Static and Dynamic Method Unboxing for Python

Anonymous

Abstract: The Python programming language supports object-oriented programming using a simple and elegant model that treats member variables, methods, and various metadata as instances of a single kind of 'attribute'. While this allows a simple implementation of an interpreter that supports advanced metaprogramming features, it can inhibit the performance of certain very common special cases. This paper deals with the optimization of code that loads and then calls object methods.

We modify Python's compiler to emit special bytecode sequences for load/call pairs on object attributes to avoid unnecessary allocation of method objects. This can result in considerable speedups, but may cause slowdowns at call sites that refer to builtin functions or other special attributes rather than methods. We therefore extend this static compile-time approach by a dynamic runtime quickening scheme that falls back to the generic load/call sequence at such call sites.

The experimental evaluation of dynamic unboxing shows speedups of up to 8 % and rare slowdowns caused by as yet unresolved excessive instruction cache misses. A comparison with a common manual optimization of method calls in Python programs shows that our automatic method is not as powerful but more widely applicable.

1 Introduction

Dynamically typed interpreted programming languages like Python[1] are increasingly popular due to their ease of use and their flexibility that allows their application to a large range of problems. Such languages enable rapid prototyping and high programmer productivity because even large changes to a codebase are often local and do not involve changing many static type declarations.

Such flexibility comes at a price in program performance. A completely dynamic language cannot predict at compile time all the information that would be needed to specialize operations to certain data types. This dynamic behavior makes implementations using simple interpreters very attractive; just-in-time (JIT) compilation is also possible, but doing it right even given an existing JIT compiler involves considerably more effort than interpretation [CEI+12].

The main Python implementation is therefore an interpreter written in C and often referred to as CPython. It interprets a custom bytecode instruction set that is tightly coupled to Python's language semantics and uses an evaluation stack to pass data from one instruction to the next. In Python, everything is an object that can be assigned, inspected, and modified. This enables great expressivity but can lead to inefficient behavior in common cases. In particular, methods are first-class objects as well, and method calls involve both

[1] `http://www.python.org/`

a lookup in a dynamic data structure and allocation of a method object on the heap which is then consumed and deallocated by the call.

Method calls are known to be slow in Python for these reasons. Previous work has proposed to cache some of the complexity of the lookup by caching call targets [MKC+10] or call types [Bru10b]. We propose an orthogonal optimization:

The rest of this paper is organized as follows. In Section 2 we describe how Python looks up, boxes, and calls methods and similar objects and where inefficiencies lie. In Section 3 we propose a static compile-time solution to the problem of repeated boxing and unboxing of method objects and observe that it performs much worse than it should; Section 4 shows a solution to this problem by using quickening at runtime. Section 5 looks at two micro-benchmarks for an estimation of the best speedups possible using our method under maximally friendly and maximally adverse conditions. Section 6 discusses related work, and Section 7 concludes.

2 Method Calls in Python

As a prerequisite and motivation for our program transformation, we must first discuss which kinds of callable objects are available in Python, and how calls are implemented in the interpreters's bytecode. There are three major types of callable objects: Functions, methods, and 'builtin' operations. Functions behave as expected and are called with explicitly provided arguments. Methods are functions defined within class definitions and have a special first function argument `self` that refers to the call's receiver. Methods are called as expressions of the form `o.f(a, b)` where the `self` argument is bound to the value of `o` and the other arguments are passed as expected. A builtin is a piece of code implemented in C that may or may not have a `self` pointer and may or may not be called using the `o.f` attribute access syntax.

Given a variable (or more complex expression) `o`, a name `f`, and expressions `a` and `b`, the Python expression `o.f(a, b)` may mean one of several things:

- A call to a method named `f` associated with `o` or with `o`'s class or one of its superclasses, with the three arguments `o`, `a`, `b`, binding the value of `o` to the method's special `self` parameter.
- A call to a function implemented in Python and stored in `o`'s `f` field that is not a method, i. e., does not have a `self` pointer, and is called with only *two* arguments `a` and `b`.
- A call to a 'builtin' function or method implemented in C and associated with `o`'s `f` field that may or may not have a `self` pointer and is called with two or three arguments, depending on its internal flags.
- A call to some other callable object (very rare).

