

CSE P 501 – Compilers

Java Implementation – JVMs, JITs &c
Hal Perkins
Summer 2004

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-1

Agenda

- Java virtual machine architecture
- .class files
- Class loading
- Execution engines
 - Interpreters & JITs – various strategies
- Garbage Collection
- Exception Handling
- Managed code execution in C#/CLR is similar

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-2

Java Implementation Overview

- Java compiler (javac, jikes) produces machine-independent .class file
 - Target architecture is Java Virtual Machine (JVM) – simple stack machine
- Java execution engine (java)
 - Loads .class files
 - Executes code
 - Either interprets stack machine code or compiles to native code (JIT)

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-3

JVM Architecture

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

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-4

JVM Data Types

- Basic data types found in Java – byte, short, int, long, char, float, double, boolean
- Reference Types

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-5

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

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-6

JVM Runtime Data Areas (2)

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

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-7

Frames

- Created when method invoked; destroyed when method completes
- Allocated on stack of creating thread
- Contents
 - Local variables
 - Operand stack for JVM instructions
 - Reference to runtime constant pool
 - Symbolic data that supports dynamic linking
 - Anything else the implementer wants

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-8

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

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-9

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

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-10

Instruction Sampler (1)

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

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-11

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

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-12

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.

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-13

Instruction Sampler (4)

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

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-14

Instruction Sampler (5)

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

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-15

Instruction Sampler (6)

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

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-16

Instruction Sampler (7)

- Exceptions: athrow
- Synchronization
 - Model is *monitors* (cf any standard operating system textbook)
 - monitorenter, monitorexit

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-17

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, lots of metadata – much of the symbol table/type/other attributes data found in a compiler
 - Supports dynamic class loading
 - Allows runtime compilation (JITs), etc.

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-18

Contents of Class Files (1)

- Starts with magic number (0xCAFEBABE)
- Constant pool - symbolic information
 - String constants
 - Class and interface names
 - Field names
- All other operands and references in the class file are referenced via a constant pool offset
- Constant pool is essentially a "symbol table" for the class

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-19

Contents of Class Files (2)

- Class and superclass info
 - Index into constant pool
- Interface information
 - Index into constant pool for every interface this class implements
- Fields declared in this class proper, but not inherited ones (includes type info)
- Methods (includes type info)
 - Includes byte code instructions for methods that are not native or abstract

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-20

Constraints on Class Files (1)

- Long list; verified at class load time
 - ∴ execution engine can assume valid, safe code
- Some examples of static constraints
 - Target of each jump must be an opcode
 - No jumps to the middle of an instruction or out of bounds
 - Operands of load/store instructions must be valid index into constant pool
 - new is only used to create objects; anewarray is only used to create arrays
 - Only invokespecial can call a constructor
 - Index value in load/store must be in bounds
 - Etc. etc. etc.

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-21

Constraints on Class Files (2)

- Some examples of structural constraints
 - Instructions must have appropriate type and number of arguments
 - If instruction can be executed along several paths, operand stack must have same depth at that point along all paths
 - No local variable access before being assigned a value
 - Operand stack never exceeds limit on size
 - No pop from empty operand stack
 - Execution cannot fall off the end of a method
 - Method invocation arguments must be compatible with method descriptor
 - Etc. etc. etc.

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-22

Class Loaders

- One or more class loaders (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 initial built-in bootstrap class loader

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-23


Reading .class Files for Execution

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

8/17/2004

© 2002-04 Hal Perkins & UW CSE


T-24



Loading

- Class loader locates binary representation of the class (normally a .class file) and reads it
- Once loaded, a class is identified in the JVM by its fully qualified name + class loader id
 - A good class loader should always return the same class object given the same name
 - Different class loaders generally create different class objects even given the same class name


8/17/2004 © 2002-04 Hal Perkins & UW CSE T-25



Linking

- Combines binary form of a class or interface type with the runtime state of the JVM
- Always occurs after loading
- Implementation has flexibility on timing
 - Example: can resolve references to other classes during verification (static) or only when actually used (lazy)
 - Requirement is that verification must precede initialization and semantics of language must be respected
 - No exceptions thrown at unexpected places, for example


8/17/2004 © 2002-04 Hal Perkins & UW CSE T-26



Linking: Verification

- Checks that binary representation is structurally correct
 - Verifies static and structural constraints (see above)
 - Goal is to prevent any subversion of the Java type system
- May cause additional classes and interfaces to be loaded, but not necessarily prepared or verified


8/17/2004 © 2002-04 Hal Perkins & UW CSE T-27



Linking: Preparation

- Creation of static fields & initialization to default values
- Implementations can optionally precompute additional information
 - Method tables, for example


8/17/2004 © 2002-04 Hal Perkins & UW CSE T-28



Linking: Resolution

- Check symbolic references and, usually, replace with direct references that can be executed more efficiently

8/17/2004 © 2002-04 Hal Perkins & UW CSE T-29



Initialization

- Execute static initializers and initializers for static fields
- Direct superclass must be initialized first
- Constructor(s) not executed here
 - Done by a separate instruction as part of new, etc.

8/17/2004 © 2002-04 Hal Perkins & UW CSE T-30

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 in initial class is executed to drive all further execution

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-31

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)

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-32

Hybrid Implementations

- Interpret or use very dumb compiler most of the time
- Identify "hot spots" by dynamic profiling
 - 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

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-33

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, ...)
 - GC must prove not needed before reclaiming
- 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
- Can't do this for C/C++ because of incomplete type info & weak type system; best you can hope for is a conservative GC

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-34

Garbage Collection

- Basic idea
 - Identify root set of references
 - Registers
 - Active stack frames
 - Static fields in classes
 - Trace closure of root set references
 - Reclaim any allocated objects that are not reachable

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-35

Garbage Collection Variations

- Compacting collectors
 - Move active objects so they are adjacent in new heap space
 - Advantage: better locality
 - Need bookkeeping during move/compact sweep to handle pointers between old space and new space

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-36

Generation Garbage Collection

- Observation: Programs written in most O-O languages create many short-lived objects
- Implication: Scanning entire heap on each GC is mostly wasted effort
- Strategy
 - Allocate new objects in a small part of the heap
 - Routine GC just collects in this nursery
 - Objects that survive some number of GCs are moved to more permanent part of heap
 - Still need to GC full heap occasionally

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-37

Escape Analysis

- Another idea 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

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-38

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.

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-39

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

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-40

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, and the 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

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-41

Summary

- Object-oriented languages introduce new implementation issues, and different tradeoffs for classical compiler techniques
- Wide interest in the compiler research community, particularly since Java burst onto the scene
- Still an active area of research/development

8/17/2004

© 2002-04 Hal Perkins & UW CSE

T-42