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

The Python language

Motivation

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

The Python language

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 ?

Python's implementations

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

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

Basic block versioning

```
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Basic block versioning

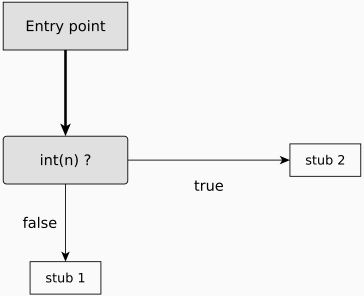
```
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Basic block versioning

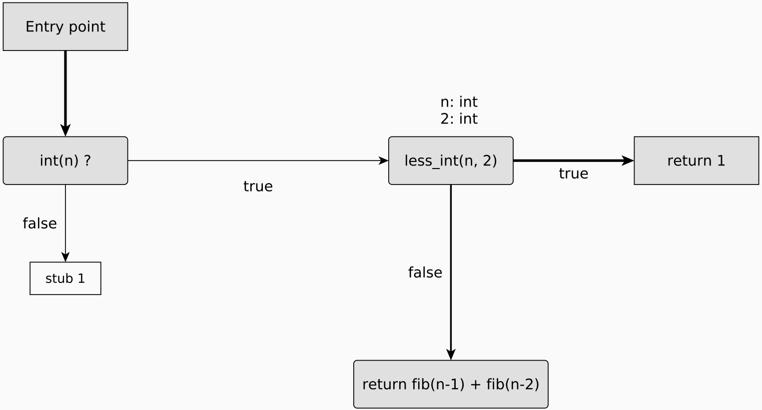
Basic Block Versioning (BBV) [Chevalier-Boisvert and Feeley, 2015]:

- 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



Internal of basic block versioning



Frontend and simple model

Official Python bytecode

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

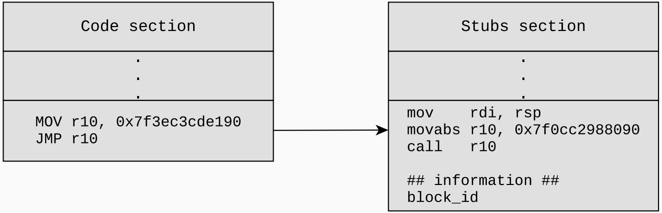
The frontend is low-development effort thanks to Python's API.