Due to Python's dynamic nature which allows adding, deleting and modifying attributes

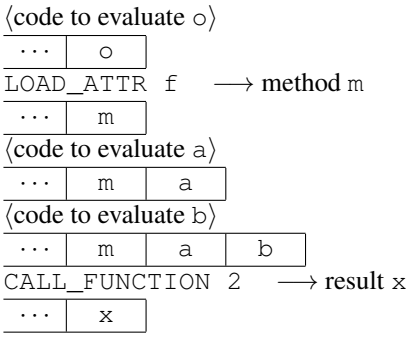


Figure 1: Abstract bytecode sequence and stack states for method call `o.f(a, b)`.

of classes and any object instances at runtime, it is impossible to distinguish between these cases statically. The bytecode compiler therefore emits the same code sequence for each of these cases, and the interpreter must make a dynamic decision how to execute each call based on the actual type of callable object encountered.

The bytecode sequence for this expression is illustrated in Figure 1. Code snippets are interleaved with an illustration of the state of the evaluation stack, which grows to the right. The basic call mechanism in Python is to push the function or method to call onto the stack, then push its arguments above it, and to execute a call instruction to pop off all of these values and push the function's return value. The call instruction has a constant argument telling it how many arguments are on the stack above the function or method to call.

In the concrete case of method calls via an expression `o.f`, first the value of `o` is computed and placed on the stack, then a `LOAD_ATTR` instruction is executed to replace it with its attribute `f`. Assume that `f` is defined as method in the class of object `o` or some superclass of it. Method definitions in Python are simply function definitions located within class definitions and taking a special '`self`' pointer as their first argument. When a method is looked up, the lookup actually finds this function first and consults a 'descriptor' associated with it. This descriptor may be modified, but it will typically cause allocation of an actual *method* object containing both the function `f` and the receiver `o`. We call this operation *boxing* of the function, and refer to functions with a descriptor like this as *boxable* functions. Once the method object containing the function is allocated, it replaces the original object `o` on the stack.

Besides boxable functions, callable object attributes may also refer to plain Python functions that are returned by the lookup without boxing, or builtins that are contained in a different type of object. Each of these also replace `o` on the stack.

Therefore, whatever the type of `o.f`, `o` itself is no longer directly reachable from the stack. If the loaded value of `o.f` is a function without a `self` pointer, it will only be called with the two arguments `a` and `b` that are on the stack, so `o` is not necessary. However, if it is a method, the value of `o` must be recovered in order to pass it as the first of *three* arguments to the function call.

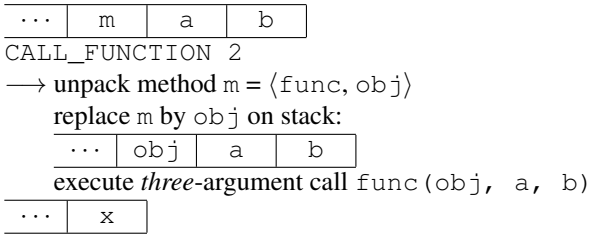


Figure 2: Unpacking a method object `m` to recover its `self` pointer.

Figure 2 illustrates the Python interpreter's solution to this problem: When the object to be called is a method, its receiver is reachable from a pointer within it. The `CALL_ATTR n` instruction checks the type of the callable object below the arguments, at index $-\mathtt{n}-1$ from the stack top, and if it is a method, unboxes it. The receiver object replaces the method on the stack, and the function is executed with $\mathtt{n}+1$ arguments.

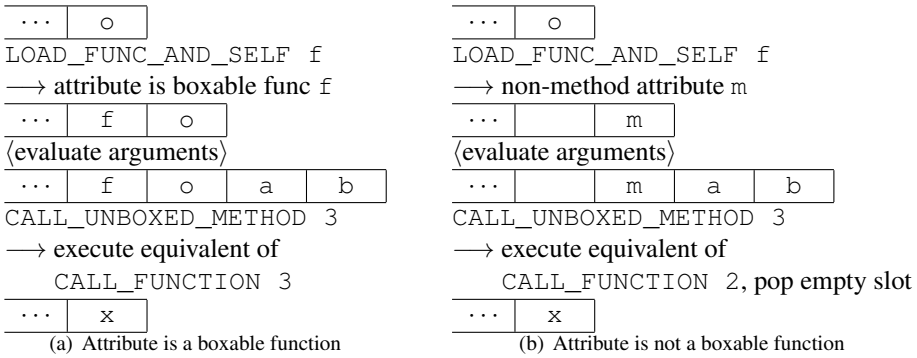
In summary, in the very common case of method calls of the form `o.f`, the Python interpreter performs a lookup for the code of `f` and boxes it up into a heap-allocated reference-counted method object ⟨`o`, `f`⟩which is then almost immediately taken apart and deallocated by the corresponding call instruction. The method object cannot escape this bytecode sequence, its lifetime extends only from the attribute load to the function call. Thus, although the allocations are almost always served very quickly from a free list, all of these operations aren't really necessary, and we propose to replace them with an unboxing scheme.

