Introduction to tVM

Leonardo Banderali

Overview

Introduction: About this Tutorial

Who this tutorial is for

- You want to learn how Virtual Machines work
- You've been working on Eclipse OMR/OpenJ9 for a few weeks (months?)
- You have some idea of how a language VM works...
  - Maybe you know how one specific part works
  - Maybe you know bits and pieces of everything
- ... but you don't yet understand how a complete VM works

Goals

- Learn how the internals of a VM work
- Practice reading and writing code

What we will do

- Use a very simple (toy) VM/language to show how all the parts of a VM work and fit together
  - Most language VMs have a similar basic structure
  - Concepts from a simple VM can be applied to large scale VMs like OpenJ9
- We will alternate between theory and application
  - Start by going over some theory
  - Then, apply the theory by writing code

Schedule

1. Tutorial Overview + Intro to tVM Interpreter (this!) [theory]
2. Complete interpreter implementation [code]
3. Intro to tVM code generator [theory]
4. Complete code generator [code]
5. . . .

Introduction to Virtual Machines

What is a virtual machine?

- A program that executes other programs
  - A software version of a physical machine (“real” computer)
- Manages execution of a program
  - Manages resources like memory, file handles, locks/mutexes, etc.
  - VMs are also sometimes called “Managed Runtimes” (or “Runtimes” for short)
- In this tutorial, we will focus on stack-based VMs
  - Operations pop arguments from a stack and push results
  - Most common kind of VM; e.g. OpenJ9 is stack-based
  - Other VM kinds exist (e.g. Lua is register-based) but we won’t cover these

What are Bytecodes?

- VMs typically execute “Bytecodes”
- A Bytecode is a type of instruction that can be executed by a virtual machine
  - Analogous to Assembly instructions of a physical computer
- Bytecodes are not dependent on a specific processor’s instruction set
  - They are platform agnostic or portable
  - “Compile once, run anywhere” principle
- They are often designed to be efficiently decoded and interpreted (executed) by a computer program
  - Are not necessarily in a human readable format!

Example bytecodes for the Base9 (stack-based) VM

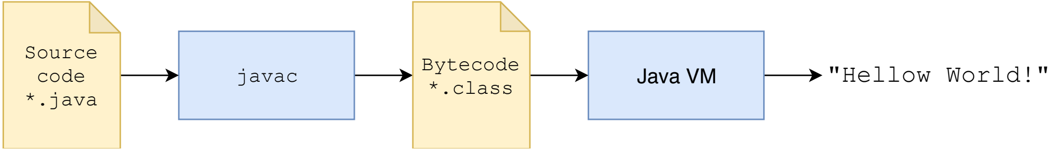
32-bit Encoding:	8-bit opcode	24-bit immediate

Name	Encoding		Action
	Opcode	Immediate	
`INT_PUSH_CONSTANT`	0xf	constant	Push the constant immediate onto the stack
`INT_ADD`	0xb	*(unused)*	Add values on the stack
`INT_SUB`	0xc	*(unused)*	Subtract values on the stack
`PUSH_FROM_VAR`	0x7	index	Push value from variable at `index` onto the stack
`POP_INTO_VAR`	0x8	index	Pop value from the stack and store into variable at `index`

`https://github.com/b9org/b9/blob/master/b9/include/b9/instructions.hpp`

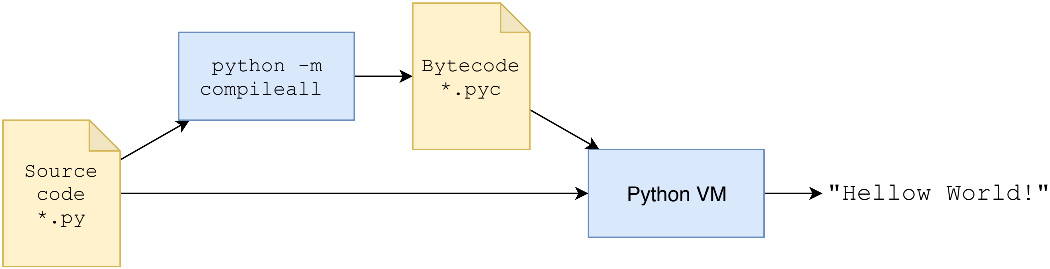
Source code and bytecode

- Source code is usually translated to bytecode before execution
- For some languages, source code is compiled “ahead of time” (AOT) to bytecode
  - Also called “static” compilation
  - Similar to how C and C++ get compiled to machine code
  - E.g., `javac` compiles Java code to bytecodes in a class file



