A Peephole Optimizer for Python

Skip Montanaro
Automatrix, Inc.
Rexford, NY
skip@calendar.com

Abstract

This paper describes the implementation of a peephole optimizer for Python byte codes. Python's byte code compiler currently generates code that can easily be improved. The peephole optimizer implemented presented here implements a number of common optimizations, including jump chaining, constant expression evaluation and elimination of unecessary loads. Some optimizations rely on specific properties of Python or its virtual machine. Some optimizations common to statically typed languages, such as algebraic simplification and expression rearrangement, are prevented by Python's dynamic typing.
Preliminary results obtained using pybench[Lemb] suggest that the optimizer has a positive benefit for specific operations. With the current measurement tools available however, it is difficult to quantify benefits that can be derived by using the optimizer. Situations in which the benchmarks may not yield reliable results are considered. The limitations Python places on the optimizer are discussed, especially restrictions caused by the dynamic nature of the language. Other performance improvements to the Python interpreter are discussed briefly.

1 Introduction

Python code is compiled to a high-level virtual machine in a straightforward fashion. We should be able to apply a peephole optimizer to the code to improve performance. This paper investigates an experimental peephole optimizer written in Python which is integrated with the existing byte compiler. In general, a peephole optimizer searches for patterns of opcodes that can be replaced by more efficient ones. Peephole optimization works because it can "see" more of the instruction stream at once than the compiler can during code generation. For instance, the simple

```
x = hairy_calc()
y = x + 15
```

would generate a store into x followed by an immediate load from x. The compiler wouldn't know to make a copy of the function value on the stack before storing it in x because it hasn't yet seen the statement after the one it is compiling. A peephole optimizer will see a very simple pattern[1]:

```
STORE x
LOAD x
```

which it can replace with:

```
DUP_TOP
STORE x
```

Two obvious optimizations are implemented by the byte code compiler itself. When Python is run with the -O flag, SET_LINENO opcodes and code to implement assert statements are not generated. There is much more that can be done, however. Some common constructs such as tuple assignments, while convenient for the programmer

```
a,b,c = 1,2,3
```

yield inefficient code which can easily be improved. Peephole optimization of Python byte code is performed by the xfreeze module[2], a contributed enhancement of the freeze module available from the Python Software Association's web site[3]. Several of the optimizations implemented here were derived from those found in xfreeze. Unfortunately, xfreeze is a specialized program mixing two purposes: optimizing byte code and binding a Python interpreter and a set of associated modules into a single monolithic executable. The relative rarity of the latter task means the optimizations it performs get little use.

[1] A notation similar to that generated by the Python disassembler is used in this paper. Labels and statement addresses are omitted and variable references are simplified.

2 Byte Code Interpreter

Python code is compiled into virtual machine code, that is, code that defines a computer that is not realized in hardware, but is executed at run-time by a byte code interpreter. Python's virtual machine is stack-oriented. With one exception, all instructions operate on the top one to three Python objects on the stack or transfer objects between the stack and fixed locations (representing local or global variables).

The main portion of the interpreter is implemented in the eval_code2 function in `ceval.c`. A large switch statement is used to decode and execute individual instructions:

```
switch (opcode) {
    case POP_TOP:
        ...
        continue;

    case ROT_TWO:
        ...
        continue;
    ...
    case BINARY_MULTIPLY:
        ...
        break;
    ...
}
```

There are several different ways to improve the performance of Python programs. Some of the most obvious ones are:

Optimize the Python code.
Straightforward reorganization of Python code, replacing less efficient algorithms with more efficient ones, reimplimenting some critical code in extension modules or rearchitecting the application altogether can have dramatic effects on the program's performance. In this regard Python is no different than most other programming languages.

Improve the efficiency of the interpreter itself.
The C code that implements the interpreter has been improved dramatically, and it can be made still more efficient. If the interpreter can make assumptions about the properties of some objects, some of the underlying functions the interpreter relies on can also be improved, or replaced altogether with versions that avoid needless error checks and thus streamline their execution.

Improve the sequence of instructions executed.
The peephole optimizer described here is one example of this. It works by matching fairly straightforward patterns of instructions and replacing them with faster, though equivalent, sequences. Much more sophisticated optimizations are possible as well, as has been demonstrated by Chambers for Self[Cham].

