pylibjit: A JIT Compiler Library for Python

Gergö Barany
Institute of Computer Languages
Vienna University of Technology
gergo@complang.tuwien.ac.at

Abstract

We present pylibjit, a Python library for generating machine code at load time or run time. The library has two distinct modes of use: First, it aims to provide a high-level interface for generating code at run time. This is achieved by using language features such as operator overloading and Python's context managers and decorators.

Second, Python's reflection features allow us to access functions' abstract syntax trees. pylibjit can make use of this to generate machine code for functions originally written in Python. Compiling a Python function can be as easy as attributing it with a decorator providing type information, without changing the function itself in any way. Such compiled functions are executed transparently within interpreted Python programs. This makes it convenient to develop applications in Python and then speed up only the hot spots using compilation.

The development of pylibjit is at a very early stage, but we can already present some working examples. For simple numeric programs, we achieve speedups of up to $50\times$ over standard interpreted Python.

1 Motivation

Interpreted high-level programming languages are often criticized for their comparatively poor performance. Still, they are popular due to their ease of use and high-level features. Various techniques are used to bridge the performance gap to more traditional compiled languages: tracing just-in-time compilers (JITs) [CSR+09, BCFR09], JITs for subsets of languages marked by special annotations [asm], or ahead-of-time compilers that rely on static type annotations [Sel09]. A partial solution is to use interpreters that specialize the program at run time [Bru10].

This paper introduces the pylibjit library, which aims to provide a JIT for fragments of Python code marked by annotations ('decorators' in Python parlance). The compiled code runs within the normal Python interpreter embedded in a traditional Python program; program parts without compiler

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

Figure 1: The naïve Fibonacci function in Python.

annotations are interpreted as usual, while the compiled parts can make use of static type information provided by the developer.

JITs are mostly associated with languages that are compiled to some form of 'bytecode' intermediate representation. However, machine code generation at run time is an important topic even for some programs written in traditional, statically compiled languages. Code generation can be useful for programs that perform expensive computations with control flow that heavily depends on user input. For example, image processing applications can benefit from generating specialized JIT-ed code for user-defined image filters [Pet07]. Even the Linux kernel contains a JIT compiler for a simple domain-specific language describing network packet filtering rules [Cor11].

Our `pylibjit` project builds on a JIT library meant for building code at run time in such applications. However, since Python also allows execution of code at load time (or rather, since there is no clear distinction between 'run time' and 'load time'), we can apply the same library to compile entire Python function definitions as they appear in a source file. The following sections describe how `pylibjit` builds up from a simple JIT library interface to what is a de facto ahead-of-time compiler for a subset of Python.

2 Compiling functions using the low-level API

Our `pylibjit` library builds on the GNU `libjit` just-in-time compiler library[1] and an existing Python wrapper library for it[2].

We will use the naïve Fibonacci function in Figure 1 to illustrate the use of this existing library and our improvements to it. The library exports a class `jit.Function` which users must subclass for each function they wish to build and compile. Figure 2 shows the code needed to build the Fibonacci function using this interface. The core is the `build` function which performs the API calls to build up the intermediate code step by step: API calls provide for computations, labels, conditional and unconditional jumps, and function calls. Note that, since the intermediate code is meant to be compiled to machine code, all computations are typed, although type annotations are

[1]`http://www.gnu.org/software/libjit/`

[2]`https://github.com/eponymous/libjit-python`

```
class fib_function(jit.Function):
    def create_signature(self):
        # return type, argument types
        return self.signature_helper(jit.Type.int, jit.Type.int)

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

Figure 2: Building the Fibonacci function using the low-level JIT interface.

mostly only needed where a value is first introduced; the types of arithmetic and other operators are inferred from their operands. The type name `int` refers to the machine's 'usual' word-sized integer type, as in C.

Note also that already at this level, Python's high-level nature offers some convenience: The arithmetic operators `+` and `-` (and others) are overloaded to work on the compiler's 'value' objects, so we can directly generate an addition instruction by writing `a + b` rather than the less natural `insn_add(a, b)`.

Having created this definition, the class can be instantiated and called as if it were a normal Python function. Wrapper code takes care of unboxing Python argument values and boxing the native code's return value in a Python object.

3 The higher-level API

While this library is usable, it makes building functions more verbose than absolutely necessary. Having to subclass `Function` and implement several methods on it can lead to a lot of boilerplate code; the only interesting function in a typical subclass is `build`, so ideally users should not have to write more than this function. Conveniently, Python provides a concept of *function decorators* that can be used for this purpose.