Source code and bytecode

- Some language VMs allow source code to be loaded directly
  - Translation to bytecode is done when the source is loaded
  - An example is JavaScript
- Some languages can do both
  - Python and Lua

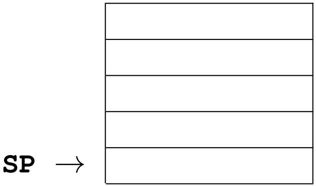


Understanding bytecode interpreters

- The interpreter is the part of a VM that <u>executes</u> bytecodes
  - Like the "CPU" of a VM
- An interpreter is essentially a loop around a giant switch statement
  - For each bytecode, execute some $\mathrm{C}{++}$ code that implements the functionality of the bytecode
- An Instruction Pointer (`IP`) keeps track of which bytecode to execute next
  - Is initialized to the start of the bytecode sequence
  - Points to the next bytecode to be executed
  - The loop terminates when the `IP` is pointing to the end of the bytecode sequence
- A Stack Pointer (`SP`) tracks (points to) the top of the stack

Bytecodes and the Stack

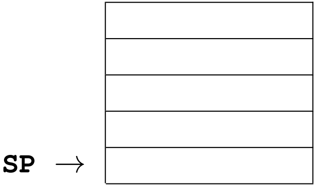
```
function simple_add() {
  return 5 + 6;
}

INT_PUSH_CONSTANT 5 ← IP
INT_PUSH_CONSTANT 6
INT_ADD
FUNCTION_RETURN
END_SECTION
```



Bytecodes and the Stack

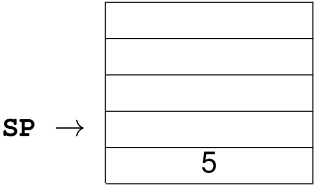
```
function simple_add() {
  return 5 + 6;
}
```

```
INT_PUSH_CONSTANT 5
INT_PUSH_CONSTANT 6 ← IP
INT_ADD
FUNCTION_RETURN
END_SECTION
```



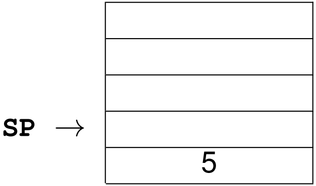
Bytecodes and the Stack

```
function simple_add() {
  return 5 + 6;
}

INT_PUSH_CONSTANT 5
INT_PUSH_CONSTANT 6 ← IP
INT_ADD
FUNCTION_RETURN
END_SECTION
```



Bytecodes and the Stack

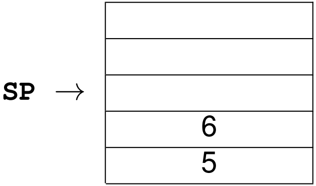
```
function simple_add() {
  return 5 + 6;
}

INT_PUSH_CONSTANT 5
INT_PUSH_CONSTANT 6
INT_ADD                ← IP
FUNCTION_RETURN
END_SECTION
```



Bytecodes and the Stack

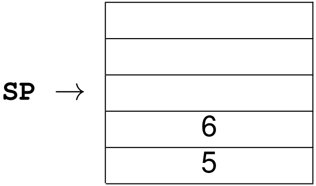
```
function simple_add() {
  return 5 + 6;
}
```

```
INT_PUSH_CONSTANT 5
INT_PUSH_CONSTANT 6
INT_ADD                ← IP
FUNCTION_RETURN
END_SECTION
```



