Assignment 7: Dynamic Compilation

Patrick S. Li

October 22, 2015

1 Introduction

One of the biggest sources of inefficiency in our bytecode interpreter is the overhead of executing all the instructions that comprise the interpreter as compared to directly executing the instructions that perform the bytecode operation. In this assignment, we will avoid this overhead by compiling the Feeny bytecode directly into the assembly instructions and then asking the processor to execute the generated instructions. Step-by-step, we will go through the process of developing the necessary techniques for writing a JIT, and eventually finish with a just-in-time compiler for Feeny.

2 Background

It is assumed in this assignment that you either know the basics of x86 assembly programming or can easily find the appropriate information. Here are all the instructions that I use in my own implementation of Feeny:

- movq
- call
- ret
- leaq
- shrq

- shlq
- subq
- addq
- cmpq
- jle
- jl
- je

For the purposes of debugging, you will also need basic familiarity with interacting with assembly code using GDB. Good tutorials are available online.

3 What is a JIT?

Here we will get to the essence of a just-in-time compiler by writing a simple program that generates and executes instructions.

Start a new C program, and include the standard libraries for handling input/output, strings, and manual memory management. Save this file as `jit1.c`.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/mman.h>
```

In our main function, the first thing we need to do is allocate memory for holding the generated instructions. Because we will be executing the instructions this memory requires execute priviledges, thus `malloc` will not suffice, and we have to use `mmap` instead. The following call allocates 1024 bytes of executable memory.

```
char* code = mmap(0, 1024,
                  PROT_READ|PROT_WRITE|PROT_EXEC,
                  MAP_PRIVATE|MAP_ANON, -1, 0);
```

Next we generate the instructions we want to execute in this memory.

```
code[0] = 0x48;
code[1] = 0xc7;
code[2] = 0xc0;
code[3] = 0x2a;
code[4] = 0x00;
code[5] = 0x00;
code[6] = 0x00;
code[7] = 0xc3;
```

The above code represents a function that takes no arguments and returns an integer. To execute this generated function, we cast our `code` pointer to a function pointer and then call it.

```
int (*f)() = (void*)code;
printf("Returned %d\n", f());
```

This program will print :

```
Returned 42
```

when we compile and run it in the terminal.

This simple program makes up the essence of a just-in-time compiler. It is no more complicated than generating the binary sequences that comprise a program at runtime and then executing the generated instructions. The difficult (and interesting) part, of course, is in knowing what to generate.

3.1 Exercises

1. (3 points) Change the line above from

   ```
   code[3] = 0x2a;
   ```

   to

   ```
   code[3] = 0x3f;
   ```

   What does the program do now?

4 Binary Encoding of Instructions

The binary data given above corresponds to the x86 assembly instructions

```
  movq $42, %rax
  ret
```

which is a simple function that simply returns the integer 42. Theoretically, this information is completely available online, for free, in the Intel x86 reference manuals, but diphering those manuals requires a manual in itself. Instead, we will take advantage of gcc's built-in assembler to translate instructions to binary for us.

Start a new file called `jit1.s` and put the following in it:

```
  .globl return42
return42 :
  movq $42, %rax
  ret
```

Now compile your program using

```
gcc jit1.c jit1.s -o jit1
```

and launch gdb using

```
gdb jit1
```

Using the command `x/2i return42` we can print out the contents of the memory location designated by label `return42`, formatted as instructions. Your system should print out something similar to:

```
   0x400701 <_return42>:        mov    $0x2a,%rax
   0x400708 <_return42+7>:      retq
```

To print out the binary encoding of the instructions, we can use the command `x/8b return42` which prints out something similar to:

```
0x400701 <_return42>: 0x48 0xc7 0xc0 0x2a 0x00 0x00 0x00 0xc3
```

which exactly corresponds to what we stored in our code buffer.

4.1 Exercises

1. (3 points) Generate the instructions required to implement a function that takes a single integer argument, x, and returns $x + 42$. Detail here the assembly instructions you used, and their binary encoding. See section 5 for details on the SystemV calling convention.

2. (3 points) Generate the instructions required to implement a function that takes two integer arguments, x and y, and returns $x - y$. Detail here the assembly instructions you used, and their binary encoding.

5 SystemV Calling Conventions

The full System V calling conventions are complicated and here we will explain only the subset of the full convention that we will need for writing our JIT.

x86-64 has 16 general purpose registers: `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%rbp`, `%rsp`, `%r8`, `%r9`, `%r10`, `%r11`, `%r12`, `%r13`, `%r14`, `%r15`.

When calling a function, the first six arguments are stored in registers:

