Assignment 9: More BF—JIT Compilation

Cayden Lund (u1182408)

28 October 2024

Repository: https://github.com/caydenlund/brainforge

JIT Compilation: Initial Implementation

The concept of just-in-time compilation is simple: to run a block of code, compile it and then jump to the newly-compiled instructions. The first task was to set up an executable region of memory, which was actually quite straightforward, with a call to `posix_memalign` and then `mprotect`:

```
let jit_mem: *mut u8 = unsafe {
    let mut contents: MaybeUninit<*mut libc::c_void> = MaybeUninit::uninit();
    libc::posix_memalign(contents.as_mut_ptr(), PAGE_SIZE, capacity);
    let contents = contents.assume_init();
    libc::mprotect(
        contents,
        capacity,
        libc::PROT_EXEC | libc::PROT_READ | libc::PROT_WRITE,
    );
    mem::transmute(contents)
};
```

With this executable region of memory, in order to run a JIT-compiled program, we simply need to encode the instructions, cast the memory region as a function pointer, and call the function.

My first tests were simple: I wanted to make sure that I could call the function with an argument and return a value without a segmentation fault. I wrote a few tests in assembly and used the GNU assembler to get the raw hex bytes, which I copied into the executable memory region before calling it. Here's an example of a typical one of these tests:

```
// |x| { 2*x + 1 };
let bytes: Vec<u8> = vec![
    // mov eax, edi
    0x89, 0xF8,
    // imul eax, 2
    0x6B, 0xC0, 0x02,
    // add eax, 1
    0x83, 0xC0, 0x01,
    // ret
    0xC3
];
for index in 0..bytes.len() {
    jit_mem[index] = bytes[index];
}
let fn_ptr: *mut fn(libc::c_int) -> libc::c_int =
    unsafe { mem::transmute(jit_mem) };
assert_eq!(fn_ptr(3), 7);
```

The real challenge came next: encoding the assembly instructions in binary. This took hours upon hours of poring over the programmer's manuals for the x86-64 ISA. Most things were clearer in AMD's document, but some things were clearer in Intel's document, so consulting both together proved to be the most effective approach to understanding the encoding for each instruction.

I defined several key new data structures to represent instructions, operands, and registers. These new data structures are shown below (slightly simplified):

```
pub enum AMD64Register {
    AL, AX, EAX, RAX,
    CL, CX, ECX, RCX,
    DL, DX, EDX, RDX,
    // ...

    R8B, R8W, R8D, R8,
    R9B, R9W, R9D, R9,
    R10B, R10W, R10D, R10,
    // ...

    XMM0, YMM0,
    XMM1, YMM1,
    XMM2, YMM2,
    // ...
}

pub enum MemorySize { Byte, Word, DWord, QWord, XMMWord, YMMWord }
```

```rust
pub enum AMD64Operand {
    Register(AMD64Register),
    Immediate(isize),
    Memory(
        Option<MemorySize>, Option<AMD64Register>, // mem. size, base reg.
        Option<AMD64Register>, Option<u8>,         // index reg., index scale
        Option<i32>                                // displacement
    ),
}

pub enum AMD64Instruction {
    /// `call <function>`
    Call(Function),
    /// `je <offset>` or `je <label>`
    Je(isize, Option<String>),
    /// `jmp <offset>` or `jmp <label>`
    Jmp(isize, Option<String>),
    /// `jne <offset>` or `jne <label>`
    Jne(isize, Option<String>),

    /// `add <dst>, <src>`
    Add(AMD64Operand, AMD64Operand),

    /// `bsf <dst>, <src>`
    Bsf(AMD64Operand, AMD64Operand),
    /// `bsf <dst>, <src>`
    Bsr(AMD64Operand, AMD64Operand),
    /// `cmovge <dst>, <src>`
    Cmovge(AMD64Operand, AMD64Operand),
    /// `cmp <dst>, <src>`
    Cmp(AMD64Operand, AMD64Operand),
    /// `imul <dst>, <src>`
    Imul(AMD64Operand, AMD64Operand),
    /// `lea <dst>, <src>`
    Lea(AMD64Operand, AMD64Operand),
    /// `mov <dst>, <src>`
    Mov(AMD64Operand, AMD64Operand),
    /// `movzx <dst>, <src>`
    Movzx(AMD64Operand, AMD64Operand),
    /// `xor <dst>, <src>`
    Xor(AMD64Operand, AMD64Operand),

    // (AVX instructions omitted for brevity...)

    /// `push <src>`
    Push(AMD64Operand),
    /// `pop <dst>`
    Pop(AMD64Operand),
    /// `ret`
    Ret(),
}
```

With these new data structures, I was effectively able to share a lot of code between the AOT compiler and the JIT compiler. In fact, the main difference between the two compilers is whether `.to_string()` or `.to_binary()` is called on these instructions, and whether the result is written to an assembly file or the executable region of memory.

