The Effect of Unrolling and Inlining for Python Bytecode Optimizations

Yosi Ben Asher
CS department, Haifa University, Israel.
yosi@cs.haifa.ac.il

Nadav Rotem
CS department, Haifa University, Israel
rotemn@cs.haifa.ac.il

Abstract

In this study, we consider bytecode optimizations for Python, a programming language which combines object-oriented concepts with features of scripting languages, such as dynamic dictionaries. Due to its design nature, Python is relatively slow compared to other languages. It operates through compiling the code into powerful bytecode instructions that are executed by an interpreter. Python's speed is limited due to its interpreter design, and thus there is a significant need to optimize the language. In this paper, we discuss one possible approach and limitations in optimizing Python based on bytecode transformations. In the first stage of the proposed optimizer, the bytecode is expanded using function inline and loop unrolling. The second stage of transformations simplifies the bytecode by applying a complete set of data-flow optimizations, including constant propagation, algebraic simplifications, dead code elimination, copy propagation, common sub expressions elimination, loop invariant code motion and strength reduction. While these optimizations are known and their implementation mechanism (data flow analysis) is well developed, they have not been successfully implemented in Python due to its dynamic features which prevent their use. In this work we attempt to understand the dynamic features of Python and how these features affect and limit the implementation of these optimizations. In particular, we consider the significant effects of first unrolling and then inlining on the ability to apply the remaining optimizations. The results of our experiments indicate that these optimizations can indeed be implemented and dramatically improve execution times.

Categories and Subject Descriptors D.1.5 [*Software Engineering*]: Programming Techniques—Object-oriented Programming

General Terms Performance

Keywords Python, Bytecode Optimizations, Dynamic Languages

1. Introduction

Python [7] is a popular programming language combining object-oriented and script languages. It is a dynamically-typed language wherein new types can be dynamically assigned to variables during execution. In the following example x may enter as a list or as an integer and return as either a real number or as a list of unknown type $if(x > 0)\ :\ \ x+ = 0.23;\ else:\ x.append(y)$.

Python, like many other dynamic languages, uses an interpreter, which adds overhead. The design of the python interpreter makes Python slower than other languages such as C, C++ and Java (all statically-typed languages).

Applying compilation and compiler driven optimizations to Python is a creative and practical challenge.

Research efforts to compile and optimize Python have been undertaken. However, more compilation techniques and optimizations are needed in order to further optimize Python programs. The dynamic typing of Python is combined with other dynamic features including:

- Polymorphic operators that are able to operate on many unknown types, e.g., $x + y$ is able to operate on lists, strings, numbers and any object that implements the $.\_\_add\_\_$ method.
- Dynamic evaluation of new or updated functions, methods and classes in instances, e.g., $if(x < y)x.foo = y.bar$.
- Call sites that may reference many possible functions, e.g., $if(x < y) : f = g;\ f(x, y)$. Thus, a static call graph cannot be computed at compile time.

In order to preserve the correctness of the original program without losing potential performances, special considerations must be taken even when implementing the most standard optimizations.

Data flow optimizations [12, 9] are a set of optimizations that are known to be very effective for imperative programming languages. In their extended form, data flow optimizations process CFGs (control flow graphs) created by by if-then-else branches and loops. 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, eliminating redundant load/store operations and removing superfluous assignments (see section on data flow optimizations).

We have extended the standard data flow Analysis [9] with specific "filtering" rules to identify cases that are safe (in terms of Pythons dynamic types and features) for data flow optimizations. The exact set of rules that has been used to extend data flow analysis is too technical to be included here. Instead we provide examples to illustrate the problematic cases that should be filtered out. One example is Invariant Code Motion which includes expressions that are not modified in a loop's body and are computed outside the loop [4], avoiding repeated execution of an invariant code.

Namely, $x * y$ in the loop

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

may seem to be an invariant expression. However, if $x.\_\_mul\_\_$ implements some kind of a logging mechanism or effects a global variable each time it is being multiplied by y, then moving $x * y$ outside the loop is incorrect. The rule used to verify that $x * y$ can be safely moved outside a loop is to add two tests before executing the version in which $x * y$ is moved outside the loop.

The program should:

- Verify that both x and y are of type "integer" as it is not possible to change $x.\_\_mul\_\_$ if x is an integer.
- Verify that there is no "store" operation in $x.\_\_mul\_\_$ and no other function is called from $x.\_\_mul\_\_$.

Many such rules have been identified and implemented. However, the description of these rules is too technical to be included.

Next, we observed that it is needed to first expand the code before applying the above data flow optimizations. This is done by applying two known transformations:

- Loop unrolling [4] is a transformation that unrolls the loop's body to include several iterations $i, i + 1, i+2, \ldots, i+k$ consecutively. Here, we implement a special variant of loop unrolling where indexed references of unrolled iterations are being updated, e.g., instead of $A[i]$ we witness $A[i], A[i + 1], A[i + 2]$.
- Inlining [5] 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. Note that in the above example if $x.\_\_mul\_\_$ is first "fully" inlined, then its "logging" side effects will be revealed.

