Top 10 Issues for Java in Production

SriSatish Ambati
Cliff Click Jr.
Azul Systems, Inc
A Decade of Java in Production

• A lot of hard-earned wisdom
• A lot of victories (quickly forgotten)
• A lot of endless pain points
• Usually the Pain Point is really

A Systems Issue

• It's Not Just the JVM (nor network, nor ...)
Tools of the Trade

• What the JVM is doing:
  – dtrace, hprof, introscope, jconsole, visualvm, yourkit, azul zvision

• Invasive JVM observation tools:
  – bci, jvmti, jvmdi/pi agents, logging

• What the OS is doing:
  – dtrace, oprofile, vtune

• What the network/disk is doing:
  – ganglia, iostat, lsof, nagios, netstat
• Symptom
  – Production monitoring can be very expensive
    Staging environment does not repro issues
  – Instrumented code changes cache profile
  – MBeans are not cheap either!

• Solutions
  – Pick the right axe for the problem!
  – Avoid expensive heap walks
  – Finish task then increment perf counters
  – Asynchronous logging, jconsole, azul zvision
9 - Leaks

• Symptom
  – App consumes all the memory you got
  – Live Heap trend is a ramping sawtooth
  – Then slows, then throws OutOfMemory

• Tools
  – yourkit, hprof, eclipse mat, jconsole, jhat, jps, visualvm, azul zvision

• Theory
  – Allocated vs Live Objects, vm memory, Perm Gen
  – Finalizers, ClassLoaders, ThreadLocal
Leaks: jconsole

- Tomcat + ActiveMQ
  - 1 week in production
  - after 9hrs in test
  - Leaks 100MB/hr
Leaks: Visual VM

14741 classes loaded
ClassLoader leak,
PermGen full
9 – Leaks: Bloat

• Cost of an 8-char String?

<table>
<thead>
<tr>
<th>8b</th>
<th>12b</th>
<th>4b</th>
</tr>
</thead>
<tbody>
<tr>
<td>hdr</td>
<td>fields</td>
<td>ptr</td>
</tr>
</tbody>
</table>

A: 56 bytes, or a 7x blowup

• Cost of 100-entry TreeMap<Double,Double> ?

<table>
<thead>
<tr>
<th>8b</th>
<th>4b</th>
<th>16b</th>
<th>4b</th>
</tr>
</thead>
<tbody>
<tr>
<td>hdr</td>
<td>len</td>
<td>data</td>
<td>pad</td>
</tr>
</tbody>
</table>

A: 7248 bytes or a ~5x blowup
### JBoss 5.1
#### Allocated

<table>
<thead>
<tr>
<th>Class name</th>
<th>Size (B)</th>
<th>Count</th>
<th>Avg (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>1,410,764,512</td>
<td>19,830,135</td>
<td>71.1</td>
</tr>
<tr>
<td>char[]</td>
<td>423,372,528</td>
<td>4,770,424</td>
<td>88.7</td>
</tr>
<tr>
<td>byte[]</td>
<td>347,332,152</td>
<td>1,971,692</td>
<td>176.2</td>
</tr>
<tr>
<td>int[]</td>
<td>85,509,280</td>
<td>1,380,642</td>
<td>61.9</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>73,623,024</td>
<td>3,067,626</td>
<td>24</td>
</tr>
<tr>
<td>java.lang.Object[]</td>
<td>64,788,840</td>
<td>565,693</td>
<td>114.5</td>
</tr>
<tr>
<td>java.util.regex.Matcher</td>
<td>51,448,320</td>
<td>643,104</td>
<td>80</td>
</tr>
<tr>
<td>java.lang.reflect.Method</td>
<td>43,374,528</td>
<td>301,212</td>
<td>144</td>
</tr>
<tr>
<td>java.util.HashMap$Entry[]</td>
<td>27,876,848</td>
<td>140,898</td>
<td>197.9</td>
</tr>
<tr>
<td>java.util.TreeMap$Entry</td>
<td>22,116,136</td>
<td>394,931</td>
<td>56</td>
</tr>
<tr>
<td>java.util.HashMap$Entry</td>
<td>19,806,440</td>
<td>495,161</td>
<td>40</td>
</tr>
<tr>
<td>java.nio.HeapByteBuffer</td>
<td>17,582,928</td>
<td>366,311</td>
<td>48</td>
</tr>
<tr>
<td>java.nio.HeapCharBuffer</td>
<td>17,575,296</td>
<td>366,152</td>
<td>48</td>
</tr>
<tr>
<td>java.lang.StringBuilder</td>
<td>15,322,128</td>
<td>638,422</td>
<td>24</td>
</tr>
<tr>
<td>java.util.TreeMap$EntryIterator</td>
<td>15,056,784</td>
<td>313,683</td>
<td>48</td>
</tr>
<tr>
<td>java.util.ArrayList</td>
<td>11,577,480</td>
<td>289,437</td>
<td>40</td>
</tr>
<tr>
<td>java.util.HashMap</td>
<td>7,829,056</td>
<td>122,329</td>
<td>64</td>
</tr>
<tr>
<td>java.util.TreeMap</td>
<td>7,754,688</td>
<td>107,704</td>
<td>72</td>
</tr>
<tr>
<td>java.lang.Class[]</td>
<td>570,808</td>
<td>45,354</td>
<td>20.3</td>
</tr>
<tr>
<td>Built-in VM methodKlass</td>
<td>922,808</td>
<td>45,354</td>
<td>20.3</td>
</tr>
<tr>
<td>Built-in VM constantPoolKlass</td>
<td>643,104</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>Built-in VM constantPoolCacheKlass</td>
<td>140,898</td>
<td>197.9</td>
<td></td>
</tr>
<tr>
<td>java.lang.Class</td>
<td>73,623,024</td>
<td>3,067,626</td>
<td>24</td>
</tr>
<tr>
<td>java.lang.reflect.Method</td>
<td>51,448,320</td>
<td>643,104</td>
<td>80</td>
</tr>
<tr>
<td>java.lang.reflect.Method</td>
<td>43,374,528</td>
<td>301,212</td>
<td>144</td>
</tr>
<tr>
<td>java.util.HashMap$Entry[]</td>
<td>27,876,848</td>
<td>140,898</td>
<td>197.9</td>
</tr>
<tr>
<td>java.util.TreeMap$Entry</td>
<td>22,116,136</td>
<td>394,931</td>
<td>56</td>
</tr>
<tr>
<td>java.util.HashMap$Entry</td>
<td>19,806,440</td>
<td>495,161</td>
<td>40</td>
</tr>
<tr>
<td>java.nio.HeapByteBuffer</td>
<td>17,582,928</td>
<td>366,311</td>
<td>48</td>
</tr>
<tr>
<td>java.nio.HeapCharBuffer</td>
<td>17,575,296</td>
<td>366,152</td>
<td>48</td>
</tr>
<tr>
<td>java.lang.StringBuilder</td>
<td>15,322,128</td>
<td>638,422</td>
<td>24</td>
</tr>
<tr>
<td>java.util.TreeMap$EntryIterator</td>
<td>15,056,784</td>
<td>313,683</td>
<td>48</td>
</tr>
<tr>
<td>java.util.ArrayList</td>
<td>11,577,480</td>
<td>289,437</td>
<td>40</td>
</tr>
<tr>
<td>java.util.HashMap</td>
<td>7,829,056</td>
<td>122,329</td>
<td>64</td>
</tr>
<tr>
<td>java.util.TreeMap</td>
<td>7,754,688</td>
<td>107,704</td>
<td>72</td>
</tr>
</tbody>
</table>

### Apache Tomcat 6.0
#### Allocated

<table>
<thead>
<tr>
<th>Class name</th>
<th>Size (B)</th>
<th>Count</th>
<th>Avg (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>21,580,592</td>
<td>228,805</td>
<td>94.3</td>
</tr>
<tr>
<td>char[]</td>
<td>4,215,784</td>
<td>48,574</td>
<td>86.8</td>
</tr>
<tr>
<td>byte[]</td>
<td>3,683,984</td>
<td>5,024</td>
<td>733.3</td>
</tr>
<tr>
<td>Built-in VM methodKlass</td>
<td>2,493,064</td>
<td>16,355</td>
<td>152.4</td>
</tr>
<tr>
<td>Built-in VM constMethodKlass</td>
<td>1,955,696</td>
<td>16,355</td>
<td>119.6</td>
</tr>
<tr>
<td>Built-in VM constantPoolKlass</td>
<td>1,437,240</td>
<td>1,284</td>
<td>1,119.30</td>
</tr>
<tr>
<td>Built-in VM instanceKlass</td>
<td>1,078,664</td>
<td>1,284</td>
<td>840.1</td>
</tr>
<tr>
<td>java.lang.Class[]</td>
<td>922,808</td>
<td>45,354</td>
<td>20.3</td>
</tr>
<tr>
<td>Built-in VM constantPoolCacheKlass</td>
<td>903,360</td>
<td>1,132</td>
<td>798</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>73,623,024</td>
<td>3,067,626</td>
<td>24</td>
</tr>
<tr>
<td>java.lang.Object[]</td>
<td>64,788,840</td>
<td>565,693</td>
<td>114.5</td>
</tr>
<tr>
<td>java.util.regex.Matcher</td>
<td>51,448,320</td>
<td>643,104</td>
<td>80</td>
</tr>
<tr>
<td>java.lang.reflect.Method</td>
<td>43,374,528</td>
<td>301,212</td>
<td>144</td>
</tr>
<tr>
<td>java.util.HashMap$Entry[]</td>
<td>27,876,848</td>
<td>140,898</td>
<td>197.9</td>
</tr>
<tr>
<td>java.util.TreeMap$Entry</td>
<td>22,116,136</td>
<td>394,931</td>
<td>56</td>
</tr>
<tr>
<td>java.util.HashMap$Entry</td>
<td>19,806,440</td>
<td>495,161</td>
<td>40</td>
</tr>
<tr>
<td>java.nio.HeapByteBuffer</td>
<td>17,582,928</td>
<td>366,311</td>
<td>48</td>
</tr>
<tr>
<td>java.nio.HeapCharBuffer</td>
<td>17,575,296</td>
<td>366,152</td>
<td>48</td>
</tr>
<tr>
<td>java.lang.StringBuilder</td>
<td>15,322,128</td>
<td>638,422</td>
<td>24</td>
</tr>
<tr>
<td>java.util.TreeMap$EntryIterator</td>
<td>15,056,784</td>
<td>313,683</td>
<td>48</td>
</tr>
<tr>
<td>java.util.ArrayList</td>
<td>11,577,480</td>
<td>289,437</td>
<td>40</td>
</tr>
<tr>
<td>java.util.HashMap</td>
<td>7,829,056</td>
<td>122,329</td>
<td>64</td>
</tr>
<tr>
<td>java.util.TreeMap</td>
<td>7,754,688</td>
<td>107,704</td>
<td>72</td>
</tr>
</tbody>
</table>

**Million Objects allocated**

- **JBoss 5.1**: 20M objects
- **Apache Tomcat 6.0**: 4M objects

**Live**

- **JBoss 5.1**: 20M objects
- **Apache Tomcat 6.0**: 4M objects

**JEE is not cheap!**

JBoss & Apache startup
- 20M objects *before* starting the app
example: yourkit memory profiling

Know footprint: use memory profiling!
(snapshots are still expensive)
I will not leak the ending of this talk!
I will not leak the ending of this talk!
I will not leak the ending of this talk!
I will not leak the ending of this talk!
I will not leak the ending of this talk!
I will not leak the ending of this talk!
I will not leak the ending of this talk!
8 – I/O: Serialization

• Symptom
  – Multi-node scale-out does not scale linearly
  – Time in both CPU and I/O (serialization costs)

• Tools
  – Cpu profiling, I/O profiling

• Solution
  – All serialization libraries are not equal!
  – Pick a high performance serialization library or roll-your-own
  – Avro, kryo, protocol-buffers, thrift
8 – I/O: Limits, Tuning

• Symptom
  – Application hangs or remote call fails after awhile
  – “Too many open File Descriptors”, “Cursors”
  – Inconsistent response times

• Tools
  – nagios, pkg, rpm info, ulimit, yum

• Solutions
  – Check for “new” OS patches, user & process limits, network & semaphore configurations
  – Close all I/O streams
  – Maybe you are I/O bound!
• Symptoms
  – Socket.create/close takes too long
  – JRMP timeouts, long JDBC calls
  – Running out of file descriptors, cursors, disk
• Tools
  – dbms tools, du, iostat, gmon, lsof, netstat
• Workaround
  – Check all O/S patches, sysctl flags, run ping/telnet test
  – Check & set SO_LINGER, TCP_LINGER2
Excessive logging will cause floods!
• Symptoms
  – Adding users / threads / CPUs causes app slow down (less throughput, worse response)
  – High lock acquire times & contention
  – Race conditions, deadlock, I/O under lock
• Tools
  – d-trace, lockstat, azul zvision
• Solution
  – Use non-blocking Collections
  – Striping locks, reducing hold times, no I/O
Example: IBM Visual Analyzer
(j.u.c view in eclipse)

Zillion threads acquiring same lock
j.u.c.ConcurrentLock is still a lock!
Need a non-blocking collection
(or stripe lock or lower hold times, etc)
Hot lock is usually 10x to 100x more acquire time than next lock.
Look for rapidly growing acquire times!
Lock Statistics of `org.apache.catalina.session.StandardManager`

Contention Tree

1. 94.46% 4,601,974ms 30,347 `org.apache.catalina.session.ManagerBase.generateSessionId` (ManagerBase.java:959, bci=-1)
   2. 100.00% 4,601,974ms 30,347 `org.apache.catalina.session.ManagerBase.createSession` (ManagerBase.java:801, bci=43)
      3. 100.00% 4,601,974ms 30,347 `org.apache.catalina.session.StandardManager.createSession` (StandardManager.java:291, bci=49)
      4. 100.00% 4,601,974ms 30,347 `org.apache.catalina.connector.Request.doGetSession` (Request.java:2232, bci=245)
      5. 100.00% 4,601,974ms 30,347 `org.apache.catalina.connector.Request.getSession` (Request.java:2031, bci=2)

1. 5.54% 269,912ms 7,167 `org.apache.catalina.session.ManagerBase.generateSessionId` (ManagerBase.java:959, bci=-3)
   2. 100.00% 269,912ms 7,167 `org.apache.catalina.session.ManagerBase.createSession` (ManagerBase.java:801, bci=43)
      3. 100.00% 269,912ms 7,167 `org.apache.catalina.session.StandardManager.createSession` (StandardManager.java:291, bci=49)
      4. 100.00% 269,912ms 7,167 `org.apache.catalina.connector.Request.doGetSession` (Request.java:2232, bci=245)
      5. 100.00% 269,912ms 7,167 `org.apache.catalina.connector.Request.getSession` (Request.java:2031, bci=2)

Hot Lock Backtrace
6 – Endless Compilation

• Symptom
  – Time “compiling”
  – Time in the Interpreter

• Tools
  – -XX:+PrintCompilation, cpu profiler
  – Find endlessly-recompiling method

• Workaround
  – Exclude using .hotspot_compiler file

• Root cause: It's a JVM Bug! File a bug report!
5 – Endless Exceptions

• Symptom
  – Application spends time in j.l.T.fillInStackTrace()

• Tools
  – Cpu profiler, azul zvision
  – Thread dumps (repeated kill -3, zvision)
  – Track caller/callee to find throw'r
    • Not all exceptions appear in log files

