pylibjit: A JIT Compiler Library for Python

Gergö Barany

gergo@complang.tuwien.ac.at

Institute of Computer Languages
Vienna University of Technology

KPS 2013

October 1, 2013

This talk

A compiler. . .

- Called from interpreted Python programs
- Running in standard Python interpreter
- Compiling code to run in standard Python interpreter

Architecture (1/3)

'Front end': Python AST compiler
Python convenience layer
Back end: GNU libjit

GNU libjit

- Portable JIT (x86, ARM)/interpreter library
- RISC-like intermediate code API, simple type system
- Python bindings with convenient operator overloading

GNU libjit

- Portable JIT (x86, ARM)/interpreter library
- RISC-like intermediate code API, simple type system
- Python bindings with convenient operator overloading

Example: Function $\lambda xyz.(x * y + z)$

```
def create_signature(func):
    return func.signature_helper(jit.int, jit.int,
                                 jit.int, jit.int)
def build(func):
    x, y, z = (func.get_param(i) for i in range(3))
    product = func.insn_mul(x, y)
    sum = func.insn_add(product, z)
    func.insn_return(sum)
```

GNU libjit

- Portable JIT (x86, ARM)/interpreter library
- RISC-like intermediate code API, simple type system
- Python bindings with convenient operator overloading

Example: Function $\lambda xyz.(x * y + z)$

```
def create_signature(func):
    return func.signature_helper(jit.int, jit.int,
                                 jit.int, jit.int)
def build(func):
    x, y, z = (func.get_param(i) for i in range(3))
    func.insn_return(x * y + z)
```

GNU libjit: Verbose code

```
def build(func):
    # values: n, 1, 2


    # if n < 2: goto return_label

    # return fib(n-1) + fib(n-2)


    # return_label: return n
```

GNU libjit: Verbose code

```
def build(func):
    # values: n, 1, 2
    n = func.get_param(0)
    one = func.new_constant(1, jit.Type.int)
    two = func.new_constant(2, jit.Type.int)
    # if n < 2: goto return_label
    return_label = func.new_label()
    func.insn_branch_if(n < two, return_label)
    # return fib(n-1) + fib(n-2)
    fib_func = func
    fib_sig = func.create_signature()
    a = func.insn_call('fib', fib_func, fib_sig, [n-one])
    b = func.insn_call('fib', fib_func, fib_sig, [n-two])
    func.insn_return(a + b)
    # return_label: return n
    func.insn_label(return_label)
    func.insn_return(n)
```

Invisible control flow!

GNU libjit: Boilerplate

```
class fib_function(jit.Function):
    def __init__(self, context):
        super().__init__(context)
        self.create()

    def create_signature(self):
        return self.signature_helper(jit.int, jit.int)

    def build(func):
        ...

fib = fib_function(context)
fib.__name__ = 'fib'
```

Mostly boring stuff.

Architecture (2/3)

'Front end': Python AST compiler

Python convenience layer

Back end: GNU libjit

Metaprogramming to the rescue!

```
@jit.builder(return_type=jit.Type.int,
             argument_types=[jit.Type.int])
def fib(func):
    n = func.get_param(0)
    one = func.new_constant(1, jit.Type.int)
    two = func.new_constant(2, jit.Type.int)
    with func.branch(n < two) as (false_label, end_label):
        func.insn_return(n)
    # else:
        func.insn_label(false_label)
        func.insn_return(
            func.recursive_call('fib', [n - one]) +
            func.recursive_call('fib', [n - two]))
```

Metaprogramming to the rescue!

```
@jit.builder(return_type=jit.Type.int,
             argument_types=[jit.Type.int])
```

Function decorator:

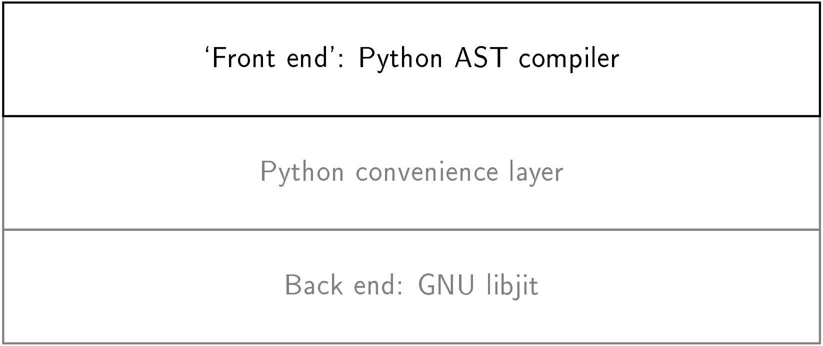
- Attached to function definition
- Arbitrary analyses/transformations on function object
- ➙ Here: hide class definition boilerplate

```
with func.branch(n < two):
```

