Just In Time Compilation

JIT Compilation: What is it?

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

Java Bytecode

int i = 0;
i++;

```
iconst_0
istore_1
iinc      1, 1
return
```

int i = 0;
i++;
int j = 1;
j = i;

```
iconst_0
istore_1
iinc      1, 1
iconst_1
istore_2
iload_1
istore_2
```

Java Bytecode

```
int sum = 0;
for (int i = 0; i < 10; i++)
{     sum+=i;
}
```

```
iconst_0
istore_1
iconst_0
istore_2
4: iload_2
bipush      10
if_icmpge   20
iload_1
iload_2
iadd
istore_1
iinc        2, 1
goto        4
20: return
```

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 Optimization Example
- JIT Compilation elsewhere

Goals in JIT Compilation

- combine speed of compiled code with the flexibility of interpretation
- Goal
  - surpass the performance of static compilation
  - maintaining the advantages of bytecode interpretation

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), so 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
- Compilers for different types of architectures
  - for some JITs like .NET

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 Optimization Example
- JIT Compilation elsewhere

Optimization techniques

- Detect frequently used bytecode instructions and 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

Optimization techniques

- Server & Client specific optimizations
- More useful in longer running programs
  - have time to reap benefits of compiling/optimizing
- Compilation and optimizations are performed on java bytecode in the JVM.

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

A look at a traditional JVM

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

JRockit JVM

- The industry's highest performing JVM as claimed by Oracle
- Currently integrated with Sun's (now Oracle' s) HotSpot JVM
- Why?
  - JIT

When to use which?

- **Hotspot**
  - Desktop application
  - UI (swing) based application
  - Fast starting JVM
- **JRockit**
  - Java application server
  - High performance application
  - Need of a full monitoring environment