```
@jit.builder(return_type=jit.Type.int, argument_types=[jit.Type.int])
def fib2(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]))
```

Figure 3: Building the Fibonacci function using the higher-level interface.

A decorator is a callable object that can be attached to a function definition using the `@` operator. After the function has been parsed and compiled to Python bytecode, the decorator is applied to it and can perform any analysis or transformation. Finally, the decorator's return value replaces the original function object. This enables various higher-order programming techniques.

Using this mechanism, it is easy to write a decorator for JIT builder functions that hides all the boilerplate involved in defining a class and instantiating it. This is encapsulated in the `jit.builder` decorator exported by `pylibjit`. This decorator creates an internal class subclassing `jit.Function`, defines the `build` method in that class to call the decorated function provided by the user, and finally takes care of instantiating the JIT-ed function.

Besides boilerplate, another inconvenience when using the pure `libjit` interface is the lack of structure. In Figure 2, the control flow in the generated program is difficult to see as it is implicit in a number of labels and jump statements. We can, however, build constructs for more structured programs using Python's *context managers* combined with its `with` statement. In essence, a context manager is a pair of two functions `__entry__` and `__exit__` which are called when execution enters and leaves a `with` statement, respectively.

This is a good match to the work that must be done to set up a branch or a loop: At the head of the control structure, a branch condition must be evaluated, and at the end a label (and, for loops, a jump back to the loop head) must be generated. `pylibjit` defines context managers `branch` and `loop` for this purpose. Figure 3 shows how the code building the Fibonacci function can be simplified by using a decorator and the `branch` context manager. The context manager's entry function generates the labels needed to distinguish the true and false branches of execution; unfortunately, the abstraction is not perfect because the user must still take care of placing some jumps and labels.

This interface makes it convenient to build functions at run time and, if needed, specialize them to user input that is partly known (such as user-defined filters [Pet07]). However, the API is still too verbose for functions encapsulating algorithms that we do not want to specialize in this way. The next section puts everything together by showing how `pylibjit` can leverage Python's own syntax for specifying compiled functions.

4 Compiling Python functions

As the final step in the development, we note that Python's `inspect` module allows function decorators to access metadata for functions. In particular, it can be used to obtain the function's original source code (if the program is available in source form, which is typically the case); that code can be extracted and passed to the `ast` module to obtain an abstract syntax tree (AST) for the function [BJ13].

We are therefore in a position to write a decorator function which accesses its decorated function's AST and traverses that AST emitting appropriate `pylibjit` instructions for each AST node. That is, we obtain a very simple way to replace interpreted Python functions by compiled code implementing the same semantics. The Python code itself need not change at all: Whether the code is interpreted or compiled depends only on whether an appropriate decorator is present.

A compiled version of the Fibonacci function is shown in Figure 4. Note, again, that apart from the decorator it is identical to the original Python implementation in Figure 1. At any point during development, the decorator can be removed (e. g., by commenting it out) to switch back to the original interpreted function, or inserted again to obtain the benefits of compilation.

As with all the previous examples, this also produces an object that can be called like a function. On our development machine (Intel Atom N270 at 1.60 GHz running Linux 3.2.0), it takes about 8.2 seconds to evaluate `fib(32)` for the original Python version, while the compiled version takes about 0.155 seconds, for a speedup of about 53 ×.

```
@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)
```

Figure 4: The compiled naïve Fibonacci function.

```
def eval_A (i, j):
    return 1.0 / (((i + j) * (i + j + 1) >> 1) + i + 1)