Again, special care must be taken not to violate correctness and apply these transformations only when it is safe. For example, in the following loop, in every iteration the two functions $f()$ and $g()$ may be swapped. Hence, inlining the function calls to $f()$ or to $g()$ should not be done. Similarly, unrolling is also prohibited since $in\ f(i)$ can return an iterator which does not have a $x.\_\_getitem\_\_(i)$ method which is needed for the unrolling.

```
i = 0
for i in f(i):
  if (i<g(i)):
    f,g = g,f
  else:
    i += 1
```

Note that inlining helps unrolling by substituting explicit values. For instance, assume that $f(0)$ is the list $[1, 2, 3]$. After inlining the above loop, it can be safely unrolled:

```
i = 0
if(1 < g(1)):
    f,g = g,f; else: i += 1
if(2 < g(2)):
    f,g = g,f; else: i += 1
if(3 < g(3)):
    f,g = g,f; else: i += 1
```

This works in the opposite direction as well. Unrolling can help apply safe inlining. For instance, in the following code, we cannot inline $func(t)$ without first unrolling:

```
def func_2(): def func_2(): def func_2():
  t = 123       F1(123)       print 123
  for func in   F2(123)       log(123)
    [F1,F2,F3]: F3(123)       sleep(123)
    func(t)
```

The main contribution of this work is the observation that for Python programs aggressive unrolling and aggressive inlining increase the number of optimization opportunities that may be applied. The experimental results verify this observation. A significant effort was made to enable safe unrolling and inlining. In addition, the combination of data flow optimizations, loop unrolling and inlining has not been considered in previous works and therefore was an area of focus in this study.

2. Related works

Several researchers in the past decade have addressed the obstacle of accelerating Python programs. Optimizations on the bytecode level, the approach taken in this work, is one approach to make Python faster. An early effort [11] to optimize Python's bytecode utilized "Peephole" optimizations i.e., locating small sequences of bytecode instructions and replacing them with an optimized sequence. For example, eliminating redundant "store" and "load" operations of the same variable. Peephole optimizations to basic blocks are addressed by [11]. In this work, however, in addition to addressing basic blocks we also optimize a variety of control flow graphs.

Different optimizations, replacing hot sequences of bytecode instructions with new special bytecode instructions, were implemented by [3].

The code for executing the new instructions were the combined C-codes used by the interpreter to execute the original instructions. A minor effort was made to optimize the C-code of the new instruction. This approach cannot be regarded as a compilation of Python to C, but is in fact another form of Peephole optimization.

Starkiller [14] promotes the idea of using type analysis to statically determine types of variables, objects and functions in Python source code. For example, once the type of function $f(x, y)$ is known, it is possible to safely create a C++ version for $f(x, y)$ and use it instead of the original bytecode of $f(x, y)$. Assuming that all occurrences of $f(x, y)$ in the code agree with the type $int \times int \longrightarrow float$, then a C-code to evaluate $f(x, y)$ can be generated. This approach is based on the claim that usually Python programs do not use dynamic features, and thus, small static types can be associated with most of the functions and variables (i.e., cases like $int \times (float\ or\ int) \longrightarrow (float\ or\ list\_int)$ are rare).

Starkiller does not handle iterators and exceptions and thus its use does not nullify the system proposed in this work. Shed skin [8] also uses the idea of static type analysis to enable compilation from Python to C++. Shed skin promotes "class splitting" in which variables with multiple types are split in to different variables, such that each new variable is assigned a unique simple type. This type of conversion to C++ is more efficient than that of Starkiller. In this way, many of the dynamic types associated with Python variables are resolved. Shed skin uses type analysis [2] to determine the types of each assignment and heuristically backtrack to find optimized splitting points. In spite of the sophisticated analysis, the outcome may be complicated, e.g., $2 * [1]$ is converted to the following C++ code $\_\_mul\_\_(2, new\ list < int > (1))$. One problem with this approach is that variables must be merged at the end of loop iterations. Another problem of Shed skin is that it restricts the use of dynamic features to those supported by Shed skin. Shed skin does not fully implement the Python language.

Psyco [13] focuses on specializing functions at runtime, i.e., generating special and more efficient code for executing a specific case of a given function. For example, a function $def\ f(n, x) :\ return(x*n+x)$ may be "specialized", or adjusted, to the case where both x, n are integers (as opposed to other possible types such as strings or lists) or even to the case when $x = 0$. Once a specialized version is created it can be converted to machine code, e.g., in the case of $f(n, x)$, Psyco may use machine-level arithmetic. Psyco handles loops by converting them to tail recursion. In addition, it frequently switches back to interpretation when specialization ceases to work.

Bytecode optimizations, the approach proposed here, are mainly focused on loop unrolling and inlining. Regarding language differences, C/C++ is comprised of identified types while Python's literals are not type-identified at compile time.

The approach taken in our study is that it is not necessary for most bytecode optimizations to be type-aware. In many cases, weaker properties (bytecode hints) are sufficient to perform a given optimization.

For example, it is always safe to remove the statement $x = x$ (copy propagation) regardless of x's type. Another example is the Peephole-pass, a feature introduced in CPython-2.5, which resolves constant statements such as $4 + 5$ at compile time. Unlike CPython-2.5, our solution performs optimizations not only on constants, but also on local variables and global variables. Importantly, the key benefit of our optimizer is its ability to change the control flow graph and perform the optimizations on multiple basic blocks.

Psyco, Starkiller and Shed Skin must determine the types of all variables in order to compile some of the code. A unified system could be obtained by combining our proposed bytecode optimizations with Psyco, a process which allows the optimizer to apply specialization "after" the bytecode has been optimized.

3. Python Bytecode Overview

In order to understand how the proposed optimization works and how the optimizer bypasses a number of Python's area of weakness, a brief overview of Python's structure is in order.

Python, like Java, uses a stack-based bytecode. The source code is parsed into an AST structure and the bytecode is generated directly from it. 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 python stack holds references to "PyObjects." Python bytecode is interpreted by the CPython interpreter that is implemented in C. The interpreter implementation is a big-switch statement that runs different routines based on the bytecode being read.

The smallest code unit in Python is the 'Function'. Each Function in Python holds its own code section, constants array, temporary/fast storage and stack.