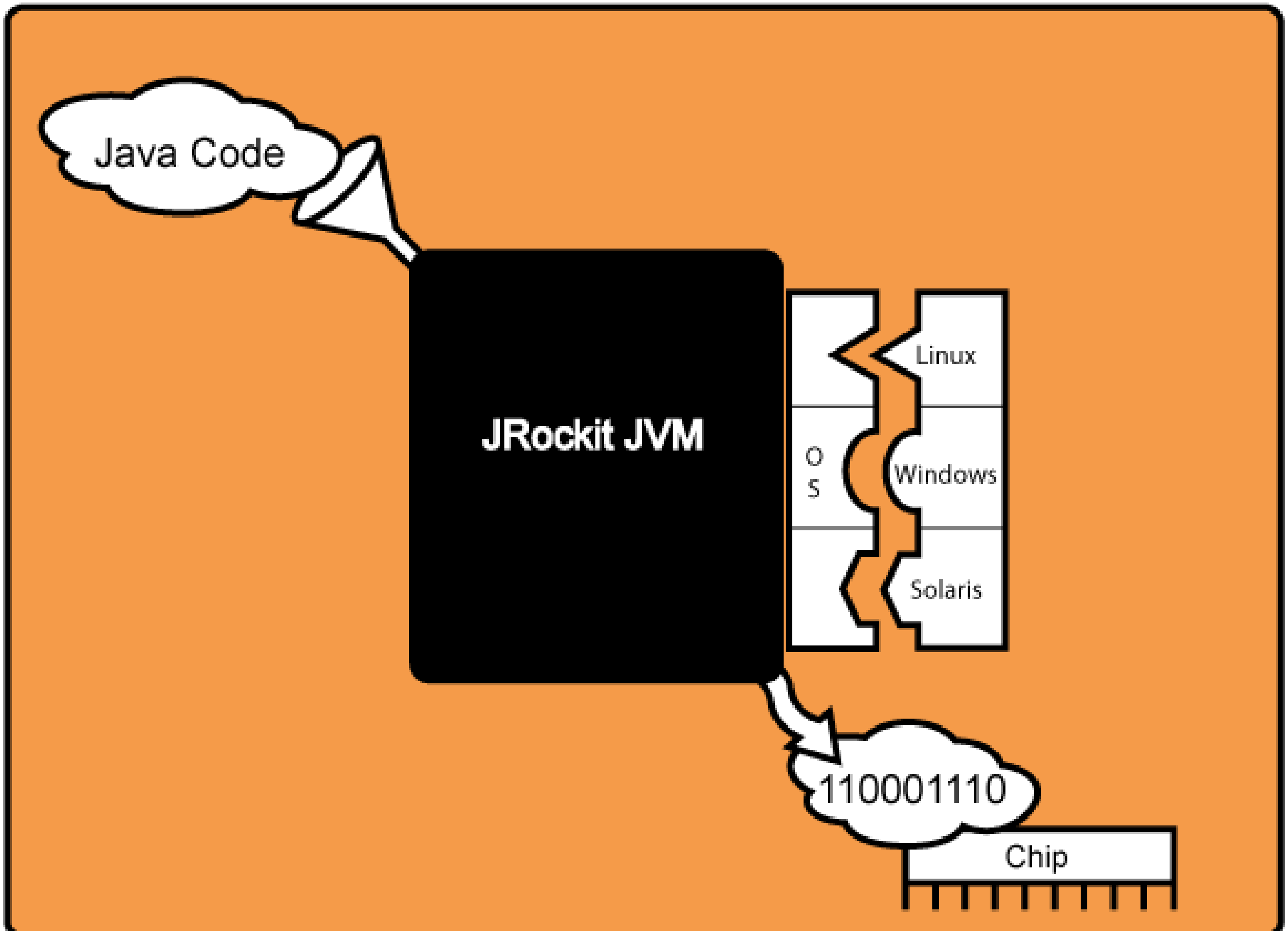
HotSpot’s JRockit Integration

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

Outline

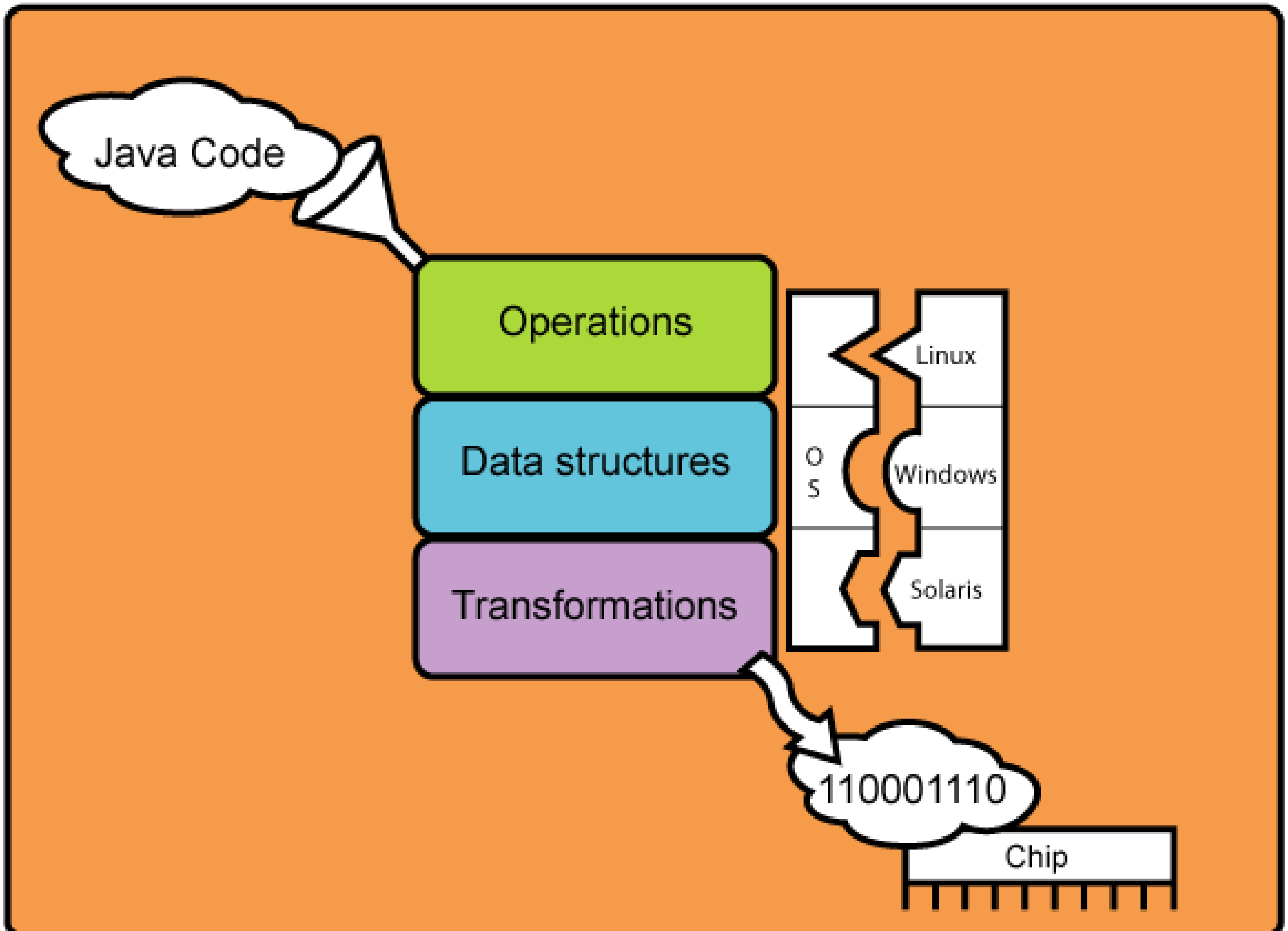
- Traditional Java Compilation and Execution
- What JIT Compilation brings to the table
- Optimization Techniques
- JIT Compilation in JRockit/HotSpot JVMs
- JRockit Breakdown Optimization Example
- JIT Compilation elsewhere