3 Unboxed Method Calls

We propose to unbox method calls based on the observations that method calls are frequent in object-oriented Python programs (but not in numeric and some library-intensive benchmarks; we discuss this issue later), that boxing and unboxing is slow and redundant, and on the fact that a fast alternative exists: Once the method is unpacked and the `self` pointer is on the stack, the interpreter treats an n-argument method call identically to an $n+1$-argument call of a 'plain' Python function. Such calls are simple, so if we can set up the stack in the desired format right away, we can immediately execute the simple fast function call path.

3.1 Compile-Time Unboxing

The optimization is based on introducing two new bytecode instructions that specialize a `LOAD_ATTR`/`CALL_FUNCTION n` pair to unboxed method calls. During compilation from the program's abstract syntax tree (AST) to bytecode, we detect cases where a call's target is an attribute access like `o.f`. In this special case, the modified compiler generates our new `LOAD_FUNC_AND_SELF` bytecode instead of `LOAD_ATTR`, and the new byte-



(a) Attribute is a boxable function

(b) Attribute is not a boxable function

Figure 3: Unboxed attribute calls of the form `o.f(a, b)`.

code `CALL_UNBOXED_METHOD n+1` instead of `CALL_FUNCTION n`. The meaning of these bytecodes is illustrated in Figure 3.

The `LOAD_FUNC_AND_SELF` bytecode is like `LOAD_ATTR` in that it performs a lookup of an attribute in the base object on the top of the stack. However, this bytecode is only used with `CALL_UNBOXED_METHOD`, and its result takes up two stack slots. In the common case illustrated in Figure 3(a), the attribute lookup yields a function that `LOAD_ATTR` would box up with the receiver into a method object. We simply omit this boxing and place *both* the function and the receiver on the stack, which the function below the receiver. Function arguments are evaluated and pushed as usual, and when the corresponding `CALL_UNBOXED_METHOD n+1` instruction is reached, the order of the function and its arguments on the stack is exactly as is needed for a plain function call, without any unboxing or on-stack replacement operations.

There is also the possibility, shown in Figure 3(b), that the attribute found by the lookup performed by `LOAD_FUNC_AND_SELF` is not a function that would be boxed up as a method. These are typically 'builtin' functions with an external implementation in C, or sometimes 'plain' Python functions that do not take a `self` argument. (These can be put into any slot of any object by simple assignment.) In these cases we do not need to put a receiver object onto the stack, but the following `CALL_UNBOXED_METHOD n+1` needs to be able to distinguish between the unboxed method case and this non-method case. `LOAD_FUNC_AND_SELF` therefore leaves a stack slot empty (that is, we push a NULL pointer or other special sentinel value) and places the attribute it found above it. The corresponding call always checks this slot. If it is not empty, it must be a function, and execution proceeds with a fast function call as in Figure 3(a); otherwise, a more elaborate call sequence based on the exact dynamic type of the looked-up attribute is initiated. After the call completes, the empty slot is removed from the stack before the call's result is pushed.