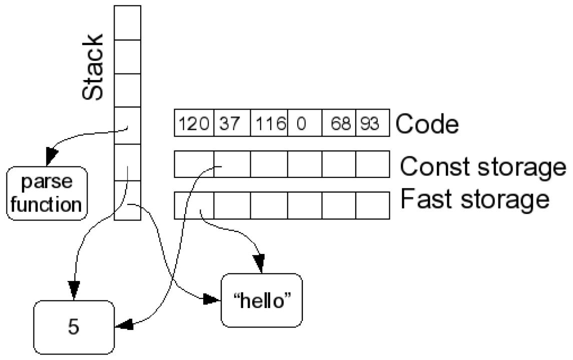


Figure 1. Bytecode architecture for each function call.

When a function is called, a new stack is allocated and is destroyed when the function resumes. When a function is compiled into bytecode a type-function object is created. This object holds an array of constants that the function uses. Another array that is present in the function object is the 'fastlocals' array, which is a register-like temporary storage. Each variable in the scope of the function, i.e each literal, has a cell in this array. The compiler, during compile time, allocates a unique address in the array to each variable declared. During runtime, the store/load opcodes refer to this address. Each item in the array is a Python Object which implements the PyObject interface, a part of the CPython implementation. When the virtual machine encounters the 'CALL_FUNCTION' opcode, it loads the function object from the stack and loads the 'func_code' object which holds the bytecode to be executed, the constant storage array, the fast storage array and the rest of the data needed to execute the function. The Python Virtual machine starts executing the function in a new blank space which is designated for the execution of this specific call. Calling a recursive function creates a new stack and a new set of arrays for each step of the recursion. Figure 1 illustrates the stack immediately before calling the function 'parse' with two arguments.

Unlike Java, Python has no primitive types. Each object must implement its own basic operations. When a 'BINARY_MUL' opcode is encountered, the interpreter executes the "__mul__" function in the 'left' object. This is true for all object operations, including array access, object creations and execution calls. For example, the code in the box below is a method that is being called recursively. When passed with a list, the function will recursively call itself with the 'tail' part of the list, namely with a new copy of the list except for the first item. The function will return when the list's size is one.

```
def x(myList):
  if len(myList) == 1: return myList
  return x(myList[1:len(myList)])
```

The code below is a disassembled version of the code above. The first part is an example of how the global function 'len' is loaded and then called. Later, the return value that was saved on the stack is compared with the constant value '1' (which in an arbitrary manner is saved in the constant array in cell #1). If the result is true, then "myList" is loaded from the fast storage array (cell #0) and is pushed into the stack to be returned. The second part is an example of how "x", a global function that is not a part of a class, is pushed into the stack to be called at the end of the calculations. Later the parameters that are needed for the 'slice' operation are loaded. The slice operation will call the '__getslice__' function of the list object and will create a new list that starts at the second item in the list. Function x is called with the value of the slice operation, which is saved on the stack, and the return value is returned.

```
LOAD_GLOBAL 0 (len)      L1: POP_TOP
LOAD_FAST 0 (myList)     L2: LOAD_GLOBAL 2 (x)
CALL_FUNCTION 1              LOAD_FAST 0 (myList)
LOAD_CONST 1 (1)             LOAD_CONST 1 (1)
COMPARE_OP 2 (==)            LOAD_GLOBAL 0 (len)
JUMP_IF_FALSE 8 (to L1)      LOAD_FAST 0 (myList)
POP_TOP                      CALL_FUNCTION 1
LOAD_FAST 0 (myList)         SLICE+3
RETURN_VALUE                 CALL_FUNCTION 1
JUMP_FORWARD 1 (to L2)       RETURN_VALUE
```

As indicated befiore, some of the proposed optimizations and their utilities are directly related to the bytecode architecture described in figure 1. For example, moving an item from the global storage to the local storage of a function reduces the overhead of searching by performing a direct access. Similarly, inlining saves the relatively large overhead involved with generating the data structures described in figure 1.

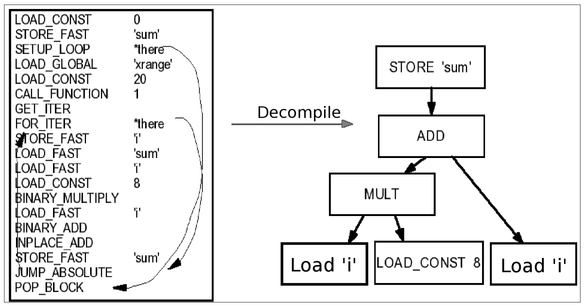
4. Converting Bytecode to Intermediate Representation

Converting the Python bytecode into a useful intermediate representation is a preliminary stage in implementing bytecode optimizations. Like regular compilers, this intermediate representation includes: control-flow graph (CFG), wherein edges correspond to jump operations, and data dependency graph (DDG), wherein edges indicate dependencies between bytecode instructions (see [12] for an elaborate discussion). Some of the difficulties that are unique to bytecode are:

- In the bytecode mechanism everything is allocated on a stack and there is no use of temporary results (registers). Thus, dependencies must be calculated based on stack operations.
- Since the Python bytecode is designed to be interpreted by a virtual machine, it holds relative and absolute jump calls. It also holds indexes to constant arrays. This design makes it difficult to build the CFG.