Change the architecture of the virtual machine.
Identifying special case instructions, such as integer loads or instructions that resolve dotted names (e.g., `string.join`) in one instruction instead of two can improve the performance of the system. Changing the underlying architecture - by adding registers to the architecture, for instance - can improve performance. For instance, adding registers to the virtual machine and implementing a decent register allocation scheme has the potential to reduce data movement within the virtual machine significantly.

Replace the interpreter altogether.
JPython replaces the Python virtual machine with the Java virtual machine. While this was not done necessarily for performance reasons, increased performance may result as JVM technology continues to improve. Python-to-C converters also fall into this category.

3 Architecture

The peephole optimizer[Mon1] is organized as a set of Python classes. Each class is a subclass of the OptimizeFilter base class, specialized for applying a single optimization. An entire optimizer is built by chaining several optimizers together in a pipeline. OptimizeFilter instances accept either a code object or another OptimizeFilter instance as input, as well as optional lists of local variable names, values, and constants. The optimize method breaks up the code into basic blocks, then calls optimize_block for each basic block, replacing it with the return value of the call.

```
def optimize(self):
    blocks = self.input
    self.output = [None]*len(blocks)
    for i in range(len(blocks)):
        self.output[i] = \
            self.optimize_block(blocks[i])
```

For the purposes of this work, a basic block is a straight section of code whose only entry point is the first instruction in the block. This differs slight from that used by Aho, et. al.[Aho], who further constrain basic blocks to have only one exit point. While the current definition isn't strictly incorrect, it may not expose as many opportunities for optimizations as Aho's does.

In most cases, OptimizeFilter subclasses need only implement an `optimize_block` method. One optimizer class (JumpNextEliminator) overrides the `optimize` method however, because it looks across basic block boundaries.

The optimizer is invoked by the byte code compiler if run-time optimization has been selected with the -O flag and the `optimize` module can be located and imported. An alternative implementation would be to post-process `.pyc` files, emitting `.pyo` files as a result. This was not considered because it would require reliance of the optimizer on the `new` module to generate new code objects which is not built into the Python executable by default.

Chaining individual optimizer instances into a pipeline allows large optimizers to be constructed easily. It is easy to plug new optimizer instances into the pipeline at the appropriate point(s). The main optimization function is:

```
def optimize5(code, v=None, n=None, c=None):
    lsc = LoadStoreCompressor(code, v, n, c)
    sle = StoreLoadEliminator(lsc, v, n, c)
    lnr = LineNumberRemover(sle, v, n, c)
    ...
    jne2 = JumpNextEliminator(dcr2, v, n, c)
    jpo2 = JumpOptimizer(jne2, v, n, c)
    return jpo2.code()
```

Note that each optimizer instance except the first takes another optimizer instance as input. The first optimizer step, the `LoadStoreCompressor` instance, takes the byte code string as input.

Several new instructions were added to the Python virtual machine. The `LOADI` instruction uses the `small_ints` cache maintained in `intobject.c` to speed up loading of small positive integers. `LOAD_TWO_FAST` and `STORE_TWO_FAST` encode two arguments to load in their 16-bit arguments. `LOAD_ATTR_FAST` and `STORE_ATTR_FAST` encode a local argument reference and an attribute name reference in their 16-bit arguments.

In the paragraphs that follow, each of the optimization classes currently implemented is described briefly, with examples, where appropriate, of the specific optimizations they perform.

3.1 TupleRearranger

This class optimizes Python code like

```
a,b,c = 1,2,3
```

into the byte code equivalent of

```
c = 3
b = 2
a = 1
```

avoiding the overhead of building and unpacking a tuple. This is an obvious pattern likely to appear in Python byte code, but less likely to occur in languages without similar operations. The execution order of the right-hand side of the statement is not affected. The assignment to the variables on the left-hand side is optimized, however. The above assignment statement generates a sequence of instructions that look like

```
LOAD_CONST    1
LOAD_CONST    2
LOAD_CONST    3
BUILD_TUPLE   3
UNPACK_TUPLE  3
STORE_FAST    a
STORE_FAST    b
STORE_FAST    c
```

The net effect of the `BUILD_TUPLE/UNPACK_TUPLE` pair is simply to reverse the order of the three values on the top of the stack. Instead of reversing the order of the operands, it's simpler to reverse the order of the assignments:

```
LOAD_CONST    1
LOAD_CONST    2
LOAD_CONST    3
STORE_FAST    c
STORE_FAST    b
STORE_FAST    a
```

and avoid tuple creation and deletion altogether.

