Gergö Barany

gergo@complang.tuwien.ac.at

Institute of Computer Languages
Vienna University of Technology

ATPS 2013
February 26, 2013
Coming up...

- Specialize Python method calls for unboxed representation
- Use quickening to fix mis-speculation
- Speedups up to 8% (and 13% on microbenchmarks)
Python ‘method’ calls

```
o.f(42)
What does this mean?
```
Python ‘method’ calls

What does this mean?

```
class O:
    def f(self, arg):
        ...

o = O()
o.f(42)
```

→ Call *method* with 2 arguments
Python ‘method’ calls

What does this mean?

```
Function call (via attribute)
class O:
    pass

def foo(arg):
    ...

o = O()
o.f = foo  # create new field f
o.f(42)
```

→ Call *function* with 1 argument
Python ‘method’ calls

What does this mean?

External function call

```python
o = ExternalClass()  # defined in C
o.f(42)
```

→ Call *external function* with 1 or 2 arguments
Compilation of `method` calls

Compilation of \( o.f(a_1, \ldots, a_n) \):

<table>
<thead>
<tr>
<th>Source</th>
<th>Bytecode</th>
<th>Stack effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>( o )</td>
<td></td>
<td>( \ldots)</td>
</tr>
<tr>
<td>( .f )</td>
<td>LOAD_ATTR f</td>
<td>( \ldots)</td>
</tr>
<tr>
<td>( \langle \text{args} \rangle )</td>
<td></td>
<td>( m ) ( a_1 ) ( \ldots) ( a_n )</td>
</tr>
<tr>
<td>( ) )</td>
<td>CALL_FUNCTION n</td>
<td>( \ldots)</td>
</tr>
</tbody>
</table>

\[ \rightarrow m = f \text{ or } m = \langle o, f \rangle \text{ or } m = \langle o, external \rangle \text{ or } \ldots \]
Static unboxing

Our solution: Special handling of attribute calls

- Assume common case \( m = \langle o, f \rangle \)
- Compile \( o.f(...) \) calls to new bytecodes:
  - LOAD_FUNC_AND_SELF and CALL_UNBOXED_METHOD

<table>
<thead>
<tr>
<th>Source</th>
<th>Bytecode</th>
<th>Stack effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>( o )</td>
<td>:</td>
<td>( \ldots )  ( o )</td>
</tr>
<tr>
<td>( .f )</td>
<td>LOAD_FUNC_AND_SELF ( f )</td>
<td>( \ldots )  ( f )  ( o )</td>
</tr>
<tr>
<td>( \langle \text{args} \rangle )</td>
<td>:</td>
<td>( \ldots )  ( f )  ( o )  ( a_1 )  ( \ldots )  ( a_n )</td>
</tr>
<tr>
<td>( ) )</td>
<td>CALL_UNBOXED_METHOD ( n )</td>
<td>( \ldots )  ( x )</td>
</tr>
</tbody>
</table>

→ No boxing/unboxing of \( \langle o, f \rangle \) needed, \( n + 1 \) arg function call
Behavior if \( m \neq \langle o, f \rangle \) (i.e., not a method)

<table>
<thead>
<tr>
<th>Source</th>
<th>Bytecode</th>
<th>Stack effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td></td>
<td>( \cdots ) ( o )</td>
</tr>
<tr>
<td>.f</td>
<td>LOAD_FUNC_AND_SELF f</td>
<td>( \cdots ) ( m )</td>
</tr>
<tr>
<td>( \langle \text{args} \rangle )</td>
<td>CALL_UNBOXED_METHOD n</td>
<td>( \cdots ) ( m ) ( a_1 ) ( \cdots ) ( a_n )</td>
</tr>
</tbody>
</table>

→ Must check for empty slot, unbox \( m \) if needed
Static unboxing: results

Static unboxing: speedup versus baseline Python interpreter

% speedup

Static and Dynamic Method Unboxing for Python
Static unboxing: speedup versus baseline Python interpreter

Object-oriented programs
Static unboxing: results

Static unboxing: speedup versus baseline Python interpreter

% speedup

Static unboxing: speedup versus baseline Python interpreter

heavy use of external classes
Solution: Dynamic unboxing

The problem with static unboxing

The compiler often mis-speculates assuming `o.f(...)` will be a method call.
This mis-speculation can be expensive.
Solution: Dynamic unboxing

The problem with static unboxing

The compiler often mis-speculates assuming o.f(...) will be a method call. This mis-speculation can be expensive.

Quickening to the rescue!

→ Method or not? Decide at first execution of call site.
Dynamic unboxing

compiler

LOAD_CALLABLE_ATTR

PSEUDO_CALL_ATTR

first call is to method, quicken

LOAD_FUNC_AND_SELF

CALL_UNBOXED

first call is to non-method

LOAD_ATTR

CALL_FUNCTION

later call to non-method, fall back

LOAD_ATTR

CALL_FUNCTION

Static and Dynamic Method Unboxing for Python
Dynamic unboxing: results

Dynamic unboxing: speedup versus baseline Python interpreter

% speedup

-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10

chaos fannkuch fastpickle fastunpickle float go hexiom2 iterative_count json_dump_v2 json_load mako_v2 meteor_contest nbody nqueens pathlib pidigits raytrace regex_compile richards simple_logging spectral_norm threaded_count

10/13  G. Barany (TU Vienna)  Static and Dynamic Method Unboxing for Python
Dynamic unboxing: results

Dynamic unboxing: speedup versus baseline Python interpreter

% speedup

instruction cache misses and branch mispredictions
Performance counter data

Detailed analysis

- Count branch mispredictions and L1 instruction cache misses (using PAPI)
- Run on interpreter with extra bytecodes, with unmodified compiler
- Measure overhead of extra instructions *that are never executed*
Method unboxing vs. method caching

Excerpt from method_call microbenchmark

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

Manual optimization: 39% speedup, our unboxing method: 13%

But: Our method also applicable in cases where caching impossible
Method unboxing vs. method caching

Excerpt from method_call microbenchmark

Common manual optimization:

```python
def foo(self, a, b, c):
    # 20 calls
    self.bar(a, b, c)
    self.bar(a, b, c)
    ...
```

```python
def foo(self, a, b, c):
    self_bar = self.bar
    # 20 calls
    self_bar(a, b, c)
    self_bar(a, b, c)
    ...
```

Manual optimization: 39% speedup, our unboxing method: 13%

But: Our method also applicable in cases where caching impossible
Method unboxing vs. method caching

Excerpt from method_call microbenchmark
Common manual optimization:

```python
def foo(self, a, b, c):
    # 20 calls
    self.bar(a, b, c)
    self.bar(a, b, c)
    ...

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

Manual optimization: 39% speedup, our unboxing method: 13%
But: Our method also applicable in cases where caching impossible
Summary

- Specialize Python method calls for unboxed representation
- Use quickening to fix mis-speculation
- Speedup up to 8%, but it’s not that simple...
Summary

- Specialize Python method calls for unboxed representation
- Use quickening to fix mis-speculation
- Speedup up to 8%, but it’s not that simple...

Thank you for your attention!

This work was supported by the Austrian Science Fund (FWF) under contract P23303, *Spyculative*. 