```
%rdi: Argument 1
%rsi: Argument 2
%rdx: Argument 3
%rcx: Argument 4
%r8: Argument 5
%r9: Argument 6
```

and the result has to be stored in register `%rax`. The following registers: `%rsp`, `%rbx`, `%rbp`, `%r12`, `%r13`, `%r14`, `%r15` are callee saved registers, which means that if you use these registers, it is *your* responsibility to restore them to their original values before returning from your function.

6 Simple JIT

As is evident now, the most time consuming part of writing a full-featured JIT is the assembler that encodes instructions into the specific binary format needed by the processor. We will be using a different technique that avoids having to do this.

Section 4 shows that we can directly write our assembly instructions in a separate file and have gcc translate it into binary for us. The translated code is included as part of the compiled program. Therefore, there is no need for us to figure out the details of the binary encoding for us to generate the

instructions. We can simply copy the encoded instructions from where its stored in the compiled program to our allocated executable memory.

Modify `jit1.s` to mark the end of the `return42` snippet.

```
  .globl return42
  .globl return42_end
return42 :
  movq $42, %rax
  ret
return42_end :
```

Now in `jit1.c`, add the following declarations to tell the C compiler that there exists some binary data with the following labels.

```
  extern char return42[];
  extern char return42_end[];
```

Thus the binary encodings for the instructions that comprise the `return42` instruction starts at address `return42` and ends at address `return42_end`.

Now instead of populating our `code` array manually:

```
  code[0] = 0x48;
  code[1] = 0xc7;
  code[2] = 0xc0;
  code[3] = 0x2a;
  code[4] = 0x00;
  code[5] = 0x00;
  code[6] = 0x00;
  code[7] = 0xc3;
```

We will simply copy the data over.

```
  memcpy(code, return42, return42_end - return42);
```

Compile and run the program and ensure that your JIT still works.

Thus, with this technique we can directly write assembly snippets in text form and then rely upon our C compiler to translate them to their appropriate binary encodings. This saves us the work of having to write an assembler ourselves. Note that almost all of the optimizations possible on a modern JIT are still possible using this technique. The only optimization that really necessitates a full assembler is register allocation, which is beyond the scope of this course.

7 Code Holes

With the technique developed in section 6 we wrote a snippet of assembly code that returns the integer 42 to the caller, and we can copy this snippet wherever needed in the construction of our JIT.

However, at some point, we will likely have to return integers other than 42. We *could* write a separate assembly snippet for each possible value that we want to return, but this is too tedious of a solution to consider. Instead, we will write our assembly snippets using a placeholder value first, and then later fill it in with our value of choice.

Modify `jit1.s` to use the placeholder value `0xcafebabecafebabe` in the return snippet. You may substitute `cafebabe` with `cafed00d` to suit your individual preferences.

```
  .globl return_ins
  .globl return_ins_end
return_ins :
  movq $0xcafebabecafebabe, %rax
  ret
return_ins_end :
```

Now modify `jit1.c` to generate instructions for returning 42 to the caller by copying over the binary data comprising `return_ins` and replacing the first occurrence of `0xcafebabecafebabe` with the number 42.

Note that this technique makes two relatively strong assumptions. The first is that the binary pattern `0xcafebabecafebabe` does not appear anywhere else in the encoded instructions except in our placeholder location. In the unlikely event that this is not true, we can simply choose a different placeholder value. The second assumption is that the literal `0xcafebabecafebabe` is stored contiguously in the encoded instruction stream. This is true for the x86 processor, but not for some others. For those processors, a more robust replacement mechanism will be necessary.

8 Position Independent Code

Modify `jit1.c` to declare a global variable called counter.

```
  long counter = 0;
```

Now modify `jit1.s` to include the following snippet for incrementing the value of counter.