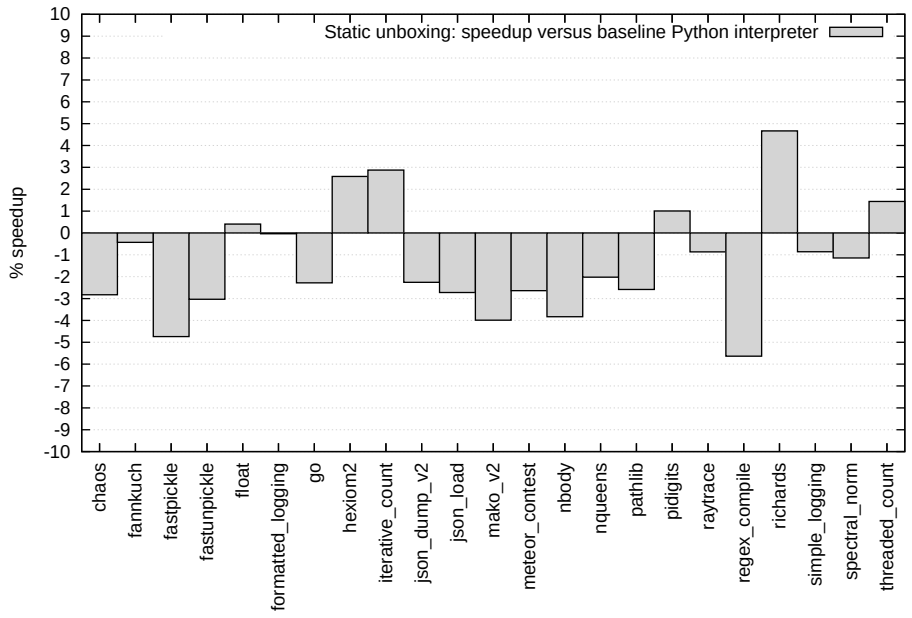


Figure 4: Performance impact of static method call unboxing at compile-time.

3.2 Experimental Evaluation

We implemented this method unboxing scheme in the Python 3.3.0 interpreter and evaluated it on a system with an Intel Xeon 3 GHz processor running Linux 2.6.30. Python was compiled with GCC 4.3.2 using the default settings, resulting in aggressive optimization with the `-O3` flag.

Despite the fact that Python 3 has existed for several years, its adoption by the community has been slow, which means that many popular benchmarks, such as Django, are only available in incompatible Python 2 versions. This restricts our choice of useful benchmark programs, but there is still a collection of useful though mostly quite small Python 3 compatible benchmarks available from `http://hg.python.org/benchmarks/`. We used these programs but excluded all those that took less than 0.1 seconds to execute a benchmark iteration because we found that these result in very inaccurate timings. We also excluded the micro-benchmarks that only test the speed of method calls (we look at these separately in Section 5) as well as the `2to3` benchmark that failed in some configurations even using an unmodified Python interpreter.

Figure 4 shows the speedups we obtained by using static method unboxing versus the baseline, unmodified Pyton interpreter. We were surprised to find that our proposed optimization tended to result in slowdowns on most programs. In a number of cases this is easy to explain as mis-speculation by the compiler that assumes that all attribute calls will

go to unboxable methods implemented in Python. However, at least in the `fastpickle`, `fastunpickle`, `json_dump_v2`, and `json_load` benchmarks, this is simply not true: These benchmarks test the speed of libraries that are implemented as Python classes but with important operations implemented as callable attributes that are not Python methods but 'builtins' implemented in C. As we described above, we treat such calls by leaving the result of the lookup boxed, placing an empty slot on the stack below it, and checking that slot to decide how to perform the call. This overhead is not great, but it can result in a measurable slowdown if this slow path is hit in a benchmark that performs many of these calls. The following section proposes a solution for such cases of static mis-speculation.

Other slowdowns are more baffling: `nbody`, for instance, is a numeric benchmark that is aimed squarely at mathematical computation and contains no calls via attributes at all. No matter how we treat method calls, that treatment should not make an observable difference on such programs. We used the PAPI library[2] to collect information from hardware performance counters to better understand the behavior of the interpreter on such benchmarks. PAPI indeed verifies that the number of instructions executed on `nbody` for the baseline and our modified interpreter is essentially identical. However, the number of L1 instruction cache misses increases by a factor of 6–7. Our changes to the interpreter are small, but it appears that inserting the code for handling our two new bytecode instructions caused the compiler to make changes to the code layout that lead to this very detrimental effect.

Overall, we observe that the benchmark suite contains quite few programs written in an object-oriented style and is thus inherently biased against optimizations aimed at method calls. Instruction cache misses may be difficult to fix because they depend on the code layout generated by the underlying compiler, but at least we must make an effort to correct the static mis-speculation at call sites that typically call builtins, not Python methods.

