Static and Dynamic Method Unboxing for Python

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

`o.f(42)`

What does this mean?

Method call

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

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

→ Call *method* with 2 arguments

Python 'method' calls

`o.f(42)`

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

`o.f(42)`

What does this mean?

External function call

```
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, ..., a_n)`:

Source	Bytecode	Stack effect				
`o`	⋮	···	`o`			
`.f`	`LOAD_ATTR f`	···	`m`			
`(`⟨*args*⟩	⋮	···	`m`	`a_1`	···	`a_n`
`)`	`CALL_FUNCTION n`	···	`x`			

➞ $\mathtt{m} = \mathtt{f}$ or $\mathtt{m} = \langle \mathtt{o}, \mathtt{f} \rangle$ or $\mathtt{m} = \langle \mathtt{o}, \mathit{external} \rangle$ or . . .

Static unboxing

Our solution: Special handling of attribute calls

- Assume common case $\mathtt{m} = \langle \mathtt{o}, \mathtt{f} \rangle$
- Compile `o.f(...)` calls to new bytecodes `LOAD_FUNC_AND_SELF` and `CALL_UNBOXED_METHOD`

Source	Bytecode	Stack effect					
`o`	⋮	···	`o`				
`.f`	`LOAD_FUNC_AND_SELF f`	···	`f`	`o`			
`(`⟨*args*⟩	⋮	···	`f`	`o`	`a_1`	···	`a_n`
`)`	`CALL_UNBOXED_METHOD n`	···	`x`				

→ No boxing/unboxing of $\langle \mathtt{o}, \mathtt{f} \rangle$ needed, $n + 1$ arg function call

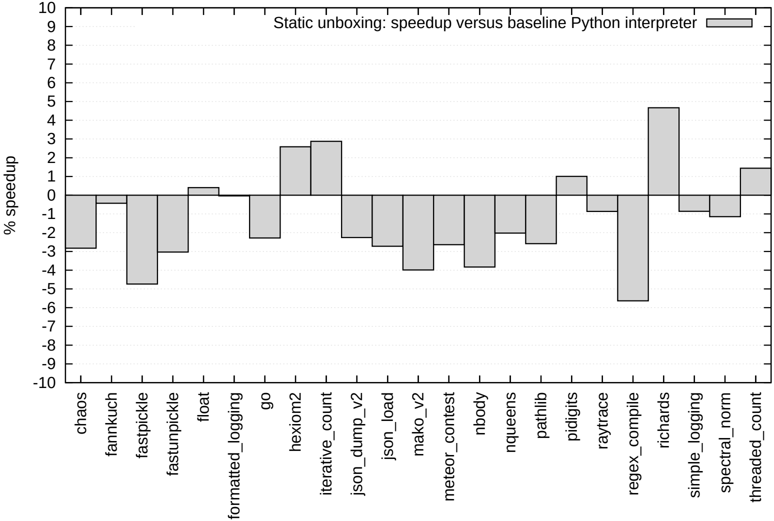
Static unboxing: non-method case

Behavior if $\mathtt{m} \neq \langle \mathtt{o}, \mathtt{f} \rangle$ (i. e., not a method)

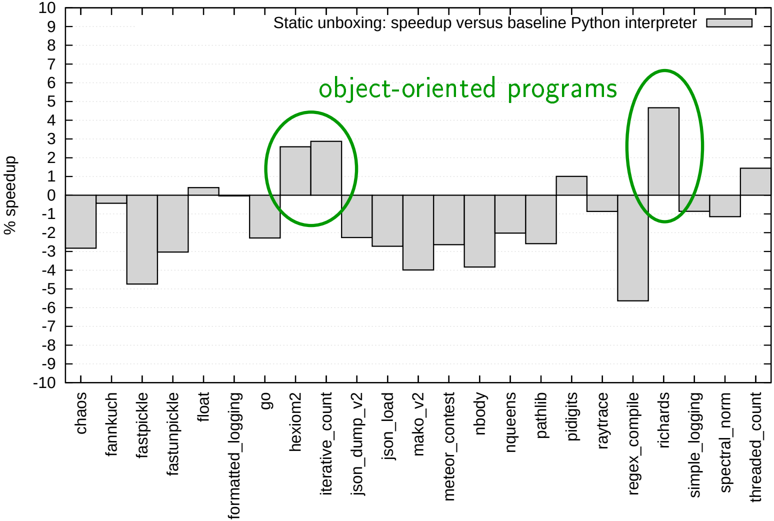
Source	Bytecode	Stack effect					
o	⋮	···	o				
.f	LOAD_FUNC_AND_SELF f	···		m			
(⟨*args*⟩	⋮	···		m	a_1	···	a_n
)	CALL_UNBOXED_METHOD n	···	x				

→ Must check for empty slot, unbox m if needed

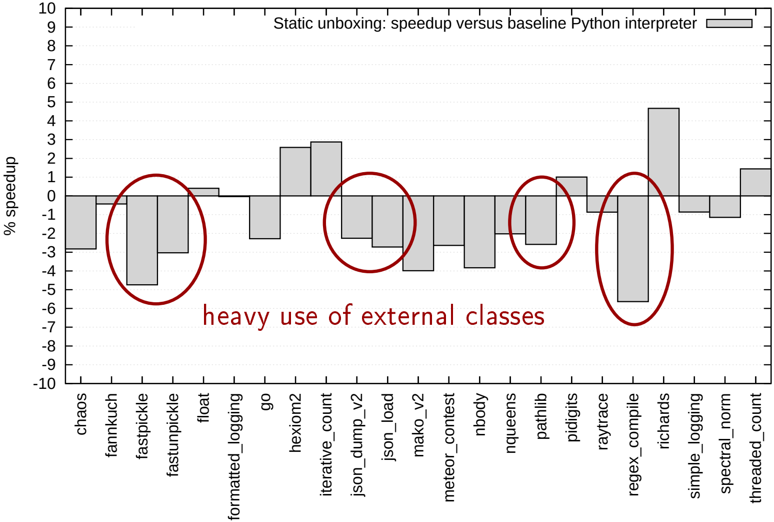
Static unboxing: results



Static unboxing: results



Static unboxing: results



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

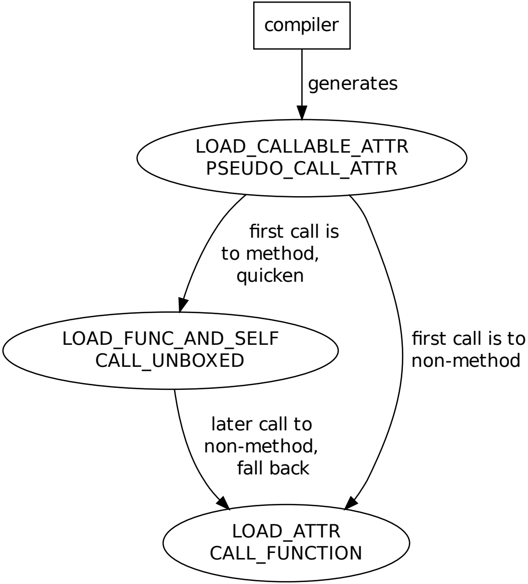
