# Virtual Method Tables in Python

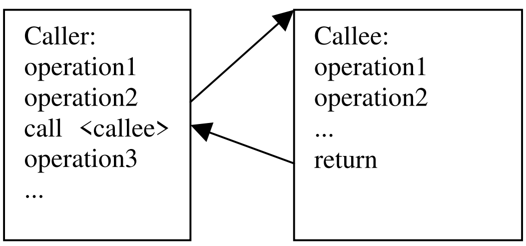
Martin von Löwis
*Humboldt-Universität zu Berlin*

## 1. Abstract

Virtual tables are a mechanism to find methods of a class efficiently. Typically, they are used for statically-typed languages. This paper discusses an implementation of this mechanism for Python. Different design choices are analyzed, and performance measurements are presented.

## 2. Introduction

In an object-oriented language, invocation of methods on objects is a frequent operation. In traditional procedural languages, it is usually possible to determine the procedure being used without looking at the parameters. Therefore, *early binding* of the procedures is used: the code of the calling procedure contains a direct reference to the code of the called procedure (see the figure below).




In Python, a different implementation of method invocations is necessary. Let's consider the program

```
class Foo:
 def f(self):
  print 42
 def g(self):
  print -42
class Bar(Foo):
 def f(self):
  print "Hello, World!"
foo = Bar()
foo.f()
```

In the example, the statement `foo.f()`is not enough to determine whether `Foo.f` or `Bar.f` will be called. Instead, the method to invoke can be determined only when the value of `foo` is known (*late binding*).

In the different languages that employ late binding, a variety of approaches is used. In statically-typed languages, virtual method tables often result in good execution speed. As explained below, this approach fails for more dynamic languages. In Python, the lookup procedure searches in various places to find the method. Both techniques are explained in the next sections.

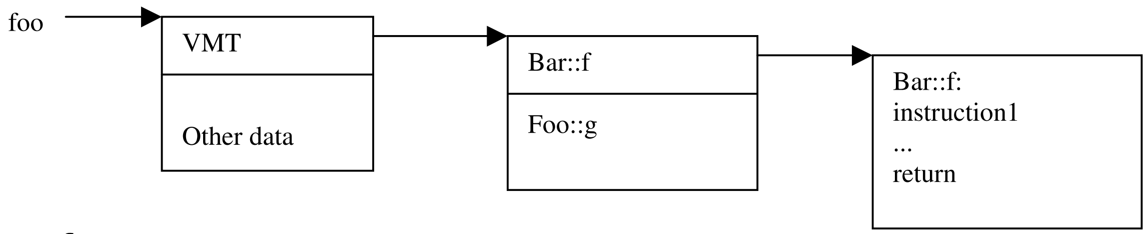
## 3. Virtual Method Tables

In implementations of the C++ and Java languages, tables of virtual methods are used during method invocation [1]. The Python example from the introduction would read in C++ as

```
class Foo{
 virtual void f(){
  cout<<42<<endl;
 }
 virtual void g(){
   cout<<-42<<endl;
       }
};
class Bar: Foo{
 void f(){
  cout<<"Hello, world!"<<endl;
 }
};

main(){
 Foo *foo = new Bar;
 foo->f();
}
```

It should be noted that the `class Bar` needs to inherit from `class Foo`; otherwise, a polymorphic call to `f` is not supported in C++. As it turns out, this is an important restriction which allows to use method tables efficiently.

When performing the call to `f` in `main`, the compiler sees that the object has a static type of `Foo`, and that it therefore implements the method `f`. At run-time, the object will have a lay-out as shown in the figure below.




The pointer `foo` references the actual object, which is a `Bar` instance. Each object has the pointer to the virtual method table as its first field. The method table for `class Foo`, and all derived classes, contains a pointer to the method `f()`. Therefore, the call to f requires three indirections:

- using the address of the object, retrieve the VMT at offset 0,
- using the VMT and the offset of the method (0), retrieve the address of the function, and
- call the function.

