HotSpot profiling with JITWatch

Chris Newland - 16th April 2014
WhatSpot?

• Java HotSpot Virtual Machine
  – Bytecode interpreting stack machine
    • No registers
    • Variables pushed onto stack
  – Just In Time (JIT) compilers
    • Profile Guided Optimisation (PGO)
    • Compile bytecode to native code

<table>
<thead>
<tr>
<th>Tiered</th>
<th>Non-Tiered</th>
<th>-Xint</th>
<th>-Xcomp</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.9s</td>
<td>2.6s</td>
<td>80.5s</td>
<td>4.4s</td>
</tr>
</tbody>
</table>

*Horrible unscientific benchmark
(com.chrisnewland.jitwatch.demo.MakeHotSpotLog)
Talking JIT

• Client compiler (C1)
  – Starts quickly, simple compilation to native

• Server compiler (C2)
  – Waits until more information available
  – Loop unrolling, **Inlining**, Dead Code Elimination, Escape analysis, Intrinsics, **Branch prediction**

• Tiered Compilation (C1 + C2)
  – Default in Java 8
  – Enable in Java 7 with -XX:+TieredCompilation
  – Best of both worlds?
Explain yourself!

- Enable JIT logging
- `-XX:+UnlockDiagnosticVMOptions`
- `-XX:+LogCompilation`
- `-XX:+TraceClassLoading (JITWatch)`
- `-XX:+PrintAssembly`
  - Required hsdis binary in jre/lib/<arch>/server
  - Significant performance overhead
  - [http://www.chrisnewland.com/building-hsdis-on-linux-amd64-on-debian-369](http://www.chrisnewland.com/building-hsdis-on-linux-amd64-on-debian-369)
I heard you like to grep?

- Logs can be > 50MB
- Much bigger with disassembly!
- Let's build a visualiser!
JITWatch

- [https://github.com/AdoptOpenJDK/jitwatch/](https://github.com/AdoptOpenJDK/jitwatch/)
- **JIT Compilation**
  - When? (time, invocations)
  - How? (C1, C2, Tiered, OSR)
- **Decompiles**
  - Back to bytecode interpretation (Why?)
- **Inlining** - successes / failures
- **Branch probabilities** - taken / not taken
- **Intrinsics**
Inlining (C1 + C2)

```java
int a = 3;
int b = 4;
int result = add(a, b);
...
public int add(int x, int y) {  return x + y;  }

int result = a + b;
```
Branch Prediction (C2)

// make an array of random doubles 0..1
double[] bigArray = makeBigArray(1_000_000);

for (int i = 0; i < bigArray.length; i++)
{
    double cur = bigArray[i];
    if (cur > 0.5) { doThis();} else { doThat();}
}

// branch will be taken ~50% of time
// sorting the array will make it more predictable
Setting up

JITWatch Configuration

Source locations
- /home/chris/jdk1.8.0/src.zip
- /home/chris/workspace/jw/src/main/java

Class locations
- /home/chris/workspace/jw/target/classes

Buttons:
- Add File(s)
- Add Folder
- Remove
- Add JDK src
- Add File(s)
- Add Folder
- Remove
Compile tree
Compilations timeline
Toplists

• **Bytecode size**
• Native code size
• **Inlining failure reasons**
• Most-used intrinsics
• Compilation order
• Most-decompiled methods
  – Compiler assumption was wrong
# Toplists – Inline failure reasons

<table>
<thead>
<tr>
<th>Value</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1526</td>
<td>too big</td>
</tr>
<tr>
<td>523</td>
<td>already compiled into a big method</td>
</tr>
<tr>
<td>499</td>
<td>call site not reached</td>
</tr>
<tr>
<td>476</td>
<td>already compiled into a medium method</td>
</tr>
<tr>
<td>457</td>
<td>executed &lt; MinInliningThreshold times</td>
</tr>
<tr>
<td>242</td>
<td>hot method too big</td>
</tr>
<tr>
<td>233</td>
<td>never executed</td>
</tr>
<tr>
<td>77</td>
<td>native method</td>
</tr>
<tr>
<td>18</td>
<td>unloaded signature classes</td>
</tr>
<tr>
<td>1</td>
<td>NodeCountInliningCutoff</td>
</tr>
</tbody>
</table>
Compile times
Histogram – Inlined method sizes