This optimization is not applied in all cases. If the `UNPACK_TUPLE` instruction is not followed immediately by at lest the same number of simple stores (stores to individual global or local variables) to unique variables, the optimization is not performed. Transforming

```
LOAD_CONST    1
LOAD_CONST    2
LOAD_CONST    3
BUILD_TUPLE   3
UNPACK_TUPLE  3
STORE_FAST    a
STORE_FAST    b
STORE_FAST    b
```

into

```
LOAD_CONST    1
LOAD_CONST    2
LOAD_CONST    3
STORE_FAST    b
STORE_FAST    b
STORE_FAST    a
```

is incorrect, because `b` has the value `3` after executing the first sequence, but `2` after executing the second sequence.

3.2 ConstantExpressionEvaluator

This class precomputes a number of constant expressions and stores them in the function's constants list, including obvious binary and unary operations and tuples consisting of just constants. Of particular note is the fact that complex literals are not represented by the compiler as constants but as expressions, so 2+3j appears as

```
LOAD_CONST n (2)
LOAD_CONST m (3j)
BINARY_ADD
```

This class converts those to

```
LOAD_CONST q (2+3j)
```

which can result in a fairly large performance boost for code that uses complex constants. In general Python code this optimizations is probably not as useful as in languages like C that make heavy use of macro processors to define manifest constants.

3.3 UnreachableCodeRemover

This class deletes all instructions in a basic block that follow an unconditional transfer. It does not improve the performance of the program directly, though by eliminating unused code it can sometimes clear the way for further optimizations.

3.4 LoadStoreCompressor

This class performs two similar optimizations. It replaces pairs of `LOAD_FAST` or `STORE_FAST` instructions with single `LOAD_TWO_FAST` or `STORE_TWO_FAST` instructions, respectively. It also replaces the sequences `LOAD_FAST/LOAD_ATTR` or `LOAD_FAST/STORE_ATTR` with `LOAD_FAST_ATTR` or `STORE_FAST_ATTR` opcodes, respectively. The `LOAD_FAST_ATTR` and `STORE_FAST_ATTR` instructions seem to speed up attribute lookup, but it's not clear that the `LOAD_TWO_FAST` and `STORE_TWO_FAST` opcodes result in much of a performance improvement (they do save a trip around the interpreter loop), and they further complicate downstream optimizations by increasing the number of different types of load or store instructions that must be considered by later optimization classes. For this reason this optimizer is currently executed late in the pipeline. It should probably be split into two separate classes.

3.5 MultiLoadEliminator

This class converts n consecutive loads of the same object to a single load and n-1 `DUP_TOP` opcodes.

3.6 StoreLoadEliminator

This class converts a store to an object then a load of the same object into a `DUP_TOP` followed by a store. This construct occurs frequently in code that saves an expensive-to-compute quantity then reuses it:

```
foo = spam = some_hairy_function(...)
bar = foo * math.sin(math.pi/3)
baz = aunt_mabel * foo
```

This class currently eliminates the first load of foo but not the second.

3.7 JumpNextEliminator

This class eliminates an unconditional jump to the following basic block if it is the last opcode in its basic block.

3.8 JumpOptimizer

This class traverses a chain of n jumps, short-circuiting the second through n-th jumps if they are unconditional.

3.9 ConstantShortcut

This class converts `LOAD_CONST` instructions for None or small positive integers to `LOAD_NONE` or `LOADI` instructions, respectively. `LOAD_NONE` short-circuits the constant lookup of `Py_None`. `LOADI` uses the `small_ints` cache in intobject.c to avoid looking up small positive integer constants. Both optimizations are useful because they reduce a series of memory references of the form

```
(PyTupleObject *)(f->f_code->co_consts)->ob_item[i]
```

where f is the current frame, to either `small_ints[i]` or `Py_None`.
Generation of `LOADI` instructions is a bit fragile, since it relies on the size of the `small_ints` cache

in intobject.c being the same when the optimizer is run as when the actual byte codes are executed later. A more general approach to efficient access of a function's constants would be to create a `fastconstants` variable in `eval_code2`, similar to the `fastlocals` variable already used to speed up access to local variables.