For each of these instruction variants, and for each variant of the included operands, I had to write code to encode the instruction as a vector of bytes. Once I had done this a few times, I was able to identify the code that could be reused between different instructions, and pulled that code out into dedicated functions, along with a couple of helper data types: `Rex` (for the REX prefix), `ModRm` (for the ModR/M operand byte), `Sib` (for the SIB operand byte). With these helper methods and data structures, many instructions were as simple as chaining together function calls and an opcode. Here's an example of encoding `Add(Register(_), Register(_))` instructions:

```
match self {
    Add(dst, src) => {
        match (dst, src) {
            // `add <reg>, <reg>`
            (Register(dst_reg), Register(src_reg)) => {
                // Registers must the same size
                if dst_reg.size() != src_reg.size() {
                    return self.encoding_err();
                }

                // 16-bit operand prefix: Option<u8>
                let prefix_reg_16 = (dst_reg.size() == 16).then_some(0x66);

                // REX prefix: Option<u8>
                let rex = self.encode_rex(Some(src), Some(dst))?;

                let opcode: u8 = if dst_reg.size() == 8 { 0x00 } else { 0x01 };

                // ModR/M byte: Vec<u8>
                let rmi =
                    self.encode_reg_rmi(Some(src), Some(dst), dst_reg.size())?;

                // Simply string together `prefix_reg_16`, `rex`, `opcode`, rmi`
                Ok(vec![prefix_reg_16, rex, Some(opcode)]
                    .into_iter()
                    .flatten()
                    .chain(rmi)
                    .collect())
            }
        }
        // ...
    }
    // ...
}
```

Once I was able to encode all of the assembly instructions in binary, write them to the executable memory, and call the memory as a regular function, the JIT compiler was functional.

Truly Compiling Blocks Just-in-Time

Once I had a functional JIT compiler, I was tempted to just call it good enough, but I wasn't quite satisfied: rather than compiling blocks of instructions immediately before executing them, I was compiling the entire program upfront, which I felt wasn't exactly in line with what a true JIT compiler does.

I initially wanted to compile each *basic block* of instructions independently, where each basic block has exactly one entry and one exit point. In BF, basic blocks begin and end at loop boundaries, and so a loop that has nested loops will be broken into multiple basic blocks. This makes it very difficult to encode jump instructions, because the jump instructions are encoded using the exact number of displacement bytes from the end of the jump instruction. Because the AMD64 instruction width is variable, I would need to encode all of the instructions between a jump and its target in order to known the displacement. My initial approach was to—instead of jumping to an unencoded target—return from the JIT function with an integer ID of the basic block destination, and then the Rust code would encode all of the basic blocks in the middle before handing execution back to the JIT-compiled code. After working on that for some time, though, I decided to just keep things simple and encode an outermost loop all together as a single block. By doing this, I didn't need to do any crazy tricks to compile blocks out-of-order or to dynamically patch jump targets.

Following this decision, I realized that by using this approach, the JIT code will never backtrack from one block to a previous block, because there weren't any loops split across multiple blocks. Therefore, this implementation just wrote over previously-compiled instructions, because they would never be used again.

I just had one task left to make this JIT compiler functional: preserve the memory tape pointer across function calls. My original strategy was to pass a pointer to a pointer to the memory tape location as an argument when calling the JIT code. I saved this pointer into register `R14`, and dereferenced this pointer into register `R12`, which was used as the memory tape pointer throughout the JIT code. At the end of the function, the code would save the current memory tape pointer from `R12` into the memory location pointed to by register `R14`. In this manner, the memory tape pointer was preserved, and it worked, but I pretty quickly realized that I could just return the memory tape pointer in register `RAX` when breaking out of the JIT code. The reason why I didn't think of doing this at first was just because I had originally intended to return an integer ID of a basic block for the purposes of jumping, but because I wasn't doing that anymore, I could just return the memory tape pointer, and then pass the new memory tape pointer on the next call to the function.

This JIT compiler worked great. The performance was great, and I was really happy with how it didn't encode a block of instructions until it was about to execute it.

