VIRTUAL MACHINES IN DYNAMIC LANGUAGE INTERPRETERS

RYAN EGGERT

Computer Architecture has shown me how human-readable programs written in C or C++ can be translated into assembly or machine code which can be directly executed by a computer processor. This led me to wonder about a language such as Python. Python is generally my language of choice, but I found I was entirely unfamiliar with how the programs I wrote became instructions executable by a CPU. Just as I could compile C to executable instructions, certainly Python must, at some level, be 'turned into' CPU instructions. I wanted to see

Languages such as C/C++ are compiled languages. A text file written in C/C++ can be processed by a compiling program ["compiler"] such as gcc or g++ to create a file of instructions executable by a computer. These compiled programs are specific to one particular type of computer. A C/C++ file compiled for an x86 processor will not run on a MIPS processor and vice versa. As a result, one C/C++ file distributed to many different machines must first be recompiled on each computer before running.

However, another category of languages are interpreted instead of compiled. Instead of being converted directly into a file of machine instructions, they are converted incrementally to machine-specific instructions on-the-fly by another program [1]. At the expense of the additional computational overhead of running this other program (generally called the "interpreter"), one can simply run an interpreted program on just about any computer.

Interpreter structures can vary—there exist tens of different Python interpreters. We will focus on the reference implementation, CPython. This is the standard-issue "Python" downloadable from python.org. It is written in C, and uses a nifty computer architecture trick in order to execute Python code on any computer—it makes its own computer. Any time one runs a Python program, the interpreter creates its own virtual machine. Though written in C, it has memory, a program counter, and a computational means of executing instructions.

Unlike most physical computer processors, which are register machines, CPython's process virtual machine is a stack machine. Whereas a register machine stores and retrieves data to an array of addressable registers, a stack machine pops and pushes data to a handful of Last-In-First-Out [LIFO] stacks [2]. A diagram of a potential implementation of a stack machine is shown in Figure 1. Some components are familiar from our MIPS register CPU lab. This has a program counter which keeps track of the next instruction to be executed. The instructions are stored in RAM in the program memory—this implementation includes a memory address register ("MAR")

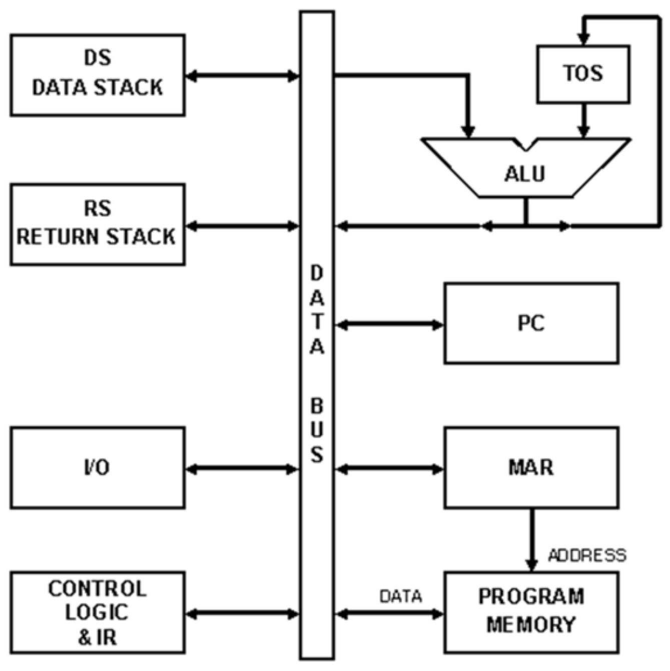

FIGURE 1: A SIMPLE ARCHITECTURE FOR A STACK MACHINE.

which stores the address to be read from (or written to) the program memory. The instruction is loaded into an instruction register ("IR"); a control logic module parses the instruction from the IR and sets control signals across the machine. It has an ALU for performing logical and arithmetical operations on sets of two data elements.

