Just in Time Compilation

Louis Croce
JIT Compilation: What is it?

“Compilation done during execution of a program (at run time) rather than prior to execution” -Wikipedia

● Seen in today’s JVMs and elsewhere
Outline

● Traditional Java Compilation and Execution
● What JIT Compilation brings to the table
● Optimization Techniques
● JIT Compilation in JRockit/HotSpot JVMs
● JRockit Breakdown Optimization Example
● JIT Compilation elsewhere
Outline

● Traditional Java Compilation and Execution
  ● What JIT Compilation brings to the table
  ● Optimization Techniques
  ● JIT Compilation in JRockit/HotSpot JVMs
  ● JRockit Breakdown and Optimization Example
● JIT Compilation elsewhere
Traditional Java Compilation and Execution

- 2 steps
- A Java Compiler compiles high level Java source code to Java bytecode readable by JVM
- JVM interprets bytecode to machine instructions at runtime
Traditional Java Compilation and Execution

- **Advantages**
  - platform independence (JVM present on most machines)
  - reflection: modification of program at runtime

- **Drawbacks**
  - need memory
  - not as fast as running pre-compiled machine instructions
Outline

- Traditional Java Compilation and Execution
- What JIT Compilation brings to the table
- Optimization Techniques
- JIT Compilation in JRockit/HotSpot JVMs
- JRockit Breakdown and Optimization Example
- JIT Compilation elsewhere
Goals in JIT Compilation

- combine speed of compiled code w/ flexibility of interpretation

**Goal:** “surpass the performance of static compilation, while maintaining the advantages of bytecode interpretation” -Wikipedia
JIT Compilation (in JVM)

- Builds off of bytecode idea
- A Java Compiler compiles high level Java source code to Java bytecode readable by JVM
- JVM compiles bytecode at runtime into machine readable instructions as opposed to interpreting
- run compiled machine readable code
- Seen in many JVM implementations today
Advantages of JIT Compilation

- Compiling: can perform AOT optimizations
- Compiling bytecode (not high level code) => can perform AOT optimizations faster
- can perform runtime optimizations
- executing machine code is faster than interpreting bytecode
Drawbacks of JIT Compilation

- **Startup Delay**
  - must wait to compile bytecode into machine-readable instructions before running
  - bytecode interpretation may run faster early on
- **Limited AOT optimizations b/c of time**
- **JVM needs compiler packaged in now**
- **Compilers for different types of arches**
  - for some JITs like .net
  - (not for JVM)
Security issues

● Executable space protection
  ○ Bytecode compiled into machine instructions that are stored directly in memory
  ○ those instructions in memory are run
  ○ Have to check that memory
Outline

- Traditional Java Compilation and Execution
- What JIT Compilation brings to the table
- Optimization Techniques
- JIT Compilation in JRockit/HotSpot JVMs
- JRockit Breakdown and Optimization Example
- JIT Compilation elsewhere
Optimization techniques

- Detect frequently used bytecode instructions & optimize
  - # of times a method executed
  - detection of loops

- Combine interpretation with JIT Compilation
  - method used in popular Hotspot JVM incorporated as of Java8’s release

- Server & Client specific optimizations

- More useful in longer running programs
  - have time to reap benefits of compiling/optimizing
Outline

- Traditional Java Compilation and Execution
- What JIT Compilation brings to the table
- Optimization Techniques
- JIT Compilation in JRockit/HotSpot JVMs
- JRockit Breakdown and Optimization Example
- JIT Compilation elsewhere
A look at a traditional JVM

- HotSpot JVM (pre-Java 8)
  - straight bytecode interpretation
  - limited optimizations
JRockit JVM

- “The industry’s highest performing JVM now built into Oracle Fusion Middleware.” - Oracle
- Currently integrated with Sun’s (now Oracle’s) HotSpot JVM
- Why?
  - JIT
When to use which?