The first step in converting the bytecode to the DDG structure is to convert the list of integers that represent the bytecode to a higher representation where each bytecode is an object. The relative and absolute jumps are translated into 'labels', such as those you may find in an assembly language, and the constants are translated into object-references to allow for easy manipulation. This new data structure, the 'op-list', can be compiled at any time back to a functional bytecode that can be executed by the machine. The 'op-list' data structure provides the methods to insert into or remove from arbitrary bytecode the sequence of opcodes without needing to adjust offsets in the bytecode. In the 'op-list', each jump call is a reference to a special 'label' opcode. Once the 'op-list' data structure is implemented, it is simple to add, delete and replace any opcode in the list and also modify any parameter of opcode. For example, an opcode that would load a string from the fifth index in the constant array can be modified in such a way so that there is no need to take the constant array into account. Once the bytecode is organized in the 'op-list' structure, it can be extracted into a 'Control Flow Graph' (CFG) structure. This structure is a graph that connects a list of basic blocks. Each basic block is a sequence of bytecode that has no jump opcodes, starts with a 'label' and ends with a jump opcode.

The CFG makes the processing of data in the graph simpler. Calculating dependencies is easier on a basic block where no jump or conditional branches need to be considered. After converting the code to a sequences of 'op-list', the bytecode is arranged in 'Data Dependency Trees' (DDT). Due to the structure of the pro-



```
LOAD_CONST        0
STORE_FAST        'sum'
SETUP_LOOP        *there
LOAD_GLOBAL       'xrange'
LOAD_CONST        20
CALL_FUNCTION     1
GET_ITER
FOR_ITER          *there
STORE_FAST        'i'
LOAD_FAST         'sum'
LOAD_FAST         'i'
LOAD_CONST        8
BINARY_MULTIPLY
LOAD_FAST         'i'
BINARY_ADD
INPLACE_ADD
STORE_FAST        'sum'
JUMP_ABSOLUTE
POP_BLOCK
```

Figure 2. Converting bytecode to CFG and trees like DDGs.

gram, the stack is balanced, and it is possible to organize the opcodes in a tree structure where each opcode is dependent on one or more opcodes. The only exceptions are opcodes residing in the leaves of the tree that explicitly load values onto the stack, such as the 'LOAD_FAST' opcode. Each node in the tree is an opcode and each link is a dependency. This tree (referred as DDT) is very similar to the original 'AST'.

The DDT is created by "bottom-up" process in which the parser begins reading the 'op-list' from the bottom, treating the bottom as the head of a tree, and fills the dependency of each node recursively.

Opcodes that are problematic for this algorithm, such as the 'DUP_TOP' or 'ROT_TOP', that duplicate the top of the stack or modify it, are translated to a sequence of simple 'load' and 'store' opcodes that can represent a small tree. The tree is a de-compiled version of the source code. It is easy to translate this CFG and DDT back into source code via a simple code generation visitor.

5. Loop unrolling and Inlining of Python's byte code

The consideration of the application of loop unrolling [4] to Python's loops is an essential aspect of the optimization. Importantly, loop unrolling duplicates the body of one iteration $k > 1$ times creating a new loop wherein each iteration contains k iterations of the original loop. There are several possibilities regarding whether or not loop indexes are substituted in the unrolled code. Loop unrolling includes substitution of the loop's index and re-use of temporary values. We remark that, in general, one can substitute not only loop indexes but also the induction variables (see [12] IVR) of the loop. Loop unrolling reduces the overall number of iterations and can be used to eliminate some array references. In addition, loop unrolling increases the size of the loop bodies, yielding new opportunities for other optimizations such as constant propagation. For example, in C, unrolling the loop $for(i = 1; i < n; i++)A[i] = A[i-1] + A[i-2]$ three times will yield the following code including index substitutions and using temporary values:

```
for(i=0;i<n-3;i+=3){
   A[i] = A[i-1]+A[i-2];
   A[i+1] = A[i]+A[i-1];
   A[i+2] = A[i+1]+A[i];
}

int tmp1,tmp2,tmp3;
for(i=0;i<n-3;i+=3){
   tmp1 = A[i-1];
   tmp2=A[i] = tmp1+A[i-2];
   tmp3=A[i+1] = tmp2+tmp1;
   A[i+2] = tmp3+tmp2;
}
```

Note that this type of unrolling requires that the loop execution is "predictable," and that if the loop condition is met, then the execution of the next three iterations is guaranteed.

Unlike C, Python loops use iterators to loop over a sequence of items. The iterators mechanism provides a 'next' function that retrieves the values for the next iteration. When no more items are in the sequence, a 'StopIteration' exception is thrown. Thus, the iterator function must be re-evaluated every iteration, making the loop's execution dynamic and "unpredictable". For general iterators, loop unrolling with index substitution cannot be used in Python. Consider for example the loop:

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

Though nothing is known about the literal *bar*, general unrolling is possible as follows:

```
def f(bar):
   sum = 0
   it = bar.__iter__()
   try:
      while(1):
         p = it.next()
         sum += p
         p = it.next()
         sum += p
         p = it.next()
         sum += p
         p = it.next()
         sum += p
   Except StopIteration:
    pass # resume
```

However, direct index substitution of the following form may not be possible unless it is known that sum and $p1, p2, p3, p4$ are not dependent on one another (e.g, affected by some change of the iterator application).

```
def f(bar):
   sum = 0
   it = bar.__iter__()
   try:
      while(1):
         p1 = it.next()
         p2 = it.next()
         p3 = it.next()
         p4 = it.next()
         sum += p1+p2+p3+p4
   Except StopIteration:
     pass # resume
```

While this rule does not hold true for the examples above, there are special, yet common, cases of loops in Python for which unrolling with indexed substitution can be used.

The 'xrange' iterator is an example of an iterator that is predictable, enabling the use of loop unrolling.

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.

Because the iteration range may not be a multiplication of the unroll parameter, a duplication of the original body loop must remain in order to execute the remaining tail.

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

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

Another important case of "predictable" loops in Python is when the object that is being iterated also provides a random access interface. The class 'list' is an example of such object. A 'list' may be accessed via the 'iter' interface or directly through the i'th element using the 'get_element' interface.