• Solution
  – Don't Throw, alternate return value (e.g. null)
5 – Endless Exceptions

• Related
  – Exception paths are typically failure paths
  – JVMs do not optimize them much
  – Often found when a server collapses
4 - Fragmentation

• Symptom
  – Performance degrades over time
  – Inducing a “Full GC” makes problem go away
  – Lots of free memory but in tiny fragments

• Tools
  – GC logging flags, e.g. for CMS
    –XX:PrintFLSStatistics=2
    –XX:+PrintCMSInitiationStatistics
4 - Fragmentation

• Tools
  – “Fragger”
    • Tiny cpu cost, low memory cost
    • Frag's heap in 60sec like an hour in production
    • Get FullGC cycles at dev's desk

• Solution
  – Upgrade to latest CMS (CR:6631166)
  – Azul Zing & Gen Pauseless GC
  – Pooling similar sized/aged objects
    • (really hard to get right!)
3 – GC Tuning

- **Symptom**
  - Entropy(gc) == number_of_gc_flags
    - Too many free parameters
    - 64-bit/large heap size is not a solution
  - Constant 40-60% CPU utilization by GC
  - Scheduled reboot before full-GC
  - Full time Engineer working GC flags;

- **Workarounds**
  - Ask JVM Vendor to give 1 flag solution
  - G1 GC, Azul’s Zing GPGC
3 – GC Tuning
Oracle Weblogic GC Flags

-server -Xloggc:gc.log -XX:+PrintGCDetails
-XX:+PrintGCTimeStamps -XX:MaxPermSize=128m
-XX:+UseParNewGC -XX:+UseConcMarkSweepGC
-XX:MaxNewSize=64m -XX:NewSize=64m
-Xms1536m -Xmx1536m -XX:SurvivorRatio=128
-XX:MaxTenuringThreshold=0
-XX:CMSInitiatingOccupancyFraction=60
-Dsun.rmi.dgc.server.gcInterval=0x7FFFFFFF
FFFFFE
-Dsun.rmi.dgc.client.gcInterval=0x7FFFFFFF
FFFFFE
2 - Spikes

• Symptoms
  – Rush hour traffic, tax day, Black Friday
  – Outages under spikes, power law of networks

• Solution
  – Measure.
  – Test with realistic load & realistic multi-node setup
  – Build redundancy & elasticity into infrastructure
  – Don’t ignore Exceptions & retries under load
Busiest online day is...
• Symptom
  – Different nodes have different configurations, different stack components, versions
  – classpath has dist/*, -verbose:class
  – subtle hard to reproduce issues

• Solution
  – Good ol’ fashioned rigor

“It can only be attributable to human error” - HAL
0 – Collapse Under Load (pick any 3 above!)

• Runs fine as load Ramps Up
  – At peak load, system is unstable
  – Slightly above peak: Collapse!
• Heavy load triggers exception (e.g. timeout)
• Exception path is slow already (e.g. logging)
• Transaction retried (so more work yet again)
• So NEXT transaction times-out
• Soon all time spent throwing & logging exceptions
• No forward progress
example:
Driving into San Francisco
References:
Java.util.concurrent lock profiling
  http://infoq.com/jucprofiler
Java serialization benchmarks
Memory profiling with yourkit
  http://yourkit.com
Tuning gc
  http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html
  http://blog.codecentric.de/en/2010/01/
    java-outofmemoryerror-a-tragedy-in-seven-acts/
Cliff Click's High Scale lib, Non-Blocking HashMap
  http://sourceforge.net/projects/high-scale-lib/
References:

Memory Leak

http://deusch.org/blog/?p=9

Handy list of jvm options


Fragger (with source code)

http://www.azulsystems.com/resources

Garbage Collection: Algorithms for Automatic Dynamic Memory Management, Richard Jones, Rafael D Lins
Backup slide–Fragmentation

- Works well for hours at 300-400MB
  - Same workload
- Suddenly haywire
  - Promotion
    - Too frequently
  - Back to back FullGCs
  - May not all be completing.