Profiling Dynamically Compiled Java

Tia Newhall

cnewhall@cs.wisc.edu

Computer Sciences
University of Wisconsin
1210 W. Dayton St.
Madison, WI 53706
http://www.cs.wisc.edu/~newhall
Why Java?

- Java is slow, but it is being used for all kinds of things
  - Web-based computing
  - Meta-computing (Globus, Javelin, Charlotte)
  - High performance numeric applications (NPAC, JNT, JAMA)
  - Parallel computing (jPVM, JPVM, Java-MPI, HPJava, Titanium)
- Java VM’s are getting faster
  - HotSpot as fast as equivalent C++
Dynamically Compiled Java

- Java application changes form at run-time
- Even in native code form, an application method interacts with Java VM
Performance measurement important...oh yeah, prove it

- Our hypothesis
  - dynamic compilation might not be the only answer

- Tested 3 application kernels on Sun’s ExactVM dynamic compiler
  - Platform2 release of JDK

- ExactVM’s run-time compiling heuristic:
  - if a method contains a loop, compile it immediately
  - else, wait until a method is called 15 times
Test Applications

- Application kernels test cases where we suspect dynamic compilation may not win
  - method’s whose time not dominated by interpreting byte-code (I/O or synchronization)
  - method’s whose native code form still has a lot of interaction with Java VM (object creates)
  - small method functions

- A mainloop method calls methods implementing one of the three cases
Results

The graph shows the execution time in seconds for different methods as a function of iterations. The methods include obj DC, obj Interp, I/O DC, I/O Interp, small DC, and small Interp. The graph indicates that as the number of iterations increases, the execution time for all methods also increases, with obj DC being the fastest and small Interp being the slowest.
What did we learn?

- There is something going on in this execution that we would like to see...
  - performance measures with native code form and byte-code form of a method function
    - did run-time compilation help? why not?
  - Java VM interactions with native code form of a method
    - what are these interactions?
    - how much do they affect the application’s execution?
Paradyn-J

- Profiles dynamically compiled Java
  - simulate dynamic compilation
  - wrapper calls byte-code & JNI native versions
- Performance data that:
  - explicitly describes interactions between the VM and the Java application
  - associated with multiple execution forms of Java application methods
  - describes run-time costs of dynamically compiling a Java method
Let's see what we can do...

<table>
<thead>
<tr>
<th>Method with object creates</th>
<th>I/O intensive method</th>
<th>Small methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Byte-code</td>
<td>Native</td>
</tr>
<tr>
<td>Total CPU</td>
<td>2.35</td>
<td>5.65</td>
</tr>
<tr>
<td>Obj. create overhead</td>
<td>1.57</td>
<td>0.01</td>
</tr>
<tr>
<td>Total I/O time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total CPU</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Performance tuning study

- Java neural network application

23 classes and 15,800 lines of Java source
A method that doesn’t benefit from run-time compilation

<table>
<thead>
<tr>
<th>File</th>
<th>Actions</th>
<th>View</th>
</tr>
</thead>
</table>

**Time Histogram Display**

Phase: Global

**CPUs**

- cpu_inclusive `/APCode/ArtificialNeuralNetworkLearner.class/calculateHiddenLayer_interp()`
- cpu_inclusive `/APCode/libann1_g.so/Java_ArtificialNeuralNetworkLearner_calculateHiddenLayer_1native PAN`
Why not?

- VM still handles all memory management
How can we use this data to tune the Java application?

- remove some object creates

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Tuned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>3.96</td>
<td>3.53</td>
</tr>
<tr>
<td>%</td>
<td>20.8</td>
<td>18.7</td>
</tr>
<tr>
<td>Total</td>
<td>24.76 secs</td>
<td>22.23 secs</td>
</tr>
</tbody>
</table>

⇒ improved method’s performance by 10%
Applying tuning to a real dynamically compiled execution

- Run tuned version on real Java dynamic compiler (Sun’s ExactVM)

<table>
<thead>
<tr>
<th>original</th>
<th>tuned</th>
<th>change</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.09</td>
<td>18.97</td>
<td>10%</td>
</tr>
</tbody>
</table>

- Why does this make sense?
  - simulation adds extra overheads not in ExactVM
  - object creation overheads about the same
What about the VM?

- Tune the VM routines responsible for handling object creates in the Java application
- Tune the dynamic compiler’s run-time compiling heuristics
  - characteristics of method that make it a bad candidate
  - incorporating profile data into the heuristics
Conclusions

- Java is here to stay
- More sophisticated VM’s will ensure this
- Performance measurement of dynamically compiled Java is complicated
- Paradyn-J provides data that
  - lets us see inside the dynamic compiler to see how it executes the application
  - characterizes the VM’s performance in terms of the application code it dynamically compiles