Java™ Terminology

HotSpot JVM

Runtime Compiler

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. Hewlett-Packard is independent of Sun Microsystems, Inc.
Learning objectives

At the end of this section you will understand:

- The vocabulary that we use
- The J2SE product and its components
- How the Java Virtual Machine works
- Benefits of the HotSpot Runtime Compiler
Agenda

• Java Terminology
  • The vocabulary with which we work
  • J2SE
  • JVM
  • Java Runtime Compiler
  • Garbage Collection
  • Synchronization
  • Benchmarks

• Java Virtual Machine

• Java Runtime Compiler
  • Benchmarking and Benchmarks
Java Virtual Machine

- **Class Loader**
  - Loads class files
- **Interpreter**
  - Executes bytecode
- **Runtime Compiler**
  - Converts bytecode to native code

```
0a0b0c0d0c6262431c1d688662a0b0c0d0c1334514726522723
```

```
01010101010010110101011011001110111011
10101011000111010101100110111011
```

```
01010101010010110101011011001110111011
10101011000111010101100110111011
```

```
bytecode
```

```
native code
```

© 2001 Hewlett-Packard Company
Products

• J2SE - Java 2 Software Developer’s Kit (SDK), Standard Edition
  • Java development tools
    – Debugger - jdb
    – Compiler – javac - java source file to java class file
  • Java Virtual Machine + libraries required to execute Java class files
  • Libraries that include the basic operations (and the sizes and numbers of libraries are growing)

• JDK – Java Developer’s Kit
  • Short name for J2SE
Products

- JRE – Java Runtime Environment
  - Short name for J2SE JRE
  - Distributable version that permits J2SE applications to be run
Product Components

- JVM - The Java Virtual Machine
  - Reads class files
  - Interpreter
    - Executes bytecodes
  - Native runtime
  - Runtime Compiler that converts bytecodes to native instructions
JVM Types

• **HotSpot JVM**
  • The name of the default JVM on HP-UX
    – Includes Server-side runtime compiler
  • Has optimizations for running code faster
    – Fast interpreter
    – Java Heap and Garbage Collector
    – Runtime compilation of bytecodes to native code

• **Classic JVM**
  • First generation JVM
  • Invoked using: `java -classic`
  • Planned obsolescence in JDK 1.4
Runtime Compilers

- Just-In-Time (JIT) Compilers (Classic JVM)
  - Translation of each bytecode to corresponding native instructions
  - Little optimization - No internal data structures created to do:
    - Register allocation
    - Instruction scheduling
- HotSpot Runtime Compiler (Server-side)
  - Translation of bytecodes in method to an intermediate representation (IR)
  - Inlining of called methods
  - Optimizer performs optimization of the IR
  - Code generator targets the architecture explicitly
HotSpot JVM Overview

- Created by a company named “LongView Technology”
  - LongView acquired by Sun
  - Technology licensed by HP
  - Optimized by HP for HP-UX
- Interpreter
  - Runtime
  - Libraries
  - Java Heap and Garbage Collector
- Analyzes code execution
  - Identifies “hot” spots in program execution
- Compiles code using the optimizing Runtime Compiler
  - Inlines critical methods (in caller)
HotSpot JVM Overview

• Major benefits:
  • Lightweight, low overhead object model
  • Fast garbage collector
  • Fast synchronization
  • Adaptive compilation

• Designed for long-running, server-side applications
  • Can be configured to run for client-side applications using options to the JVM
HotSpot Runtime Compiler

- Classic JIT Compiler
  - Compile based on frequency of execution

- HotSpot Runtime Compiler
  - **Adaptive optimization**
  - Profile application
    - Method call frequency
    - Stack trace
  - Compile based on time spent executing
    - Inline based on stack traces
HotSpot Runtime Compiler

Advantages

• Less code compiled
  • Smaller code cache required
  • Allows higher optimization level for compiled code

• Reoptimization
  • Switch back to executing interpreted method
    – Interpreter calls the bytecode version of the method
    – When appropriate, method is re-compiled
  • Permits adaptive response to program behavior
    – Different paths -> different inlining
  • Required for correct program execution
Future Directions