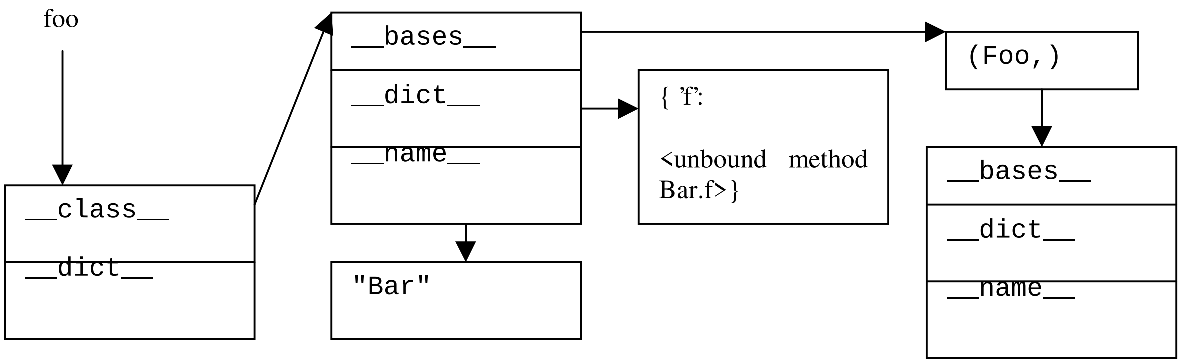
Each method of `class Foo` has its own index in the virtual method table. When derived classes redefine a method, they replace the slot in the method table at the index of this method. Therefore, each class needs its own copy of the method table. Since all instances of a class bind to the same set of methods, the instances can share the method table. Instances then reference the method table of their class, using a single pointer.

If a derived class does not redefine a method from a base class, the pointer to the base class method is put into the table (see `Foo::g` in the example).

## 4. Attribute Lookup in Python

Finding methods in Python is different. Like in C++, methods are defined in the class. Unlike C++, late binding is used not only for methods, but for all class attributes. For each instance, there is a set of per-instance attributes, whose values will be different in a different instance of the same class. There is also a set of class attributes (methods and data) which is shared among all instances.

The layout of the object `foo` from the introduction is shown in the figure below.




The object itself references the dictionary of instance attributes, and the class it belongs to (in the example, `class Bar`). The class references the tuple of base classes, the class dictionary, and the name of the class.

Based on this layout, attribute lookup for instances uses the following steps:

1. Given the name of the attribute, and a reference to the instance, look into the instance dictionary. If the attribute is found, we're done.
2. Otherwise, go to the class, and look for the attribute name in the class. If the attribute is

found, and it is an instance method, build a bound method.
3. Otherwise, search in the base classes, repeating steps 2 and 3 until the attribute is found.
4. If it is not found, raise the `AttributeError` exception.

This algorithm implements the full Python semantics.[1] In particular, it is not necessary to declare in advance whether the attribute is a method or a data attribute, or whether it is per-instance, or defined in the class.

Unfortunately, this algorithm is less efficient than virtual tables would be:

- As the first step, it performs a dictionary lookup in the instance, even though methods are typically found in the class. Assuming that dictionary lookup is constant[2], this gives a constant overhead.

- If that fails, it searches the base classes. There is one dictionary lookup necessary per base-class. If the attribute is not present at all, all base classes have to be searched unsuccessfully.

The base class lookup part is what makes instance attribute lookup consume non-constant time, and the one that virtual method tables will improve. During the experimentation, it turned out that the instance dictionary lookup is also significant.

One might assume that a failed attribute lookup is a rare case (it would give an `AttributeError`, and would be a bug in the application). There are some features in Python that make failed attribute lookup a frequent case. For example, when converting objects to strings, Python will first try `__str__`. If that fails, `__repr__` will be tried. If an object is used in a Boolean expression, Python will try `__nonzero__` and `__len__` before it decides that the object counts as true. In the typical case that none of these are defined, Python will traverse all base classes twice.

[1] The invocation of `__getattr__` if present is viewed as part of step 4, here.

[2] This assumption holds in the typical case. With cached hash values, and string interning, dictionary lookup does not significantly slow down as the number of dictionary entries increases.

# 5. Changing the Interpreter

This section discusses the solution that I took after certain experimentation. Some of the alternatives that I've explored are presented later.