Small methods inlined aggressively
private void testCallChain(long iterations) {
    long count = 0;
    for (int i = 0; i < iterations; i++) {
        count = chainA1(count);
        count = chainB1(count);
    }
    logger.info("testCallChain: {}", count);
}

private long chainA1(long count) {
    return 1 + chainA2(count);
}

private long chainA2(long count) {
    return 2 + chainA3(count);
}

private long chainA3(long count) {
    return 3 + chainA4(count);
}

 approves testCallChain testCallChain

bytecode (double click for JVM)

Assembly

Compiled with C2
if_icmp<cond>

Operation
Branch if int comparison succeeds

Format

if_icmp<cond>
branchbyte1
branchbyte2

Forms
if_icmpeq = 159 (0x9f)
if_icmpne = 160 (0xa0)
if_icmplt = 161 (0xa1)
if_icmpge = 162 (0xa2)
if_icmpgt = 163 (0xa3)
if_icmple = 164 (0xa4)

Operand Stack

..., value1, value2 →
# Code Suggestion Tool

<table>
<thead>
<tr>
<th>Score</th>
<th>Type</th>
<th>Caller</th>
<th>Suggestion</th>
</tr>
</thead>
<tbody>
<tr>
<td>21904</td>
<td>Branch</td>
<td>com.chrisnewland.jitwatch.demo.MakeHotSpotLog private void randomBranchTest(int)</td>
<td>Method contains an unpredictable branch at bytecode 24 that was observed 43807 times and is taken with probability 0.50129. It may be possible to modify the branch (for example by sorting a collection before iterating) to make it more predictable.</td>
</tr>
<tr>
<td>18634</td>
<td>Branch</td>
<td>java.lang.Integer public static integer valueOf(int)</td>
<td>Method contains an unpredictable branch at bytecode 3 that was observed 37268 times and is taken with probability 0.498927. It may be possible to modify the branch (for example by sorting a collection before iterating) to make it more predictable.</td>
</tr>
<tr>
<td>18002</td>
<td>Branch</td>
<td>java.lang.Integer public static integer valueOf(int)</td>
<td>Method contains an unpredictable branch at bytecode 3 that was observed 36004 times and is taken with probability 0.499195. It may be possible to modify the branch (for example by sorting a collection before iterating) to make it more predictable.</td>
</tr>
<tr>
<td>12673</td>
<td>Inlining</td>
<td>com.chrisnewland.jitwatch.demo.MakeHotSpotLog private long chainA4(long)</td>
<td>The call at bytecode 3 to Class: com.chrisnewland.jitwatch.demo.MakeHotSpotLog Member: private long bigMethod(long,int) was not inlined for reason: 'hot method too big' The callee method is 'hot' but is too big to be inlined into the caller. You may want to consider refactoring the callee into smaller methods. Invocations: 12673 Size of callee bytecode: 350</td>
</tr>
<tr>
<td>12673</td>
<td>Inlining</td>
<td>com.chrisnewland.jitwatch.demo.MakeHotSpotLog public void tooBigToinline(int)</td>
<td>The call at bytecode 15 to Class: com.chrisnewland.jitwatch.demo.MakeHotSpotLog Member: private long bigMethod(long,int) was not inlined for reason: 'hot method too big' The callee method is 'hot' but is too big to be inlined into the caller. You may want to consider refactoring the callee into smaller methods.</td>
</tr>
</tbody>
</table>
JarScan Tool

- Statical analysis of a jar
- Methods with bytecode > inlining threshold
- These methods might not be hot
- Around 3000 non-inlineable methods in rt.jar
  - String.split
  - String.toUpperCase / toLowerCase
  - Core parts of j.u.ComparableTimSort
TL;DR

- Eliminate other performance issues first
- Keep your methods small for inlining
- Turn on JIT logging
  - JITWatch suggestion tool
    - “hot method too big”
    - Unpredictable branches
- Learn about the JVM :)

Premature optimization is the root of all evil

Donald Knuth
Resources

- JITWatch on GitHub
  - http://www.github.com/AdoptOpenJDK/jitwatch
  - AdoptOpenJDK project
  - Send a pull request!

- Mailing list
  - groups.google.com/jitwatch

- Twitter
  - @chriswhocodes

Thanks!