Adventures in JIT compilation: Part 4 - in Python (https://eli.thegreenplace.net/2017/adventures-in-jit-compilation-part-4-in-python/)

May 10, 2017 at 05:31 **Tags** Python (https://eli.thegreenplace.net/tag/python) , Compilation (https://eli.thegreenplace.net/tag/compilation) , Code generation (https://eli.thegreenplace.net/tag/code-generation)

This is part 4 in a series of posts on writing JIT compilers for BF.

- Part 1 - an interpreter (https://eli.thegreenplace.net/2017/adventures-in-jit-compilation-part-1-an-interpreter/)
- Part 2 - an x64 JIT (https://eli.thegreenplace.net/2017/adventures-in-jit-compilation-part-2-an-x64-jit/)
- Part 3 - LLVM (https://eli.thegreenplace.net/2017/adventures-in-jit-compilation-part-3-llvm/)

Even though, for performance considerations, most JIT compilers are written in down-to-metal languages like C and C++, in some scenarios it could be useful to write them in higher-level languages as well [1].

This post intends to do just that, by presenting yet another JIT for BF - this time written in Python. It presents both a low-level approach to JITing native machine code, and using a higher-level library to perform instruction encoding. The JIT implemented here is not optimized (equivalent to the "simple" JIT from part 2), so I'm not going to discuss any performance results. The idea is just to show how JITing in Python works.

How to JIT from Python

To be able to JIT-compile BF, we should first be capable of JITing any code at all. It's worth showing how to do this in plain Python without any third-party libraries. In part 2 of the series (https://eli.thegreenplace.net/2017/adventures-in-jit-compilation-part-2-an-x64-jit/) we've used the same approach as presented in my older How to JIT (https://eli.thegreenplace.net/2013/11/05/how-to-jit-an-introduction) post for basic JITing by copying machine code into an executable memory segment and jumping to it.

That was done in C and C++, but it turns out it's not much harder in Python due to the magic of `ctypes`. I've written about doing runtime calls to C code via ctypes before (https://eli.thegreenplace.net/2013/03/09/python-ffi-with-ctypes-and-cffi), and if you're not familiar with this wonderful tool, I encourage you to read more about it. What follows is a complete code sample that implements the JITing presented in the "How to JIT" post; it's heavily commented, so it should be reasonably clear what's going on:

```
import ctypes
import mmap

# Equivalent to dlopen(NULL) to open the main process; this means the Python
# interpreter, which links the C library in. Alternatively, we could open
# libc.so.6 directly (for Linux).
libc = ctypes.cdll.LoadLibrary(None)

# Grab the mmap foreign function from libc, setting its signature to:
#
#     void *mmap(void *addr, size_t length, int prot, int flags,
#                int fd, off_t offset)
#
# Per the mmap(2) man page.
mmap_function = libc.mmap
mmap_function.restype = ctypes.c_void_p
mmap_function.argtypes = (ctypes.c_void_p, ctypes.c_size_t,
                          ctypes.c_int, ctypes.c_int,
                          ctypes.c_int, ctypes.c_size_t)

CODE_SIZE = 1024

# Allocate RWX memory with mmap. Using mmap from libc directly rather than
# Python's mmap module here because the latter returns a special "mmap object"
# and we just need the raw address of the mapping. However, we will use the
# PROT_* and MAP_* constants from the mmap module rather than duplicating them.
code_address = mmap_function(None, CODE_SIZE,
                             mmap.PROT_READ | mmap.PROT_WRITE | mmap.PROT_EXEC,
                             mmap.MAP_PRIVATE | mmap.MAP_ANONYMOUS,
                             -1, 0)

if code_address == -1:
    raise OSError('mmap failed to allocate memory')

# Emit code into the allocated address. This sequence of x86-64 machine code
# encodes:
#
#  mov %rdi, %rax
#  add $4, %rax
#  ret
code = b'\x48\x89\xf8\x48\x83\xc0\x04\xc3'
```

```
assert len(code) <= CODE_SIZE
ctypes.memmove(code_address, code, len(code))

# Declare the type for our JITed function: long (*JitFuncType)(long), and cast
# code_address (which is a void*) to this type so we could just call it.
JitFuncType = ctypes.CFUNCTYPE(ctypes.c_long, ctypes.c_long)
function = ctypes.cast(code_address, JitFuncType)
print(function(-100))
```

If you execute this code it will print -96, since the JITed function adds 4 to its argument.

From this point on, it should be easy to implement a full JIT-compiler for BF using hand-rolled instruction encoding, just like in part 2 (https://eli.thegreenplace.net/2017/adventures-in-jit-compilation-part-2-an-x64-jit/). Feel free to do it as an exercise :) You'll face the same issue discussed in that post - manually encoding instructions and resolving labels/branches is tedious, and it would be nice if a library could do it for us.