Clearly, virtual method tables offer a solution to avoid the non-constant complexity of the class attribute lookup. When searching for a class attribute, a lookup in the table will tell whether the attribute is there, without requiring to check base classes. The problem is the method table index: At run-time, the interpreter has only the attribute name. It now needs to correlate this name with a number used as index in the method table. In C++, this relationship is defined at compile time. By looking at the classes, it is very easy to give numbers to each virtual method. Derived classes just copy the number assignments of the base class, and extend it with their additional virtual methods.

This assignment strategy breaks if there is no distinct root class for a given method invocation. At ECOOP'97, Onodera and Nakamura presented an approach to implement virtual method tables in Smalltalk [2]. With regard to polymorphism, Smalltalk is similar to Python: Two classes can implement the same method, even if their common base class does not define this function. Since there is no static type for a call, the index in the method table must be the same for *all* classes.

In turn, the first problem is to assign unique numbers to method names. At run-time, the interpreter has the name and needs the number, so this translation is better fast. Fortunately, Python 1.5 introduced *interned strings*, which come quite handy.

## 5.1 Interned Strings

All variable names used in Python program are strings, their associated values are stored in various dictionaries. There is a dictionary for local and one for global variables, every module, every class, and every instance has a dictionary. To make dictionary lookup in Python fast, different techniques are used:

- A dictionary is a hash table. When the hash value of the key is known, it is very easy to determine the place where the value should be. If there is no value for the hash, there is no value for the key.

- To avoid recomputation of the string hash over and over again, the hash value for a string is stored inside the string object.

With these two optimizations, lookup is very efficient if the string is not in the dictionary. If the string is in the dictionary, it is necessary to compare the key being searched with the key in the dictionary, because they might be different strings that just have the same hash.

Comparing strings is expensive, unless the strings are identical. Let's compare the statements

```
foo = "Hello"+" World"
bar = "Hello World"
if foo == bar: print "yes"
```

and

```
foo = bar = "Hello World"
if foo == bar: print "yes"
```

In the second case, `foo` and `bar` reference the very same string object. It is clear that they are also equal. In the first case, the interpreter must compare them byte-by-byte to determine equality.

Since Python 1.5, the interpreter maintains a global dictionary of "interesting" strings, the interned strings. During interning of a string, the interpreter determines whether an equal string was interned earlier, and uses the interned version of the string from then on. When a string object is encountered the first time, a dictionary lookup is performed. After that, the interned string is referenced in a field of the original string, so that a later interning of the same string object does not need to go through the interning dictionary again.

Since interning is an expensive operation, it is not performed for all strings. Instead, a string is interned:

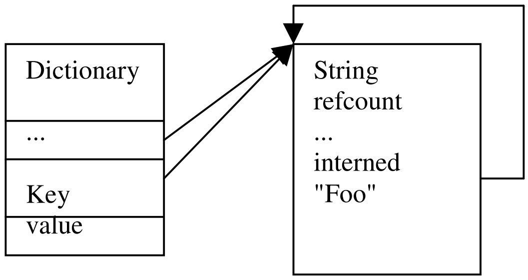
1. when it lexically appears as variable or attribute name in source code,

2. when it is used as a string in source code, but looks like an identifier (i.e. no spaces, ...), or

3. when it is used in an attribute lookup, and was not interned before.

Typically, the first condition would cover most applications. The second condition covers `getattr/setattr` calls with literal strings, and the third one covers computed attribute names. If some application explicitly modifies the `__dict__` attribute of a class, no interning is performed.

With the interning dictionary, it is possible to associate numbers with some strings, but not with others. In Python 1.5, each string has a pointer to its interned variant, if there is one. For the interned strings themselves, this gives the links shown in the figure below.




The interned string appears both as key and as value in the interned dictionary, and it references itself in its `interned` field.

In extension of this structure, I can associate another object with the interned string, as shown at the bottom.

I have introduced another type which holds additional attributes of an interned string. Not all interned strings need to have this attribute, only those used for instance attributes. During interning, the interpreter passes a flag whether this additional object is necessary. Currently, it does so in the cases 1 and 3, but not if it encounters string literals. As a result, the interned field of a string is polymorphic, and might reference either a string or the interned attributes.