Hotspot
- Desktop application
- UI (swing) based application
- Desktop daemon
- Fast starting JVM

JRockit
- Java application server
- High performance application
- Need of a full monitoring environment
HotSpot’s JRockit Integration

- Launched with Java8
- By default interprets
- Optimizes and compiles hot sections
- Runs compiled code for hot sections
Outline

● Traditional Java Compilation and Execution
● What JIT Compilation brings to the table
● Optimization Techniques
● JIT Compilation in JRockit/HotSpot JVMs
● JRockit Breakdown and Optimization Example
● JIT Compilation elsewhere
JRockit Breakdown

- NOTE: Compilation and optimizations are performed on java **BYTE**code in the JVM.
JRockit JVM

byte
Java Code

JRockit JVM

Linux
Windows
Solaris

110001110
Chip
JRockit JVM
JRockit JIT Compilation

1. JRockit runs JIT compilation
2. JRockit monitors threads
3. JRockit runs optimization

Java application

JIT-compiled machine code

Highly optimized machine code
JRockit Step 1: JIT Compilation

- When section of instructions called
  - compile bytecode into machine code just in time
  - run compiled machine code
- Not fully optimized
- May be slower than bytecode interpretation
- JVM Startup may be slower than execution
JRockit Step 2: Monitor Threads

● Identify which functions merit optimization
● Sampler thread
  ○ checks status of active threads
● Hot methods are ear-marked for optimization
● Optimization opportunities occur early on
JRockit Step 3: Optimization

In background, run compilation of optimized “hot” methods

(Compile optimized bytecode into machine readable instructions)
JRockit Optimization Example

● NOTE: Optimizations are performed on java bytecode in the JVM.
● In the following example from Oracle, the code is written in Java so that it is easier to read, but the JRockit JVM is performing the optimizations on the bytecode instructions
## JRockit Optimization Example

### Class A before optimization
```java
class A {
    B b;
    public void foo() {
        y = b.get();
        ...do stuff...
        z = b.get();
        sum = y + z;
    }
}

class B {
    int value;
    final int get() {
        return value;
    }
}
```

### Class A after optimization
```java
class A {
    B b;
    public void foo() {
        y = b.value;
        ...do stuff...
        sum = y + y;
    }
}

class B {
    int value;
    final int get() {
        return value;
    }
}```
public void foo() {
    y = b.get();
    ...do stuff...
    z = b.get();
    sum = y + z;
}
Step 2: Inline Final Method

```java
public void foo() {
    y = b.value;
    ...do stuff...
    z = b.value;
    sum = y + z;
}

● swap b.get() with get() method's contents
```
Step 3: Remove Redundant Loads

```java
public void foo() {
    y = b.value;
    ...do stuff...
    z = y;
    sum = y + z;
}
```

- swap `z=b.value();` with `z=y;`
public void foo() {
    y = b.value;
    ...do stuff...
    y = y;
    sum = y + y;
}

- no use for z
Step 5: Eliminate Dead Code

public void foo() {
    y = b.value;
    ...do stuff...
    // nothing
    sum = y + z;
}

• y=y does nothing, delete it
## JRockit Example

### Class A before optimization
```java
class A {
    B b;
    public void foo() {
        y = b.get();
        ...do stuff...
        z = b.get();
        sum = y + z;
    }
}

class B {
    int value;
    final int get() {
        return value;
    }
}
```

### Class A after optimization
```java
class A {
    B b;
    public void foo() {
        y = b.value;
        ...do stuff...
        sum = y + y;
    }
}

class B {
    int value;
    final int get() {
        return value;
    }
}```
Outline

● Traditional Java Compilation and Execution
● What JIT Compilation brings to the table
● Optimization Techniques
● JIT Compilation in JRockit/HotSpot JVMs
● JRockit Breakdown and Optimization Example
● JIT Compilation elsewhere
JIT Elsewhere: More bytecode langs

JIT in JVM has been driving force in movement of more languages to compile to java byte code

- Jython
- JRuby
- Groovy
JIT Elsewhere: C++ like languages

- by default, C++ uses AOT
- C#
  - MSIL == java bytecode
  - JIT
- *Not certain how these work
- CLANG
  - Uses LLVM on backend
  - can benefit from JIT Compilation of bytecode
- C++/CLI (Common Language Infrastructure)
  - Language from Microsoft
JIT Elsewhere: Web Browsers

- Goal: optimize javascript
- Seen today in
  - Mozilla’s Tamarin
  - safari webkit FTL JIT compiler
  - chrome’s V8
  - all browsers except ie8 and earlier
Questions?
Sources

● Oracle Docs (for JRockit JVM)
  ○ http://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagno
ds/underst_jit.html

● CLANG info
  ○ http://clang.llvm.org/comparison.html

● JVM comparison
  ○ http://www.dbi-services.com/index.php/blog/entry/a-
    comparison-of-java-virtual-machines-hotspot-jvm-vs-
    jrockit
Sources

- Wikipedia