3.10 Branches Not Taken

At first glance, a number of other code transformations seem possible. It would be nice to rearrange expressions using the algebraic properties of numbers, but this can't be done because although an object implements a particular operation, it's not required to implement the usual semantics associated with the operation on numbers, and with no side-effects. For instance, consider the expression

```
l + a + l
```

This compiles to

```
LOAD l
LOAD a
BINARY_ADD
LOAD l
BINARY_ADD
```

If `l` and `a` are numbers the expression can be rearranged to avoid a load

```
LOAD l
DUP_TOP
BINARY_ADD
LOAD a
BINARY_ADD
```

but not if `l` and `a` are strings.

This problem applies to strength reduction as well. If we know we are dealing with integers or floating point numbers the following statement

```
a = a ** 2
```

can be sped up by recasting it as

```
a = a * a
```

since multiplication is faster than exponentiation. The optimizer doesn't know that `a` is a number, however. (There may be some situations where it can be inferred `a` is an int or float object, though this is currently not possible in the general case.)

If the view of the code being optimized is expanded beyond the realm of basic blocks, other optimizations are possible. Dead code elimination identifies variables that are set but not used and eliminates them. Cross jumping merges the code common to the tail end of two basic blocks whose sole successor is the same block. The following code segments demonstrate cross jumping:

```
if type(a) == types.ComplexType:
    x = cmath.sin(a)
    foo(x)
else:
    x = math.sin(a)
    foo(x)

if type(a) == types.ComplexType:
    x = cmath.sin(a)
else:
    x = math.sin(a)
foo(x)
```

Stack pollution (deferred stack cleanup) could be performed if we added a `POP n` instruction to the virtual machine. Stack pollution saves stack cleanup until it's required. For instance, the following code

```
f(10)
g(20)
```

compiles to

```
LOADI 10
LOAD f
CALL_FUNCTION 1
POP_TOP
LOADI 20
LOAD g
CALL_FUNCTION 1
POP_TOP
```

The first `POP` could be deferred:

```
LOADI 10
LOAD f
CALL_FUNCTION 1
LOADI 20
LOAD g
CALL_FUNCTION 1
POP 2
```

This requires analysis to determine that elements on the stack below the return value of `f(10)` are not needed before the call to `g(20)` returns.

4 Results

All performance tests were run on a machine with the following configuration:

```
100MHz Pentium, 64MB main memory
RedHat Linux 5.0
EGCS 1.0.3 compiler
Python 1.5.1
pybench 0.6
```

Python was configured using "–with-threads". The interpreter was compiled using the -O2 flag of EGCS. While this does not represent the maximum optimization possible, it is perhaps the most common optimization level used. The benchmark was run four times, twice with the peephole optimizer disabled, and twice with it enabled. Between pairs of runs all .pyo and .pyc files were deleted. Only the results of the second run of each pair were used to minimize the effect of executing the optimizer (which is itself rather slow) on the benchmark results. Python was run with the -S and -O flags and the PYTHONPATH and PYTHONSTARTUP environment variables were not set.

Table 1 shows the results of the benchmark tests. The numbers represent the per cent reduction (negative values) or increase (positive values) in execution time for the optimized versus unoptimized runs.

Test	% diff
Builtin Function Calls	-2.51%
Second Import	+1.37%
Builtin Method Lookup	-11.95%
Second Package Import	+1.63%
Comparisons	+2.06%
Second Submodule Import	+0.50%
Concat Strings	-7.34%
Simple Complex Arithmetic	-14.68%
Create Instances	-0.01%
Simple Dict Manipulation	-4.24%
Create Strings With Concat	-12.30%
Simple Float Arithmetic	+3.85%
Dict Creation	-5.01%
Simple Int Float Arithmetic	+1.55%
For Loops	-5.46%
Simple Integer Arithmetic	-0.74%
If Then Else	-3.52%
Simple List Manipulation	-8.07%
List Slicing	-0.34%
Simple Long Arithmetic	-7.83%
Nested For Loops	-0.97%
Small Lists	-7.11%
Normal Class Attribute	-20.64%
Small Tuples	-3.74%
Normal Instance Attribute	+3.84%
Special Class Attribute	-23.11%
PPrint	-1.88%
Special Instance Attribute	+1.65%
Pickler	+0.31%
String Slicing	-9.55%
Python Function Calls	+0.81%
Try Except	+0.67%
Python Method Calls	-2.72%
Try Raise Except	-2.33%
Random	+4.32%
Tuple Slicing	-6.39%
Recursion	+7.30%

Table 1: The effect of applying the peephole optimizer

The most obvious result is that there are a few tests that show very large improvements, a few that show moderate improvements, but none that apparently suffer much from the changes. Why this might be the case is discussed in the discussion below.