## 5.2 The Size of the Tables

Since the method table index is globally associated with a method name, all classes have the same method table layout. That means that the method

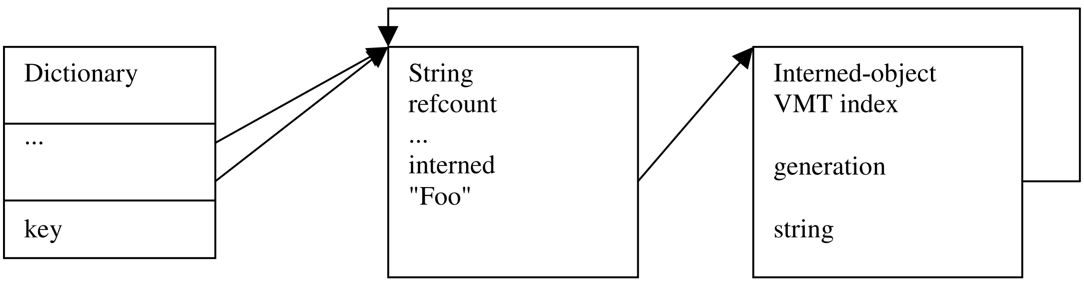


table contains the union of all method names found in the program. Since there is one such table per class, that might result in quite some memory consumption.

In order to reduce memory consumption, only a subset of the methods names relates to method table entries, as proposed in [2]. Currently, I give 64 entries to each table, and assign indices as I encounter them. Sooner or later, the table will fill up. At that time, I clear the tables, and start all over. To make the clearing of the table efficient, I keep a generation counter. Every time I clear the table, I only increment the generation counter. Each interned string not only stores the its method table index, but also the generation of the index. If the index is not from the current generation, it is just as if it didn't have a method table index at all.

The resulting layout for the example from the introduction is shown at the bottom.

### 5.3 Failing Instance Attribute Lookups

During the experiments, it became clear that the lookups for instance attributes had a considerable impact on the overall performance. For each method call, an instance lookup is performed before the class is inspected. For many frequently-called methods, it is obvious that this lookup will fail: Nobody assigns instance attributes called `__init__` or `__str__`.

This opens another optimization opportunity. If it is known that no instance in the program has an attribute of a given name, an attribute lookup will fail. Therefore, I record assignments to instance attributes in the extra attributes of an interned string.

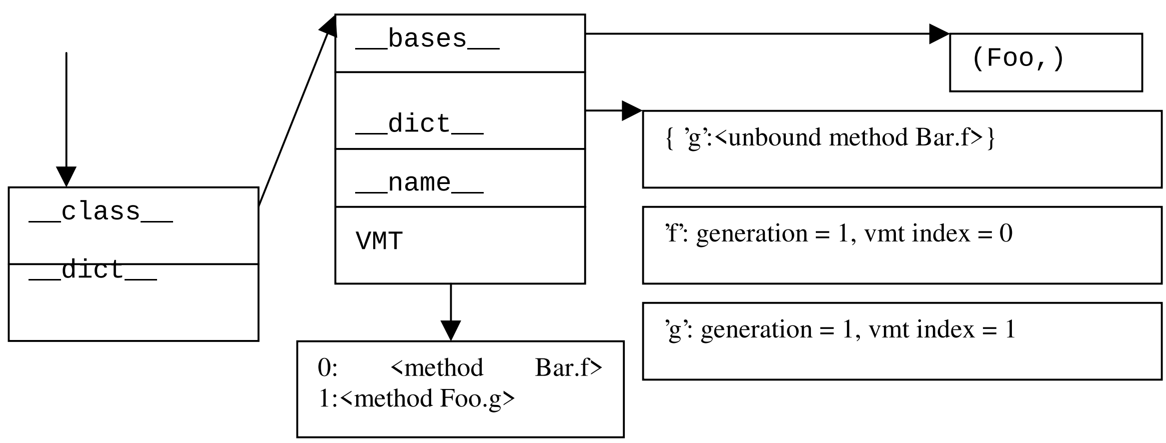
With this optimization, I get the following pseudo-code for the instance attribute lookup:

```
def is_instance_assigned(string):
    return !is_interned(string) or\
    string.instance_assigned

def class_lookup(cl, string):
 #search __dict__,__bases__

def instance_lookup(i, string):
 if is_instance_assigned(string):
  if i.__dict__.has_key(string):
   return i.__dict__[string]
 if is_interned(string):
  #If interned, check the VMT
  if !string.vmti or\
string.generation!=cur_generation:
   string.vmti = new_vmt_index()
  if !i.__class__.vmt:
   i.__class__.vmt = new_vmt()
  if i.__class__.vmt[string.vmti]:
   return
i.__class__.vmt[string.vmti]
 result = \
class_lookup(i.__class__, string)
 if is_interned(string):
  #Cache the result
  i.__class__.vmt[string.vmti] =\
result
 return result
```

