CSE 413
Programming Languages & Implementation

Hal Perkins

Spring 2021

Java Implementation – JVMs, JITs &c

Agenda

- Java virtual machine architecture
- .class files
- Class loading
- Execution engines
  - Interpreters & JITs – various strategies
- Exception Handling

Java Implementation Overview

- Java compiler (javac et al) produces machine-independent .class files
  - Target architecture is Java Virtual Machine (JVM), a simple stack machine
- Java execution engine (java)
  - Loads .class files (often from libraries)
  - Executes code
    - Either interprets stack machine code or compiles to native code (JIT)

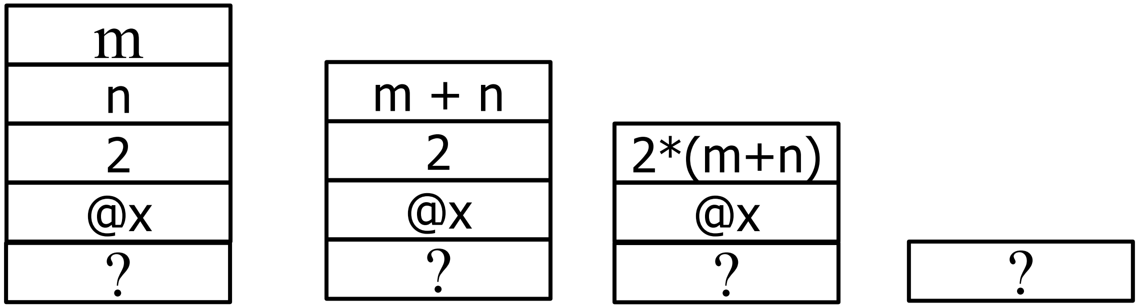
JVM Architecture

- Abstract stack machine
- Implementation not required to use JVM specification literally
  - Only requirement is that execution of .class files has specified effect
  - Multiple implementation strategies depending on goals
    - Compilers vs interpreters
    - Optimizing for servers vs workstations

Stack Machine Code Example

Hypothetical code for x = 2 * (m + n)

```
pushaddr    x
pushconst   2
pushval     n
pushval     m
add
mult
store
```



Compact: common opcodes just 1 byte wide; instructions have 0 or 1 operand

JVM Data Types

- Primitive types
  - byte, short, int, long, char, float, double, boolean
- Reference types
  - Non-generic only (more on this later)

JVM Runtime Data Areas (1)

- Semantics defined by the JVM Specification
  - Implementer may do anything that preserves these semantics
- Per-thread data
  - pc register
  - Stack
    - Holds frames (details below)
    - May be a real stack or may be heap allocated

JVM Runtime Data Areas (2)

- Per-VM data – shared by all threads
  - Heap – objects allocated here (new)
  - Method area – per-class data
    - Runtime constant pool
    - Field and method data
    - Code for methods and constructors
- Native method stacks
  - Regular C-like stacks or equivalent

Frames

- Created when method invoked; destroyed when method completes
  - This is why Java "lambdas" aren't real first-class closures – environments not retained when creating function exits
- Allocated on stack of creating thread
- Contents
  - Local variables
  - Operand stack used by JVM instructions
  - Reference to runtime constant pool
    - Symbolic data that supports dynamic linking
  - Anything else the implementer wants

Representation of Objects

- Implementer's choice
  - JVM spec 3.7: “The Java virtual machine does not mandate any particular internal structure for objects”
  - Likely possibilities
    - Data + pointer to Class object
    - Pair of pointers: one to heap-allocated data, one to Class object

JVM Instruction Set

- Stack machine
- Byte stream
- Instruction format
  - 1 byte opcode
  - 0 or more bytes of operands
- Instructions encode type information
  - Verified when class loaded

Instruction Sampler (1)

- Load/store
  - Transfer values between local variables and operand stack
  - Different opcodes for int, float, double, addresses
  - Load, store, load immediate
    - Special encodings for load0, load1, load2, load3 to get compact code for first few local vars

Instruction Sampler (2)

- Arithmetic
  - Again, different opcodes for different types
    - byte, short, char & boolean use int instructions
  - Pop operands from operand stack, push result onto operand stack
  - Add, subtract, multiply, divide, remainder, negate, shift, and, or, increment, compare
- Stack management
  - Pop, dup, swap

Instruction Sampler (3)

- Type conversion
  - Widening – int to long, float, double; long to float, double, float to double
  - Narrowing – int to byte, short, char; double to int, long, float, etc.