x86 backend

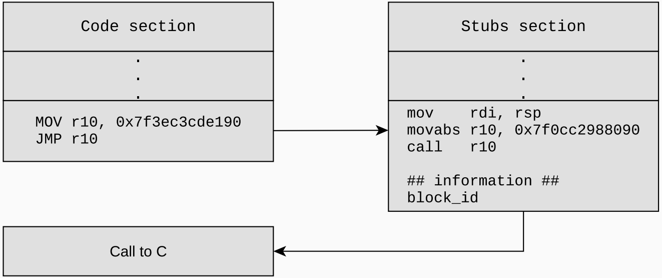
Implementation details

- 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

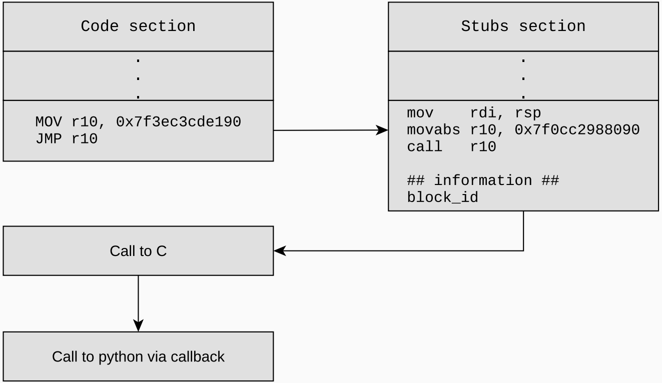
Lazy compilation: stubs



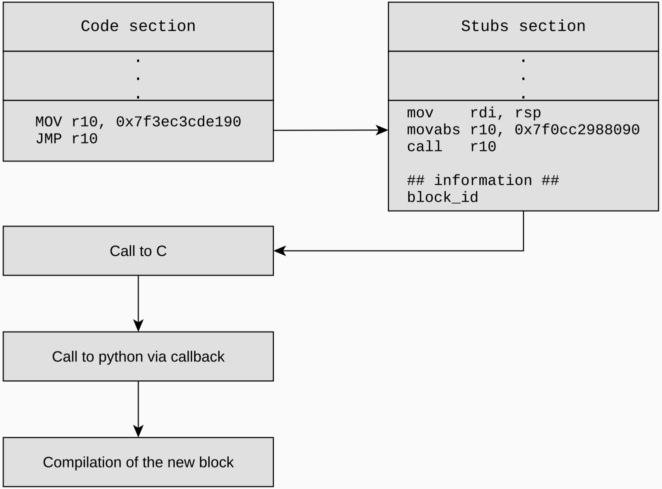
Lazy compilation: stubs



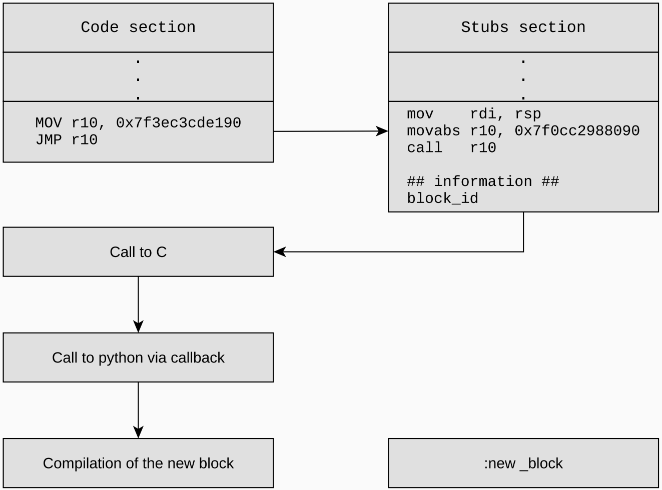
Lazy compilation: stubs



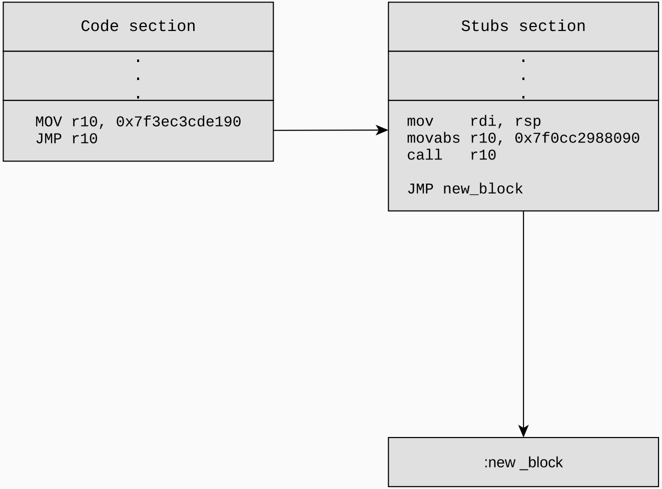
Lazy compilation: stubs



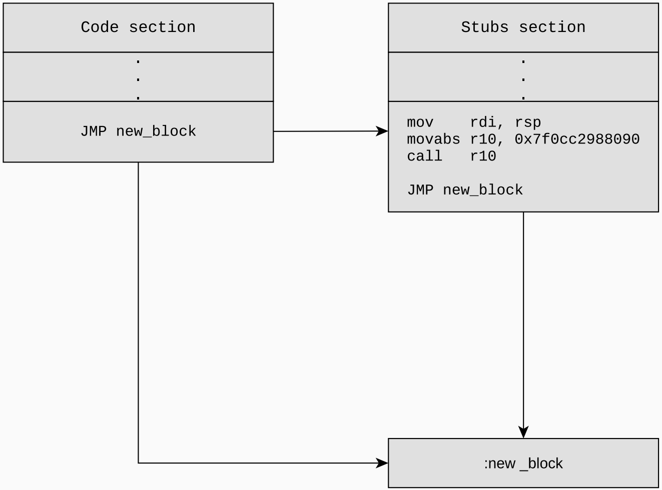
Lazy compilation: stubs



Lazy compilation: stubs



Lazy compilation: stubs



Eliminated tests thanks to BBV

	fibonnacci	for_loops	ack	tak
Without BBV	6623205608	4559999552	1072759792	1773815640
BBV	2649282244	1579999848	268206304	1267011192
Eliminated tests	**60**%	**65**%	**75**%	**29**%

Table 1: Runtime executed type-tests in Twopy

Micro benchmarks

BBV is not fully implemented

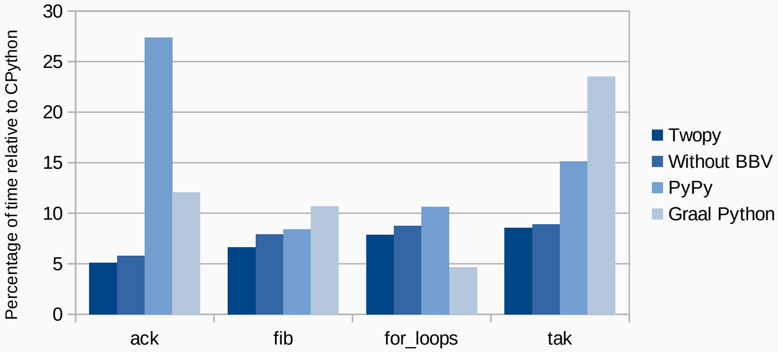


Figure 1: Time results (seconds) relative to CPython

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

Conclusion

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

Bibliography

Chevalier-Boisvert, M. and Feeley, M. (2015).
Simple and effective type check removal through lazy basic block versioning.
In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 101–123.

Dukhan, M. (2013).
Peachpy: A python framework for developing high-performance assembly kernels.
Python for High Performance and Scientific Computing.