### 5.4 Fighting Degenerate Cases

With the scheme presented above, a number of things can go wrong. When I started to modify Python, I had three design goals:

- The semantics of the language should not change,



no matter how misimplemented the program is.

- For average code using classes, the performance should be better than it is now.

- Performance should not degenerate in the worst case.

First, correctness of such a caching approach is an important issue. The problem is that class attributes are retrieved from the cache. This includes attributes from the base class. In Python, it is possible (and even common) to change class attributes, expecting that they change immediately for all instances, and that the old value becomes deallocated if it is not referenced elsewhere.

Fortunately, the number of class attributes that change over time is small in a typical program, and a caching scheme should still provide performance gains if those are not cached. In order to determine which attributes change, I assign an out-of-range method index to attributes that ever changed (no matter what class the change occurred in). In other classes (including derived classes), the attribute is then not cached anymore. The tricky part is to discard the cache if the value is ever changed. There are three ways to introduce a class attribute in Python:

```
class Foo:
 def f(self):
  pass
Foo.g = 42
Foo.__dict__["h"] = Foo
```

During class definition, assigning an attribute does not harm the cache as there is no cache yet; these are the attributes that typically should be cached later. It is considerably easy to trap class attribute modification in the second case. Trapping the third case is difficult, as the dictionary implementation doesn't know anything about classes. To trap dictionary modification, I introduced a call-back function in the dictionary object.

Whenever a class attribute is changed, the call-back function checks whether the attribute name has a method index assigned. If it does, the cached value needs to be deleted from all virtual method tables.

The same technique is used for instance attributes. Whenever an instance attribute is assigned (directly or through `__dict__`), I record this fact in the interned string. If an uninterned string was used to modify the instance, I cannot record this assignment to that particular string. Instead, I give up and have `is_instance_assigned` return always true.

From a performance viewpoint, the worst case occurs when each VMT lookup fails. In this case, the algorithm will check whether the attribute generation is correct, and find out that it isn't. In turn, it will assign a new index. The class won't have a VMT entry for that index, and needs to perform the regular lookup. This gives a constant overhead of a few function calls per attribute lookup.

Every time the VMT fills up (in worst case, every $64^{th}$ lookup), the generation counter is increased. In turn, all classes will clear their method tables the next time they are accessed. In the worst case, this are 64 classes per generation. During the next generation, it might be necessary to clear a virtual table at each access. This could add to some measurable overhead. In order to avoid such thrashing, the algorithm could turn off virtual tables for classes that suffer from it. At the moment, no such algorithm is implemented. An application that exercises this worst case would need to use a large number of different method attribute names, and apply them to instances of a large number of different classes.

## 6. Alternative Approaches

During the experiments, I explored a number of alternatives. This section presents some of the failures, and some options that still need to be explored.

### 6.1 Storing VMT indices

Initially, I planned not to modify the string type, and store the method indices in the code objects. Each code object has a tuple of attribute names (`co_names`) used in the code object. Parallel this tuple, I introduced a vector `co_vmt`, which should record the method table index. As with in the final implementation, the index was inserted lazily, when the code was executed for the first time. Each value in `co_vmt` was a tuple of the vmt index and the generation counter.

This implementation did not show the desired speed-up. Primarily, the reason was probably that I used generic Python types (tuples and vectors). The access functions for those data structures would produce overhead for reference counting and extra function calls.