Instruction Sampler (4)

- Object creation & manipulation
  - New class instance
  - New array
  - Static field access
  - Array element access
  - Array length
  - Instanceof, checkcast

Instruction Sampler (5)

- Control transfer
  - Unconditional branch – goto, jsr (originally used to implement finally blocks)
  - Conditional branch – ifeq, iflt, ifnull, etc.
  - Compound conditional branches - switch

Instruction Sampler (6)

- Method invocation
  - invokevirtual
  - invokeinterface
  - invokespecial (constructors, superclass, private)
  - invokestatic
- Method return
  - Typed value-returning instructions
  - Return for void methods

Instruction Sampler (7)

- Exceptions: athrow
- Synchronication
  - Model is *monitors* (cf any standard operating system textbook)
  - monitorenter, monitorexit
  - Memory model greatly cleaned up in Java 5

JVM and Generics

- Surprisingly, JVM has no knowledge of generic types
  - Not checked at runtime, not available for reflection, etc.
- Compiler *erases* all generic type info
  - Resulting code is pre-generics Java
  - Objects are class Object in resulting code & appropriate casts are added
- Only one instance of each type-erased class – no code expansion/duplication (as in C++ templates)

Generics and Type Erasure

- Why did they do that?
  - Compatibility: need to interop with existing code that doesn't use generics
    - Existing non-generic code and new generic libraries, or
    - Newly written code and older non-generic classes
- Tradeoffs: only reasonable way to add generics given existing world way back then, but
  - Generic type information unavailable at runtime (casts, instanceof, reflection)
  - Can't create new instance or array of generic type
- C#/CLR is different – generics reflected in CLR

Class File Format

- Basic requirements are tightly specified
- Implementations can extend
  - Examples: data to support debugging or profiling
  - JVMs must ignore extensions they don't recognize
- Very high-level, symbolic, lots of metadata – much of the symbol table/type/other attribute data produced by a compiler front end
  - Supports dynamic class loading
  - Allows runtime compilation (JITs), etc.

Class Loaders

- One or more class loader (instances of ClassLoader or its derived classes) is associated with each JVM
- Responsible for loading the bits and preparing them
- Different class loaders may have different policies
  - Eager vs lazy class loading, cache binary representations, etc.
- May be user-defined, or the initial built-in bootstrap class loader

Readying .class Files for Execution

- Several distinct steps
  - Loading
  - Linking
    - Verification
    - Preparation
    - Resolution of symbolic references
  - Initialization

Virtual Machine Startup

- Initial class specified in implementation-defined manner
  - Command line, IDE option panel, etc.
- JVM uses bootstrap class loader to load, link, and initialize that class
- `public static void main(String[])` method of initial class is executed to drive all further execution

Execution Engines

- Basic Choices
  - Interpret JVM bytecodes directly
  - Compile bytecodes to native code, which then executes on the native processor
    - Just-In-Time compiler (JIT)

Hybrid Implementations

- Interpret or use very simple compiler most of the time
- Identify "hot spots" by dynamic profiling
  - Often per-method counter incremented on each call
  - Timer-based sampling, etc.
- Run optimizing JIT on hot code
  - Data-flow analysis, standard compiler middle-end optimizations, back-end instruction selection/ scheduling & register allocation
  - Need to balance compilation cost against responsiveness, expected benefits
    - Different tradeoffs for desktop vs server JVMs

JIT optimization implications

- JVM optimized code often combines code from multiple classes
  - One particularly common optimization: inlining
    - Replace calls to getter/setter methods with copies of method bodies (load/store from mem)
  - Often extremely effective, but if any class is reloaded, other compiled code that depended on previous version is no longer valid
    - JVM has logic to detect this and invalidate previously compiled code, forcing JIT to rerun if needed to optimize

C# and Microsoft CLR

- Very similar to Java – basic compiler generates byte code files that are combined for execution
- Big implementation difference: basic CLR compiles everything to native code when assemblies created – no JIT interpreter + compiler for hot spots

- Other differences: various extensions for Microsoft-specific environments

Memory Management

- JVM includes instructions for creating objects and arrays, but not deleting
- Garbage collection used to reclaim no-longer needed storage (objects, arrays, classes, …)
- Strong type system means GC can have exact information
  - .class file includes type information
  - GC can have exact knowledge of layouts since these are internal to the JVM
- More details next hour

Escape Analysis

- Another optimization based on observation that many methods allocate local objects as temporaries
- Idea: Compiler tries to prove that no reference to a locally allocated object can "escape"
  - Not stored in a global variable or object
  - Not passed as a parameter

Using Escape Analysis

- If all references to an object are local, it doesn't need to be allocated on the heap in the usual manner
  - Can allocate storage for it in local stack frame
    - Essentially zero cost
  - Still need to preserve the semantics of new, constructor, etc.

Exception Handling

- Goal: should have zero cost if no exceptions are thrown
  - Otherwise programmers will subvert exception handling with the excuse of “performance”
- Corollary: cannot execute any exception handling code on entry/exit from individual methods or try blocks

Implementing Exception Handling

- Idea: Original compiler generates table of exception handler information in the .class file
  - Entries include start and end of section of code array protected by this handler; argument type
  - Order of entries is significant
- When exception is thrown, JVM searches exception table for first matching argument type that has a pc range that includes the current execution location

Summary

- That’s the overview – many more details, obviously, if you want to implement a JVM
- Primary reference: Java Virtual Machine Specification. Available online: https://docs.oracle.com/javase/specs/
- Many additional research papers & studies all over the web and in conference proceedings