• Additional optimizations
  • Loop transformations
  • Pipelining
  • 64 bit arithmetic
  • Inlining of new instances
HotSpot Runtime Compiler
Targeting IA-64

• Exploiting Instruction Level Parallelism (ILP)
  • Collaboration between compiler and processor
  • Compiler explicitly:
    – Orders instructions
    – Maps instructions to functional units

• Predication
  • Can eliminate branches

• Speculation
  • Early execution of an instruction
    – Control: Early instruction execution reduces latency
    – Data: Early load reduces latency
HotSpot Runtime Compiler
Targeting IA-64

• Branch prediction: Hints
  • Empirical data available at runtime helps!
• Block scheduling
• Global scheduling
  • Entire method
• Instruction scheduling
  • Bundling for IA-64: 3 instructions plus hints
• Register allocation
HotSpot Runtime Compiler Benchmarking

- Original Benchmark
  
  ```java
  static public void main(String [] args) {
    do all the work;
  }
  ```
HotSpot Runtime Compiler Benchmarking

• Original Benchmark

```
static public void main(String[] args) {
    do all the work;
}
```
• **Original Benchmark**
  
  ```java
  static public void main(String [] args) {
    do all the work;
  }
  ```

• **Modified**
  
  ```java
  static public void testit() {
    do all the work;
  }
  
  static public void main(String [] args) {
    for (int i=0; i<6; i++)
      testit();
  }
  ```
HotSpot Runtime Compiler

Benchmarking

• Original Benchmark
  ```java
  static public void main(String [] args) {
    do all the work;
  }
  ```

• Modified
  ```java
  static public void testit() {
    do all the work;
  }

  static public void main(String [] args) {
    for (int i=0; i<6; i++)
      testit();
  }
  ```
HotSpot Runtime Compiler Benchmarking

• Original Benchmark
  static public void main(String [] args) {
    do all the work;
  }

• Modified
  static public void testit() {
    do all the work;
  }
  static public void main(String [] args) {
    for (int i=0; i<6; i++)
      testit();
  }
HotSpot Runtime Compiler Benchmarking

- Original Benchmark
  ```java
  static public void main(String [] args) {
    do all the work;
  }
  ```

- Modified
  ```java
  static public void testit() {
    do all the work;
  }

  static public void main(String [] args) {
    for (int i=0; i<6; i++)
      testit();
  }
  ```

<table>
<thead>
<tr>
<th>Outer Loop Iteration</th>
<th>Read</th>
<th>Compute</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>35.6</td>
<td>27.8</td>
</tr>
<tr>
<td>2</td>
<td>7.5</td>
<td>27.8</td>
</tr>
<tr>
<td>3</td>
<td>35.4</td>
<td>27.8</td>
</tr>
<tr>
<td>4</td>
<td>6.2</td>
<td>4.2</td>
</tr>
</tbody>
</table>
### Runtime Compiler Benchmarking

**Results of Iterating**

- **JDK 1.2.2.05**

```java
java -Xmx128m -Xms128m -Xmn32m -XX:SurvivorRatio=8 JavaReadAndCompute
```

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Read</th>
<th>Compute</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>35.6</td>
<td>27.8</td>
</tr>
<tr>
<td>2</td>
<td>7.5</td>
<td>27.8</td>
</tr>
<tr>
<td>3</td>
<td>35.4</td>
<td>27.8</td>
</tr>
<tr>
<td>4</td>
<td>6.2</td>
<td>4.2</td>
</tr>
<tr>
<td>5</td>
<td>6.2</td>
<td>4.2</td>
</tr>
<tr>
<td>6</td>
<td>6.2</td>
<td>4.2</td>
</tr>
</tbody>
</table>

Times are in seconds
## Runtime Compiler Benchmarking

### Comparison of C++ and Java

<table>
<thead>
<tr>
<th>Platform</th>
<th>Compiler</th>
<th>Read</th>
<th>Compute</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP-UX</td>
<td>gcc</td>
<td>0.9</td>
<td>4.4</td>
</tr>
<tr>
<td>HP-UX</td>
<td>HP aCC</td>
<td>0.9</td>
<td>2.8</td>
</tr>
<tr>
<td>HP-UX</td>
<td>JDK 1.2 HotSpot</td>
<td>6.1</td>
<td>4.2</td>
</tr>
<tr>
<td>HP-UX</td>
<td>JDK 1.1 Classic/JIT</td>
<td>55.6</td>
<td>8.0</td>
</tr>
<tr>
<td>HP-UX</td>
<td>JDK 1.1 Classic/noJIT</td>
<td>52.7</td>
<td>35.4</td>
</tr>
<tr>
<td>Linux 800 MHz</td>
<td>JDK 1.1 HotSpot</td>
<td>49.5</td>
<td>8.1</td>
</tr>
</tbody>
</table>