This is where the differences begin. Instead of a set of registers to store operands and results, the stack machine uses a LIFO stack. Data is added and removed from the top of this stack, which is implemented in Figure 1 as a 'data stack' ("DS") and a top-of-stack register ("TOS"). This TOS always contains the "top" of the operand/result stack, so the top of the DS is actually the second item in the operand/result stack. This separation is done so that the ALU can simultaneously access the top two items in the operand/result stack. The return stack ("RS") holds address pointers to instruction addresses for returning from subroutine/function calls.

This illustrates a performance advantage of stack machines over register machines. Instruction sets for a register machine must include fields to specify register addresses for reading operands from and storing results to. This is not required for stack machines—the ALU always pops the top two elements of the operand/result stack for its two inputs and pushes the result back to the stack. This eliminates the need for address specification for logical and arithmetical operations; as a result, instructions can be much shorter. This means that programs for stack machines can be significantly smaller.

However, stack machines can, in some scenarios require more instructions than their register counterparts. Take the example of implementing a simple expression like

$$z = x + y$$

Assume that x and y are values which were calculated some time ago (i.e., neither x nor y would be near the top of the data/operand stack). In a register machine, x and y can be stored in registers. In this case, this addition operation can be implemented in one line of MIPS instructions as shown in Figure 3. x and y are retrieved from their respective registers and their sum is stored into another, different register. The stack machine, on the other hand, must first load x and y from a data memory to the top of the stack before adding them. The sum, which the add operation pushed to the top of the stack, must also be stored into data memory to be preserved for future use. Admittedly, this is not always necessary. Figure 2 shows an example of how a stack machine may efficiently load and use results when performing sequential arithmetic operations [3]. While it is imaginable that a program could be organized in such a way that values could be left on the stack in strategic locations to be accessed and used later, it is probably fair to say that this optimization would not always be applicable. In fact, a study comparing stack- and register-based virtual machines for interpreting the Java language found that register architectures, on average, required 47% fewer instructions but required, on average, 25% more program memory [4].

MIPS Register Machine

```
add $z, $x, $y
```

Pseudo-MIPS Stack Machine

```
push x_addr
push y_addr
add
pop z_addr
```

FIGURE 3: A MIPS REGISTER MACHINE AND A STACK MACHINE (WHICH USES MIPS-ESQUE INSTRUCTIONS) IMPLEMENT A SIMPLE ADDITION OF PREVIOUSLY CALCULATED VALUES, X AND Y. THE SUM, Z, IS STORED FOR LATER USE.

Pseudo-MIPS Stack Machine

```
push x
push y
push z
mult
add
push u
add
```

FIGURE 2: HOW A STACK MACHINE MIGHT EFFICIENTLY COMPUTE $x + y * z + u$.

Stack machines are well-suited to run programs with many function or subroutine calls. Compared to a register machine, it is easy to switch to a subroutine. With a stack machine, there are no banks of registers which must be stored and restored when calling and leaving functions. A stack machine simply needs to store the current PC to the return stack ("RS") and push any additional values the subroutine needs to the stack [5]. As we'll see, this may be a primary reason why the writers of CPython chose to build their interpreter around a stack machine architecture.

CPython's interpreter creates a virtual stack machine to run instructions derived from Python code. This stack machine is implemented in C. At the core of this is `ceval.c` (in the CPython source code, this can be found in the `/Python` directory) [6]. This file, ceval, contains the interpreter's main loop. This function, `PyEval_EvalFrameEx()`, describes how to iterate through instructions, decode instructions, and execute instructions. An infinite loop (`for (;;) {...}`, line 1057) first grabs the next opcode by incrementing the pointer to the next instruction (`*next_instr++`, line 1167); this is effectively the PC and the program memory. From here, the instruction decode, some of the control logic, and the ALU-esque computation are handled by a large switch statement starting on line 1199. The switch statement allows specific C code to be run for each instruction. The switch statement is where the virtual machine defines how to perform an AND operation.