In addition, this approach stores the VMT index in each method where the interned string is used, requiring a dictionary lookup the first time the method is executed.

## 6.2 Cache Overflow

Another issue is the selection of attribute names that have method indices assigned. Ideally, only those attribute names that are frequently used should be associated with numbers, since the size of the virtual tables is small (compared to the total number of class attributes). The algorithm assigns a method index to the name the first time it is used, which guarantees that names never used in a program will not fill the cache at all. On the other hand, the algorithm will run out of numbers eventually. There are several strategies that could be used in that case:

- Stop assigning numbers, and continue to use the cache.
- Clear the cache, and start all over.
- Try to group the attribute names into different clusters, guaranteeing uniqueness of numbers only within a group.

Different inheritance hierarchies usually use different sets of method names, so grouping the names along these hierarchies seems like a good idea. Unfortunately, some method names in Python are identical across all such hierarchies, and some of them will be even frequently used (such as `__init__` or `__str__`). This rules out the third possibility.

The first possibility works in cases were the program will call the same methods during its entire operation. Unfortunately, Python itself, as well as most applications, have a start-up phase where the program calls methods that are used only once. Therefore, there have to be precautions to clear the cache.

With the implemented strategy, there is the potential risk of cache thrashing. If more methods are in active use than there are indices, the cache will be cleared very often. In order to avoid this, two strategies can be employed:

- Keep track of the miss ratio. If there are more new index assignments than successful accesses to the cache, disable the cache.
- Introduce new generations lazily. If the cache is full, keep the current assignment of method indices as long as the hit ratio is good.

Since both strategies can be used simultaneously, it is easy to combine them in the implementation. The tricky part is the definition of a ‘good’ hit ratio, which means that only few cache accesses fail.

None of these alternatives have been implemented, basically because I have no good heuristics.

# 7. Measuring the Performance

During the implementation process, I compared the performance of a stock Python 1.5 implementation with one that had my modifications. It turned out that good measurements were difficult to obtain, especially since there are no well-designed benchmark suites for the area of execution I modified.

Since I suspected that the dictionary lookup was the most costly part of class attribute lookup, I wrote a simple benchmark based on the source code

```
class X:
 a1 = 1
 a2 = 1
 ...
 a1000 = 1
 def f():
  pass
for i in xrange(1,1000000):
 X.f
```

Then, in a loop, I accessed the value X.f a large number of times. On a 250 MHz UltraSparc, this needed 10 seconds. With my early modifications, it took 13 seconds. The results were discouraging. What's worse, the benchmark was flawed.

There were several problems with the benchmark:

- The class X was accessed through its global name, which required a dictionary lookup.
- The test did not involve instances.
- The loop construct would consume some time, as the interpreter was constructing new integers over an over again.

- The interpreter loop took some time itself, partially since it consumed time running `SET_LINENO`.

To reduce the cost of the loop, I unrolled it. Also, there was a multiply involved in the iteration process, and it turned out that gcc would generate `.imul` calls on the Sparc. Since recent Sparc machines have built-in multiplication, I used the `–mcpu=v8` flag for gcc 2.8.1. To further reduce the cost of the loop, I used a pre-computed list to iterate. To reduce the impact of the interpreter itself, I used `python -O` for the measurements.

Finally, I switched to using identical code bases for further measurements, i.e. Python 1.5.1 for both the baseline code, and my modifications.

The revised benchmark is

```
class X:
 f = None
class Y(X):
 a1 = 1
 a2 = 1
 ...
 a1000 = 1
def f():
    x=Y()
    l=[1]*1000
    for a in l:
        for i in l:
            x.f
            x.f
            #repeat 6 more times
f()
```

## 8. Benchmark Results

During the experiments, I used two types of measurement. First, I timed the execution on an idle processor using the Unix `time(1)` command. While this allows to compare two implementations, it doesn't tell `why` some implementation is slower than another. To understand time consumption better, I profiled the interpreter using the GNU C compiler profiling (-pg). This tool counts the number a function is called, and helps reconstructing the call graph (i.e it tells why a function is called frequently).

Using the sample code above, I compared Python 1.5.1 and my modified version. Plain Python needs 26s, the modified version needs 16s.