PeachPy

Enter PeachPy (https://github.com/Maratyszcza/PeachPy) - an x86-64 assembler embedded in Python. It's a relatively new project mostly aimed at writing highly-optimized assembly kernels for numerical computations without leaving the domain of Python. It even has a cute logo:

PeachPy has some examples strewn around the web, and my code for this post can serve as another example. Let's start by replicating the simple JIT functionality shown above. PeachPy takes care of all the memory mapping behind the scenes [2], so the full code we need to write is just this:

```
import peachpy
import peachpy.x86_64

x = peachpy.Argument(peachpy.int64_t)

with peachpy.x86_64.Function("Add4", (x,), peachpy.int64_t) as asm_function:
    peachpy.x86_64.LOAD.ARGUMENT(peachpy.x86_64.rax, x)
    peachpy.x86_64.ADD(peachpy.x86_64.rax, 4)
    peachpy.x86_64.RETURN(peachpy.x86_64.rax)

abi = peachpy.x86_64.abi.detect()
encoded_function = asm_function.finalize(abi).encode()
python_function = encoded_function.load()

# The JIT call
print(python_function(-100))
```

PeachPy lets us write our code in *assembly* rather than directly in machine code, and it also handles the loading for us. As we'll see soon, it lets us use symbolic labels (just like `asmjit`) and takes care of jump target resolution automatically. In other words, it's just the tool we need to focus on our domain - compiling BF - without worrying too much about the low level details.

Note how PeachPy uses Python's context managers (a very common approach for DSLs embedded in Python). The `with peachpy.x86_64.Function` statement creates a context manager within which all assembly instructions (like `ADD` and `RETURN`) are appended to the function.

JIT-compiling BF via PeachPy

With the `Add4` sample above, we actually already have most of the building blocks we need to JIT-compile BF. What remains is just a matter of digging in PeachPy's source and examples (sadly there's very little documentation) to figure out how to invoke and properly use its assembly primitives. The following is very similar to how `simpleasmjit` is implemented in part 2. The full code sample is available here (https://github.com/eliben/code-for-blog/blob/master/2017/bfjit/peachpyjit.py); I'll just highlight the important snippets of code. First the function "declaration":

```
# Create a JITed function named "ppjit", with the C-style signature:
#   void ppjit(uint8_t* memptr)
#
memptr = peachpy.Argument(peachpy.ptr(peachpy.uint8_t))

with peachpy.x86_64.Function("ppjit",
                             [memptr],
                             result_type=None) as asm_function:
    # Use r13 as our data pointer; initially it points at the memory buffer
    # passed into the JITed function.
    dataptr = peachpy.x86_64.r13
    peachpy.x86_64.LOAD.ARGUMENT(dataptr, memptr)
```

In this code we'll be passing `memory` from the host side (from Python, in this case), so the function signature is `void (*ppjit)(uint8_t*)`. We then give `r13` the symbolic name `dataptr`. The usual instruction-by-instruction BF compilation loop follows:

```
for pc, instr in enumerate(parse_bf_program(bf_file), start=1):
    if instr == '>':
        peachpy.x86_64.ADD(dataptr, 1)
    elif instr == '<':
        peachpy.x86_64.SUB(dataptr, 1)
    elif instr == '+':
        peachpy.x86_64.ADD([dataptr], 1)
    elif instr == '-':
        peachpy.x86_64.SUB([dataptr], 1)
```

PeachPy dresses Python in assembly-like flavor. Registers placed in brackets like `[dataptr]` imply dereferencing. While `dataptr` refers to the value of the `r13` register itself, `[dataptr]` refers to the value of the memory word whose address is held in `dataptr` - this is similar to the syntax of many assembly languages.

For emitting I/O operations, we resort to syscalls again [3]:

```
elif instr == '.':
    # Invoke the WRITE syscall (rax=1) with stdout (rdi=1).
    peachpy.x86_64.MOV(peachpy.x86_64.rax, 1)
    peachpy.x86_64.MOV(peachpy.x86_64.rdi, 1)
    peachpy.x86_64.MOV(peachpy.x86_64.rsi, dataptr)
    peachpy.x86_64.MOV(peachpy.x86_64.rdx, 1)
    peachpy.x86_64.SYSCALL()
elif instr == ',':
    # Invoke the READ syscall (rax=0) with stdin (rdi=0).
    peachpy.x86_64.MOV(peachpy.x86_64.rax, 0)
    peachpy.x86_64.MOV(peachpy.x86_64.rdi, 0)
    peachpy.x86_64.MOV(peachpy.x86_64.rsi, dataptr)
    peachpy.x86_64.MOV(peachpy.x86_64.rdx, 1)
    peachpy.x86_64.SYSCALL()
```