For example, see Figure 4, which shows how the interpreter's virtual stack machine executes a binary logical AND instruction. Notice that it removes sequentially removes `w` and `v` from the top of the stack, and then performs an AND operation on them. The two Py_DECREF calls are used by the interpreter's garbage collection system. It maintains a count of the number of references to every object (in this case, w and v). These two lines note that w and v have served their purpose here and are no longer useful. If the number of references to any object reaches zero, the object can be cleaned up by other garbage collection functions [7]. The result of the binary AND operation is placed at the top of the stack, and a quick error-check is performed. If the result of the binary AND operation is NULL, the `DISPATCH()` sequence (which increments a counter and prepares for the next instruction) is skipped and the `break` is hit, exiting the main interpreter loop.

```
TARGET_NOARG(BINARY_AND)
{
    w = POP();
    v = TOP();
    x = PyNumber_And(v, w);
    Py_DECREF(v);
    Py_DECREF(w);
    SET_TOP(x);
    if (x != NULL) DISPATCH();
    break;
}
```

FIGURE 4: AN EXAMPLE FROM THE PYTHON 2.7.11 SOURCE CODE [LINE 1571, CEVAL.C] SHOWING HOW THE VIRTUAL STACK MACHINE EXECUTES A BINARY AND INSTRUCTION.

The instruction set for this machine is Python bytecode. While Python is not ever compiled directly to machine-executable instructions, Python will instead first convert written Python into a sequence of integer instructions. These instructions are internal to a Python release and are subject to change between releases. They are meant to be used solely by the interpreter. A thorough discussion of disassembling Python to bytecodes lies outside the scope of this paper, but those interested in learning more can find a complete list of bytecodes in the Python documentation and can use Python's built-in `dis` module for interactively exploring generated bytecodes [8].

That's how the magic happens. Python is translated into bytecode instructions which are hardware-agnostic. The bytecode is executed by a virtual machine, written in C and compiled for the host machine, which runs on the host machine, using each instruction in turn to specify which C—and hence which machine code—to run. This is why one cannot easily "compile" Python to machine code. The interpreter's virtual machine makes it possible to run any arbitrary Python code on-the-fly.

REFERENCES

[1] “java - Compiled vs. Interpreted Languages - Stack Overflow.” [Online]. Available: http://stackoverflow.com/questions/3265357/compiled-vs-interpreted-languages. [Accessed: 15-Dec-2015].

[2] “Stack Machine.” [Online]. Available: http://www.cp.eng.chula.ac.th/˜piak/teaching/ca/stack.htm. [Accessed: 14-Dec-2015].

[3] “Stack machine - Wikipedia, the free encyclopedia.” [Online]. Available: https://en.wikipedia.org/wiki/Stack_machine. [Accessed: 15-Dec-2015].

[4] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, “Virtual machine showdown: Stack versus registers,” *Acm Trans. Archit. Code Optim.*, vol. 4, no. 4, p. 21, 2007.

[5] “Stack Computers: 6.2 ARCHITECTURAL DIFFERENCES FROM CONVENTIONAL MACHINES.” [Online]. Available: https://users.ece.cmu.edu/˜koopman/stack_computers/sec6_2.html. [Accessed: 14-Dec-2015].

[6] *Python 2.7.11*. Python Software Foundation, 2015.

[7] “python - What is the purpose of Py_DECREF and PY_INCREF? - Stack Overflow.” [Online]. Available: http://stackoverflow.com/questions/24444667/what-is-the-purpose-of-py-decref-and-py-incref. [Accessed: 15-Dec-2015].

[8] “32.12. dis — Disassembler for Python bytecode — Python 2.7.11 documentation.” [Online]. Available: https://docs.python.org/2/library/dis.html. [Accessed: 15-Dec-2015].