The Effects of Unrolling and Inlining on Python Bytecode Optimizations

Yosi Ben Asher, Nadav Rotem
Haifa University
The Python Programming Language

- Very popular dynamic programming language combining object-oriented and scripting concepts
- Features a fully dynamic type system named 'duck typing'
- Compiled into bytecode and executed by an interpreter
- Known to be hundreds of times slower than C or Java
Data Flow Optimizations

- Data flow optimizations are a set of optimizations that are known to be very effective.

- Typically, this set includes constant propagation, common sub-expression elimination, algebraic simplifications, copy propagation and dead code elimination.

- In general, these optimizations create a more dense code by simplifying expressions and removing dead code.
Example of Dynamic Typing

```python
>>> def add(a, b):
...     return a + b
... # define a new function

>>> add(1, 2)
3 # integers

>>> add([1,2,3], [4,5,6])
[1, 2, 3, 4, 5, 6] # lists

>>> add("hello", "world")
"hello world" # strings
```
Failed Data Flow Optimizations

- The following algebraic simplification is valid for integers: $(a*2 + b*2)$ becomes $(a+b) * 2$

- However, if $a$ and $b$ are strings, it is not valid.
Optimizing Python

- Applying compiler optimizations is challenging due to Python's dynamic typing system.

- In order to preserve the correctness of the original program, special considerations must be taken even when implementing the most standard optimizations.
Bytecode Optimization

- In this work, we developed optimizations which are unique to dynamic languages.
- We disassembled the precompiled Python bytecode and reconstructed into data-dependency trees and optimize them.
- We recovered compiled bytecode files (.pyc files) which contain no AST information.
- We have extended the standard data flow analysis with specific rules to identify cases that are safe.
Bytecode Structure

- Python uses a stack-based bytecode which is generated from the AST.

- The Python opcodes operate directly on the stack.

- A 'BINARY_ADD' instruction, for example, pops two items from the stack and pushes a single item, which is the sum of the two original items.

- The add instruction tells the lower stack object to call the internal '__add__' method with the other object as a parameter.
Bytecode Structure

LOAD_FAST 0 // "a"
LOAD_FAST 1 // "b"
BINARY_ADD
RETURN_VALUE
Python 'Duck Typing' System

class Person():
    def talk(self): print "I am a person"

p = Person()  # Create a new Person object

def quack(): print "I am a duck"

p.talk = quack  # Override a function

>>>p.talk()
I am a duck
Unsafe Optimizations and Side Effects

- Consider the following code:

  ```python
  for i in xrange(100):
      sum += x*y
  ```

- In Java, CSE pass would evaluate "x*y" only once.
- However, in Python, a method could be overridden by another method which has a side effect. This method could potentially write a log file every time x is multiplied by y.
- We have no way of knowing in advance what x would do when multiplied by y.
Loop Unrolling

- Loop unrolling is a well-known transformation.
- The first unrolling pass we implemented unrolls numeric loops (xrange loops).
- The unrolling of the 'xrange' iterator is done by changing the 'xrange' constructor when it is created in order to yield values in steps that are greater than one.
- Then, the body of the loop is duplicated and modified to accommodate the changes and execute the next iteration.
xrange unrolling

Original loop:

for i in xrange(n):
    z = i*7 + i*2

Transformed loop:

m = n-(n % unroll)
# unrolled loop body
for i in xrange(0,m-1,unroll):
    z = i*7 + i*2
    z = (i+1)*7 + (i+1)*2
    ...

# loop tail
for i in xrange(m,n, 1):
    z = i*7 + i*2

The iteration range may not be a multiplication of the unroll parameter.
A 'tail' must finish the last iterations.
Complete Unrolling of Lists

- Using iterators is the 'native' way to iterate over data in Python.
- We have implemented two variants of unrolled iterations.
- The first unroll pass is for lists of known size and content. For example:

```python
for x in [1,2,3,4]:
    print x
```

```
print 1
print 2
print 3
print 4
```
Unrolling Iterators of Unknown Size

```python
def f(bar):
    sum = 0
    for p in bar:
        sum += p

def f(bar):
    sum = 0
    it = bar.__iter__()
    try:
        while(1):
            p1 = it.next() ; i = 1
            p2 = it.next() ; i = 2
            p3 = it.next() ; i = 3
            p4 = it.next() ; i = 4
            sum += p1+p2+p3+p4
    except StopIteration:
        # handle tail if needed
        based on value of i
        if i > 1: ...
        if i > 2: ...
```
Inlining of Functions

- Python function calls are time-consuming in comparison to other compiled languages.
- Inlining is a transformation where a call to a function or a method is replaced by its body, and the called arguments are inserted into the body of the loop.
- Each return call in the original inlined function is translated into a 'store' and 'jump to end' set of opcodes.
Inlining and Unrolling may assist one another

- These transformations help to reduce the 'type uncertainty'.
- Inlined functions have access to type information from the calling function. Parameters may become constants.
- Complete unrolling of constant lists gives concrete knowledge of type.
Example

def func_2():
    t = 123
    for func in [F1,F2,F3]:
        func(t)
User-Guided Optimizations

- Some of the possible optimizations are not type-safe.
- We allow the user to specify which methods should be optimized by Python 'decorators' which are source code annotations.
- This method can be further extended to indicate other safety features.

```python
@NumericCode
def func(x, y):
    return x*2 + y*2
```
Bytecode Optimizations

**Basic Block Optimization**
- Value propagation
- Constant propagation
- Common sub-expression elimination
- Loop invariant
- Strength reduction
- Memory optimizations
  - Load elimination
  - Store elimination
- Global variable cache

**CFG Optimizations**
- Loop Unrolling:
  - Complete unroll
  - Iterator unroll
  - Range unroll
  - Random access transformation
- Method Inlining
Benckmarks

- The proposed optimizations were tested using several benchmarks: Pystone, Pybench, Crypto, PyPy and several micro tests.
- Results show significant improvement.

![Graph showing performance improvements for various benchmarks](image-url)
Thank you. Questions ?
Backup Slides
def func(a,b,c):
    return a[b]*c + b*c + a[0]

>>> dis.dis(func)
  2           0 LOAD_FAST          0 (a)
  3           3 LOAD_FAST          1 (b)
  6          6 BINARY_SUBSCR
  7          7 LOAD_FAST          2 (c)
 10         10 BINARY_MULTIPLY
 11         11 LOAD_FAST          1 (b)
 14         14 LOAD_FAST          2 (c)
 17         17 BINARY_MULTIPLY
 18         18 BINARY_ADD
 19         19 LOAD_FAST          0 (a)
 22         22 LOAD_CONST          1 (0)
 25         25 BINARY_SUBSCR
 26         26 BINARY_ADD
 27         27 RETURN_VALUE
def f(x):
    v = 5
    if (x==9):
        return x + v
    return x*3

def g():
    sum = 0
    for i in xrange(n):
        sum += f(7+i)
    return sum

def new_g():
    sum = 0
    for i in xrange(n):
        $inline_x = 7+i
        $local_v = 5
        if ($inline_x==9):
            _inline_return=x+$local_v
            *goto END_TAG
        _inline_return = x*3
        *goto END_TAG
    END_TAG:
    sum += _inline_return
    return sum