For emitting the loops, we use a stack of PeachPy `Label` objects with one label for the loop and another for the "after loop". Again, this is *very* similar to how it was done with `asmjit` in part 2 (https://eli.thegreenplace.net/2017/adventures-in-jit-compilation-part-2-an-x64-jit/).

```
elif instr == '[':
    # Create labels for the loop start and after-loop.
    loop_start_label = peachpy.x86_64.Label()
    loop_end_label = peachpy.x86_64.Label()
    # Jump to after the loop if the current cell is 0.
    peachpy.x86_64.CMP([dataptr], 0)
    peachpy.x86_64.JZ(loop_end_label)
    # Bind the "start loop" label here.
    peachpy.x86_64.LABEL(loop_start_label)
    open_bracket_stack.append(
        BracketLabels(loop_start_label, loop_end_label))
elif instr == ']':
    if not len(open_bracket_stack):
        die('unmatched closing "]" at pc={}'.format(pc))
    labels = open_bracket_stack.pop()
    # Jump back to loop if the current cell is not 0.
    peachpy.x86_64.CMP([dataptr], 0)
    peachpy.x86_64.JNZ(labels.open_label)
    # Bind the "after-loop" label here.
    peachpy.x86_64.LABEL(labels.close_label)
```

Finally, this is how the JITed function is invoked:

```
# Allocate memory as a ctypes array and initialize it to 0s. Then perform
# the JIT call.
memsize = 30000
MemoryArrayType = ctypes.c_uint8 * memsize
memory = MemoryArrayType(*([0] * memsize))
```

There's just a little bit of trickiness here in our usage of `ctypes`. Since we have to pass arguments to C functions, `ctypes` lets us declare C-like types like `MemoryArrayType` which is a `uint8_t[30000]`. The funky syntax on the following line is nothing more than Python's `*args` destructuring of a list of 30000 zeros. So `memory` is now an object we can safely pass to our JITed function which expects a `uint8_t*` argument:

```
python_function(memory)
```

This calls the JITed function, the result of which is I/O and modified memory cells in `memory`.

PeachPy - more features and some limitations

My usage of PeachPy in this post has been fairly limited, and the library has many more features. For example:

1. Since PeachPy was mainly developed to write fast mathematical kernels, it has fairly good support for the newest vector extensions like AVX512.
2. There's some support for automatic register allocation. In the BF JIT, the `r13` register is used directly, but we don't *really* care which register it is. We could ask PeachPy for a "general purpose 64-bit register" and it would allocate one for us. When writing a much more complicated piece of assembly code, this can be very useful.
3. There's also some support for generating ARM code, though I don't know how mature it is.

However, PeachPy is also a fairly new library which means there are still limitations and rough edges. When developing the JIT I ran into a bug and sent a PR (https://github.com/Maratyszcza/PeachPy/pull/72) - which was promptly accepted. PeachPy also doesn't do well with a large number of assembly instructions. It has some recursive analysis logic that blows up the Python stack when run on large code

(https://github.com/Maratyszcza/PeachPy/issues/74). In the BF JIT, I'm setting the stack to a higher limit artificially; with that, PeachPy doesn't crash but takes quite a while to assemble large programs like Mandelbrot.

According to PeachPy's maintainer, this is due to the design of PeachPy being aimed towards writing assembly code manually rather than generating it from some other language. Fair enough - but definitely something to keep in mind. All in all, the maintainer is responsive and the library seems to be improving quickly - so these limitations may go away in the future.

Python JIT alternative - llvmlite

A somewhat higher-level alternative to JITing from Python is using `llvmlite`, the new Python binding to LLVM. I wrote about llvmlite before (https://eli.thegreenplace.net/2015/building-and-using-llvmlite-a-basic-example/) and also ported the LLVM official tutorial to Python using it (https://eli.thegreenplace.net/2015/python-version-of-the-llvm-tutorial/).

`llvmlite` would definitely do the job here (and most likely not have the limitations of PeachPy), but I wanted to go with something different in this part. After all, a BF JIT with LLVM was already covered in part 3 (https://eli.thegreenplace.net/2017/adventures-in-jit-compilation-part-3-llvm/), and `llvmlite` is a binding to the same library, just in Python. PeachPy offers an altenative approach for machine code generation from Python - with more direct control of the emitted instructions, though none of the optimizations LLVM provides automatically.

---

[1] For example Numba (http://numba.pydata.org/) - a JIT compiler for numeric code written in Python; it takes Python code that uses Numpy and JIT-compiles it to optimized native code.

[2] No magic is involved here - in fact PeachPy uses precisely the same approach as I've shown to invoke `mmap` from Python via `ctypes`.

[3] As an exercise, modify this code to invoke `putchar` and `getchar`.

For comments, please send me ✉ an email (mailto:eliben@gmail.com).