4 Quickening to the Rescue!

The program transformation described in the previous section is completely static; the call sites where method unboxing is applied are determined by the bytecode compiler based only on the shape of the AST. However, we do not know at compile time which of these sites will actually call Python methods and which will call builtins, and as the experimental evaluation shows, this speculation on 'methods always' can fail and lead to diminished performance.

Since purely static speculation does not seem like a good approach, we therefore move on to a more dynamic approach using quickening. Quickening [Bru10a] is an operation that modifies code at runtime, typically after some sort of initialization has taken place or where data collected during previous execution of an instruction suggests that another variant of the instruction may execute more efficiently. Quickening can be applied to specialize code to particular data types; this often pays off because even in highly dynamic languages like Python, many pieces of code are 'dynamically static' in their usage of types, i. e., even

[2] `http://icl.cs.utk.edu/papi/`

if operand types for given instructions cannot be predicted beforehand, they often tend to remain relatively constant over single program runs.

4.1 Quickening of Non-Unboxed Calls

In our method unboxing implementation, we specialize every call site separately depending on the type of the attribute loaded to serve as a call target. If a lookup/call pair referred to an unboxable method in the past, we expect it to also do so in the future; if, however, the attribute was a builtin or other non-method function, we assume that that call site will tend to call such non-methods in the future.

We therefore modified our implementation of the `LOAD_FUNC_AND_SELF` bytecode to use quickening in cases where the attribute it finds is not a boxable function. In such cases, we immediately fall back to the behavior of the baseline interpreter. Instead of using the path shown in Figure 3(b) with an empty stack slot, in this case `LOAD_FUNC_AND_SELF` simply pushes the attribute to the stack and immediately quickens itself to (i. e., replaces itself by) `LOAD_ATTR`. As the call target is now boxed, the following call can no longer be `CALL_UNBOXED`, so we immediately quicken it to `CALL_FUNCTION`. That is, at this call site this and all future calls will be treated exactly as in the regular Python interpreter.

To quicken the call corresponding to a given attribute lookup, we need to locate that call; this is implemented simply as a scan forward in the instruction stream over the argument evaluation code to the call. Computation of the arguments may also involve lookup/call pairs, but these are always strictly nested, so there is no ambiguity as to which lookup belongs to which call. A linear scan over the instrucions may seem expensive, but the call is typically very close to the lookup (never more than 200 bytes later in our benchmarks, and typically much closer), and quickening is rare, so the total overhead is acceptable.

An alternative implementation without a forward sweep over the code would be to place a special marker on the stack as a signal to the call instruction to quicken itself. This does not work in practice, however: Exceptions may (and do!) occur during argument evaluation, so the call instruction may not be reached. The next time the lookup/call pair would be executed, the lookup would already be quickened, but the call would not. As they have different expectations about the organization of the stack, this would lead to program failures. We must therefore quicken both the lookup and the call at the same time. A possibly slightly more efficient variant would be to associate an offset with the lookup that tells it exactly the distance to the corresponding call. This would be considerably more tedious to implement than the current scheme, so we have not evaluated this possibility yet.

If the attribute found by `LOAD_FUNC_AND_SELF` is a boxable method, we perform an unboxed method call as before, and no quickening is needed.

Figure 5 shows a graph of all the states that an attribute call site can be in, and the transitions between the states. In the quickening variant we introduced yet another pair of lookup/call instructions, called `LOAD_CALLABLE_ATTR` and `PSEUDO_CALL_ATTR`, respectively. The idea is to generate only these instructions in the compiler. The first

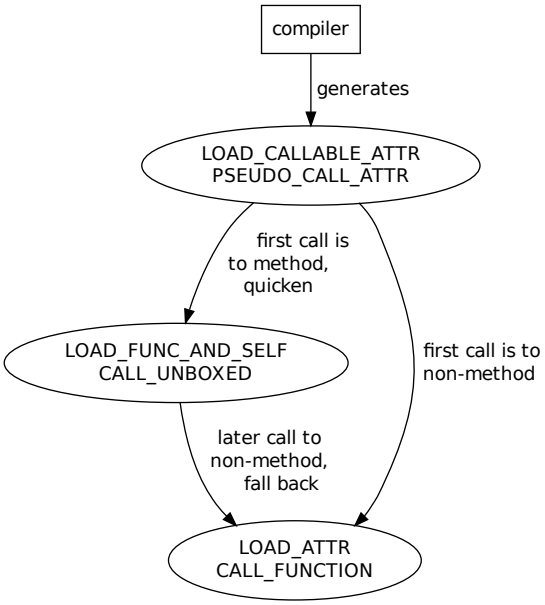


Figure 5: Possible states of bytecode pairs at each attribute call site.