The problem with static unboxing

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

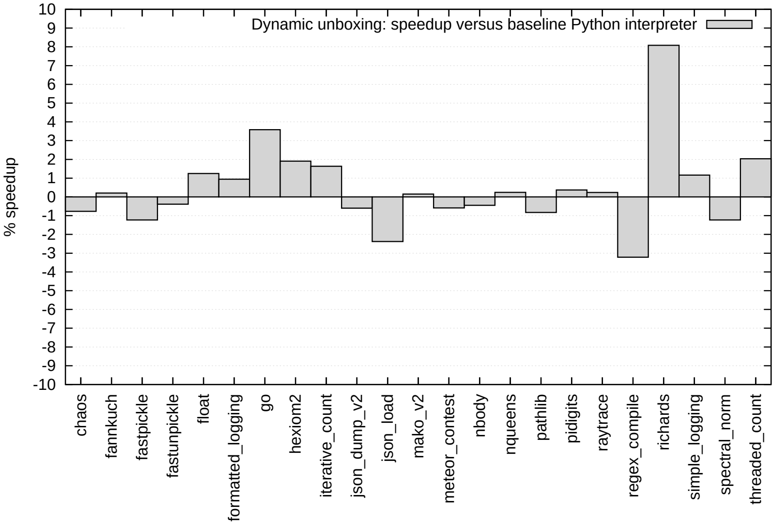
Quickening to the rescue!

➙ Method or not? Decide at first execution of call site.

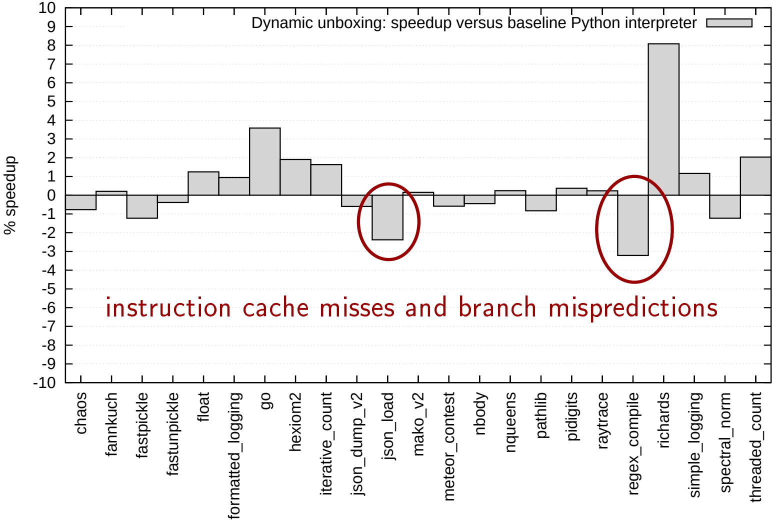
Dynamic unboxing



Dynamic unboxing: results



Dynamic unboxing: results



instruction cache misses and branch mispredictions

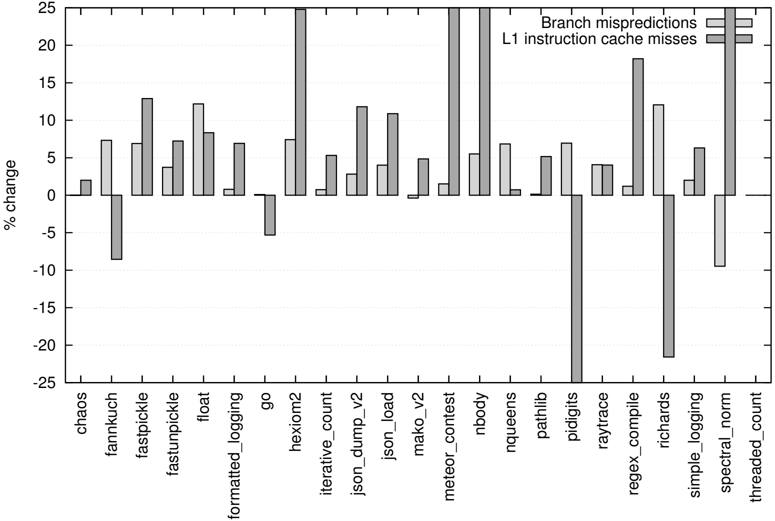
Performance counter data

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*

Performance counter data



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)
    ...
```

Method unboxing vs. method caching

Excerpt from `method_call` microbenchmark
Common manual optimization:

```
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)
    ...
```

Method unboxing vs. method caching

Excerpt from `method_call` microbenchmark
Common manual optimization:

```
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*.