Converting the serial access iterator to a random access iterator allows the loop to be treated as a regular 'xrange' loop that can be unrolled. Since unrolling creates many optimization opportunities, the conversion of the iterator to a random access may be beneficial in some cases. In other cases, it may hurt the performance since the random access interface and the added checks may be slower.

In the following example, unrolling is applied in three stages: 1) conversion to random access; 2) unrolling with index substitution; and 3) copy+value propagation.

```
1) for item in lst:
    sum += item * 7

2) len = len(lst)
for i in xrange(len):
    sum += lst[i] * 7

3) len = len(lst)
for i in xrange(0,len-3,3):
   sum += lst[i] * 7
   sum += lst[i+1] * 7
   sum += lst[i+2] * 7

4) len = len(lst)
for i in xrange(0,len-3,3):
   sum += lst[i]*7+ lst[i+1] *7+lst[i+2]*7
```

In the case that the loop range is a small constant, "full" unrolling can be applied, eliminating the loop altogether. For example, the loop first will be replaced by the second.

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

becomes:
```

```
print 1
print 2
print 3
print 4
```

Note that unrolling should work for Python's 'tuple' structure which is a fixed list.

Another known optimization is function inline [6] in which a function call is replaced by a copy of its body and the parameter values are substituted as the initial values of the former parameters. The main disadvantages of an over-aggressive function inline is an increased number of instruction cache misses caused by the increased code size. In Python, code size is not a consideration. Function inline in Python can be beneficial. Like unrolling, it increases the opportunities for other optimizations.

Python function calls are time-consuming in comparison to other compiled languages. Inlining is done by inserting the body of the inlined function into the body of the function, thus eliminating a much slower function call.

Each return call in the original inlined function is translated into a 'store' and 'jump to end' set of opcodes. The 'yield' opcode is not supported in this case and a function with 'yield' will not be inlined.

The parameters that are passed on to the function are manually saved to some local variable. In order to avoid namespace conflicts with the outer function's local variables, they are given a unique random name.

```
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 major_1():
 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
```

Bytecode 'goto' operations are used to simulate multiple returns of the inlined function. We note that only explicit calls to functions are inlined. For example, $map(f, List)$ is not inlined since it may involve many inline invocations which depends on the number of elements in $List$.

Related to inlining is "caching" of global functions. Due to the special structure of the bytecode interpreter (Figure 1), it can be beneficial to cache global functions. Access to global variables requires searching a dictionary, and therefore, is slower than accessing local variables which are looked up by their index values. By "caching" function calls locally, the searching of the global storage is eliminated.

Global functions such as 'int' or 'len' are invoked inside loops and unlikely to be modified by the program. One optimization which is commonly done manually by the programmers is to store a reference to these functions in the local storage area, resulting in a much faster function call inside a loop.

The optimization is unsafe in cases in which these global functions are modified by other threads or modified by the body of the optimized function itself.

The "caching" itself is carried out using local variables as follows:

```
local_convert_data  = convert_data
for i in xrange(100):
    local_convert_data(i)
```

6. Special cases of unrolling and inlining in Python

Due to the dynamic typing of Python, unrolling and inlining may be expressed in unexpected, non-standard forms.

One possible obstacle in implementing these optimizations in the standard way is Python's ability to dynamically change methods during runtime. Consider the replacement of the $'in'$ operator by random access in list-based loops:

```
for item in lst:
  sum += item * 7

becomes:

lst_len = len(lst)
for i in xrange(lst_len):
    sum += lst[i] * 7
```

The previously described methods may fail due to one of the following reasons:

- The variable lst is not of type 'list' making the random access interface fail. The solution here is to add a check at bytecode level $if(lst.\_\_class\_\_ == list)$: and execute the unrolled version only if lst is of type 'list'. Similar checks can be used to verify that lst provides 'len' and random access ($[i]$) as expected.
- The 'xrange' method has been deleted or modified. In this case the optimizer can save the 'xrange' method during load time as $\_\_safe\_xrange$ and use it instead of the original 'xrange'.
- The lst is dynamically changed (appending/removing of elements) during the execution. Clearly (as is depicted by the following example), in such case, loops should not be unrolled. The optimizer may need to check that the lst is not modified.

```
lst = [1,2,3]
for i in lst:
  if(A[i] > 10):
    lst.append(i)

lst_len = len(lst)
for i in xrange(0,lst_len,3):
  if(A[i] > 10):   lst.append(i)
  if(A[i+1] > 10): lst.append(i+1)
  if(A[i+2] > 10): lst.append(i+2)
```

In some cases unrolling may be applied despite the modification of $'lst'$. This form of unrolling is not supported in the current version of the optimizer. An example of such an optimization:

```
for i in L:
  if(m()):
    L.remove(0)
    print L[i]

try:
 for i in L:
  if (m()): L.remove(0)
       print L[i]
  if (m()): L.remove(0)
       print L[i+1]
except Exception, e:
  pass
```

Solving this problem by using the built-in Python exceptions mechanism may work for some methods. However, throwing exceptions to catch the termination of the loop (for example when $LIST[i + 1]$ does not exist) will fail in cases where this exception is also thrown by the loop's body.

Similar problems exist with $'xrange'$ based loops, e.g., the built-in $'xrange'$ method may be modified. The modification of the loop's index is a particularly interesting case. Unrolling with substitution of the indices $i + 1, i + 2$ yields a loop that is not equivalent to the original loop. However, this is solvable if we use the load increase and store operation ($i+ = 1$) instead of direct index substitution. Consider the following example:

```
for i in xrange(90):
    if (A[i] > 10):
       i+=1
       print A[i]

for i in xrange(90):
    if A[i] > 10: i+=1
    print A[i]
    if A[i+1] > 10: i+=1
    print A[i+1]
    if A[i+2] > 10: i+=1
    print A[i+2]

for i in xrange(0,90,3):
    if A[i] > 10: i+=1
    print A[i]
    i+=1
    if A[i] > 10: i+=1
    print A[i]
    i+=1
    if A[i] > 10: i+=1
    print A[i]
```

The reason the original method fails is that the $'xrange'$ iterator is not affected by the modification of i inside the loop's body. The current system does not apply unrolling in cases where the index of any other parameter of $'xrange'$ can be changed by the loop's body.

We remark that loops with $'break'$ and $'return'$ statements can be safely unrolled. However, loops with $'continue'$ must be unrolled using explicit *goto* operation:

```
for i in xrange(n):
  print i
  if d == i:
    continue

for i in xrange(0,n,3):
    print i
    if d == i:
     bytecode_goto L1:
L1:
    print i+1
    if d == i+1:
      bytecode_goto L2:
L2:
    print i+2
    if d == i+2: continue
```

In addition, loops with $'yield'$ instruction can be safely unrolled. The yield instruction [1] is a special return statement of Python that returns a function generator. Since the yield instruction returns to the body of the called function at the instruction proceeding the yield instruction, the unrolling preserves the regular order of execution and therefore is safe to unroll.

Like unrolling, inlining has several problematic cases:

- Functions containing '$yield'$ cannot be inlined because they are designed to be called several times and continue to execute, each time, from the last location at which they were stopped. Additionally, a function that has a yield instruction may be called from more than one function.
- Global functions which are modified in runtime may not be inlined since each time a different function is called and therefore the optimizer cannot inline more than one function per location.
- Built-in functions, which are implemented in C may not be inlined.

7. Unsafe Optimizations

In addition to the safe optimizations, unsafe optimizations may be very beneficial. These optimizations, however, are safe when placing certain assumptions the code.

A good example of why it is so difficult to optimize inter-procedural python code is the ”logging side-effect”. Imagine that a method in an instance of a class is replaced with a wrapper method which logs the call. In this case, there is no way to know ahead of time if it is safe to perform the optimization since the method was replaced dynamically after the object was created. Notice that in the algebraic simplifying pass interchanging addition and multiplication is very common. If the correctness of the code depends on the ”logging side-effect” then any inter-procedural optimization in python which does not implement runtime checking is invalid.

One of the implemented unsafe optimizations is the “general iterator unrolling”. In this optimization, we may unroll any iterator of any type. Even iterators that resume at random times and return random values may be unrolled. The body of the unrolled loop is assembled to provide opportunities for other optimization passes.

We first extract from the iterator the next k values and substitute them into the bodies of the unrolled loop. We record how many values of the iterator we are able to retrieve before we have reached the end of the iterator. In case that the number of items that are retrieved from the iterator is not a multiple of k, the exception handler will execute the remaining iterations.

In the example below, the content of an iterator is written into the object of unknown type A. This unroll operation is unsafe since A might implement a method that will write to a global variable while the iterator of the variable, 'storage', might implement a 'next' method that reads from the same global variable. Changing the order of execution may impair the correctness of the code since the order of the writing and the reading of the global variable may change.

```
def func(storage):
  OLD = None
  for i in storage:
      A[i] = OLD; OLD = i; t+=1
```

Unsafe unrolling will produce the following code:

```
def f(storage):
```

```
OLD = None
it = storage.__iter__()
while(1):
  try:
    counter = 0
    i0 = it.next();  counter +=1
    i1 = it.next();  counter +=1
    i2 = it.next();  counter +=1
    i3 = it.next();  counter +=1
    A[i0] = OLD; OLD = i0
    A[i1] = OLD; OLD = i1
    A[i2] = OLD; OLD = i2
    A[i3] = OLD; OLD = i3
  except StopIteration:
    if counter < 1: pass
    if counter < 2: A[i0] = OLD; OLD = i0
    if counter < 3: A[i1] = OLD; OLD = i1
    if counter < 4: A[i2] = OLD; OLD = i2
    break
```

The data-flow optimizations will propagate $i0, i1, i2, i3$, eliminating redundant load/store operations, and thus optimizing the above loop as follows:

```
After:
A[i0] = OLD;
A[i1] = i0
A[i2] = i1;
A[i3] = i2; OLD = i3
```

The user must explicitly indicate that it is safe to apply unsafe-unrolling to a given loop using Python's decorator syntax, e.g.,

```
@NoGlobalCoherencyOptimize
def func(x):
 ...
```

8. Dataflow Optimizations

Once the bytecode has been unrolled and inlined, it contains opportunities for further optimizations. Due to the fact that the bytecode interpreter is implemented in software, unlike machine code, scheduling and resource allocation optimizations are not beneficial in the same way they would be for assembly code. Thus, the second phase, after unrolling and inlining, focuses on simplifying the bytecode in a high level mode using well-known optimizations and new optimizations.

The following example illustrates the set of optimizations implemented at this stage (assuming that a is a list):