time an occurrence of `LOAD_CALLABLE_ATTR` is executed, it quickens itself to either `LOAD_FUNC_AND_SELF` or to `LOAD_ATTR`, depending on whether the attribute it found is a method or not. If it is a method, it is unboxed on the fly, and the instruction is quickened to `LOAD_FUNC_AND_SELF`.

`PSEUDO_CALL_ATTR` is never executed and does not even have an implementation. As it is always preceded by `LOAD_CALLABLE_ATTR`, it will always be quickened to either `CALL_UNBOXED` or `CALL_FUNCTION` before it is reached.

Note the direction of transitions in the graph: Once a call site is quickened to the baseline case of `LOAD_ATTR`/ `CALL_FUNCTION`, it can never move back to unboxing, even if many of the future calls might profit from it. During testing, we found that such cases appear to be very rare or nonexistent in real Python programs, although it is easy to write simple counterexamples. In any case, `LOAD_ATTR`/ `CALL_FUNCTION` is a good, heavily optimized baseline implementation to fall back to.

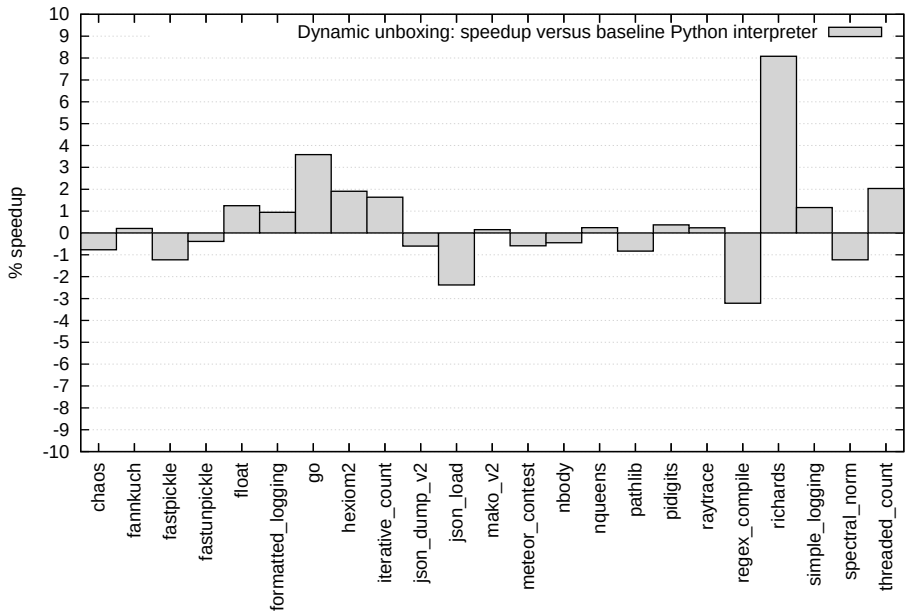


Figure 6: bar

4.2 Experimental Evaluation

Figure 6 shows the result of evaluating unboxing with quickening on the same benchmark set as in Figure 4; the scales are identical to enable a quick visual comparison. We note the general trend that most results are much better with quickening than with the purely static compile-time method. In particular, the number of benchmarks exhibiting slowdowns has been reduced, and where slowdowns remain, they are less pronounced than before. Several object oriented benchmarks that showed speedups before are now even faster. The most impressive case is `richards`, where even more attribute calls are now handled as efficiently as possible.

We investigated the curious case of the large 3 % slowdown on `regex_compile` in some detail. Using the PAPI library[3] to collect information from hardware performance counters, we verified that the quickened version needed about 3 % more processor cycles than the baseline interpreter. This is despite the fact that the number of *instructions* executed was reduced by about 0.5 %. However, the quickened version exhibits about 20 % more instruction cache misses than the baseline! We believe this is due to the larger and more complex code in the interpreter loop due to our bytecode instruction set extensions.