Bytecodes and the Stack

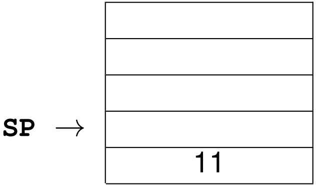
```
function simple_add() {
  return 5 + 6;
}

INT_PUSH_CONSTANT 5
INT_PUSH_CONSTANT 6
INT_ADD
FUNCTION_RETURN      ← IP
END_SECTION
```



Bytecodes and the Stack

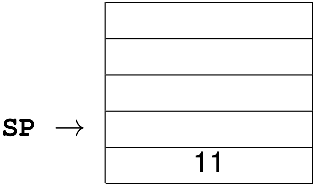
```
function simple_add() {
  return 5 + 6;
}
```

```
INT_PUSH_CONSTANT 5
INT_PUSH_CONSTANT 6
INT_ADD
FUNCTION_RETURN      ← IP
END_SECTION
```



Bytecodes and the Stack

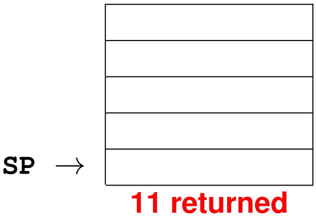
```
function simple_add() {
  return 5 + 6;
}

INT_PUSH_CONSTANT 5
INT_PUSH_CONSTANT 6
INT_ADD
FUNCTION_RETURN
END_SECTION          ← IP
```



Bytecodes and the Stack

```
function simple_add() {
  return 5 + 6;
}
```

```
INT_PUSH_CONSTANT 5
INT_PUSH_CONSTANT 6
INT_ADD
FUNCTION_RETURN
END_SECTION          ← IP
```



tVM: A Very Simple Virtual Machine

What is tVM?

- An educational Virtual Machine
  - Internals are designed to be easily understood
  - No details are hidden
  - Intentionally does not use Eclipse OMR
- The name stands for "tutorial Virtual Machine"
  - The letter "t" also sounds like "tea," which is a caffeinated drink that is lighter than Java ~~code~~ coffee
  - "t" is also the first letter of "Testarossa," the name of the JIT compiler I work on
- Code originated from Base9 VM source code
  - `https://www.base9.xyz/`
  - Base9 is designed for teaching how to use OMR
  - tVM was created by taking the Base9 code, stripping out all but the essentials, and removing all the OMR parts

The tVM bytecode language

- Language is very minimal
- Base implementation is just a calculator
  - Similar to Reverse Polish Notation calculator:
  - `https://en.wikipedia.org/wiki/Reverse_Polish_notation`
- Only supports 32-bit integer types
- Base implementation is also incomplete
- This tutorial will complete and extend the VM implementation
  - Complete interpreter to support all bytecodes
  - Add a very simple “JIT” compiler
  - Add more bytecodes: comparisons, control flow, etc.

The tVM bytecode language

- Bytecodes use similar encoding as Base9
- * = *Implementation left as an exercise to the reader*

Name	Encoding		Action
	8-bit Opcode	24-bit Immediate	
`END`	0x0	*(unused)*	Signal end of bytecode sequence
`INT_ADD`	0x1	*(unused)*	Add values on the stack
`INT_SUB`*	0x2	*(unused)*	Subtract values on the stack
`INT_MUL`	0x3	*(unused)*	Multiply values on the stack
`INT_DIV`*	0x4	*(unused)*	Divide values on the stack
`INT_PUSH_CONSTANT`	0x5	constant	Push the constant immediate onto the stack

`https://github.com/b9org/tVM/blob/master/tvm/include/tvm/instructions.hpp`

How does one go about writing an interpreter?

- Start with a class to represent instructions

```
class Instruction {
  public:
  OpCode opCode() const;
  Immediate immediate() const;
  // ...
};
```

INT_PUSH_CONSTANT 5

- OpCode
- Immediate

- See tvm/include/tvm/instructions.hpp

How does one go about writing an interpreter?