Context manager:

- Perform actions on entry to/exit from block
- ➙ Here: hide some labels and jumps
- Similar `func.loop`

Architecture (3/3)



Compiling Python

```
@jit.compile(return_type=jit.Type.int,
             argument_types=[jit.Type.int])
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Compiling Python

```
@jit.compile(return_type=jit.Type.int,
             argument_types=[jit.Type.int])
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Obtain source:

```
src = inspect.getsource(function)
```

Compiling Python

```
@jit.compile(return_type=jit.Type.int,
             argument_types=[jit.Type.int])
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Obtain source:

```
    src = inspect.getsource(function)
```

Build AST:

```
    AST = ast.parse(src, mode='exec')
```

Compiling Python

```
@jit.compile(return_type=jit.Type.int,
             argument_types=[jit.Type.int])
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Obtain source:

```
src = inspect.getsource(function)
```

Build AST:

```
AST = ast.parse(src, mode='exec')
```

Compile and profit! (up to 50 × speedup)

pylibjit Features (1/6)

Machine integer and floating-point arithmetic

```
@pyjit.compile(return_type=jit.Type.float64,
               argument_types=[jit.Type.int] * 2)
def eval_A(i, j):
   return 1.0 / (((i + j) * (i + j + 1) >> 1) + i + 1)
```

spectral_norm benchmark: 53 × speedup

pylibjit Features (2/6)

Arrays and lists

```
@pyjit.compile(
    return_type=jit.Type.int,
    argument_types=[jit.Type.array_t(jit.Type.ubyte)])
def array_stuff(an_array):
    an_array[0] += 1
    return len(an_array)
```

pylibjit Features (3/6)

Variables, unboxed for loops

```
@pyjit.compile(
    return_type=jit.Type.void,
    argument_types=[object, ubyte_array, ubyte_array],
    variables={'i': jit.Type.int})
def sub_bytes(self, block, sbox):
    for i in range(16):
        block[i] = sbox[block[i]]
```

AES crypto benchmark: 20 $\times$ speedup

pylibjit Features (4/6)

Arbitrary-precision Python types still available

```
@jit.compile(return_type=jit.Type.int,
             argument_types=[jit.Type.int])
def square_unboxed(n):
    return n * n

@jit.compile(return_type=int, argument_types=[int])
def square_boxed(n):
    return n * n

>>> square_unboxed(2**30)   # 32-bit system
0
>>> square_boxed(2**30)
1152921504606846976
>>> square_boxed(2**300)
41495155688809929585124078636911611510124462322424368999956...
```

pylibjit Features (5/6)

Parallel assignments

```
@jit.compile(...,
             intrinsics={'math.log'})
def mandel_point(i, j, N):
    cx = 2*i / N - 1.5
    cy = 2*j / N - 1
    x, y = 0, 0
    iteration = 0
    max_iteration = 255
    while x*x + y*y < 4 and iteration < max_iteration:
        x, y = x*x - y*y + cx, 2*x*y + cy
        iteration += 1
    if iteration == max_iteration:
        return 1
    else:
        return math.log(iteration) / math.log(2) / 8
```

pylibjit Features (6/6)

Math intrinsics

```
@jit.compile(...,
             intrinsics={'math.log'})
def mandel_point(i, j, N):
    cx = 2*i / N - 1.5
    cy = 2*j / N - 1
    x, y = 0, 0
    iteration = 0
    max_iteration = 255
    while x*x + y*y < 4 and iteration < max_iteration:
        x, y = x*x - y*y + cx, 2*x*y + cy
        iteration += 1
    if iteration == max_iteration:
        return 1
    else:
        return math.log(iteration) / math.log(2) / 8
```

Some other Python compilers

Numba

- Very similar decorator-based JIT
- \+ Good performance (based on LLVM)
- – Too complex for my needs

PyPy

- Tracing Python JIT written in Python
- \+ $1.25\times$–$50\times$ faster than CPython interpreter, $6.3\times$ on average
- – Not compatible with all Python extension libraries

What's this good for?

Possible applications of `pylibjit`:

- Fun metaprogramming exercise!
- Promote idea of compilation of Python fragments
- Basis for hybrid static/dynamic type system research
- Basis for research into Python interpreter performance

Summary

- JIT compiler for Python, in Python, atop Python interpreter
- Nice speedups vs. interpreted code
- Not-entirely-trivial subset of Python supported, more to come

Summary

- JIT compiler for Python, in Python, atop Python interpreter
- Nice speedups vs. interpreted code
- Not-entirely-trivial subset of Python supported, more to come

Thank you!

This work was supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung, FWF) under contract P23303, *Spyculative*.