During initial testing of the quickening implementation, we also observed unexpectedly bad results for a number of benchmarks that should be able to take good advantage of

[3] `http://icl.cs.utk.edu/papi/`

unboxing. In particular, the `hexiom2` program showed a 2 % slowdown, which is strange given that the non-quickened version performs well. We suspected that the blame might lie with type checks in our implementation of `CALL_FUNC_AND_SELF`, which must always check whether the attribute to return is a boxable function. In the hope to optimize code layout, we tried using GCC's `__builtin_expect` mechanism to annotate two `if` statements with the information that we typically expected their conditions to succeed. These annotations were indeed what was needed to turn a 2 % slowdown into a 2 % speedup.

These observations suggest that the control flow of a typical bytecode interpreter—a loop containing a large `switch` statement containing computed gotos to other cases—may be confusing enough to compilers to make such manual annotations worthwhile. It might also be useful to try to minimize code size within the interpreter loop as much as possible. In particular, researchers modifying such bytecode interpreters should be aware that adding new bytecode instructions can come with an intrinsic cost even if the modified instruction set helps perform 'less work' in terms of the number of executed instructions.

Implementation size. Overall, our changes to the Python interpreter to accommodate both static and dynamic method unboxing comprise adding 569 lines of code including extensive comments and commented-out debug code. The implementations are cleanly separated by preprocessor directives that must be specified at compile time. This way, we can build the baseline, static, and dynamic unboxing versions of the interpreter from the same codebase, and the purely static implementation does not have the code size overhead (and resulting possible additional instruction cache misses) of the dynamic version.

5 Method Unboxing vs. Method Caching

We excluded call-oriented micro-benchmarks from the experimental evaluation in the previous sections because the speedups we achieve here do not translate to speedups in real applications, which perform other work besides simple function calls. However, we perform two experiments here to obtain a rough comparison with optimizations based on caching of method call targets.

The `call_method` benchmark performs a large number of method calls via method attributes; Figure 7(a) shows an illustrative code snippet from the benchmark. All the call sites in this program are monomorphic, so aggressive optimizations are possible in theory: All calls could be resolved before runtime, or at least cached at each call site. We do not perform any caching but are able to unbox all call sites, saving the method allocation and deallocation overhead associated with every call. On this micro-benchmark we achieve a speedup of 13 %, which we therefore assume is the best speedup that can ever be achieved using our method unboxing scheme (until we manage to fix instruction cache issues).

To put this into perspective, we compared our result to an optimization that is folklore in the Python community and applied widely in the Python library sources and other projects: Repeated calls (such as in a loop) of the same method on the same receiver object can be

```
class Foo(object):
   def foo(self, a, b, c, d):

     # 20 calls
     self.bar(a, b, c)
     self.bar(a, b, c)
     ...

def test_calls(iterations):
   f = Foo()

   for _ in xrange(iterations):
     # 20 calls
     f.foo(1, 2, 3, 4)
     f.foo(1, 2, 3, 4)
     ...
```

(a) Original benchmark

```
class Foo(object):
   def foo(self, a, b, c, d):
     self_bar = self.bar
     # 20 calls
     self_bar(a, b, c)
     self_bar(a, b, c)
     ...

def test_calls(iterations):
   f = Foo()
   f_foo = f.foo
   for _ in xrange(iterations):
     # 20 calls
     f_foo(1, 2, 3, 4)
     f_foo(1, 2, 3, 4)
     ...
```

(b) Manual caching of methods

Figure 7: The `call_method` micro-benchmark and a common manual optimization, caching methods in local variables.

optimized by assigning the method object to a local variable and performing the calls through that variable. Figure 7(b) illustrates this transformation. The optimization effect here is due to avoiding repeated lookups of the same value and allocating/deallocating a method object at each call site. At the place of the call, the method object need only be fetched from the local variable, which is fast, and unboxed for the call, which is also fast since the object need not be garbage collected because a reference to it remains in the local variable. This manually optimized program achieves a speedup of 39 % over the original; as this is is considerably larger than our unboxing speedup, we conclude that there is still some room for better caching of method calls in Python.