def eval_A_times_u (u, results):
    u_len = len (u)
    for i in range (u_len):
        partial_sum = 0
        j = 0
        while j < u_len:
            partial_sum += eval_A(i, j) * u[j]
            j += 1
        results[i] = partial_sum
```

Figure 5: The `spectral_norm` benchmark.

5 Larger benchmarks

The ultimate goal for `pylibjit` is to be able to compile a nontrivial subset of Python to speed up the innermost loops of computationally intensive programs. While development is at a very early stage, we can already use it to speed up some interesting benchmark applications. The decorator-based approach with an embedding in the Python interpreter is convenient for incremental development of the compiler: We never need to worry about handling all the intricacies of the Python language, only the features actually used in the few functions of interest.

At the time of writing, we can compile Python functions containing arithmetic, branching, counting loops, function calls, and accesses to tuples, lists, and arrays. Compilation is implemented using an AST traversal comprising about 600 lines of code. Here we present results for two benchmark programs in which the hot spots use only these features.

5.1 Case study: The `spectral_norm` benchmark

First, consider the implementation of the core of the `spectral_norm` benchmark[3] in Figure 5. The `eval_A_times_u` function takes two arrays of floating-point numbers as its arguments and populates the second based on values from the first. The other half of the benchmark is an almost identical function that differs only in that the arguments `i` and `j` to `eval_A` are swapped.

Running this benchmark with an input of 1000 (denoting the length of the input array) takes 190.7 seconds using the Python interpreter. We can cause the core functions to be compiled by `pylibjit` by defining a decorator

[3]`http://benchmarksgame.alioth.debian.org/u32/program.php?test=spectralnorm&lang=python3&id=8`

that declares the types of the arguments and variables used within each function.

Decorating only the `eval_A` function, the benchmark's performance deteriorates to 304.9 seconds, a 1.6 × slowdown. This is caused by the fact that calls from Python to code compiled by pylibjit are expensive. Such calls follow a slow path in the interpreter in which the code to be called is looked up in an object's `__call__` slot. The wrapper code must then check and unbox the Python argument objects and finally box up the function's result in a Python object. It is therefore not worthwhile to compile only very small leaf procedures such as `eval_A`.

However, if we also compile the outer `eval_A_times_u` function and its almost identical twin, we obtain a version of the program that runs in 3.5 seconds. This is a speedup of about 53 × over the Python interpreter. The time for the compiled version includes JIT compilation time, which for this benchmark is on the order of 0.05–0.1 seconds. For comparison, the C version of this benchmark compiled with GCC 4.6.3 takes about 3.4 seconds when compiled at `-O1`, 2.8 seconds at `-O2`, and 2.6 seconds at `-O3`. This shows that for numerical computations on array-like structures we can sometimes come close to the performance of optimized C code while enjoying the benefits of developing in Python.

5.2 Case study: The `AES` benchmark

As a larger and more realistic application, we compiled parts of a Python implementation of the AES encryption algorithm[4]. AES encryption and decryption consists essentially of XOR operations and permutations of arrays of bytes. Figure 6 shows two representative functions. Compiling these and a few other similar leaf procedures which together account for 81 % of execution time, `pylibjit` improves benchmark runs from 12.4 seconds (interpreted) to 3.42 seconds for a speedup of about 3.6 ×. (In other words, 72 % of execution time is optimized away.)

This can be improved further by also compiling the outer loops that call such leaf functions. In this implementation of AES, the entire algorithm is encapsulated in a Python class, and the calls to the auxiliary functions are therefore implemented as virtual method calls. However, in the context of this benchmark, the targets for these calls can, in principle, always be determined statically. `pylibjit` allows users to annotate such functions to resolve targets at compile time. This is useful because it saves not only lookup time but also surprisingly expensive boxing and unboxing operations on method objects [Bar13].

Applying this user-guided static devirtualization when compiling the encryption and decryption driver functions, we obtain a total benchmark ex-

[4]Adapted to Python 3 from a version available from `https://bitbucket.org/pypy/benchmarks/src`

```
def add_round_key(self, block, round):
    offset = round * 16
    exkey = self.exkey
    for i in range(16):
        block[i] ^= exkey[offset + i]

def sub_bytes(self, block, sbox):
    for i in range(16):
        block[i] = sbox[block[i]]
```

Figure 6: Two of the hottest functions in the `AES` benchmark, showing simple manipulation of arrays of bytes.

ecution time of only 0.607 seconds. This corresponds to a total speedup of about $20\times$ versus interpretation.

6 Conclusions and future work

We presented `pylibjit`, a Python wrapper for the GNU `libjit` just-in-time compiler library. Besides just exposing the underlying API, `pylibjit` allows the use of decorators to cause existing Python functions to be compiled to machine code. At the time of writing, the library supports a fragment of Python supporting integer and floating-point arithmetic, counting loops, function calls, and accesses to Python tuples, lists, and arrays.

The current version of `pylibjit` is not fit for general use, but the compiler is simple and extensible and will soon grow more features as needed. Since the compiled code runs within the Python interpreter, all of the functions used internally by Python to implement operations are accessible and can be called from the compiled code. This means that to support more language features, `pylibjit` need only mimic the sequence of internal API calls that the interpreter would itself execute for a given program. However, the presence of type annotations means that we can elide certain operations that add overhead, such as dynamic typechecks and boxing/unboxing operations.

A more complete version of `pylibjit` will be made available through the author's website at `http://www.complang.tuwien.ac.at/gergo/`.

Acknowledgements

This work was supported by the Austrian Science Fund (FWF) under contract no. P23303, Spyculative.

References

[asm] asm.js: an extraordinarily optimizable, low-level subset of JavaScript. `http://asmjs.org/`.

[Bar13] Gergö Barany. Static and dynamic method unboxing for Python. In *6. Arbeitstagung Programmiersprachen (ATPS 2013)*, included in volume 215 of *Lecture Notes in Informatics (LNI)*. GI - Gesellschaft für Informatik, February 2013.

[BCFR09] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[BJ13] David Beazley and Brian K. Jones. *Python Cookbook, 3rd Edition*, chapter Parsing and Analyzing Python Source. O'Reilly, 2013.

[Bru10] Stefan Brunthaler. Efficient interpretation using quickening. In *Proceedings of the 6th symposium on Dynamic languages*, DLS '10, pages 1–14, New York, NY, USA, 2010. ACM.

[Cor11] Jonathan Corbet. A JIT for packet filters. `https://lwn.net/Articles/437981/`, 2011.

[CSR$^+$09] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 71–80, New York, NY, USA, 2009. ACM.

[Pet07] Charles Petzold. On-the-fly code generation for image processing. In Andy Oram and Greg Wilson, editors, *Beautiful Code*. O'Reilly, 2007.

[Sel09] Dag Sverre Seljebotn. Fast numerical computations with Cython. In *Proceedings of the 8th Python in Science conference (SciPy 2009)*, 2009.