To obtain more realistic data, I looked for a large Python program that makes use of classes and inheritance. I settled with Grail [3], because it is perhaps one of the most well-known Python applications. Unfortunately, it is difficult to benchmark. It involves user and network interaction, both having unpredictable timing properties. To obtain reliable data, I profiled function counts, concentrating on the functions where I expected most significant changes.

During the sampling of Python 1.5.1, I got the following call frequencies:

| Function | Number of calls |
|---|---|
| **lookdict** | 2 million |
| **PyDict_GetItem** | 1.7 million |
| **instance_getattr** | 490,000 |

| Function | Number of calls to PyDict_Getitem |
|---|---|
| **eval_code2** | 670,000 |
| **instance_getattr** | 480,000 |
| **class_lookup** | 448,000 |

This statistic shows that a significant number of dictionary lookups results instance attribute lookups, yet many lookups fail and are performed in the class again. Each call to `instance_getattr` generates a call to `PyDict_GetItem`, which in turn needs one or more accesses to the dictionary representation. A call to `class_lookup` generates also a dictionary lookup. `eval_code2` is invoked each time a Python function is invoked, and generates lookups primarily to the dictionary of global names of a function. `instance_getattr` is invoked each time an instance attribute is accessed. Both functions together therefore give a rough estimate of the computational complexity of a Python run.

The next trace, for the modified version, works on roughly the same data. Unfortunately, it is not easy to repeat the exact sequence of operations when browsing through the Web.

| Function | Number of calls |
|---|---|
| **lookdict** | 1.5 million |

| | |
|---|---|
| **PyDict_GetItem** | 1.2 million |
| **instance_getattr** | 418,000(*) |

| **Function** | **Number of calls to PyDict_GetItem** |
|---|---|
| **eval_code2** | 580,000(*) |
| **instance_getattr** | 280,000 |
| **class_lookup** | 211,000 |

As the marked numbers show, this sample was about 85% computational effort of the first run, yet the number of dictionary data accesses (`lookdict`) is goes down to 75%. Also, `instance_getattr` performs dictionary lookups only two thirds of the time. Also, the number of dictionary lookups performed during `class_lookup` is significantly reduced.

## 9. Thinking about Memory

Every caching scheme is a trade-off between time and memory. In the approach outlined here, part of the memory consumption comes from associating new objects with interned strings. This could be partially reduced by not giving every interned string additional attributes.

Another concern is the size of the virtual tables. The Onodera-Nakamura implementation reserved a pool of virtual tables to the most actively used classes. This results in even more arbitrary choices, such as: How much tables are there, and which classes should get one? Instead, I decided that only most-derived classes should get virtual tables. Since most class attribute lookups ultimately come from instance attribute lookups, only classes that have instances need to get virtual tables. The base class lookup will then still perform dictionary lookups; after this is performed for the first time, the derived classes will cache the value anyway, and not look in the base classes again. There have been no measurements, yet, how much memory is consumed by virtual tables in a typical application.

## 10. Future Directions

A number of modifications to that code have been considered, but not implemented. First, it is not possible to administrate the strategies used. It is not clear whether the application programmer needs access to the various tuning parameters (virtual table size, whether to use virtual tables at all, ...). If there is a tuning need, it is not clear whether tuning at run-time is necessary.

The implementation itself needs more testing. It introduces considerable changes to the Python runtime, which need careful review before being applied to mainstream Python.

## 11. Conclusions

With the introduction of virtual tables in Python, selected applications will see improved performance. While it is possible to construct programs that show a loss of performance, it seems unlikely that existing applications would suffer.

Further work is necessary to determine effects of such caching techniques in real applications.

## 12. Literature


[1] Timothy Budd. “An Introduction to Object-Oriented Programming”, Addison Wesley Longman, 1997.

[2] Tamiya Onodera, Hiroaki Nakamura. “ Optimizing Smalltalk by Selector Code Indexing Can Be Practical”, in “ECOOP’97 – Object Oriented Programming”, Jyväskylä, Spring, 1997.

[3] CNRI. “Grail 0.4”, http://grail.cnri.reston.va.us/grail/