```
  .globl inc_counter_ins
  .globl inc_counter_ins_end
inc_counter_ins :
  movq counter(%rip), %rax
  addq $1, %rax
  movq %rax, counter(%rip)
  ret
inc_counter_ins_end :
```

Verify that the `inc_counter_ins` assembly code is correct by casting it to a function pointer and calling it.

Now, try and copy the instructions to our code buffer and execute it from there. If we are lucky, our system will crash with a segmentation fault. If not, the program will continue running with some corrupted value in memory.

The reason that the same code does not work when copied to a different location is because modern systems use offsets that are relative to where the code is stored. Thus the line:

```
  movq counter(%rip), %rax
```

is compiled to load from an address at a specific offset from where that line of code is stored. However, our desire is to write a snippet of instructions that load from the absolute address of the variable `counter` no matter where this code is located. To do this, we can accept the address of the `counter` variable, `&counter`, as a placeholder to be filled in after copying the code snippet.

```
  .globl inc_counter_ins
  .globl inc_counter_ins_end
inc_counter_ins :
  movq $0xcafebabecafebabe, %rcx
  movq (%rcx), %rax
  addq $1, %rax
  movq %rax, (%rcx)
  ret
inc_counter_ins_end :
```

Use this updated snippet to generate the instructions necessary to increment the counter variable.

This technique will be necessary whenever you require an absolute address in your instruction snippets, such as setting and retrieving global variables, and jumping and branching to specific locations.

9 The Feeny JIT Compiler

9.1 Simplifying Assumptions

For the remainder of this course, we will be focused on optimization. To simplify things, you may restrict your virtual machines to only handle valid Feeny programs. Thus, for example, you may assume there is never a reference to a non-existent global variable, and that there will always exist an appropriate method for a method call expression, etc. In other words, given some Feeny program, if the AST interpreter completes execution and prints the message s to the screen, your implementation must also complete execution and print s to the screen. However, if the AST interpreter becomes stuck and exits with an error, the behaviour of your implementation is free to be left undefined.

9.2 Structure of the JIT

We will use the following general strategy in structuring our just-in-time compiler for Feeny. We will reserve these registers for the following purposes:

```
  %rdi: Top of Heap
  %rsi: Heap Pointer
  %rdx: Stack Pointer
  %rcx: Frame Pointer
```

To initialize and restore these registers before and after executing your JIT generated code, we will use the following function:

```
  .globl call_feeny
call_feeny:
  movq %rdi, %rax
```

```
  movq top_of_heap(%rip), %rdi
  movq heap_pointer(%rip), %rsi
  movq stack_pointer(%rip), %rdx
  movq frame_pointer(%rip), %rcx
  call *%rax
  movq %rsi, heap_pointer(%rip)
  movq %rdx, stack_pointer(%rip)
  movq %rcx, frame_pointer(%rip)
  ret
```

where we assume the existence of global variables `top_of_heap`, `heap_pointer`, `stack_pointer`, `frame_pointer`, for holding their respective machine states. The `call_feeny` function takes a single argument representing the starting address of the JIT code to execute, and returns whatever is returned by the generated JIT code.

We will generate assembly sequences to implement as many bytecodes as possible, but some bytecodes are too tedious to code in assembly. For these bytecodes we will implement a simple trap mechanism to return to C where we will perform the operation and then afterwards resume execution of the JIT code.

9.3 Trapping to C

To suspend execution of the JIT and trap to C, we will record our current position in the JIT code, and then return a special integer that indicates which operation we want to perform in C.

```
  leaq after_trap(%rip), %rax
  movq $ADDRESS_OF_IP, %r8
  movq %rax, (%r8)
  movq $OPERATION_ID, %rax
  ret
after_trap:
```

where `ADDRESS_OF_IP` is the placeholder for the address of the global variable `instruction_pointer`, and `OPERATION_ID` is the integer representing the operation to perform.

Thus the driver for the JIT code must check the returned integer of `call_feeny` to know whether the program is done, or whether some op-

eration needs to be performed.

