

Java Performance Tools

Dr Holly Cummins Tooling Technical Lead Java Technology Centre IBM

cumminsh@uk.ibm.com





Outline

- Introduction
- Identifying performance problems
- Fixing performance problems
 - Performance tools for ...
 - Space bound applications
 - IBM Monitoring and Diagnostic Tools for Java[™] GC and Memory Visualizer
 - IBM MDD4J
 - CPU bound applications
 - Method trace
 - I/O bound applications
 - Lock bound applications
 - IBM Lock Analyzer for Java



When would you use a performance tool?

• When you have a performance problem!





What's a performance problem?

- It doesn't go as fast as you think it ought to
- It doesn't go as fast as your users demand
- It starts out fine and then after some period doesn't go as fast as it used to
- It hangs
 - This is a quite severe example of a performance problem



Outline

- Introduction
- Assessing performance problems
- Fixing performance problems
 - Performance tools for ...
 - Space bound applications
 - IBM Monitoring and Diagnostic Tools for Java[™] GC and Memory Visualizer
 - IBM MDD4J
 - CPU bound applications
 - Method trace
 - I/O bound applications
 - Lock bound applications
 - IBM Lock Analyzer for Java



Assessing performance problems

- Performance must be measured before problems can be fixed
 - Otherwise you risk making things worse with a clever fix
- We don't provide a tool for this!
 - A performance tool cannot do your performance measurement for you
- Performance measurement must be based on your application and your quality of service requirements
 - Throughput
 - Response times
 - Mean response time
 - 90th percentile response time
 - Worst-case response time

The perils of benchmarks

- Sometimes measuring the performance of your own application is difficult
- Measuring the performance of a benchmark is not good enough
 - If it's your application you care about, measure your application





The perils of simulated workloads

- Generating a "real" workload can be hard in a test environment
- Tuning a system against a simulated workload can be misleading
 - Example: garbage collection can be very sensitive to the exact distribution of object sizes and the pattern of connections between objects
 - Example: Insufficient variation in data can lead to artificially warm caches and disguise I/O bottlenecks
- Care must be taken to ensure simulated workloads are sufficiently realistic



The perils of inference

- The performance metrics from performance tools cannot tell you how well your application is performing
 - Pause times cannot tell you what your application response times are
 - Time in GC cannot tell you how fast your application is running
 - Generational garbage collectors often use more of the CPU but give better throughput, and shorter maximum response times
 - A profiler may show more time is being spent in a method, but that may be because a change prompted the JIT to inline other methods, so total time may be reduced



How well is your application performing?

- The simplest and most effective way to measure performance is to invoke System.currentTimeMillis() in a test harness to time properties of interest
- Performance can be very variable, so measurements must be repeated
- Allow unmeasured warm-up period
 - (If that's how the application will run)
 - Allows caches to be populated and methods to be compiled



Exception: Use GC to measure

- throughput 🗖 Data set 2 🔀 set 1
- Rate of garbage collection = rate of garbage generation
- If the code doesn't change, generating garbage faster is good, because garbage is a side effect of work
 - IBM Monitoring and **Diagnostic Tools for** Java – GC and **Memory Visualizer** reports the rate of

commendation

Tuning recommendation

rences cleared

The mean occupancy is 10%. This is a bit low, so you have room to save space by lowering y the heap size will probably degrade overall application performance slightly but improve pa

The memory usage of the application does not indicate any obvious leaks.

Summary

Mean interval between collections (sec)	0.02
Largest memory request (bytes)	416
Mean heap unusable due to fragmentation (MB)	0.0
Allocation failure count	13890
GC Mode	opthruput
Proportion of time spent in garbage collection pauses (%)	6.59
Concurrent collection count	0
Proportion of time spent unpaused (%)	93.41
Forced collections with	U
rumber of collections	13890
Rate of garbage collection	197.044 MB/sec
Mean garbage collection pause (ms)	1.78

Mean	Minimum	Maximum	Total
number (#)	number (#)	number (#)	number (#)
7683	6954	7735	106715277

garbage collection .vgc Report Data Line plot Structured Data Displayer



Outline

- Introduction
- Assessing performance problems
- Fixing performance problems
 - Performance tools for ...
 - Space bound applications
 - IBM Monitoring and Diagnostic Tools for Java[™] GC and Memory Visualizer
 - IBM MDD4J
 - CPU bound applications
 - Method trace
 - I/O bound applications
 - Lock bound applications
 - IBM Lock Analyzer for Java

Fixing performance problems

- Performance problems are caused by limited resources
- Which resource is limited?
- Applications may be
 - CPU bound
 - I/O bound
 - Space bound
 - "Lock bound" (contended)





How to decide which it is?