```
def derive(a, b,  rounds, co):
    x = 0
    for i in xrange(rounds):
     y = x + b*co
     x = y + 4*(i+2)*(i+2) + (a[b])*i
    return x
```

Value propagation is an optimization pass which replaces a variable's occurrences with the expressions that compute them. In the above example, $y = x + b * co$ can be eliminated by substituting it with $x = y + 4 * (i + 2) * (i + 2) + (a[b]) * i$ yielding $x+ = b*co+4*(i+2)*(i+2)+(a[b])*i$. This optimization can save the unnecessary load/store operations of $x+ =$. Note that after substitution of $x + b * co$ instead of y in $x = y+4*(i+2)*(i+2)+(a[b])*i$ the assignment $y = x+b*co$ is not used. It becomes "dead code" and is eliminated by a dead code elimination pass, which is a sub-pass executed at this stage. Dead code elimination [10] will also eliminate Write-after-Write assignments such as $g = 3g = 6$.

Constant propagation is a well known optimization in which constant variables and values are computed and propagated. Here, in the call to $derive(\ldots)$ if $co == 0$ then $b * c$ is equal to zero, then the expression would become $x+ = b * 0 + 4 * (i + 2) * (i + 2) + (a[b]) * i$. Other algebraic optimizations such as multiplication by 1 are also simplified. In Python, operations like $List * 0, List * 1, string + string, string*string, \ldots$ are allowed during constant propagation and algebraic simplifications, e.g., $len(List * 0)$ is zero.

Common sub-expression elimination is another well-known optimization wherein multiple occurrences of common sub-expressions are replaced by temporary values. For example, $(i + 2)$ occurs twice and hence can be computed to a temporary variable, $u = i + 2$, yielding $x+ = 4 * u * u + t * i$.

Loop invariants are expressions whose variables are not modified inside the loop and can be safely computed to a temporary variable once, outside the loop. In this example, $a[b]$ is a loop invariant whose value $t = a[b]$ can be computed outside the loop once.

Strength reduction- is the last stage wherein "heavier" operations such as multiplications are replaced by lighter operations such as addition. Here, instead of $t * i$, we can use a temporary value $x+ = 4 * u * u + v$ such that $v+ = t$. Thus, the final code is:

```
def derive(a, b,  rounds):
  x = 0
  t = (a[b])
  v = 0
  for i in xrange(rounds):
   u = (i+2)
   x += 4*u*u + v
   v += t
  return x
```

We note that algebraic simplifications that are based on the distributive law of commutative operations may not work for all types of variables. For example, $a * b + a * c \ \ => \ a * (b + c)$ is valid for integers but invalid for strings wherein the multiplication operation is not commutative:

```
b = "aa"
c = "bb"
a * (b+c) ->  'aabbaabb'
a*b + a*c ->  'aaaabbbb'
```

The above optimizations can be used not only for numerical types but also for lists and strings. For instance, $len(List * 7)$ can be replaced by $len(List) * 7$ since $List * 7$ concatenates seven copies of the variable $List$ to form one large list.

The following example illustrates the combined effect of the two optimization stages for Python's strings.

```
def string_builder(list_of_strings):
  sb = ""
  for s in list_of_strings:
    sb += s
  return sb
```

Unrolling twice, using direct access conversion, yields:

```
def string_builder(list_of_strings):
  sb = ""
  for i in xrange(0,len(list_of_strings),2):
    s = list_of_strings[i]
    sb += s
    s = list_of_strings[i+1]
    sb += s
  return sb
```

Value propagation and dead code elimination will produce a dense form of code:

```
def string_builder(list_of_strings):
  sb = ""
  for i in xrange(0,len(list_of_strings),2):
    sb += list_of_strings[i]
        +  list_of_strings[i+1]
  return sb
```

Finally, the optimizer can use built-in functions that can combine the two string concatenations into one operation:

```
def string_builder(list_of_strings):
  sb = ""
  for i in xrange(0,len(list_of_strings),2):
    sb +=  string.join(list_of_strings[i:i+1])
  return sb
```

9. Experimental Results

The benefits of the proposed optimizations and the necessity of unrolling and inlining were tested using several benchmarks: Pystone, Crypto-1.2.5 - Rijndael test, Pypy MD5, Pypy SHA, Pybench and several micro tests.

Pystone is the 'de facto' Python benchmark. It is distributed with every Python installation and is considered the main benchmark for Python virtual machine implementations. This benchmark contains a big loop which calls several simpler functions. It involves accessing global variables, array references and numeric operations. Here, inlining of the smaller functions improved the results by a large margin. Unrolling was applied as well but it did not dramatically improve the performance. In terms of algebraic simplifications and other passes, there were some minor changes to the bytecode.

Crypto-1.2.5 Rijndael test is a pure Python implementation of cryptographic algorithms. The algorithm that was benchmarked was Rijndael, the base for the AES encryption algorithm. In this algorithm, like most hash algorithms, most of the execution time is spent on xor-ing and shifting of bits and constant values which depend on the input key. The code was implemented as a loop which called the "SubBytes", "ShiftRows", "MixColumns" and "AddRoundKey" methods. These methods operate on small lists of numbers and were unrolled and then inlined into the main function. Additionally, the outside loop was unrolled. The main benefit of this optimization is the removal of unnecessary function call overhead and removal of unnecessary loops for arrays of known size. Additional optimization passes were able to remove unneeded load/store operations that were created when the arguments for the inlined functions were fused.

Pypy (MD5 and SHA) is a complete Python virtual machine implementation including parts of the Python library which is written in Python itself. The functions that were chosen from Pypy were the two cryptographic functions which are present in the standard CPython distribution. They were chosen because they were the most obvious computationally-intense functions in the Pypy library. Very much like the Rijndael test, both MD5 and SHA were implemented as a main loop which called sub-functions to shift and xor the data. Both functions benefited from inlining of basic cryptographic functions such as 'rotate'. In the case of MD5, some performance was gained from load/store elimination and constant propagation. In the case of SHA, complete loop unrolling helped in reducing the overhead of setting up the loops.

Pybench is a list of tests that provides a way to benchmark the performance of the Python implementations. It has a large number of duplicated code sections. Because this code can be eliminated, the dead code elimination pass, which removed all of the dead code, boosted the performance by a large factor. This is due to the sole fact that the dead code was removed. Write-after-write pass was also able to remove a large number of expressions. Another pass that showed great performances gain was the algebraic simplifications pass. Some of the tests that were executed included real dependencies in which, for example, an object was appended to a list. In these benchmarks, the optimizer showed little or no boost in performance.

Microkernels is a set of self-made programs designed to test a specific combination of optimizations. Most of the test contained loops that were unrolled. Once the loops were unrolled, the optimizer found opportunities to remove Write-after-write expressions, to eliminate load/store expressions and to propagate constants.

The times of the programs were measured in seconds. The results presented in figures 3, 4, 5 show 10-30 percent improvements for most programs, with high peaks of improvement in certain programs (e.g., 300 times improvements in Pybench). These peaks are mainly due to large segments of dead code that were eliminated. The benchmark was designed to measure the performance of the interperter and in some cases some of the code was "dead". For example the expression $5 + 3$ has no store target and can be remove. It was easy for the optimizer to eliminate such code. The necessity of unrolling and inlining has been verified experimentally as little improvement was obtained when unrolling and inlining were disabled.

Name	Before	After	Gain
Arith.SimpleIntegerArith	3.58	0.5	6.18
Arith.SimpleFloatArith	3.99	0.59	5.74
Arith.SimpleIntFloatArith	3.89	0.86	3.51
Arith.SimpleLongArith	1.61	0.22	6.29
Arith.SimpleComplexArith	2.3	0.27	7.56
Strings.ConcatStrings	4.02	0.01	335.91
Strings.CompareStrings	4.14	0.04	111.6
Strings.CompareInternedStr	2.96	0.04	82.23
Strings.CreateStringsWithCon	1.71	0.02	75.67
Numbers.CompareIntegers	4.51	0.02	223.03
Numbers.CompareFloats	3.46	0.01	334.72
Numbers.CompareFloatsInt	3.96	0.01	377.85
Numbers.CompareLongs	2.91	0.01	264.32
Lists.SimpleListManipul	2.16	1.81	0.2
Lists.SmallLists	2.96	2.56	0.16
Lookups.SpecialClassAttr	3.66	2.08	0.76
Lookups.NormalClassAttr	3.71	2.11	0.76
Lookups.SpecialInstanceAttr	5.29	2.95	0.79
Lookups.NormalInstanceAttr	3.56	2.06	0.73

Figure 3. Pybench results.

10. Conclusions

We have studied the potential of using a complete set of data-flow optimizations proceeded by loop unrolling and inlining to optimize Python's bytecode. In Python, the type of variables or functions is dynamically changed. Thus, there is a need to adapt these known compiler optimizations to the fact that variables and functions may change their type dynamically. In this work, we have used simple filtering rules to identify safe cases wherein the proposed optimizations can be implemented in light of possible dynamic changes. Through the use of this technique, we prevent the need

Name	Before	After	Gain
Total (Sum)	38.59	9.58	3.03
BasicLoop0	0.04	0.04	0.11
BasicLoop1	0.08	0.14	-0.45
BasicLoop1	1.03	0.57	0.82
BasicLoop2	0.05	0.02	0.99
LoopArith0	1.48	0.21	5.98
LoopArith1	0.74	0.46	0.61
LoopArithList0	1.37	0.16	7.74
LoopArithBranch0	1.89	1.03	0.84
LoopArithForEach0	5.43	0.07	77.07
LoopCall0	3.54	3.13	0.13
LoopExp0	3.12	0.95	2.29
LoopExp1	4.89	0.52	8.32
LoopArithInduction0	12.83	0.66	18.4
micro.LoopList0	0.62	0.5	0.25
micro.LoopRoll0	0.93	1.12	-0.17
micro.ListRoll0	0.42	0.0	37450
micro.LoopRollLstR0	0.13	0	72.71

Figure 4. Micro kernels results.

Name	Before	After	Gain
Pystone	2.55	2.09	0.22
Pypy MD5	3.24	2.61	0.24
RSA MD5	8.73	6.95	0.25
Pypy SHA	25.02	20.83	0.20
Crpt.Rijnd	144.44	109.94	0.31

Figure 5. Pystone, MD5, SHA and Crypto-1.2.5 Rijndael test results.

to implement static-type analysis. A special variant of loop unrolling with array index substitution was implemented. This variant creates more opportunities for the data flow optimizations executed after unrolling. In particular, after this unrolling, redundant array references may be optimized by fast immediate variables. The experimental results show that this set of optimizations can significantly improve the performances of Python programs. We plan to release the research code in hope that it will benefit the python community.

References

[1] *The Yield instruction*. http://www.python.org/dev/peps/pep-0255/.

[2] Ole Agesen. *Concrete type inference: delivering object-oriented applications*. PhD thesis, Stanford, CA, USA, 1996.

[3] John Aycock. Converting python virtual machine code to c. In *8th International Python Conference*, 1999.

[4] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[5] Henry G. Baker. Inlining semantics for subroutines which are recursive. *SIGPLAN Notices*, 27(12):39–46, 1992.

[6] P. P. Chang and W.-W. Hwu. Inline function expansion for compiling C programs. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 246–257, 1989.

[7] Wesley J. Chun. Keeping up with python: the 2.2 release. *Linux J.*, 2002(99):5, 2002.

[8] M. Dufour. Shed skin: An optimizing python-to-c++ compiler. In *Master Thesis, Delft University of Technology*, 2006.

[9] Gary A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, New York, NY, USA, 1973. ACM Press.

[10] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. Partial dead code elimination. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 147–158, 1994.

[11] S. Montanaro. A peephole optimizer for python. In *7th International Python Conference*, 1998.

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

[13] Armin Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26, 2004.

[14] M. Salib. Starkiller. A static type inferencer and compiler for python. In *In Proceedings of the EuroPython conference*, 2004.