The JIT driver looked like this:

```
let fn_ptr: *mut fn(*mut u8) -> *mut u8 =
    unsafe { mem::transmute(fn_mem.contents) };

// When called, copy the given memory tape location (function argument)
// into register R12.
let fn_prologue = AMD64Instruction::encode_block(&vec![
    Mov(Register(R12), Register(RDI)),
])?;
let prologue_len = fn_prologue.len();
fn_mem.extend(fn_prologue.into_iter());

let instr_blocks =
    AMD64Instruction::convert_instructions(instrs)
    .into_iter()
    .rev()  // (Reverse because we pop from the end)
    .collect::<Vec<_>>();

// A mutable pointer to the current location in the memory tape.
// Initially, this is the center of the memory tape.
let mut memory_ptr = memory_center;

// This will be added to the end of the function every time a new block
// is encoded, returning the current memory tape pointer in register RAX.
let fn_epilogue = AMD64Instruction::encode_block(&vec![
    Mov(Register(RAX), Register(R12)),
    Ret(),
])?;

// For each block of instructions in the program:
while let Some(block) = self.instr_blocks.pop() {
    // JIT-compile the instructions in the new block
    let bytes = AMD64Instruction::encode_block(&*block)?;

    // Save the encoded instructions to the executable memory,
    // overwriting the old instructions.
    self.fn_mem.position = prologue_len;
    self.fn_mem.extend(bytes.into_iter());
    // Also, add the epilogue that returns the new memory pointer
    self.fn_mem.extend(fn_epilogue.clone().into_iter());

    // Finally, we can just call this as an FFI function
    memory_ptr = fn_ptr(memory_ptr);
}
```

Preserving JIT-Compiled Blocks

I ended up deciding that I want to come back to this and only encode one basic block at a time, but that will take more time than I have before the assignment is due. In the meantime, I decided to not overwrite previously-compiled instructions, and instead jump over them.

This was easily done by encoding an unconditional `jmp` instruction immediately after the function prologue, which jumps directly to the block to execute. Every time I encoded a new block, I would overwrite the `jmp` instruction's destination displacement. This was a relatively minor change, so I won't paste the entire code again, but the changes are visible in the following section.

Fixing a Key Bug

At this point, the JIT compiler worked great, and I was very happy with it. When compiled in the (default) "debug" profile, it ran and produced the right outputs for every single test in the `bfcheck` suite, and it performed quite quickly, too.

I then compiled the binary in "release" mode, which uses several optimizations, because I wanted to see whether there were significant speedups. The resulting binary worked on all of the benchmarks in the `bf-benchmark` repository, but it crashed with a segmentation fault on two of the tests in the `bfcheck` suite. I spent a couple of all-nighters trying to figure it out. Was it optimizing away a memory access that I needed to somehow mark as "volatile"? Was there undefined behavior somewhere in my program? I spent hours comparing the disassembled binaries, and I used all of the sanitizers I could find. I tried to make smaller reproducible examples, but I couldn't identify any kinds of patterns in working and non-working tests. I increased the size of the JIT memory space. I increased the size of the memory tape. Confusingly, when I ran the program in GDB, the program ran normally, so I couldn't identify exactly where the problem was coming from. I figured that I must be corrupting the memory somehow, but I couldn't figure out what the source of the issue could be.

At some point, I was thinking about the System V ABI specification on registers, and remembered *why* I used registers `R12` and `R13` throughout my instructions: they're preserved across function calls, so when I called `getchar` and `putchar` it wouldn't mess up my memory tape pointers. I didn't bother to save and restore these registers in my AOT-compiled program because when the BF function finishes, the program exits. This assumption isn't valid in the JIT-compiled code, though, so I was overwriting the `R12` and `R13` registers with my own values without restoring them. When my compiler was compiled in release mode, the driver code uses a pointer to some memory in `R12`, so when I was overwriting it with a pointer to the memory tape and not restoring it, it caused a program crash. I added "`push r12`" and "`push r13`" to the function prologue, and "`pop r13`" and "`pop r12`" in the function epilogue. Once I did this, I had no more problems.