Times are in seconds (lower is faster)

HP-UX on 440 MHz machines

- gcc == Java Compute performance
- 2x advantage for aCC with this benchmark
### Runtime Compiler Benchmarking
### Comparison of C++ and Java

<table>
<thead>
<tr>
<th>Platform</th>
<th>Compiler</th>
<th>Read</th>
<th>Compute</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP-UX</td>
<td>gcc</td>
<td>0.9</td>
<td>4.4</td>
</tr>
<tr>
<td>HP-UX</td>
<td>HP aCC</td>
<td>0.9</td>
<td></td>
</tr>
<tr>
<td>HP-UX</td>
<td>JDK 1.2</td>
<td>6.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HotSpot</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HP-UX</td>
<td>JDK 1.1</td>
<td>55.6</td>
<td>8.0</td>
</tr>
<tr>
<td></td>
<td>Classic/JIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HP-UX</td>
<td>JDK 1.1</td>
<td>52.7</td>
<td>35.4</td>
</tr>
<tr>
<td></td>
<td>Classic/noJIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux</td>
<td>JDK 1.1</td>
<td>49.5</td>
<td>8.1</td>
</tr>
<tr>
<td>800 MHz</td>
<td>HotSpot</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Times are in seconds (lower is faster)

HP-UX on 440 MHz machines

J2SE 1.3
I/O speed improvements
J2SE 1.4
nio for more performance
Garbage Collection

• Built-in feature of the JVM
• The process of reclaiming memory no longer in use
• Removes objects which have no references to them and frees up the space they occupied
• For best application performance, garbage collection needs to be minimized for one or both of:
  • Time required to do a single collection
  • Total time required to all collections
Synchronization, Monitors, Contention

- Control access to shared resources

- Java monitors control access
Synchronization, Monitors, Contention

• Java synchronized method:
  ```java
  public synchronized void syncMethod() {
      // safe to modify shared resource
  }
  ```

• Java synchronized block:
  ```java
  static private Object syncObject = new Object();
  public void nonSyncMethod() {
      synchronized (syncObject) { // Can be “this”
          // safe to modify shared resource
      }
  }
  ```
JNI – Java Native Interface

• Interface for calling:
  • C/C++ programs from Java
  • Java runtime from C/C++ programs
Other Useful Terminology

• JNDI
  • Java Naming and Directory Interface
  • Used for looking up objects/beans in J2EE application servers
  • Can be an interface to LDAP Server

• J2EE
  • Java 2 Enterprise Edition
  • J2SE + libraries required for application servers
Terminology
HotSpot JVM

Benchmarks
Frequently Used

• SPEC JVM98
  • Measures JVM performance (CPU)
• SPEC JBB2000 (Java Business to Business)
  • Measures String allocation performance
  • White paper available on HP’s website
• Volano
  • Measures ability of OS and Java to support very high numbers of thread/sockets
  • White paper available on HP’s website
• Graphical User Interface
  • SPECglperf 3.1 - OpenGL performance
  • BenchJ3d - Java 3D
  • JMark 2.0 - Swing text
Java Profiling
Improving the Paradigm

Current situation ...
Java Profiling Today

- Application Tuning
- Java Application Run
- JVM Tuning
- Application Performance Analysis
- JVM Performance Analysis
- Profile Data

Terminology
HotSpot JVM
HP’s Java Profiling Vision

Goal: *Automatic* Feedback
Resources Available

• Access to all information about J2SE products:
  • www.hp.com/java
    – Latest SDK and JRE releases
    – HPjmeter
    – HPjconfig
    – See the “developer’s resource”

• Access to information about J2EE products:
  • www.bluestone.com
    – See the “hp bluestone developer gallery”