```
void drive () {
   int running = 1;
   while(running){
      switch(call_feeny(instruction_pointer)){
      case PROGRAM_FINISHED :
         running = 0;
         break;
      case PERFORM_OP1 :
         perform_op1();
         break;
      case PERFORM_OP2 :
         perform_op2();
         break;
      ...
      case PERFORM_OPN :
         perform_opn();
         break;
      }
   }
}
```

9.4 Order of Implementation

The ability to trap to C gives us the ability to incrementally develop the JIT by initially implementing the majority of the bytecodes as traps to C. Then we can individually replace each bytecode with equivalent assembly sequences, and test the JIT as we go.

The following order of implementation is recommended.

1. Labels, Branches, and Gotos: The control flow operators require the addresses of targets in the generated instructions, and thus must be implemented together.
2. Literals, Set/Get Locals, Set/Get Globals, Drop: These can each be individually developed as simple assembly sequences.
3. Return: This is a simple assembly sequence but requires some care to get right as it forms half of Feeny's calling convention.

4. Call: This operation is implemented in two parts. First we need to set up the new frame, record the return address, and jump to the target address. Second we need to somehow record the number of locals in the new frame. However this value changes depending on which function we are calling. To do this, insert a special prelude at the beginning of each function to record the number of locals in its frame before it begins execution of its body.

5. Object: This operation is complicated by the fact that *most of the time* it can be implemented as a simple assembly sequence. However, if allocation requires a garbage collection call, then it requires trapping to C.

6. Array: If allocation requires a garbage collection call, then trap to C. Otherwise the bytecode can be implemented in assembly. However, this operation is not straightline code, and requires a loop.

10 Harness Infrastructure

You will be expected to execute Feeny programs from a given AST, and will be using the harness from the bytecode compiler assignment.

The function

```
Program* compile (ScopeStmt* stmt)
```

in the file `src/compiler.c` will be the entry point of your bytecode compiler, and will be called with the AST datastructure.

As in the last assignment

```
void interpret_bc (Program* p)
```

in the file `src/vm.c` will be the entry point of your bytecode interpreter and will be called with the result of your bytecode compiler.

Your assembly instructions are expected to reside in the file `src/vm.s`.

10.1 Test Harness

The provided bash script `run_tests` will parse the test programs in the `test` directory into ASTs and run your bytecode compiler and interpreter on the result. To run it, type:

```
./run_tests compiler.c vm.c vm.s
```

at the terminal. For each test program, e.g. `cplx.feeny`, the output of your bytecode interpreter will be stored in `output/cplx.out`. The default implementations of `compile` and `interpret_bc` do nothing except print out the AST and bytecode IR respectively.

Please ensure that your implementation can be driven correctly by `run_tests`. Your assignment will be graded using this script.

10.2 Compiling and Running Manually

To compile and run your implementation manually, you may also follow these steps. Compile your interpreter by typing:

```
gcc -std=c99 src/cfeeny.c src/utils.c src/ast.c
   src/bytecode.c src/compiler.c src/vm.c src/vm.s
   -o cfeeny -Wno-int-to-void-pointer-cast
```

at the terminal. This will create the `cfeeny` executable.

Next, you have to run the supplied parser which will read in the source text for a Feeny program and dump it to a binary AST file.

```
./parser -i tests/cplx.feeny -oast output/cplx.ast
```

Finally, call the `cfeeny` executable with the binary AST file as its argument to start the interpreter.

```
./cfeeny output/cplx.ast
```

11 Report

Implement your bytecode compiler, interpreter, and assembly snippets and place it in the `src` directory, naming it `compiler_xx.c`, `vm_xx.c`, and `vm_xx.s`, where `xx` is replaced with your initials. Then ensure that it can be properly driven by the testing harness by typing:

```
./run_tests compiler_xx.c vm_xx.c vm_xx.s
```

Include your implementation of Towers of Hanoi, Stacks, and More Towers of Hanoi from exercise 1, and ensure that your compiler and interpreter works correctly for those as well.

1. (90 points) **Compiler Statistics**: Create a table in your report that measure and calculate the following statistics for the test programs: `bsearch.feeny`, `inheritance.feeny`, `cplx.feeny`, `lists.feeny`, `vector.feeny`, `fibonacci.feeny`, `sudoku.feeny`, `sudoku2.feeny`, `hanoi.feeny`, `stacks.feeny`, `morehanoi.feeny`.
   (a) Total amount of time (in milliseconds) for `interpret_bc` to run and return.
   (b) Total amount of time (in milliseconds) to generate the instructions needed for the JIT.
   (c) Percentage of total time spent generating instructions.
   (d) Reduction in total time spent in `interpret_bc` relative to before implementing the JIT compiler (new/old).

12 Deliverables

Students may work in pairs or alone. Please submit your answers to section 11 as *report_XX.pdf*, and your programs as *compiler_XX.c*, *vm_XX.c*, *hanoi_XX.feeny*, *stacks_XX.feeny*, and *morehanoi_XX.feeny*. Zip all files together in a file called *assign7_XX.zip*. Replace *XX* above with your initials. Mail the zip file to `patrickli.2001@gmail.com` with `[Feeny7]` in the subject header.