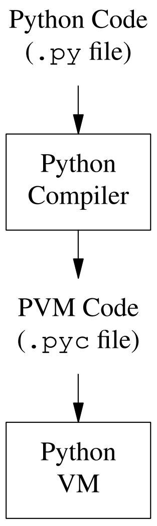

Figure 1: Execution of Python programs.

Internally, Python does not directly interpret Python programs. Instead, it has a compiler which translates Python programs into code for an idealized abstract computer — the Python Virtual Machine (PVM). The Python interpreter then executes the PVM code (Figure 1). The PVM itself is a "stack machine," meaning that PVM instructions get their arguments from the stack, and place their results onto the stack.

Virtual and stack machines are not new. The idea is decades-old, but went out of vogue in the late 1970's [12]. It has enjoyed a renaissance lately with the advent of Java, which uses a virtual machine.

The good news is that there is a wealth of research available which addresses the efficient execution of these machines. In particular, work has been done speeding up Forth [8, 7], Smalltalk [17, 4], and of course Java [16]. They have had great success in making faster language implementations, and a

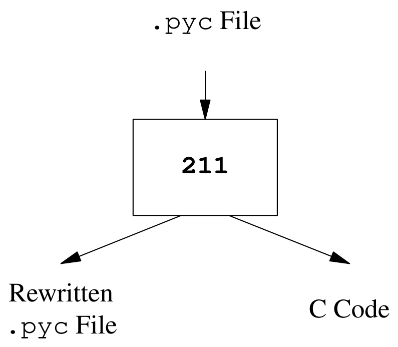

Figure 2: A model of 211.

lot of this material can be applied to speeding up Python.

The remainder of this paper is divided into two parts. Section 2 describes my work on 211, an experimental program which converts PVM code into C. In Section 3 I suggest some changes to the inner workings of Python which I argue will make it more amenable to optimization and high-speed implementations.

2 The 211 Program

My 211 program was inspired by the Toba program for Java, whose authors bill it as a "way-ahead-of-time compiler" [16]. The idea is that a Python application is profiled, and has the PVM code for its hot spots converted into C code by 211. The Python interpreter is recompiled with 211's C code output, producing a customized Python interpreter that can be used by the Python application (Figure 2).

Because it works at the PVM level, 211 is able to co-exist with any high-level PVM optimizers. This includes PVM optimizers that add new opcodes — 211 only converts PVM opcodes that it has been taught, so PVM code output by 211 can be a mixture of normal VM instructions and calls to specialized C code.

Only one line needs to be added to the Python interpreter code to make 211 work: a `#include` statement to include 211's C code output.

211 itself consists of 1320 lines of Python code, about 635 lines of which are bits of Python's interpreter code which 211 uses for its conversion. The program is based on Python 1.5.1.

To understand how I convert PVM to C code, it is useful to first look at the Python interpreter.

2.1 The Python Interpreter

The Python interpreter is what is sometimes called a "classical" interpreter [10]. Its algorithm is straightforward:

1. Fetch the opcode of a VM instruction, and its arguments (if any).
2. Execute the instruction.
3. Repeat steps 1 and 2 until the novelty wears off.

This algorithm is implemented in the Python interpreter by C code like that shown below (greatly simplified from the original).

```
while (1) {
    opcode = NEXTOP();
    if (HAS_ARG(opcode))
        oparg = NEXTARG();

    switch (opcode) {
    ...
    }
}
```

The cases in the `switch` statement implement the various VM instructions. As discussed in the next section, 211 specializes the Python interpreter by adding code into this `switch` statement.

2.2 What 211 Does

211 has four basic steps.

1. Read in a `.pyc` file and extract the PVM code.
2. Break the PVM code into extended basic blocks. An extended basic block (EBB) is the longest sequence of instructions for which there is only one entry point (there may be multiple exits from an EBB, though) [13]. The hypothetical code in Figure 3 contains two EBBs.

   Why use EBBs? The Python interpreter code expects branch targets to be expressed as offsets into PVM code. Rather than modify the interpreter extensively, I opted to use EBBs, which ensures that all branch targets correspond to PVM code and not C code.
3. Convert PVM instructions within each EBB into C code, effectively creating a new PVM opcode for each EBB. 211's conversion is a matter of pasting together code from the Python interpreter's main `switch` statement. For example, an EBB containing three PVM instructions would be converted into

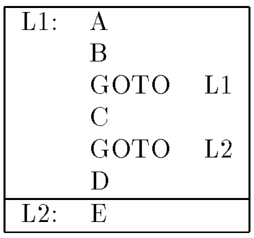

Figure 3: Extended basic blocks.

code for instruction #1
code for instruction #2
code for instruction #3

For the most part, the code from the Python interpreter is used with very little modification. In a few important cases, however, `211` will customize the C code to produce better output. In effect, `211` performs a limited amount of partial evaluation [9]. One example is the conversion of branches in the PVM code to `goto` statements wherever possible. Another example involves the PVM comparison instruction — usually, the Python interpreter decides at run-time what type of comparison (*e.g.*, `<`, `>=`) to perform, but `211` can determine this at compile time.

4. Rewrite the PVM code in the `.pyc` file so that it uses the new opcodes.

Consider the function below. Obviously this is a contrived example, but it nicely illustrates the operation of `211`.

```
def foo():
    pass
```

The PVM code for this function is shown below, as disassembled by `dis.dis(foo)`. (Recall that functions without an explicit `return` statement return the value `None`.)

```
0  SET_LINENO       1
3  SET_LINENO       2
6  LOAD_CONST       0  (None)
9  RETURN_VALUE
```

When executed by the PVM, the first two PVM instructions store the source line number from the original Python code, for use in tracebacks. The third instruction pushes the function's zero$^{\text{th}}$ constant, `None`, onto the PVM's stack. Finally, the fourth instruction pops the value off the top of the stack and returns.

Inside the Python interpreter, this would be implemented by the code:

```
while (1) {
    opcode = NEXTOP();
    if (HAS_ARG(opcode))
        oparg = NEXTARG();

    switch (opcode) {
    ...
    case RETURN_VALUE:
        retval = POP();
        why = WHY_RETURN;
        break;
    case LOAD_CONST:
        x = GETCONST(oparg);
        Py_INCREF(x);
        PUSH(x);
        break;
    case SET_LINENO:
        f->f_lineno = oparg;
        break;
    ...
    }
}
```

My 211 program takes all four PVM instructions and rewrites them into the single instruction `211_OPCODE 0`. It then outputs the C code below, which is `#include`d into the Python interpreter's main `switch` statement.

```
    case 211:
        switch (oparg) {
        case 0:
            f->f_lineno = 1;

            f->f_lineno = 2;

            x = GETCONST(0);
            Py_INCREF(x);
            PUSH(x);

            retval = POP();
            why = WHY_RETURN;
            break;
        }
        break;
```

`211` then goes through the C code it outputs, and performs "peephole optimization" — in other words, it searches for inefficient or redundant code and replaces it with improved code [1]. This does not imply that the Python interpreter's code was inefficient or redundant to begin with; this effect is caused solely by `211` piecing together chunks of the interpreter's code in new ways. The resulting C code is shown below.

```
case 211:
    switch (oparg) {
    case 0:
        f->f_lineno = 2;

        x = GETCONST(0);
        Py_INCREF(x);

        retval = x;
        why = WHY_RETURN;
        break;
    }
    break;
```

In this example, `211`'s peephole optimizer was able to remove the adjacent `PUSH/POP` operations, and delete the unnecessary assignment to `f_lineno`. These patterns are detected in the C code by some simple regular expressions.

The Python interpreter is recompiled next, incorporating `211`'s C code output. The rewritten (single-instruction) PVM code for `foo()` can now be run, and it will use the new customized interpreter code.

2.3 What 211 Doesn't Do

When a series of PVM instructions has been replaced with `211`'s specialized C code, the interpreter loop is not executed by the C code unless necessary. The speed increase that `211` gets is in part due to avoiding this loop overhead.

Unfortunately, the Python interpreter loop contains code to deal with periodic tasks, such as checking for signals[1] and changing thread states. By executing the interpreter loop less frequently, programs converted to C using `211` will be less responsive to these events.

Another thing to note is that `211` only addresses one area of Python performance. If a Python program spends most of its time in the run-time library, or is I/O-bound, then converting it into C is unlikely to make a big impact.

2.4 Results

The initial results of `211` can best be described as "mixed." All the results in this section were generated on a machine with a 200MHz Pentium-MMX processor, 512K of cache and 64M of RAM, using Windows NT 4.0 and Visual C++ 5.0. The version of `211` used converts less than half the PVM instructions: it can convert 37 out of a total of 84 PVM instructions.

Table 1 shows the results for Pystone 1.1, the Dhrystone benchmark included in the Python distribution. I ran three tests: an unmodified Python interpreter straight from the Win32 distribution; a modified Python interpreter which was customized for Pystone but ran the unmodified `pystone.pyc`; a "`211`" test which used a modified interpreter and a modified `pystone.pyc`. The file `ceval.c` is the C source file containing the Python interpreter.

From the numbers, it is fair to say that `211` speeds up the Pystone code, but at a cost too high to justify. On the other hand, the PyBench suite of benchmarks [11] shows some more interesting (and promising!) results in Figure 4.

Pybench converted into C is too much for Visual C++ on my machine to optimize, so instead I selected one of its source files, `Constructs.py`, and ran it through `211` alone. This means that the "IfThenElse," "NestedForLoops," and "ForLoops" tests have been compiled into C — this can be seen as picking a hot spot in a Python program and converting it to C. This results in 17000 lines being added to `ceval.c`, and an object file size of 137383 bytes.

The three tests directly affected by `211` show a dramatic speed improvement. I conjecture that slight speed decreases seen when using unconverted PVM code and a modified interpreter are architecture-related: `ceval.obj` passes a size threshold which necessitates use of slower branch instructions in the Pentium machine code. However, there are some artifacts — the results for "BuiltinMethodLookup" and "StringSlicing" — which require further study.

The next question is, how far can this conversion technique be pushed?

3 How to do Better

In this section I discuss five avenues which I think should (and, in one case, should not) be taken to get better results when converting PVM code to C; some of these suggestions are also applicable to Python optimization in general.

These comments should not be construed as a criticism of Python's current design. Some designs

[1]Depending on the platform.

```
PYBENCH 0.6

Benchmark: 211-python (rounds=10, warp=20)

Tests:                              per run    per oper.  diff *
-------------------------------------------------------------------------
          BuiltinFunctionCalls:     791.87 ms    6.21 us   +2.35%
           BuiltinMethodLookup:     944.35 ms    1.80 us   +8.73%
                 ConcatStrings:    2297.37 ms   15.32 us   -0.39%
               CreateInstances:    1198.43 ms   28.53 us   -2.26%
       CreateStringsWithConcat:     889.78 ms    4.45 us   -0.77%
                  DictCreation:    1161.54 ms    7.74 us   -0.18%
                      ForLoops:     523.39 ms   52.34 us  -62.11%
                    IfThenElse:     684.27 ms    1.01 us  -38.49%
                   ListSlicing:    1006.17 ms  287.48 us   +2.18%
                NestedForLoops:     383.25 ms    1.10 us  -49.84%
          NormalClassAttribute:     957.16 ms    1.60 us   +0.35%
       NormalInstanceAttribute:     931.60 ms    1.55 us   +0.38%
           PythonFunctionCalls:     890.95 ms    5.40 us   +3.89%
             PythonMethodCalls:     725.18 ms    9.67 us   -0.02%
                     Recursion:     690.87 ms   55.27 us   +2.17%
                  SecondImport:     937.91 ms   37.52 us   -4.78%
           SecondPackageImport:     961.25 ms   38.45 us   -5.25%
         SecondSubmoduleImport:    1174.28 ms   46.97 us   -4.81%
       SimpleComplexArithmetic:    1150.36 ms    5.23 us   -1.00%
        SimpleDictManipulation:     842.63 ms    2.81 us   -0.08%
         SimpleFloatArithmetic:     814.21 ms    1.48 us   +0.01%
      SimpleIntFloatArithmetic:     855.92 ms    1.30 us   +0.71%
       SimpleIntegerArithmetic:     857.16 ms    1.30 us   +0.72%
        SimpleListManipulation:    1033.49 ms    3.83 us   +3.93%
          SimpleLongArithmetic:     929.77 ms    5.63 us   +0.11%
                    SmallLists:    1569.67 ms    6.16 us   +3.16%
                   SmallTuples:     950.77 ms    3.96 us   +2.78%
         SpecialClassAttribute:     962.20 ms    1.60 us   +0.14%
      SpecialInstanceAttribute:    1062.08 ms    1.77 us   -0.54%
                 StringSlicing:     949.31 ms    5.42 us  -11.92%
                     TryExcept:    1658.73 ms    1.11 us   +2.33%
                TryRaiseExcept:     990.82 ms   66.05 us   -3.89%
                  TupleSlicing:     903.79 ms    8.61 us   -5.57%
-------------------------------------------------------------------------
            Average round time:   37401.40 ms              -4.80%

*) measured against: unmodified-python (rounds=10, warp=20)
```

Figure 4: PyBench results (partial compilation to C).

	Unmodified	Modified	211
Results (Pystones/second)	2736.42	2708.75	2921.17
`ceval.c` (lines)	2916	14890	14890
`ceval.obj` (bytes)	49838	116540	116540
`pystone.pyc` (bytes)	8270	8270	6018

Table 1: Pystone results (full compilation to C).

are better suited to certain applications than others, however, and it is safe to say that applications like `211` were not a priority when Python was created.

3.1 Opcode Education

In Section 2.4 I stated that `211` was not yet able to convert all of the PVM's opcodes into C. No technical difficulties preclude teaching `211` about the remaining opcodes. As a matter of expediency, I added opcodes lazily as required by the PVM code input to `211`.

When run, `211` prints a "top ten" list of PVM opcodes it was unable to convert, ordered by the frequency in which they occurred. This gave a static indication of opcodes to teach `211`; in addition, I used Python's dynamic execution profiler to identify frequently-executed opcodes. I used both these measures when deciding which opcodes to implement.

This means that, at present, a rewritten `.pyc` file invariably contains some regular PVM instructions. If these should happen to fall in a critical spot in the program, such as an inner loop, then performance is going to suffer. Teaching `211` the complete set of PVM opcodes would eliminate the transitions between specialized and nonspecialized code, avoiding one source of overhead. The longer C code sequences would also create more optimization opportunities, for both `211`'s peephole optimizer as well as for the C compiler's optimizer.

3.2 Conversion Heuristics

Currently, `211` always tries to convert the longest sequences of PVM instructions it is able to. This may not be the best strategy, since it can result in reams of C code being output; this time-space tradeoff has been noted in similar projects [14]. A better approach may be to selectively convert PVM code, based on either heuristics or thresholds. For instance, conversion could be restricted to frequently-seen pairs of instructions.

Ideally, the conversion heuristics could be tuned so that `211` would yield an acceptable speed increase, but with a much smaller amount of C code than it now outputs. It is unlikely that a single heuristic would work well for all inputs; run-time profiling feedback could perhaps be employed to select an appropriate heuristic.

3.3 Register Machines

As mentioned in the introduction, the PVM is a stack machine. As such, there is overhead involved in moving objects to and from the stack. An alternative is a register-based machine, which places intermediate values into registers rather than on a stack, and incurs no stack overhead. It is telling that programs which convert Forth and Java VM code into C begin by mapping stack locations into "registers" by placing their values into C variables [16, 6]. To implement this would require changing the PVM as well as the Python compiler.

There is another compelling reason to consider a register-based VM. There are a number of good optimizing C compilers available, despite it being a notoriously hard language to optimize. In terms of converting PVM to C, the key is to get the C output in such a form that an optimizing C compiler can make good use of it. But even the best C compilers are not likely to discover how values on a stack interrelate — stack manipulations appear just as pointer operations on memory. By going to a register-based VM, the data flow through the C code becomes clearer (as uses of C variables) and the compiler has a better chance of making more optimizations.

The following C program abstracts stack manipulation in the Python interpreter. The space for the stack is allocated externally, and elements of the stack are referenced using a pointer.

```
int i, j;

void interpreter(void) {
    extern int *get_frame(void);
    int *stack_pointer = get_frame();

    #define PUSH(x)  (*stack_pointer++ = x)
    #define POP()    (*--stack_pointer)

    PUSH(123);
    PUSH(456);
    i = POP();
    j = POP();
}
```

Even in theory, it is difficult for an optimizing C compiler to determine one key property about the values pushed onto the stack: their liveness. In other words, it cannot know if another function will want to use the contents of the memory allocated to the stack, so it must write both numbers to memory (as opposed to just keeping them in registers). In practice, C compilers fared even worse: neither Visual C++[2] nor gcc[3] with full optimization were even able to discover that `j` was supposed to have the value `123` without reloading it from memory.