JRockit JVM



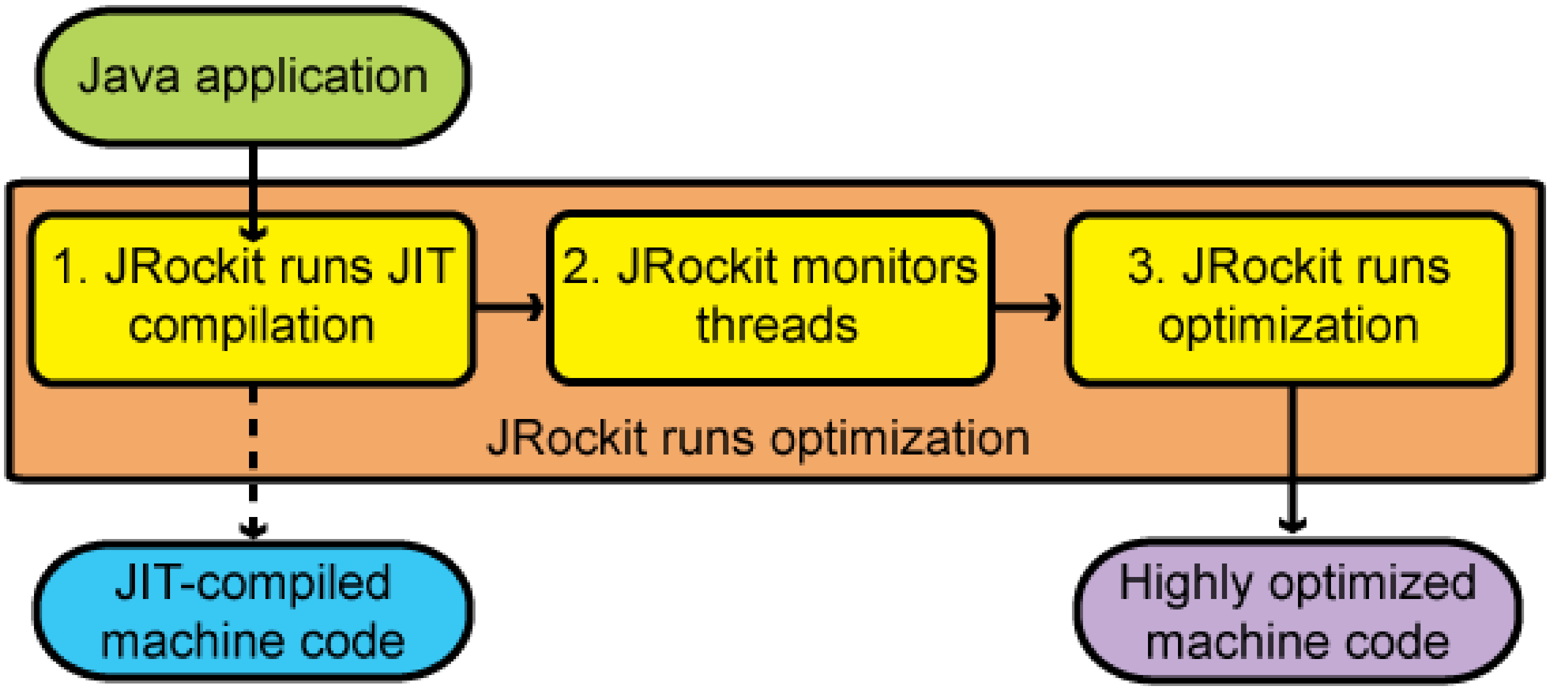
Java Code
JRockit JVM
O
S
Linux
Windows
Solaris
110001110
Chip

JRockit JVM



Java Code
Operations
Data structures
Transformations
Linux
O
S
Windows
Solaris
110001110
Chip

JRockit JIT Compilation



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

<table>
<tr><th>Class A before optimization</th><th>Class A after optimization</th></tr>
<tr><td><pre>
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;
    }
}
</pre></td><td><pre>
class A {
B b;
public void foo() {
    y = b.value;
    ...do stuff...
    sum = y + y;
}
}

class B {
    int value;
    final int get() {
        return value;
    }
}
</pre></td></tr>
</table>

Step 1: Starting Point

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

Step 2: Inline Final Method

```
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

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

- swap z=b.value(); with z=y;

Step 4: Copy Propagation

```
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 + y;
}
```

- y=y does nothing, delete it

JRockit Optimization Example

<table>
<tr><th>Class A before optimization</th><th>Class A after optimization</th></tr>
<tr><td><pre>
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;
    }
}
</pre></td><td><pre>
class A {
B b;
public void foo() {
    y = b.value;
    ...do stuff...
    sum = y + y;
}
}

class B {
    int value;
    final int get() {
        return value;
    }
}
</pre></td></tr>
</table>

Step 6: Choose Instruction

```
public void foo() {
    y = b.value;
    // do stuff
    sum = y << 1;
}
```

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

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
- CLANG
  - Uses LLVM on backend
  - can benefit from JIT Compilation of bytecode

JIT Elsewhere: Web Browsers

- Goal: optimize JavaScript
- Seen today in
  - Mozilla’s Tamarin
  - Safari’s WebKit
  - Chrome’s V8
  - all browsers except IE8 and earlier