As a second micro-benchmark, we consider `call_method_unknown`, which involves four classes containing identically-named methods that call each other. However, this benchmark is designed to foil any attempt at caching of receivers or their types, as its documentation explains: 'To make the type of the invocant unpredictable, the arguments are rotated for each method call. On the first run of a method, all the call sites will use one type, and on the next, they will all be different.' (The code is too convoluted to be excerpted here; see ...) Since the types will be different at different times, a code transformation as in Figure 7 is not applicable, and any automatic caching would presumably incur many cache misses. Our unboxing approach, however, achieves a speedup of 12 % on this micro-benchmark. This is similar to the speedup on the simple call benchmark and is to be expected, since in our scheme lookups are always performed as usual, so changing targets should not impact the interpreter's performance at all.

6 Related Work

The most directly relevant work by Mostafa et al. [MKC+10] investigates various optimizations for the Python interpreter, including caching of attribute loads and loads from global variables. Using different variants of a caching scheme for `LOAD_ATTR` they achieved speedups of about 3–4 % on average, more if caching of global lookups was also enabled. Recall that we do not report an average speedup because our benchmark suite is biased against optimizations targeted at method calls, while the benchmarks used by Mostafa et al. appear to be mostly object-oriented, and optimized attribute lookups benefit most of their programs.

It is not clear how these speedups relate to ours because they used Python 2.6 and do not specify the minor version number; at some point during the 2.6 series, the official Python interpreter got some caching of attribute lookups built in, and we cannot tell whether their caching implementation came before or after this change. In any case, our results in comparison to Python 3.3 which definitely uses caching show that unboxing (which Mostafa et al. do not perform) is orthogonal to caching and can be combined with it in a profitable way.

We use the term quickening adapted from Brunthaler [Bru10b, Bru10a] to refer to the operation of replacing an instruction with a different version, although the operation itself is folklore. It is related to inline caching, which replaces not opcodes but instruction arguments in the instruction stream [DS84]. A similar optimization is also possible in interpreters that work not on bytecode but directly on abstract syntax trees [WWS+12].

7 Conclusions and Future Work

This paper is the first step in a project aimed at a detailed understanding of Python and other high-level language interpreters. Instrumenting the interpreter to collect statistics about the frequency and execution time of bytecode instructions, we noticed a correlation between occurrences of `LOAD_ATTR` and `CALL_FUNCTION`, which are both frequent and expensive, and designed an optimization to remove some of their shared overhead.

The next step is a larger-scale study of Python 3 programs and their behavior during interpretation. We intend to generalize the results of previous investigations that focused on studies of reflective and other dynamic behavior in Python programs [HH09] and on optimizing certain classes of instructions based on such quantitative data [MKC+10].

Finally, the observation that adding just two bytecodes to the Python interpreter can lead to significant slowdowns due to instruction cache misses, which we have not seen mentioned in the literature, motivates us to look at ways to optimize the interpreter loop's code locality. This might be possible by moving the implementations of rarely executed bytecodes into a separate interpreter function.

References

[Bru10a] Stefan Brunthaler. Efficient interpretation using quickening. In *Proceedings of the 6th symposium on Dynamic languages*, DLS '10, pages 1–14, New York, NY, USA, 2010. ACM.

[Bru10b] Stefan Brunthaler. Inline caching meets quickening. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 429–451, Berlin, Heidelberg, 2010. Springer-Verlag.

[CEI+12] Jose Castanos, David Edelsohn, Kazuaki Ishizaki, Priya Nagpurkar, Toshio Nakatani, Takeshi Ogasawara, and Peng Wu. On the benefits and pitfalls of extending a statically typed language JIT compiler for dynamic scripting languages. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 195–212, New York, NY, USA, 2012. ACM.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[HH09] Alex Holkner and James Harland. Evaluating the dynamic behaviour of Python applications. In Bernard Mans, editor, *ACSC*, volume 91 of *CRPIT*, pages 17–25. Australian Computer Society, 2009.

[MKC+10] Nagy Mostafa, Chandra Krintz, Calin Cascaval, David Edelsohn, Priya Nagpurkar, and Peng Wu. Understanding the Potential of Interpreter-based Optimizations for Python. Technical Report 2010-14, UCSB, 2010.

[WWS+12] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST interpreters. In *Proceedings of the 8th symposium on Dynamic languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM.