Debugging Java performance problems

Ryan Matteson
matty91@gmail.com
http://prefetch.net
Overview

• Tonight I am going to discuss Java performance, and how opensource tools can be used to debug performance problems on production servers
• I plan to split my 60-minutes into three parts:
  – Part 1 will provide an overview of the tools
  – Part 2 will show these tools in action
  – Part 3 will be a Q&A period
• Note: Several of the tools described in the presentation require Solaris 10 and a recent Java 6 release (Java 6 update 4 was used for testing)
Typical Performance problems

• Performance and scalability problems typically fall into three major categories:
  – Inefficient use of CPU resources
  – Inefficient use of memory resources
  – Lock contention

• The following slides will describe how to locate CPU, memory and lock contention problems in Java applications
CPU performance problems

- CPU performance problems occur when one or more threads saturate the available CPU resources, resulting in system-wide scheduling latencies
- Typical CPU problems
  - Object allocation leading to excessive garbage collection (GC)
  - Inefficient methods
  - Improper use of class libraries
  - Runaway threads
Identifying CPU problems

• If mpstat or vmstat shows high CPU utilization, the Solaris prstat utility can be run with the “-L” option to break down CPU utilization by thread:

```
$ prstat -Lam
PID   USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/LWPID
19512  matty    50  0.0  0.1  0.0  0.0  0.0  50   0  151  0  0 java/2
```

• The output above contains the process id and process name, microstate accounting data, the thread name and the thread identifier
Identifying CPU problems (cont.)

• Once the id of a CPU bound thread is located, the jstack utility can be used to retrieve a stack trace for each thread in a process:

  $ jstack <VMID>

  "main" prio=10 tid=0x080ce400 nid=0x2 runnable [0xfe698000..0xfe698d48] \
    java.lang.Thread.State: RUNNABLE
    at com.sun.demo.jvmti.hprof.Tracker.nativeReturnSite(Native Method)
    at com.sun.demo.jvmti.hprof.Tracker.ReturnSite(Tracker.java:74)
    at java.lang.Math.random(Math.java:695)
    at TestMain.foo(TestMain.java:15)
    at TestMain.main(TestMain.java:9)

• Jstack will print the name of each thread, the thread id, the full class name, the thread state and a Java stack trace

• The stack trace can be used to determine which code to analyze for potential problems, or as a starting point for further debugging.
Debugging method calls with DTrace

- The Dtrace hotspot provider contains probes that fire upon method entry and exit:
  - method-entry - fires each time a method is entered
  - method-return - fires each time a method is exited
- The method probes are extremely useful for identifying which methods are consuming CPU resources on a system, and can be used to correlate system events to application events
- To use the method probes, you will first need to enable them with the jinfo utility:
  $ jinfo -flag -XX:+DTraceMethodProbes<VMID>
- The probes will add additional overhead to the Java runtime, and should be disabled once profiling is complete:
  $ jinfo -flag -XX:-DTraceMethodProbes <VMID>
Debugging method calls with Dtrace (cont.)

- To view the most active Java call stacks on a system, the DTrace profile provider can be used to sample the most frequently executing Java call stacks:

  $ dtrace -o busymethod.out -n 'profile-1001hz / pid == 19512 & tid == 2 /
  \{
    @callstacks[jstack()] = count();
  }$

- After you hit cntrl+c to stop DTrace, the output file can be filtered with c++filt to demangle C++ symbols:

  $ c++filt busymethods.out |more
Debugging method calls with Dtrace (cont.)

• This will produce output similar to the following (output edited for readability):

```
Libjvm.so`long long java_lang_Thread::thread_id(oopDesc*)+0x2a
libjvm.so`long long SharedRuntime::get_java_tid(Thread*)+0x3a
libjvm.so`intSharedRuntime::dtrace_method_entry \
  (JavaThread*,methodOopDesc*)+0x5c
java/util/Random.next(I)I*
TestMain.main([Ljava/lang/String;;)V*
StubRoutines (1)
libjvm.so`void
libc.so.1`_thr_setup+0x52
libc.so.1`_lwp_start
24
```
Debugging method calls with Dtrace (cont.)

- To list the busiest methods in an application by CPU time consumed, the DTraceToolkit j_cputime.d script can be used:

```
$ j_cputime.d 3
Top 3 counts,
PID TYPE NAME COUNT
20221 method java/lang/String.getChars 76183
20221 method java/lang/System.arraycopy 167600
0 total - 944774

Top 3 exclusive method on-CPU times (us),
PID TYPE NAME TOTAL
20221 method java/util/Arrays.copyOf 174943
20221 method java/lang/System.arraycopy 189579
0 total - 1346227

Top 3 inclusive method on-CPU times (us),
PID TYPE NAME TOTAL
20221 method java/lang/AbstractStringBuilder.expandCapacity 339847
20221 method java/lang/AbstractStringBuilder.append 664615
20221 method java/lang/StringBuilder.append 772450
```
Offline CPU analysis

• The Java heap profiling agent can be used to profile Java method utilization

• There are two profiling methods:
  – **Sampling** - samples the call stacks at periodic intervals to determine the top methods
  – **Time based** - Uses byte code injection (BCI) to instrument the entry and return points in each method

• Time based profiling is much more accurate, but introduces significantly more overhead
Time based CPU profiling

• Time based CPU profiling can be enabled by loading the heap profiling agent with the cpu flag set to times:
  $ java -agentlib:hprof=cpu=times App

• When the process exits or receives a control signal, the agent will write the profiling results to a file named java.hprof.txt
Time based CPU profiling (cont.)

• The profiling report will contain output similar to the following:

```
CPU TIME (ms) BEGIN (total = 1022085444) Fri Jan 25 17:21:16 2008
rank  self  accum  count  trace  method
 1 18.65% 18.65% 252985704 301144 java.util.Random.nextDouble
 2 11.93% 30.58% 252985704 301145 java.lang.Math.random
 3 10.13% 40.72% 505971408 301142 java.util.concurrent.atomic.AtomicLong.compareAndSet
 4 10.06% 50.77% 505971409 301141 java.util.concurrent.atomic.AtomicLong.get
 5  9.39% 60.16% 126492852 301146 TestMain.foo
 6  2.58% 62.74% 126492853 301078 TestMain.<init>
```

• Trace identifiers can be used to correlate methods to stack traces:

```
TRACE 301144:
    java.util.Random.nextDouble(Random.java:Unknown line)
    java.lang.Math.random(Math.java:Unknown line)
    TestMain.foo(TestMain.java:Unknown line)
    TestMain.main(TestMain.java:Unknown line)
```
Fixing CPU performance problems

• Identifying the problem is the key to fixing issues related to CPU utilization
• Use the tools described above to collect data that you can show your developers
• If you are the developer, check the Java API reference for unintended behaviors, and write test cases to see why code is not performing as expected
Memory performance problems

• Java memory problems are probably the single biggest source of performance problems and scalability bottlenecks

• Several types of problems exist:
  – Improper allocation strategies
  – Unintentional object references

• Memory related problems typically surface in the way of OOM (out of memory) errors, long pauses due to garbage collection and heap fragmentation
Viewing memory utilization

- The Java SDK comes with the jstat utility to print utilization statistics for each Java generation:

  $ jstat -gc `pgrep java` 1000

<table>
<thead>
<tr>
<th>S0C</th>
<th>S1C</th>
<th>S0U</th>
<th>S1U</th>
<th>EC</th>
<th>EU</th>
<th>OC</th>
<th>OU</th>
<th>PC</th>
<th>PU</th>
<th>YGC</th>
<th>YGCT</th>
<th>FGC</th>
<th>FGCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>17856.0</td>
<td>17856.0</td>
<td>0.0</td>
<td>8004.2</td>
<td>357504.0</td>
<td>194217.5</td>
<td>131072.0</td>
<td>0.0</td>
<td>16384.0</td>
<td>6178.4</td>
<td>1</td>
<td>0.186</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0.057</td>
<td>0.242</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17856.0</td>
<td>17856.0</td>
<td>0.0</td>
<td>8004.2</td>
<td>357504.0</td>
<td>235835.7</td>
<td>131072.0</td>
<td>0.0</td>
<td>16384.0</td>
<td>6178.4</td>
<td>1</td>
<td>0.186</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0.057</td>
<td>0.242</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17856.0</td>
<td>17856.0</td>
<td>0.0</td>
<td>8004.2</td>
<td>357504.0</td>
<td>263580.6</td>
<td>131072.0</td>
<td>0.0</td>
<td>16384.0</td>
<td>6178.4</td>
<td>1</td>
<td>0.186</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0.057</td>
<td>0.242</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17856.0</td>
<td>17856.0</td>
<td>8002.5</td>
<td>0.0</td>
<td>357504.0</td>
<td>0.0</td>
<td>131072.0</td>
<td>0.0</td>
<td>16384.0</td>
<td>6178.4</td>
<td>2</td>
<td>0.381</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0.057</td>
<td>0.437</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- There are also options to print class load activity, hotspot compiler statistics, new and old generation memory usage, and the reason why a GC event occurred.

  Additional information is available in the jstat(1) man page.
Debugging memory problems with DTrace

- The DTrace hotspot provider contains object allocation and garbage collection (GC) probes that can be used to observe object allocation, and collections that result from these allocations:
  - `gc-begin` - fires when a collection is about to occur
  - `gc-end` - fires when a collection finishes
  - `mem-pool-gc-begin` - fires when an individual memory pool is about to be collected
  - `mem-pool-gc-end` - fires when an individual memory pool collection finishes
  - `object-alloc` - fires when an object is allocated
Debugging memory problems with DTrace (cont.)

• The newobjects.d, objectsize.d and whoallocatebybytes.d DTrace scripts utilize these probes to measure the number the objects allocated, the size of allocations and the Java call stacks responsible for the allocations.

• Prior to using the object allocation probes, the probes need to be enabled with jinfo:

  \$ \texttt{jinfo -flag -XX:+DTraceAllocProbes<VMID>}

• The probes will add additional overhead to the Java runtime, and should be disabled once profiling is complete:

  \$ \texttt{jinfo -flag -XX:-DTraceAllocProbes<VMID>}

Viewing object allocations

- The newobjects.d script will print the number of objects created and the total number of bytes allocated for each type of object:

```
$ newobjects.d

<table>
<thead>
<tr>
<th>Class</th>
<th>Objects created</th>
<th>Bytes Allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>[l</td>
<td>112</td>
<td>16064</td>
</tr>
<tr>
<td>Wasteful</td>
<td>17943</td>
<td>287088</td>
</tr>
<tr>
<td>java/lang/StringBuilder</td>
<td>17943</td>
<td>287088</td>
</tr>
<tr>
<td>java/lang/String</td>
<td>17943</td>
<td>430632</td>
</tr>
<tr>
<td>[C</td>
<td>107658</td>
<td>52393560</td>
</tr>
</tbody>
</table>
```
Viewing object sizes

- The objectsize.d script will print a distribution of object sizes:

```
$ objectsize.d
value  ------------ Distribution ------------ count

8      |0
16     |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 95897
32     |@@@@@@@@@@@@@@@@@@@@@@@@@@    31965
64     |1
128    |@@@@@@@@@@@@@@@@@@@@@@@@@@   32157
256    |@@@@@@@@@@@@@@@@@@@@@@@@@@  31965
512    |@@@@@@@@@@@@@@@@@@@@@@@@@@@  63932
1024   |@@@@@@@@@@@@@@@@@@@@@@@@@@ @ 31967
2048   |0
```
Identifying object allocation points

- The whoallocatebybytes.d script can be used to correlate Java call stacks to the number of bytes they allocated:

```bash
$ whoallocatebybytes.d
Wasteful.main([Ljava/lang/String;)V
StubRoutines (1)
libjvm.so`_pnGThread__v_+0×1a3
libjvm.so`jni_CallStaticVoidMethod+0×15d
java`JavaMain+0xd30
libc.so.1`_thr_setup+0×52
libc.so.1`_lwp_start
  817632
```
Monitoring memory usage with visualgc

• The visualgc utility is a freely downloadable tool from Sun, which can be used to graphically monitor the following Java runtime subsystems:
  – Classloader activity
  – Garbage collection activity
  – Hotspot compiler activity

• To use visualgc, you can run visualgc with the process id of the Java process to monitor:

  $ visualgc <VMID>
VisualGC output
Offline memory analysis

- The Java heap profiling agent can be used to profile Java object allocations.
- Heap profiling can be enabled by loading the heap profiling agent with the heap flag set to sites:
  
  ```
  $ java -agentlib:hprof=heap=sites App
  ```

- When the process exits or receives a control signal, the agent will write the profiling results to a file named java.hprof.txt.
Offline memory analysis (cont.)

- The profiling report will contain data similar to the following:

<table>
<thead>
<tr>
<th>rank</th>
<th>percent</th>
<th>live</th>
<th>alloc'ed</th>
<th>stack class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>84.58%</td>
<td>84.58%</td>
<td>11088704</td>
<td>java.util.Arrays.copyOfRange(Objects.java:49)</td>
</tr>
<tr>
<td>2</td>
<td>92.34%</td>
<td>1017784</td>
<td>1876</td>
<td>java.lang.String.&lt;init&gt;(String.java:216)</td>
</tr>
<tr>
<td>3</td>
<td>95.20%</td>
<td>374928</td>
<td>15622</td>
<td>java.lang.StringBuilder.toString(StringBuilder.java:430)</td>
</tr>
<tr>
<td>4</td>
<td>97.63%</td>
<td>68992</td>
<td>98</td>
<td>Objects.allocateObjects(Objects.java:49)</td>
</tr>
</tbody>
</table>

- Trace identifiers can be used to correlate object allocations to the Java stack that is responsible for allocating the objects:

```
TRACE 300295:
    java.util.Arrays.copyOfRange(Objects.java:49)
    java.lang.String.<init>(String.java:216)
    java.lang.StringBuilder.toString(StringBuilder.java:430)
    Objects.allocateObjects(Objects.java:49)
```
Is memory allocation impacting performance?

- This depends on the nature of the application, the workload being performed, the type of client accessing the application, and the impact of latency on service delivery.
- You can measure application pauses due to GC (and the objects that resulted in the collection being performed) by adding the “-XX:+PrintGCApplicationStoppedTime” and “-Xloggc:gc.log” options to the java command line.
- This will produce log entries similar to the following:

  Total time for which application threads were stopped: 0.0068135 seconds
  105.616: [GC 61914K->50759K(129472K), 0.0067066 secs]
  Total time for which application threads were stopped: 0.0070856 seconds
Fixing memory allocation problems

• Detecting the type of allocation issue and the problem site is the key to fixing memory related problems

• The Dtrace hotspot provider can be used to identify problems in production, and the Java heap profiling agent and jhat utility can be used to identify problems offline
Lock contention problems

- Lock contention issues occur when two or more threads need access to a specific resource to perform a unit of work, and are required to wait for a period of time to acquire the resource because another thread has control of the resource.

- Locking issues can be split into three categories:
  - Native lock contention (e.g., acquiring a mutex lock)
  - Java lock contention (e.g., acquiring a monitor)
Detecting native lock contention

- Solaris 10 ships with the plockstat utility, which provides an easy way to measure locks in native code:

```
$ plockstat -A -p <PID>
 Mutex hold

<table>
<thead>
<tr>
<th>Count</th>
<th>nsec</th>
<th>Lock</th>
<th>Caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>78</td>
<td>36496</td>
<td>libc.so.1`libc_malloc_lock</td>
<td>libjava.so`getString8859_1Chars+0x72</td>
</tr>
<tr>
<td>184</td>
<td>9848</td>
<td>libc.so.1`libc_malloc_lock</td>
<td>libjvm.so`_<em>1cCosGmalloc6Fl_pv</em>+0x2b</td>
</tr>
</tbody>
</table>

 Mutex spin

<table>
<thead>
<tr>
<th>Count</th>
<th>nsec</th>
<th>Lock</th>
<th>Caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7075</td>
<td>libc.so.1`libc_malloc_lock</td>
<td>libzip.so`Java_java_util_zip_Inflater_i</td>
</tr>
<tr>
<td>2</td>
<td>5383</td>
<td>libc.so.1`libc_malloc_lock</td>
<td>libjvm.so`_<em>1cCosEfree6Fpv_v</em>+0x1b</td>
</tr>
</tbody>
</table>
```
Debugging JVM lock contention with DTrace

• The Dtrace hotspot provider provides a large set of probes that can be used to measure lock contention:
  – **monitor-contended-enter** - fires when a thread attempts to enter a contended monitor
  – **monitor-contended-entered** - fires when a thread successfully enters a contended monitor
  – **monitor-contended-exit** - fires when a thread leaves a monitor that other threads are waiting to enter
  – **monitor-wait** - fires when a thread begins waiting on a monitor via a call to Object.wait()
  – **monitor-waited** - fires when a thread completes an Object.wait()
  – **monitor-notify** - fires when a thread calls Object.notify to notify threads that are waiting on a condition
  – **monitor-notifyAll** - fires when a thread calls Object.notifyAll to notify threads that are waiting on a monitor
Debugging JVM lock contention with DTrace (cont.)

- The `jlockstat.d` and `jlockpath.d` DTrace scripts can be used to measure how much time threads spend waiting for monitors, and which code paths are involved.
- Prior to using the scripts, `jinfo` needs be run to enable the Dtrace monitor probes:
  
  ```
  $ jinfo -flag -XX:+DTraceMonitorProbes <VMID>
  ```

- The probes will add additional overhead to the Java runtime, and should be disabled once profiling is complete:
  
  ```
  $ jinfo -flag -XX:-DTraceMonitorProbes <VMID>
  ```
Measuring Java monitor contention

- The jlockstat.d script measures the time spent waiting on contended monitors:

```
$ jlockstat.d
Monitor events (Cntrl+C to display events)
Classname       JVM pid  Count   Time (ms)
-----------------  ---------  -------  -------
ContentionObject1  739    748     181069190
ContentionObject2  739    532     98765
ContentionObject3  739    12      654
```
Correlating monitor contention to Java stacks

- The jlockpath.d script will collect a Java stack trace each time a monitor is contended, and display the stack traces along with the time they waited for a monitor when the script exits:

  ```
  $ jlockpath.d > lockstat.out

  $ cat lockstat.out | egrep -v '(libjvm|libc)'
  ```

- `monitorenter_nofpu Runtime1 stub`  
- `ContentionObject.update(I)V`  
- `ThreadObject.run()V`  
- `StubRoutines (1)`  
- `748`
Detecting deadlocks

- Deadlocks occur when two or more threads are waiting for a synchronization primitive that is owned by the other thread.
- The jstack utility will print deadlocks if they exist:
  $ jstack -l <VMID>
Offline lock analysis

- The Java heap profiling agent can be used to analyze an application for threads that are waiting to enter a monitor
- Monitor profiling can be enabled by loading the heap profiler agent with the monitor flag set to y:
  
  ```
  $ java -agentlib:hprof=monitor=y App
  ```
- When the process exits or receives a control signal, the agent will write the profiling results to a file named java.hprof.txt
Offline lock analysis (cont.)

MONITOR LContentionObject;
   owner: thread 200005, entry count: 1
   waiting to enter: thread 200028, thread 200027, thread 200026, thread 200025,
   thread 200024, thread 200023, thread 200022, thread 200021, thread 200020,
   thread 200019, thread 200018, thread 200017, thread 200016, thread 200015,
   thread 200014, thread 200013, thread 200012, thread 200011, thread 200010,
   thread 200009, thread 200008, thread 200007, thread 200006, thread 200004
   waiting to be notified:
MONITOR Ljava/lang/Shutdown$Lock;
   owner: thread 200030, entry count: 1
   waiting to enter:
   waiting to be notified:
Fixing lock contention issues

- Detecting the type of lock contention issue (native lock acquisition vs. Java monitor acquisition) and the problem site is the key to fixing lock contention problems
- Plockstat will show problematic code paths in native code, and the jlockpath.d Dtrace script will show problematic code paths in Java code
- Fixing lock contention issues often involves changes to the way an application is architected, since lock-free architectures and fine grained locking typically require
Summarizing JVM utilization with jconsole

• The jconsole utility can be used to summarize heap utilization, system utilization, class loader activity and much much more

• Also allows you to enable and disable options on a live JVM, which can be useful for setting profiling flags after startup (jinfo provides similar capabilities from the command line)
Jconsole output
Trending Java performance

- System utilization can be trended with a number of opensource packages:
  - MRTG
  - Orca
  - Cacti
- Java virtual machine performance can be trended using either JMX or the Java SNMP agent
- A script (jvmstat.pl) to trend JVM utilization is available on the prefetch.net website
Conclusion

• Java code can perform extremely well, and the tools described in this presentation should help you get to the bottom of any performance problems that may be occurring in your environment.

• Solaris 10 and opensolaris are ideal platforms for debugging java performance problems.

• Even if you don’t run Solaris in production, you can use Solaris and Dtrace to debug performance problems, and then transfer those performance wins back to your preferred operating system.
References

• **DTrace users guide**
  http://docs.sun.com/app/docs/doc/817-6223/

• **DTrace hotspot probes**
  http://blogs.sun.com/kamg/#Dtrace_Probes_in_Mustang_Hotspot

• **DTraceToolkit**
  http://opensolaris.org/os/community/dtrace/dtracetoolkit/

• **Garbage collection tuning**
  http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html

• **Java troubleshooting Guide:**
  http://java.sun.com/javase/6/webnotes/trouble/

• **Lock profiling scripts**
  http://prefetch.net/blog/index.php/2008/02/03/using-the-dtrace-hotspot-provider-to-observe-java-monitor-contention

• **Object allocation scripts:**
Questions?