5 Discussion

5.1 Interpreting the Benchmark Results

This work shows that under the right circumstances, peephole optimization can yield significant performance gains. Measuring those gains is currently rather difficult, however, partly because they are generally small, but also because the tools available to measure them are either crude (simple timings), they aren't yet well characterized (pybench) or they demonstrate an unrealistic mix of operations that makes extrapolation to real programs difficult (pystone and pybench). In addition, differences between compilers, optimization levels and the processors they target can mask improvements. The range of optimizations applied here is limited to patterns the author saw while scanning byte code or those suggested by others. It is likely that over time more

optimizations will be suggested and that some of the current optimizations will be improved.
Tim Peters cautions against too great a reliance on measurements of individual optimizations[Pete]:

> A good change should be made even if timing shows it slowing down on some combo; if the operation count has been reduced, the timing result is a fluke due to bad compiler or load-map luck that's probably specific to every detail of the combo in use, and that will go away by magic anyway.

This advice is particularly germane if the optimization has not been tested on multiple architectures, it is still in an early stage of development or the tools used to measure the change are not yet well understood. Performance will be affected by many factors, including processor architecture, the compiler, optimization level and cache size. It's best to make a full range of measurements before deciding whether or not to use the output of the optimizer.
Pybench contains a series of tests which each try to exercise a single operation. It is useful as an objective gauge for measuring changes to the Python interpreter where the byte code generated by the compiler is not modified and suggesting the general direction of those changes, however it must be understood that the code in the tests doesn't resemble typical code. Since the same operation is executed repeatedly, the peephole optimizer will tend to show no effect (when it couldn't improve the particular operation being tested) or a very dramatic positive effect (when it was able to optimize a particular sequence). As an example, consider the SpecialClassAttribute test in Lookups.py. It repeatedly executes statements like:

```
c.__a = 2
x = c.__a
```

where c is a class defined in the local scope and x is a local variable. The peephole optimizer make significant improvements in the code generated for these statements. First, constants will be loaded using the more efficient `LOADI` instruction instead of `LOAD_CONST`. Second, attribute stores and loads will be performed by either `STORE_FAST_ATTR` or `LOAD_FAST_ATTR`. Assignment to c.__a looks like

```
LOADI           2
STORE_ATTR_FAST c, _SpecialClassAttribute__a
```

while assignment to x looks like

```
LOAD_ATTR_FAST  c, _SpecialClassAttribute__a
STORE_FAST      x
```

Contrast this with the unoptimized code

```
LOAD_CONST      3
LOAD_FAST       c
STORE_ATTR      _SpecialClassAttribute__b
```

and

```
LOAD_FAST       c
LOAD_ATTR       _SpecialClassAttribute__c
STORE_FAST      x
```

Only two-thirds as many instructions are executed per assignment, so even if they are individually no more efficient, the code speeds up significantly.
The SimpleComplexArithmetic test falls prey to the same magnified result, but for a different reason. Complex constants are not stored as constants in the unoptimized case, but as a pair of loads followed by an add or a subtract. The optimizer precomputes these constant expressions and saves the constants for later reuse. While both tests demonstrate the positive effect of one or more of the optimizations, their magnitude has little direct bearing on what one could expect if the optimizer was applied to production code. Also, Python code optimizers improve they will probably progress to the point where much of the repeated code in pybench's tests is deleted, making them much less useful as tests of either the optimizer or improvements to the Python virtual machine.
Pybench might or might not test most commonly occurring operations, or may test them out of context. If so, the optimizer won't get the opportunity to strut its stuff. For instance, it doesn't currently test the TupleRearranger optimization at all
The only test that seemed to run a significant amount slower was the Recursion test. It's not clear why that was the case. The optimizer made only a couple transformations to the recursive function that traded a couple `LOAD_CONST`s for `LOADI`s and removed some unreachable code. Studying the C code that implements `LOAD_CONST` and `LOADI`, it's difficult to see how the former would be faster than the latter. Executing the following simple function with and without peephole optimization enabled suggests that `LOADI` is much faster than `LOAD_CONST`.

```
def f():
    import time

    t = time.clock()
    for j in xrange(10000):
        for i in xrange(100):
            pass
    ohd = time.clock()-t
```

```
t = time.clock()
for j in xrange(10000):
    for i in xrange(100):
        a = 2; a = 2; a = 2; a = 2
print time.clock()-t-ohd
```

With the optimzer disabled, executing the above function displayed 6.88 on my 100 MHz Pentium. With it enabled, it displayed 3.38. Inspection of the disassembled byte code showed that the substitution of `LOADI` for `LOAD_CONST` instructions was the only change to the code of the inner loop.
Over the long-run, a series of benchmarks should be developed that use a mix of actual Python application code. A small attempt was made to move in that direction by adding two benchmarks to pybench (PPrint and Pickler) which are based on test code in two standard Python modules. More complex "real-life" tests should be developed, however. (Who wants to implement HSPICE in Python?)

5.2 Optimizer Speed

The speed of the optimizer itself needs to be improved, probably by recoding parts of it in C. In its current state it is a hindrance to startup of programs with short runtimes or that execute most of their code in the `__main__` module whose byte code is never saved to a .pyo file for later reuse. In the absence of an actual Python-to-C compiler, it may be possible to rely on the fairly regular structure of the optimizer classes to aid in manual conversion. Perhaps some sort of template matching "compiler" such as Strout's Python2C[Stro] specially tailored to this application can be written to automate much of the conversion. This would make it easy for people to test new optimizations without resorting to coding them directly in C, but once they are tested to fairly painlessly migrate them to C.

5.3 Miscellaneous Improvements

The technique of speeding up constant loads should be investigated further. While this appears to improve performance, it could probably be generalized to speed up access to all of a function's constants without relying on coupling between the interpreter and Python's integer object.
In theory, it is possible to generate both conservative and aggressive code based upon presumed knowledge of object types, though it may be difficult to realize much of a performance gain from it unless the aggressive optimizations are fairly significant. It seems unlikely that it would yield much better performance than the current inlining of integer adds and subtracts unless a large block of heavily optimized code could be selected with a single test.

5.4 Do We Need Type/Protocol Declarations?

To allow the prospect of efficient module compilation to C or machine code, at various times people have discussed adding some form of (optional) static type declarations to Python. At issue are what the syntax of such declarations might look like, what needs to be specified and how much type inference can be done by the compiler without them (which would lessen the need for static type declarations).
Recently the topic of protocols was raised in the Python newsgroup. John Skaller implemented a basic protocol module[Skal]. The current module may not be sufficient for the needs of an optimizer without enhancement, however, since it allows one to state what operations a module implements, but not their semantics. Also, protocols are implemented by executing Python code. It's unclear how their properties could affect the compiler or optimizer without actually executing the code.
Chambers[Cham] demonstrated that a significant amount of optimization can be performed in highly dynamic object-oriented languages. Many of the techniques applied to Self should be applicable to Python as well.

5.5 Are We Even Headed in the Right Direction?

Peephole optimization is better than nothing, but doesn't look like it will yield major improvements in performance, probably not more than 10-20 per cent. More sophisticated optimization strategies can be tried, but while they've been demonstrated in other environments, they are not trivial to implement. Python's dynamic typing makes it challenging to infer types and compile-time, though Chambers[Cham] clearly shows this is possible.
Chambers showed that code generation for multiple cases can be used to generate code for common, fast cases (and allow function inlining in some cases) while still preserving the full generality of the language.
Instead of applying effort trying to work with incomplete type information, it might be worthwhile to look at more radical changes to the run-time system. For instance, adding registers to the Python virtual machine in the form of a set of register variables

local to eval_code2 and using well-understood register allocation schemes might significantly improve the data movement characteristics of the interpreter for both global and local variables, especially in inner loops. Suppose the following Python code is executed:

```
from math import sin
for x in range(100):
    plot(x, sin(x))
```

The body of the loop compiles to something like:

```
LOAD_FAST plot
LOAD_FAST x
LOAD_FAST x
LOAD_FAST sin
CALL_FUNCTION 1
CALL_FUNCTION 2
POP_TOP
```

Each LOAD_FAST instruction copies a local variable reference to the stack from the fastlocals array – two array accesses and a reference count increment per local variable on each iteration. If register variables were available to store references to x, sin and plot, setup code for the loop would load them into registers once where they could be accessed directly by register-oriented instructions, and with much smaller penalties. Loop finalization code would copy x back to its slot in the fastlocals array. Assuming references to sin, x and plot are in R1, R2 and R4, respectively, this yields a loop body something like

```
CALLR1 R1, R2          ; R3 = sin(x)
POPR R3
CALLR2 R4, R2, R3      ; plot(x, R3)
POP_TOP
```

which would reduce or eliminate much of the movement of data between variables and the stack. The tradeoff would be the increased cost of decoding register arguments. That might be nothing more than an array index operation, but wouldn't be free.