- Next, iterate over a vector of instructions and switch on the opcode

```cpp
std::stack<Immediate> stack_;
auto ip = instructions.begin();

while (ip->opCode != OpCode::END) {
  auto inst = *ip;
  ++ip;

  switch(inst.opCode()) {
    case OpCode::INT_ADD:
      doIntAdd(); break;
    case OpCode::INT_PUSH_CONSTANT:
      doIntPushConstant(inst.immediate()); break;
    // ...
}
```

- See tvm/src/ExecutionContext.cpp

How does one go about writing an interpreter?

- Implement the helpers

```cpp
void doIntAdd() {
  auto b = stack_.pop();
  auto a = stack_.pop();
  stack_.push(a + b);
}

void doIntPushConstant(Immediate x) {
  stack_.push(x);
}
```

- See tvm/src/ExecutionContext.cpp

What tVM provide

- Additional utilities and boilerplate code
  - Useful data structures for representing opcodes, immediates, stack, etc.
  - Definition of bytecodes
  - An incomplete interpreter loop
  - A small set of helpers for writing parts of the interpreter
  - A bytecode serializer and deserializer

Credits: tVM creators and contributors

- Nazim **@nbhuiyan**
- Annabelle **@a7ehuo**
- Younes **@ymanton**
- Xiaoli **@xliang6**
- Dhruv **@dchopra001**
- Leonardo **@Leonardo2718**

Setup

Install dependencies

- git
- C++ Compiler toolchain ("build-essential" for Ubuntu)
- CMake >= 3.2.0

Clone repository

- `git clone https://github.com/b9org/tVM.git`
- `cd tVM`
- `mkdir build`
- `cd build`
- `cmake ..`
- `cmake --build .`
- `./tvm-run/tvm-run ../test/simple_add.bc`

Your first task

- Start reading through the tVM code and become familiar with it
  - `tvm/` contains the core VM implementation
  - `tvm-run/` contains the `main.cpp` file for the executable VM
  - `test/` contains sample bytecode programs

Tips for Reading Code

Good programmers read code

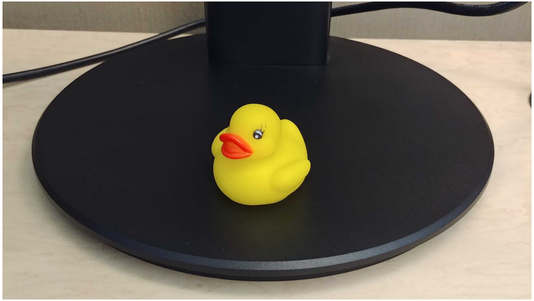
- Reading code effectively is a skill
- Reading code means:
  - Understanding the code, even if it’s the first time you see it
  - Navigating code to find the pieces you are interested in
- Reading code effectively is key to becoming a good (better!) developer

Tips for reading code

1. Divide and Conquer
   - You don’t need to understand everything right away
   - Look at the code one section/structure at a time
   - Once you understand it, mentally put a “black box” around it and move on
2. Don’t get lost in the details
   - Don’t look at the implementation of every function — it will never end
   - Try to figure out what the function probably does without looking at its code
   - Use data types, function and variable names, etc.
   - You don’t need to know how `std::sort` works exactly, you just need to know that it “sorts”

Tips for reading code

3. Use a rubber duck
   - Get a rubber duck and explain to it what the code does, one line at a time
   - `https://en.wikipedia.org/wiki/Rubber_duck_debugging`
   - Teddy bears, stuffed animals, and other inanimate objects also work
   - Real people can work but they tend to have opinions



Tips for reading code

4. Experiment!
   – Try running the code (if possible)
   – Try changing the code and see what happens
   – Try running the code in a debugger
5. Use the resources you have access to
   – [INSERT SEARCH ENGINE NAME HERE] is your friend
   – Look for answers on StackOverflow
   – Ask for help
     – Slack
     – StackOverflow
     – GitHub
     – In person :)

The Golden Rule of Programming

Write code the way you want other people to write code that you will have to read.