Twopy: A Just-In-Time Compiler For Python
Based On Code Specialization

Julien Pagès, Marc Feeley
November 4th 2018
Université de Montréal
Motivation

Twopy

First experiments

Conclusion
Motivation

Twopy

First experiments

Conclusion
The Python language

Motivation

- Python is a popular dynamic and general-purpose language
- It offers object-oriented, procedural and functional features
Motivation

- Python is a popular dynamic and general-purpose language
- It offers object-oriented, procedural and functional features

Explore Python and Basic Block Versioning

- Python’s performance is still behind other dynamic languages
- What are the challenges?
Main implementations:

- CPython: official interpreter written in C
- PyPy: Meta-tracing Just In Time (JIT) compiler written in RPython (a subset of Python)

Other implementation

- Python on Truffle/Graal: complex framework on top of a modified JVM
- Jython/IronPython: compilers to the JVM or .NET CLR
- Psyco, Pyston: abandoned project of JIT compilers
Motivation

Twopy

First experiments

Conclusion
Twopy: a new lazy JIT compiler for Python targeting x86_64 assembly.

**Implementation choices**

- Investigate basic block versioning with Python
- Use the official *bytecode* as a frontend
- Written in Python 3.7
- JIT compilation to x86 assembly
```python
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
Basic Block Versioning (BBV) [Chevalier-Boisvert and Feeley, 2015]:

- Specialize the code according to runtime types
- Types are lazily collected during type-tests
- Adapted to dynamic languages (JavaScript, Scheme, Python)
- Simple version in intraprocedural, can be extended to interprocedural
Internal of basic block versioning

- Entry point
- int(n) ?
  - true: stub 2
  - false: stub 1
Internal of basic block versioning
Official Python bytecode

- Stack-based
- High level operations: manipulating the stack, arithmetic operations, function calls, objects
- These instructions are just split in basic blocks

The frontend is low-development effort thanks to Python’s API.
Implementation details

- Tagging with 3 bits of objects to implement dynamic typing
- Integers on 61 bits and floats are boxed
- Python FFI to navigate between the compiler, C and assembly
- Generation of x86 instructions with the PeachPy [Dukhan, 2013] encoder
Lazy compilation: stubs

<table>
<thead>
<tr>
<th>Code section</th>
<th>Stubs section</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV r10, 0x7f3ec3cde190</td>
<td></td>
</tr>
<tr>
<td>JMP r10</td>
<td>mov rdi, rsp</td>
</tr>
<tr>
<td></td>
<td>movabs r10, 0x7f0cc2988090</td>
</tr>
<tr>
<td></td>
<td>call r10</td>
</tr>
<tr>
<td></td>
<td>## information ##</td>
</tr>
<tr>
<td></td>
<td>block_id</td>
</tr>
</tbody>
</table>
Lazy compilation: stubs

Code section
.
.
MOV r10, 0x7f3ec3cde190
JMP r10

Stubs section
.
.
mov rdi, rsp
movabs r10, 0x7f0cc2988090
call r10

## information ##
block_id

Call to C
Lazy compilation: stubs

```
Code section
.
.
MOV r10, 0x7f3ec3cde190
JMP r10
```

```
Stubs section
.
.
mov    rdi, rsp
movabs r10, 0x7f0cc2988090
call   r10

## information
block_id
```

Call to C

Call to python via callback
Lazy compilation: stubs

Code section

Stubs section

Call to C

Call to python via callback

Compilation of the new block

MOV r10, 0x7f3ec3cde190
JMP r10

mov rdi, rsp
movabs r10, 0x7f0cc2988090
call r10

## information ##
block_id
Lazy compilation: stubs

Code section

:  
:  
MOV r10, 0x7f3ec3cde190
JMP r10

Stubs section

:  
:  
mov rdi, rsp
movabs r10, 0x7f0cc2988090
call r10
## information ##
block_id

Call to C

Call to python via callback

Compilation of the new block

:new_block
Lazy compilation: stubs

```
Code section
.
.
MOV r10, 0x7f3ec3cde190
JMP r10

Stubs section
.
.
mov    rdi, rsp
movabs r10, 0x7f0cc2988090
call   r10
JMP new_block

:new_block
```
Lazy compilation: stubs

Code section

:  
:  
JMP new_block

Stubs section

:  
:  
mov rdi, rsp
movabs r10, 0x7f00cc2988090
call r10
JMP new_block

:new_block
Motivation

Twopy

First experiments

Conclusion
Eliminated tests thanks to BBV

<table>
<thead>
<tr>
<th></th>
<th>fibonacci</th>
<th>for_loops</th>
<th>ack</th>
<th>tak</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without BBV</td>
<td>6623205608</td>
<td>4559999552</td>
<td>1072759792</td>
<td>1773815640</td>
</tr>
<tr>
<td>BBV</td>
<td>2649282244</td>
<td>1579999848</td>
<td>268206304</td>
<td>1267011192</td>
</tr>
<tr>
<td>Eliminated tests</td>
<td>60%</td>
<td>65%</td>
<td>75%</td>
<td>29%</td>
</tr>
</tbody>
</table>

**Table 1:** Runtime executed type-tests in Twopy
Micro benchmarks

BBV is not fully implemented

Figure 1: Time results (seconds) relative to CPython
Limitations in benchmarks

Limitations of Twopy

- Only a limited version of BBV (no interprocedural)
- Small number of implemented features
- No test for stack overflow
- General *time* command for time measures
Motivation

Twopy

First experiments

Conclusion
Research directions

BBV and object mechanisms

- Collect runtime information to implement objects
- More static implementations for methods and attributes

A lot of engineering effort remains

- Garbage Collection
- Register allocation
Observations

- Python is hard to implement with a JIT due to its dynamism
- The bytecode is a good abstraction, but still too high-level
- Simple bytecode instructions can be implemented efficiently with BBV

Future work

- Interprocedural basic block versioning
- Basic block versioning for objects