Implementation of such a system would require substantial changes to the Python virtual machine and a reasonably sophisticated optimizer. The potential payoff seems significant, however. LOAD and STORE opcodes are the most frequently executed instructions in the Python virtual machine.

Another possible approach to consider is run-time native code generation and optimization. Kistler[Kist] describes a code generator and optimizer that targets its efforts as it profiles the running system. Chambers[Cham] also implemented on-the-fly translation to machine code for Self. Such a system would appear to integrate well with Python, since it is easy to replace function and method definitions at run-time. One attraction of on-the-fly native code generation is that it can be restricted to just the code that will actually be executed, thus allowing more effort to be put into optimizing the code that is generated.

Optimization of Python programs is still in its infancy. Peephole optimization is probably the easiest way to improve performance, though it is limited in how much can be achieved. By adding more techniques common to modern native code compilers[Cham], it should be possible to continue to improve the efficiency of Python byte code. Further out, more radical changes to the underlying system (which can be thought of as different implementations of the same architecture) may yield much larger performance improvements.

6 Conclusions

Peephole optimization appears to be just the tip of the optimization iceberg. In its current state, it appears to improve performance by several per cent as suggested by pybench results. The performance of the optimizer itself can be a hindrance and suggests reimplementation of parts of the optimizer itself in C. More aggressive optimizations, such as code inlining and customization, and architectural changes to the Python virtual machine, such as the addition of registers, suggest that there is plenty of room for further significant improvements in performance of Python applications.

7 Appendix: Other Changes

During the course of studying the main Python interpreter function eval_code2, several simple changes to the Python interpreter were tried[Mon2]. They included:

Optimization of stack access

: By minimizing the movement of the stack pointer the number of memory references can be reduced, resulting in small performance gains for opcodes that make heavy use of the stack such as DUP_TOP, ROT_TWO and ROT_THREE.

Caching the top element of the stack in a register

: This improves the memory access patterns of the interpreter a bit more.

Short-circuiting interpreter error checks

The error checking code at the bottom of the interpreter loop tests the values of three variables: `err`, `x` and `why`. With few exceptions, each virtual machine instruction sets at most one of those variables, so it's almost always faster to check for the one error condition an instruction can set directly the code that implements that instruction, avoiding the general set of checks at the bottom of the loop. This improvement was begun in Python 1.5, but was still incomplete as of the release of 1.5.1.

These optimizations yielded moderate performance improvements for some pybench tests, but also degraded performance in other tests. Further work is needed to identify and correct those problems, but these changes show promise.

References

[Aho] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishers (1986).

[Cham] Craig Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* Ph. D. dissertation, Computer Science Department, Stanford University, March 1992. http://self.sunlabs.com/papers-/chambers-thesis

[Kist] Thomas Kistler. *Dynamic Runtime Optimization* UC Irvine Technical Report 96-54. http://www.ics.uci.edu/~kistler/ics-tr-96-54.ps (November 1996)

[Lemb] Marc-Andre Lemburg. *Pybench.* http://starship.-skyport.net/~lemburg/pybench-0.6.zip

[Mon1] Skip Montanaro. *Peephole Optimizer Patch* http://www.automatrix.com/~skip/python-/optimizer.patch

[Mon2] Skip Montanaro. *Stack Manipulation Patch* http://www.automatrix.com/~skip/python/stack-diffs.out

[PSA] *Python Software Association.* http://www.-python.org/

[Pete] Tim Peters. *Post to the comp.lang.python newsgroup* http://www.dejanews.com-/getdoc.xp?AN=369547398, (July 8, 1998).

[Skal] John Skaller. *Post to the comp.lang.-python newsgroup* http://www.dejanews.com-/getdoc.xp?AN=369976995, (July 9, 1998)

[Stro] Joseph Strout. *Python2C* http://www.strout.net/python/ai/python2c.py

[Tutt] Bill Tutt. *Xfreeze.* http://www.python.org/ftp-/python/contrib/System/xfreeze.README