In contrast, say that we have a register-based VM. A program like `211` could trivially map the registers into C variables, resulting in a program like the one below.

```
int i, j;

void interpreter(void) {
    int reg_0, reg_1;

    reg_0 = 123;
    reg_1 = 456;
    i = reg_1;
    j = reg_0;
}
```

The liveness of the values and the flow of data is now clear to the C compiler. Both the abovementioned C compilers do an excellent job translating this program into assembly.

3.4 Instruction Granularity

One frustration when developing `211` was that a lot of information was buried inside the interpreter's C code rather than made explicit in the PVM code; stack accesses and the details of function/method invocation are but two examples. I propose finer-grained PVM instructions, where each current PVM instruction would be replaced by one or more shorter ones. In essence, this would create a more RISC-like PVM. The immediate result of this change would be larger `.pyc` files, and slower execution resulting from a greater number of instructions being dispatched in the interpreter. However, in the long run, I think this loss would be more than won back by exposing more information to `211`-type programs and high-level PVM optimizers.

3.5 Interpreter Structure

There are other methods known for implementing interpreters. "Threaded code" and "indirect threaded code" are two techniques which are used in interpreter implementation [2, 5]. While the details of these two methods are beyond the scope of this paper, suffice it to say that both eliminate the fetch-execute loop of classical interpreters. Neither are likely to make any impact on the current PVM, though. It has been shown that the speed differences between the methods become nonexistent as the amount of time taken by each VM instruction increases [10]; this is echoed in more recent work [15].

Given this, I think it is not worthwhile to change the interpreter's structure until such time as Python has a more RISC-like PVM, with finer-grained instructions. This argument might also be reasonably extended to discourage "micro-optimizations" in the PVM instruction-fetching code.

4 Conclusion

In this paper I have described `211`, an experimental program which converts Python virtual machine code into C. `211` is intended for use on Python applications that need to run faster, where manual translation of hot spots to C is not a viable option. The general conversion technique has shown merit, both in my own results and in similar work done for other programming languages. I think that changes to Python's internals will help produce even better results.

Acknowledgments

Shannon Jaeger, Jim Uhl, and Mike Zastre made numerous helpful comments on early drafts of this paper. The anonymous referees' suggestions also brought out some important points which I had overlooked.

[2]Version 5.0 for 80x86, with `/Ox` flag.
[3]Version 2.8.1 for SPARC, with `-O9` flag.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] J. R. Bell. Threaded Code. *Communications of the ACM*, 16(6):370–372, 1973.

[3] J. L. Bentley. *Writing Efficient Programs.* Prentice-Hall, 1982.

[4] L. P. Deutsch. Efficient Implementation of the Smalltalk-80 System. *ACM POPL '84 Proceedings*, pages 297–302.

[5] R. B. K. Dewar. Indirect threaded code. *Communications of the ACM*, 18(6):330–331, 1975.

[6] M. A. Ertl and M. Maierhofer. Translating Forth to Efficient C. *EuroForth '95 Proceedings.*

[7] M. A. Ertl. Stack Caching for Interpreters. *ACM SIGPLAN PLDI '95 Conference*, pages 315–327.

[8] M. A. Ertl. A New Approach to Forth Native Code Generation. *EuroForth '92 Proceedings*, pages 73–78.

[9] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[10] P. Klint. Interpretation Techniques. *Software, Practice & Experience*, 11:963–973, 1981.

[11] M.-A. Lemburg. PyBench 0.6. `http://-starship.skyport.net/~lemburg/-pybench-0.6.zip`.

[12] M. Maierhofer and M. A. Ertl. Local Stack Allocation. *Compiler Construction (CC '98)*, pages 189–203. Springer, 1998.

[13] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[14] T. Pittman. Two-Level Hybrid Interpreter/Native Code Execution for Combined Space-Time Program Efficiency. *ACM SIGPLAN*, 22(7):150–152, 1987.

[15] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. *ACM SIGPLAN PLDI '98 Conference*, pages 291–300.

[16] T. A. Proebsting *et. al.* Toba: Java For Applications. *COOTS '97 Proceedings.*

[17] D. M. Ungar. *The Design and Evaluation of a High-Performance Smalltalk System.* MIT Press, 1987.