- CPU bound
 - CPU utilisation consistently high
- I/O bound
 - CPU utilisation not consistently high
- Lock bound
 - CPU utilisation not consistently high
- Space bound
 - Any of the above!
- These heuristics aren't precise enough, so tools are required to guide diagnosis





IBM Performance Tools

- IBM provides a number of tools to identify and fix performance bottlenecks
- The tools are all freely available
- Most but not all are targeted for IBM JVMs only
- Tools available from
 - alphaWorks (alpha tools)
 - IBM Support Assistant (fully supported tools)



IBM Support Assistant (ISA)

- Hosting for Serviceability Tools across product families
- Automatic problem determination data gathering
- Assist with opening PMR's and working with IBM Support
- Documentation:
 - Aggregated search across sources
- Regular updates to GREAT INDIAN DEVERSION OF SUMMIT



http://www.ibm.com/software/support/isa



Outline

- Introduction
- Identifying performance problems
- Fixing performance problems
 - Performance tools for ...
 - Space bound applications
 - IBM Monitoring and Diagnostic Tools for Java[™] GC and Memory Visualizer
 - IBM MDD4J
 - CPU bound applications
 - Method trace
 - I/O bound applications
 - Lock bound applications
 - IBM Lock Analyzer for Java

Diagnosing space bound applications

- Space bound can be disguised as CPU bound
 - Java has garbage collection
 - If the GC is running excessively it will hog the CPU
- Space-bound can also be disguised as I/O bound
 - Excessive "in use" footprint can cause
 - Paging
 - Cache misses
- Enabling verbose garbage collection can quickly identify or rule out space issues
 - On IBM platforms, use -Xverbose:gc or -Xverbosegclog:\$file to write directly to a file
 - Logs may be analyzed with a verbose gc analysis tool



Outline

- Introduction
- Identifying performance problems
- Fixing performance problems
 - Performance tools for ...
 - Space bound applications
 - IBM Monitoring and Diagnostic Tools for Java[™] GC and Memory Visualizer
 - IBM MDD4J
 - CPU bound applications
 - Method trace
 - I/O bound applications
 - Lock bound applications
 - IBM Lock Analyzer for Java





The GC and Memory Visualizer

- IBM Monitoring and Diagnostic Tools for Java GC and Memory Visualizer (formerly known as EVTK) is a verbose GC analysis tool
- Handles verbose GC from all versions of IBM JVMs
 - 1.4.2 and lower
 - 5.0 and higher
 - zSeries
 - iSeries
 - WebSphere real time
 - ··· and Solaris platforms
 - ··· and HP-UX platforms

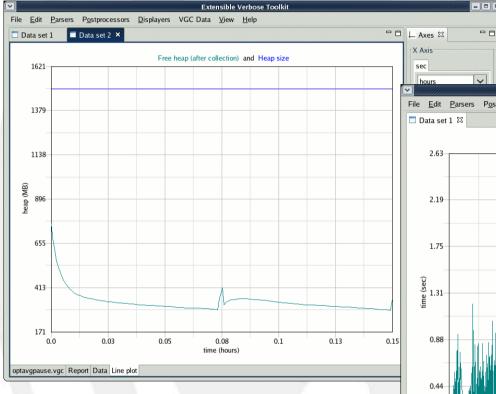


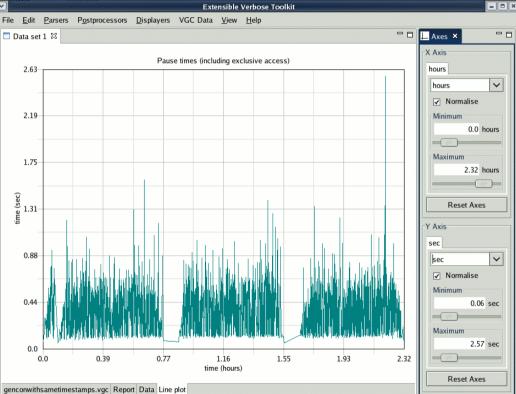
20

GC and Memory Visualizer capabilities

- Analyses heap usage, heap size, pause times, and many other properties
- Provides tuning recommendations
- Compares multiple logs in the same plots and reports
- Many views on data
 - Reports
 - Graphs
 - Tables
- Can save data to
 - HTML reports
 - JPEG pictures
 - CSV files

The GC and Memory Visualizer Heap Visualization





Heap occupancy

GREAT INDIAN

000



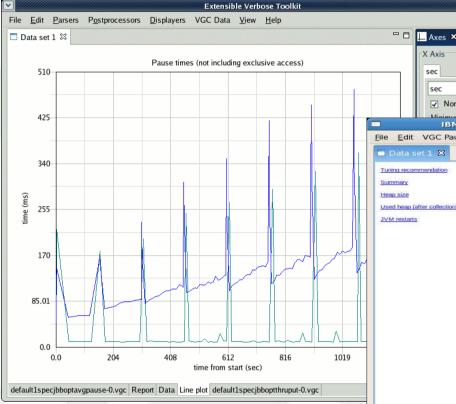
22

The GC and Memory Visualizer - Comparison &

X Axis

sec

sec ✓ Normalise



Compare runs…

Performance advisor...

IBM Monitoring and Diagnostic Tools for Java™ - Garbage Collection and Memory Visualizer

File Edit VGC Pause Data VGC Data VGC Heap Data View Help

Data set 1 🔀

- 0 ×

- -

Used heap (after collection)

- 0

Tuning recommendation

🥸 The heap size was quite variable. This will be causing unnecessary compaction. If your application's workload is relatively steady, you should consider fixing the heap size. This should improve performance in two ways; changing the heap size is rather expensive for the garbage collector because it must compact first, and fe collections will be required if your application is not running in a heap which is too small

🥺 At one point 6968 objects were queued for finalization. Using finalizers is not recommended as it can slow garbage collection and cause wasted space in the heap. Consider reviewing your application for occurrences of the finalize() method.

(b) The mean occupancy is 58%. This is high, so you may improve application performance by increasing your heap size. Increasing the heap size should reduce the narhane collection overhead from its current reported level of 496

🕭 Garbage collection is causing some large pauses. The largest pause was 8592 ms. This may affect application responsiveness. If responsiveness is a concern then a switch of policy or reduction in heap size may be helpful.

🕚 The garbage collector tried to allocate from the pinned free list and failed 17 times. Consider increasing or setting the -Xp command line parameter (the pcluster sizes).

🏝 The garbage collector increased the heap 15 times. Consider increasing the minimum heap size (with -ms) to avoid the need for heap expansions.

1 The number of collections increased by 4,800% in the last third of the log compared to the middle third. However, the change in the heap usage was 0%, which suggests that an increase in application activity or fragmentation rather than a memory leak may be the problem. If the workload is not constant then the change in the frequency of collections may be nothing to worry about.

1 The recommended command line is -ms1024K -mx1500m -Xp32K,4K -Xminf0.1 -Xloratio0.4.

Summary

Mean garbage collection pause (ms)	2395
Proportion of time spent in garbage collection pauses (%)	4.78
Number of collections	1739
Largest memory request (bytes)	65224
Mean interval between collections (minutes)	0.84
Proportion of time spent unpaused (%)	95.2
Allocation failure count	1738
Forced collection count	0
GC Mode	optthruput
Mean heap unusable due to fragmentation (MB)	3.89
Concurrent collection count	0
Full collections	0
Rate of garbage collection	213, 102 MB/minutes

Heap size

Mean	Minimum	Maximum	Total
heap (MB)	heap (MB)	heap (MB)	heap (MB)
1429	380	1500	2485752

agc.txt Report Tabbed data Line plot Structured data

What does garbage collection tell you?

- High heap occupancy indicates an application is likely space bound
 - Increasing heap size or lowering application footprint should improve performance
- If GC is using more than 10% or 20% of the CPU action may be required
 - Alternate choice of policy
 - GC tuning



Don't forget native memory

- Java applications use and may leak native memory
- Low occupancy is no guarantee an application is not space bound.
- Native memory use is not logged in verbose GC
- Memory pressure and even OutOfMemory errors may occur even though there is lots of room in the heap
- Use platform-specific tools
 - Windows perfmon tool
 - Linux ps
 - AIX vmstat



When should you size the heap?

- If performance is important
 - Fixing the heap size prevents the JVM shrinking the heap when the memory usage drops and then having to re-grow when it increases again
 - Try -Xmaxf=100 option to allow growth but prevent shrinking
- If the application uses a lot of memory
 - Most JVMs will avoid using all the memory on a box!
 - The IBM JVM has an upper limit of half the physical memory
 - If the application needs more than this intervention is required





Demonstration: Using the GC and Memory Visualizer to size the heap.

Sample
 application
 allocates
 many
 objects,
 keeps some,
 and
 regularly
 throws some
 away

GREAT INDIAN

```
🕽 Allocator.java 🗙
  package com.ibm.sample;
  import java.util.Random;

-/**

    * A class which holds objects in a hashmap
    * without ever removing them and
    * therefore leaks memory.
    */
  public class Allocator
       private static final int K = 1024;
       private Object[] things = new Object[1000];
       public static void main(String[] args) {
           new Allocator().allocate();
       }
       public void allocate() {
           int allocated = 0;
           while (allocated < 100*K*K) {</pre>
               final int[] thing = new int[new Random().nextInt(50*K)];
               int index = new Random().nextInt(things.length);
               things[index] = thing;
               allocated += things.length;
           }
       }
```

Try out various heap sizes

aring

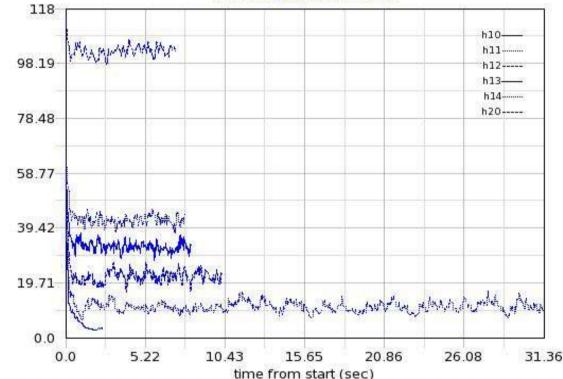
	Cumminsin@grizziy.~/projects/garbageconection/javazone-workspace/anocat = b	\sim
	<u>F</u> ile <u>E</u> dit <u>V</u> iew <u>T</u> erminal Ta <u>b</u> s <u>H</u> elp	
 Some will be obviously bad Most will seem fine 	<pre>[cumminsh@grizzly allocator]\$ heap=100 [cumminsh@grizzly allocator]\$ java -Xverbosegclog:allocator\${heap}.vgc -Xms\${hea p}m -Xmx\${heap}m -cp bin com.ibm.sample.Allocator JVMDUMP006I Processing Dump Event "uncaught", detail "java/lang/OutOfMemoryError " - Please Wait. JVMDUMP007I JVM Requesting Snap Dump using '/home/cumminsh/projects/garbagecolle ction/javazone-workspace/allocator/Snap0001.20070817.153626.20793.trc' JVMDUMP010I Snap Dump written to /home/cumminsh/projects/garbagecollection/javaz one-workspace/allocator/Snap0001.20070817.153626.20793.trc JVMDUMP007I JVM Requesting Heap Dump using '/home/cumminsh/projects/garbagecolle ction/javazone-workspace/allocator/heapdump.20070817.153626.20793.trc JVMDUMP007I JVM Requesting Heap Dump using '/home/cumminsh/projects/garbagecolle ction/javazone-workspace/allocator/heapdump.20070817.153626.20793.phd' JVMDUMP010I Heap Dump written to /home/cumminsh/projects/garbagecollection/javaz one-workspace/allocator/heapdump.20070817.153626.20793.phd JVMDUMP007I JVM Requesting Java Dump using '/home/cumminsh/projects/garbagecolle ction/javazone-workspace/allocator/javacore.20070817.153626.20793.txt' JVMDUMP012E Error in Java Dump: /home/cumminsh/projects/garbagecollection/javazo ne-workspace/allocator/javacore.20070817.153626.20793.txt' JVMDUMP013I Processed Dump Event "uncaught", detail "java/lang/OutOfMemoryError"</pre>	
📕 cumminsh@grizzly:~/projects/ga	Exception in thread "main" java.lang.OutOfMemoryError at com.ibm.sample.Allocator.allocate(Allocator.java:21) at com.ibm.sample.Allocator.main(Allocator.java:15)	
<u>F</u> ile <u>E</u> dit <u>V</u> iew <u>T</u> erminal Ta <u>b</u> s <u>H</u> elp	[cumminsh@grizzly allocator]\$	-
[cumminsh@grizzly allocator]\$ hea [cumminsh@grizzly allocator]\$ jav o}m -Xmx\${heap}m -cp bin com.ibm. [cumminsh@grizzly allocator]\$	va -Xverbosegclog:allocator\${heap}.vgc -Xms\${hea	

-

SL

Use the GC and Memory Visualizer to decidearing up

 Consider summary data and plotted data



Summary

Variant	h10	h1'	i.	Marth Musick S. Marth	in an way the the	ANIA . Re L .
Mean interval between collections (sec)	0.04	0.01	N		du an a	10 - 10 0
Largest memory request (bytes)	204392	204816	0.0			100/00
Mean heap unusable due to fragmentation (MB)	3.36	6.59	0.0	5.22	10.43 time fi	15.65 rom start (se
Allocation failure count	64	2126			Surrear	ion start (se
GC Mode	optthruput	opthruput	opthruput	optthruput	optthruput	opthruput
Proportion of time spent in garbage collection pauses (%)	91.36	78.0	35.02	19.4	14.62	8.94
Concurrent collection count	0	0	0	0	0	0
Proportion of time spent unpaused (%)	8.64	22.0	64.98	80.6	85.38	91.06
Forced collection count	0	0	0	0	0	0
Number of collections	64	2126	724	397	286	112
Rate of garbage collection	123.117 MB/sec	313.827 MB/sec	978.698 MB/sec	1,215.701 MB/sec	1,294.461 MB/sec	1,416.56 MB/sec
Mean garbage collection pause (ms)	36.09	11.9	5.47	4.66	4.61	6.27

heap (MB)

Free heap (after collection)



The trade-off between heap and performance ins

Heap size	Occupancy	GC overhead	Time taken	
100 MB	Out Of Memory crash			
110 MB	89%	77%	30s	
120 MB	82%	37%	9s	
130 MB	75%	20%	9s	
140 MB	69%	14%	8s	
200 MB	49%	9%	7s	
400 MB	24%	4%	7s	
800 MB	12%	4%	7s	



What's the right heap size?

- It depends!
- What other demands are there for heap on the system?
- Larger heaps generally give better performance
 - But ...
 - Very large heaps give diminishing returns
 - Pause times will generally be longer with larger heaps and may be very long with enormous heaps
 - Some policies are more sensitive than others to heap size
- As a rule of thumb, aim for no more than 70% used heap (occupancy)
- 50% used heap is a good balance between improving performance and avoiding waste



Assessing Footprint

- After you've sized the heap, is the footprint what you expect?
- If not, why not?
 - Excessive caching
 - Excessive cloning
 - Bloated object structures
- Solution may be to reduce application's memory usage rather than increase the heap size
- Sometimes the solution may be to increase application's memory usage if it's using less than expected
 - "If my footprint's that small then I can cache all that stuff and speed up my application"



Diagnosing footprint issues

- Understanding leaks and excessive footprint needs an understanding of what objects are on the heap
 - Take a heap or system dump
 - Heap dumps are triggered automatically on OutOfMemoryErrors
 - Dumps may be triggered with ctrl-break (windows) or kill -3 (unix)
 - Dumps may also be triggered on method entry and other events
 - Dumps may also be triggered programmatically
- Once you have a dump, the dump can be analysed to discover what's holding onto memory



Outline

- Introduction
- Identifying performance problems
- Fixing performance problems
 - Performance tools for ...
 - Space bound applications
 - IBM Monitoring and Diagnostic Tools for Java[™] GC and Memory Visualizer
 - IBM MDD4J
 - CPU bound applications
 - Method trace
 - I/O bound applications
 - Lock bound applications
 - IBM Lock Analyzer for Java



MDD4J

everything

JSPCache

String

am 16MB

13MB

String

1MB

miscellaneous

tase 13MB

35

- Java Memory Analysis tool
 - Help explain / track down OutOfMemoryError
 - Footprint analysis
 - Performance problems when object use
- 2 modes of use
 - Single snapshot to visualize a given heap
 - Delta mode to track growth between 2 points in time

Leak

13MB

- Input data types supported
 - IBM Portable Heap Dump (heapdump.phd)
 - IBM Text heap dump (heapdump.txt)
 - HPROF heap dump format (hprof.txt)
- Available through IBM Support Assistant

nalysis Summary	uspects	Explore Con	itext and Contents Brow	vse	
Data structures with large	drops in reach siz	<u>se</u>			
# Object type of suspected co	ontainer Reach size o	f the container object	Drop in reach size		
0 java/util/HashtableSEntry		3MB	3MB		
Object Types that contribution	te most to growt	h in hean size			
# Suspected Object	t Type G		ances Growth in Bytes		
0 java/lang/Integer		200,032	3,200,512		
1 java/util/Hashtable\$Hashtabl	eCacheHashEntry	100,036	2,801,008		
2 java/lang/String		1,858	52,024		
3 char[] 4 java/util/HashMap\$Entry		1,658 457	163,666 12,796		
- Java dur Hasirviapornia y		-157	12,750		
ackages that contribute n	nost to growth in	heap size			
# Suspected Package Grow	h in number of instand	ces			
0 java/lang	202,115				
1 java/util	100,692				
<u>2</u> java/io	88				
<u>3</u> sun/misc	48				
4 java/net	40				

....

🗯 Memory Dump Diagnostic for Java (MDD4J) v2.0.0 Beta - IBM Support Assistant

Support Assistant

0.0

GREAT II

Memory Dump Diagnostic for Java

Analysis Summary Suspects	Explore Context and Contents Browse
Analysis Summary Suspects 0x986800 Find Address Bookmarks: Go Remove The following table shows details of a selected object in the tree: Address : 0x986800 Object Class java/util/Hashtable\$Entry Name : [] Number of 100,001 children : Size (bytes) : Size (bytes) : 400,004 Total Reach Size 2,799,896 (bytes): Actions: Execute Parent Parent Object Name 0x4bf7c8 object java/util/Hashtable 	Explore Context and Contents Browse The objects and object references in the primary memory dump can be browsed here in a tree structure. Each node in the tree represents an object in the Java heap. Its children represent all the outgoing references from that object sorted according to their reach sizes. Its parent is any one parent object from which there is an outgoing reference to this object. To see the details of any particular object (including all its parents) select a node in the tree. Image: Context and Contents Image: Context and Contents Image: Context and Contents Image: Content and Image: Contents Image: Content and Contents Image: Content and Contents Image: Content and Contents Image: Content and Contents Image: Content and Contents Image: Content and Contents Image: Content and Contents Image: Contents Image: Content a
	Oxa 46cd0 object java/util/Hashtable\$HashtableCacheHashEntry Oxa46cd0 object java/util/Hashtable\$HashtableCacheHashEntry Oxa46d50 object java/util/Hashtable\$HashtableCacheHashEntry

•

6

0

0

IBM.

Iring

- Introduction
- Identifying performance problems
- Fixing performance problems
 - Performance tools for ...
 - Space bound applications
 - IBM Monitoring and Diagnostic Tools for Java[™] GC and Memory Visualizer
 - IBM MDD4J
 - CPU bound applications
 - Method trace
 - I/O bound applications
 - Lock bound applications
 - IBM Lock Analyzer for Java



Diagnosing CPU bound applications 7

- Code is being invoked more than it needs to be
 - Easily done with event-driven models
- An algorithm is not the most efficient
 - Easily done without algorithms research!
- Fixing CPU bound applications requires knowledge of what code is being run
 - Identify methods which are suitable for optimisation
 - Optimising methods which the application doesn't spend time in is a waste of your time
 - Identify methods where more time is being spent than you expect

"Why is so much of my profile in calls to this trivial little

Method trace and profiling

- There are two ways to work out what code your application is doing
 - Trace
 - Profiling
- Trace
 - Does not require specialist tools (but is better with them)
 - Records every invocation of a subset of methods
 - Gives insight into sequence of events
 - In the simplest case, System.out.println
- Profiling
 - Requires specialist tools
 - Samples all methods and provides statistics



- Introduction
- Identifying performance problems
- Fixing performance problems
 - Performance tools for ...
 - Space bound applications
 - IBM Monitoring and Diagnostic Tools for Java[™] GC and Memory Visualizer
 - IBM MDD4J
 - CPU bound applications
 - Method trace
 - I/O bound applications
 - Lock bound applications
 - IBM Lock Analyzer for Java





IBM Java method trace

- Traces any Java methods
- Instrumentation-free, and no extra code required
- No fancy GUI, but very very powerful
- Detailed information:
 - Entry and Exit points, with thread information and microsecond time stamps

Not overhead-free, but lower overhead than equivalent function implemented in Java

Select Command Promp	t		- 0	X
C:\j9\win32\temp\jclvi32 C:\j9\win32\temp\jclvi32 C:\j9\win32\temp\jclvi32 21:03:40.781*0x173900 static method HW 21:03:40.828 0x173900	dev-20051020 dev-20051020		Compiled	•
static method C:\j9\win32\temp\jclwi32 C:\j9\win32\temp\jclwi32 C:\j9\win32\temp\jclwi32 C:\j9\win32\temp\jclwi32 C:\j9\win32\temp\jclwi32	dev–2005102(dev–2005102(dev–2005102(dev–2005102(dev–2005102)	6> 6> 6>	0041154	•

Controlling what is traced

- Can select methods on package, class or method name:
- Package: methods={java/lang/*}
- Class: methods={java/lang/String.*}
- Method: methods={HelloWorld.main}
- Also ! operator and combination allowed:
 - methods={java/lang/*,!java/lang/String*}
- Possible to create huge volume of output, so use sensible method specifications!





Triggering events

- Can request certain actions occur when chosen methods are entered or exited
- Actions such as coredump, javadump, etc.
- Actions such as enabling trace!
- Can cause action to occur on n'th instance of trigger condition
- Can specify how many times the action occurs
- Multiple trigger types and actions can be specified



Using triggering to trace only some of the time)

- Can start trace suspended, and resume / suspend it on matching method conditions
- E.g. use start up option –Xtrace:resumecount=1 to start trace suspended.
- Trigger={method{HelloWorld.main*,resumethis,suspendthis}}
- This will cause the requested tracing to take effect only inside HelloWorld.main method
- Less work than stepping through in a debugger and creates a permanent record



Suspend / resume in action

Command Prompt

- 🗆 🗙

C:\j9\win32\temp\jc1wi32dev-20051026> C:\j9\win32\temp\jc1wi32dev-20051026>java -Xtrace:print=mt,methods={HW*,com/ibm/ jvm/io/*},trigger={method{HW.main,resumethis,suspendthis}},resumecount=1 HW > HW.main([Liava/lang/String;)V Compile 21:31:24.703×0x414c8a00 mt.4 d static method 21:31:24.703 0x414c8a00 mt.1 > com/ibm/jum/io/ConsolePrintStream.pri ntln(Ljava/lang/String;)V Compiled method, This = 414467a0 21:31:24.703 0x414c8a00 mt.1 > com/ibm/jvm/io/ConsolePrintStream.get NewlinedString(Ljava/lang/Object;)Ljava/lang/String; Compiled method, This = 414 46788 21:31:24.703 0x414c8a00 nt.1 > com/ibm/ivm/io/ConsolePrintStream.get NewlinedString(Ljava/lang/Object;Z)Ljava/lang/String; Compiled method, This = 41 446774 21:31:24.703 0x414c8a00 mt.7 < com/ibm/jum/io/ConsolePrintStream.get</pre> NewlinedString(Ljava/lang/Object;Z)Ljava/lang/String; Compiled method 21:31:24.703 Øx414c8a00 mt.7 < com/ibm/jvm/io/ConsolePrintStream.get NewlinedString(Ljava/lang/Object;)Ljava/lang/String; Compiled method HU 21:31:24.703 0x414c8a00 mt.7 < com/ibm/jvm/io/ConsolePrintStream.pri ntln(Ljava/lang/String;)V Compiled method < HW.main(ILjava/lang/String;)U Compile</pre> 21:31:24.718 0x414c8a00 mt.10 d static method

C:\j9\win32\temp\jclwi32dev-20051026>_



Triggering and Method Trace in Action

-Xtrace:print=mt,methods={myapp/MyTime*},resumecount=1,trigger=method

{myapp/MyTime.main,resume,suspend}

21:05:47.992*0x806cb00 mt.3
Bytecode static method
21:05:47.994 0x806cb00 mt.19
21:05:47.994 0x806cb00 mt.0
= 809baec
21:05:47.994 0x806cb00 mt.18
MyTime@55D8CBA8 arguments: ()
21:05:47.994 0x806cb00 mt.6

- 21:05:47.994 0x806cb00 mt.0 809baf0 21:05:47.994 0x806cb00 mt.18
- MyTime@55D8CBA8 arguments: () 21:05:48.079 0x806cb00 mt.6

21:05:48.079 0x806cb00 mt.9 Bytecode static method

- > myapp/MyTime.main([Ljava/lang/String;)V
- Static method arguments: ([L@55D8CB98)
- > myapp/MyTime.<init>()V Bytecode method, This
 - Instance method receiver: myapp/
- < myapp/MyTime.<init>()V Bytecode method > myapp/MyTime.test()V Bytecode method, This =
- Instance method receiver: myapp/
- < myapp/MyTime.test()V Bytecode method < myapp/MyTime.main([Ljava/lang/String;)V
- Only real time (79ms) is in the call to MyTime.test()
- Could now drill down into MyTime.test()



Triggering and Method Trace in Action

- Drill down into MyTime.test():
- Extend scope of methods traced, and reduce scope of tracing into MyTime.test()
- -Xtrace:print=mt,methods={myapp/*},resumecount=1,trigger=method{myapp/ MyTime.test,resume,suspend}
- 21:07:14.968*0x806cb00 mt.0 > myapp/MyTime.test()V Bytecode method, This = 809baf0
- 21:07:14.970 0x806cb00 mt.18 Instance method receiver: myapp/ MyTime@55D8CBA8 arguments: ()
- 21:07:15.067 0x806cb00 mt.3 > myapp/MyTimer.getTime()V Bytecode static method
- 21:07:15.067 0x806cb00 mt.19 Static method arguments: ()

mt.6

- 21:07:15.067 0x806cb00 mt.9 < myapp/MyTimer.getTime()V Bytecode static method
 - 21:07:15.069 0x806cb00

GREAT INDIAN

48
< myapp/MyTime.test()V Bytecode method</pre>



Other uses of trace

- Can count tracepoints using
- java -Xtrace:count={tracepoint_selection} Class
- This is almost like a sampling profiler





- Introduction
- Identifying performance problems
- Fixing performance problems
 - Performance tools for ...
 - Space bound applications
 - IBM Monitoring and Diagnostic Tools for Java[™] GC and Memory Visualizer
 - IBM MDD4J
 - CPU bound applications
 - Method trace
 - I/O bound applications
 - Lock bound applications
 - IBM Lock Analyzer for Java





Diagnosing I/O-bound applications

- A number of tools may be required to isolate the causes of I/O delays
- Use the GC and Memory Visualizer to check sweep times
 - Sweep times should be very short
 - Long sweep times indicate access to memory is slow
 - This indicates the application is probably paging
- Use method trace to trace calls to network and disk I/O



- Introduction
- Identifying performance problems
- Fixing performance problems
 - Performance tools for ...
 - Space bound applications
 - IBM Monitoring and Diagnostic Tools for Java[™] GC and Memory Visualizer
 - IBM MDD4J
 - CPU bound applications
 - Method trace
 - I/O bound applications
 - Lock bound applications
 - IBM Lock Analyzer for Java



Diagnosing lock bound applications

 Infelicitous synchronization can cause significant application delays

- IBM provides a tool to quickly diagnose and identify contended locks
 - A contended lock is the opposite of a contented lock!



- Introduction
- Identifying performance problems
- Fixing performance problems
 - Performance tools for ...
 - Space bound applications
 - IBM Monitoring and Diagnostic Tools for Java[™] GC and Memory Visualizer
 - IBM MDD4J
 - CPU bound applications
 - Method trace
 - I/O bound applications
 - Lock bound applications
 - IBM Lock Analyzer for Java



IBM Lock Analyzer for Java

- Download from http://www.alphaworks.ibm.com/tech/jla
- JLA provides profiling data on monitors used in Java applications and the JVM:
 - Counters associated with contended locks
 - Total number of successful acquires
 - Recursive acquires
 - Frequency with which a thread had to block waiting on the monitor
 - Cumulative time the monitor was held.
 - For platforms that support 3 Tier Spin Locking the following are also collected
 - Number of times the requesting thread went through the inner (spin loop) while attempting to acquire the monitor. 55

GREAT INDIAN • Number of times the requesting thread went through the outer **DEVEL** • **PGR** yield loop) while attempting to acquire the monitor.

IBM Lock Analyzer for Java

Udaring

NUNS						1		un llos	II and I and I	-
							Reset	tata Cri	ate Report Delote Report	
100				in Ke		_				
onnano	d) Monitor	u Syn	em Monit	ora S	ummary		_			_
								INTERVAL	ine: 3202210731 후 방 용 패 중	
						ATBRIDE CLARPHONE STORE SCOT (CARD)	a AllTario	AUGAN JANNANGCANDECO 2000 00000	provide teq i renutemi ungrine rupome roferno yskelin i providu ole en entrugiciogo vuluaciónesta babengeo providu ele such i subau transferiochest babengeo providu bli jerundine ges Patermight BCTT CB (Cless providu DC) - weak engle se fatermight BCTT CB (Cless providu DC) - weak engle se fatermight BCTT CB (Cless providu DC) - weak engle se fatermight BCTT CB (Cless providu DC) - weak engle se fatermight BCTT CB (Cless providu DC) - weak engle se fatermight BCTT CB (Cless providu DC) - weak engle se statu edeng proti EFBI (Clease	
						<u>E.</u> 1	. E	E E	A A A A A A A	
	a oft			and the second se		P NE	A DR. DOLLAR & D.	N AVER HT		
12 40	57	57	7	0	1568	45	23	12823689	[31A18744] CubbiHele@EC0A3CC0 (Object)	^
29	7	7	2	0	5	0	0	281281	[31A18100] Java/util/TaskQueue@EC29F9D0 (Object)	
76	4	4	3	0	1	0	0	26019	[[R1A1AA24] jovaAang/Objoch@#C006#D9 (Objoch	
67	1	1	2	n	1	0	0	702	[31A18594] comformlanghmanagementi@penaling%ysternhintificationThrout@FC0CA440 (
67	3	3	3	0	1	0	0	829	[01A1AB48] com/itm/lang/managemant/Memone/intificationThreadShutdown@EC0CBCF8	
100	1	1	1	0	0	0	10	410	[VIA183E8] convion/galagent/3.aClient8ean\$1;gECODEE98 (Object)	
U .	M.	U.	M.	10		U.	10	N.	[JTATAZ4C] SUMMISCLAUNCHER&AppClassLoader(g): CU2C4SU (Ubject)	
	10	10.	122	30	12	10.	192	10	[21A1AA90] jevalangimanegementManagementl'actory@01DEFF30 (Class)	-

What do the bars mean?

- The Lock Analyzer provides very detailed information on locking and synchronization in the table below the chart
- In most cases the chart will be enough
- The height of the bar indicates how often threads were blocked waiting for the lock
- The colour of the bar indicates what fraction of the attempts were unsuccessful



Conclusions

- Improving application performance starts with identifying limited resources
- Tools can help fix performance bottlenecks
 - Space bound
 - GC and Memory Visualizer
 - MDD4J
 - CPU bound
 - Method tracing
 - Lock bound
 - Lock Analyzer for Java





- The following terms are trademarks of International Business Machines Corporation in the United States, othe countries, or both:
 - IBM
 - z/OS
 - PowerPC
 - WebSphere
- Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
- Solaris is a trademark of Sun Microsystems, Inc.
- Intel is a trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both





Any Questions?