The driver code with these changes (plus the unconditional `jmp` described above) looks like this:

```
let fn_ptr: *mut fn(*mut u8) -> *mut u8 =
    unsafe { mem::transmute(fn_mem.contents) };

// When called, push the callee-saved registers and copy the given
// memory tape location (function argument) into register R12.
let fn_prologue = AMD64Instruction::encode_block(&vec![
    Push(Register(R12)),
    Push(Register(R13)),
    Mov(Register(R12), Register(RDI)),
])?;
fn_mem.extend(fn_prologue.into_iter());

let instr_blocks =
    AMD64Instruction::convert_instructions(instrs)
    .into_iter()
    .rev()  // (Reverse because we pop from the end)
    .collect::<Vec<_>>();

// A mutable pointer to the current location in the memory tape.
// Initially, this is the center of the memory tape.
let mut memory_ptr = memory_center;

// This will be added to the end of the function every time a new block
// is encoded, returning the current memory tape pointer in register RAX.
let fn_epilogue = AMD64Instruction::encode_block(&vec![
    Mov(Register(RAX), Register(R12)),
    Pop(Register(R13)),
    Pop(Register(R12)),
    Ret(),
])?;

// We add an unconditional `jmp` instruction in the start of the function,
// which jumps directly to the start of the new block.
let pre_jump_position = self.fn_mem.position;
let jmp_size = Jmp(0, None).to_binary()?.len();
let post_jump_position = pre_jump_position + jmp_size;

let mut next_block_position = post_block_position;

// (Continued...)
```

```
// For each block of instructions in the program:
while let Some(block) = self.instr_blocks.pop() {
    // Encode the unconditional jump to the start of the new block
    let jump =
        Jmp((next_block_position - post_jump_position) as isize, None).to_binary()?;
    for byte_index in 0..jump.len() {
        self.fn_mem[pre_jump_position + byte_index] = jump[byte_index];
    }

    // JIT-compile the instructions in the new block
    let bytes = AMD64Instruction::encode_block(&*block)?;

    // Save the encoded instructions to the executable memory
    self.fn_mem.position = next_block_position;
    self.fn_mem.extend(bytes.into_iter());
    next_block_position = self.fn_mem.position;

    // Also, add the epilogue that restores saved registers
    // and returns the new memory pointer.
    self.fn_mem.extend(fn_epilogue.clone().into_iter());

    // Finally, we can just call this as an FFI function
    memory_ptr = fn_ptr(memory_ptr);
}
```

Benchmarks

For each of the following benchmarks, I AOT-compiled a version with and without simple loop optimizations. In the previous assignment, I found that my AVX2 scan optimization didn't show a meaningful improvement, so I've left it out here. I ran each AOT-compiled binary 1,000 times, and I ran the JIT compiler on each benchmark 1,000 times with and without simple loop optimizations. I also ran the interpreter 100 times on each benchmark with and without simple loop optimizations.

These tests were all done on the same machine, a laptop with a Ryzen 5 5500U CPU and 16 GB of RAM. The median runtime is shown below.

Program	Median AOT Runtime		Median JIT Runtime		Median Interp. Runtime	
	Baseline	Loop opt.	Baseline	Loop opt.	Baseline	Loop opt.
`bench.b`	0.235s	0.001s	0.239s	0.001s	0.515s	0.008s
`bottles.b`	0.000s	0.000s	0.002s	0.001s	0.003s	0.002s
`deadcodetest.b`	0.000s	0.000s	0.000s	0.000s	0.000s	0.000s
`hanoi.b`	3.785s	0.042s	3.926s	0.055s	8.562s	0.420s
`hello.b`	0.000s	0.000s	0.000s	0.000s	0.000s	0.000s
`long.b`	3.161s	0.201s	3.255s	0.238s	10.494s	2.808s
`loopremove.b`	0.000s	0.000s	0.000s	0.000s	0.000s	0.000s
`mandel.b`	0.792s	0.747s	0.862s	0.846s	6.688s	7.457s
`serptri.b`	0.000s	0.000s	0.001s	0.001s	0.001s	0.001s
`twinkle.b`	0.000s	0.000s	0.002s	0.002s	0.001s	0.001s

The JIT compiler has to do everything that the AOT-compiled code does, as well as read a source file; parse it into BF instructions; optionally optimize on those instructions; generate corresponding assembly instructions; and then for each block of instructions, encode the instructions into binary, copy them into executable memory, and call the JIT-compiled function. Despite all this extra work, the JIT compiler's runtime is only about 10% slower than the corresponding AOT-compiled program. I was expecting it to be at least 50% slower.

Before this assignment, I hadn't added the instruction optimizations to the interpreter. Surprisingly, optimizing the `mandel.b` instructions and then running the optimized instructions was consistently slower than simply running the non-optimized instructions.