

25<sup>th</sup> September 2012



# Practical Performance

## Understanding the Performance of Your Application

## Important Disclaimers

THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY.

WHILST EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED.

ALL PERFORMANCE DATA INCLUDED IN THIS PRESENTATION HAVE BEEN GATHERED IN A CONTROLLED ENVIRONMENT. YOUR OWN TEST RESULTS MAY VARY BASED ON HARDWARE, SOFTWARE OR INFRASTRUCTURE DIFFERENCES.

ALL DATA INCLUDED IN THIS PRESENTATION ARE MEANT TO BE USED ONLY AS A GUIDE.

IN ADDITION, THE INFORMATION CONTAINED IN THIS PRESENTATION IS BASED ON IBM'S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM, WITHOUT NOTICE.

IBM AND ITS AFFILIATED COMPANIES SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION.

NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, OR SHALL HAVE THE EFFECT OF:

- CREATING ANY WARRANT OR REPRESENTATION FROM IBM, ITS AFFILIATED COMPANIES OR ITS OR THEIR SUPPLIERS AND/OR LICENSORS

## Goals of the Talk

- So...
  - You have a performance problem..
  - You are not sure what the application is doing under the covers...
  
- What next ?
  
- After this talk you will:
  - Understand when and why to use performance tools
  - Have a toolkit of performance tools and techniques
  - Get to know your Java application better

# Agenda

- Performance – why should you care?
- Approaches to performance
- Layers of the application
- Identifying bottlenecks

## Approaches to performance

- Outside in approach
  - Start from where performance can be measured
  - Work along the activity path
  - Ideal for identified performance problems
  
- Layered approach
  - Analyze and eliminate layers of the application
  - Simplify the problem as you go
  - Ideal for application health check
  
- A hybrid of both approaches can often be useful

## Performance baseline

- Important to have a repeatable performance test
- Measure baseline performance
  - Internal measurements affect the performance of what your measuring
  - External measurements have less impact on system performance

## Layers of a Java application

- Three layers of a deployment:
  - Infrastructure: Hardware and Operating System
  - Java Runtime: Garbage Collection
  - Java Application: Java application code
  
- Each can suffer from resource constraints, typically:
  - Memory
  - CPU
  - Synchronization
  - I/O

## Infrastructure

- Typical resource constraints:
  - Memory: insufficient physical memory results in paging/swapping
  - CPU: insufficient CPU time limits throughput of the application
  - I/O: insufficient I/O limits throughput of the application
  - Synchronization driven by Java runtime/Java application
  
- Easy to diagnose
  
- Easy to resolve (relatively)
  
- Note that each can also be caused by deficiencies higher up the stack!



## Infrastructure memory usage

- Infrastructure uses memory for:
  - Backing the process data: OS runtime, Java runtime, Java application
  - Caching of IO: filesystem and network buffers
  
- Lack of physical memory causes:
  - Reduction and removal of IO caching
  - Paging/swapping of process memory to disk
  
- Paging/swapping is costly for a Java process
  - Particularly affects Garbage Collection performance
    - Paging usually occurs on Least Recently Used basis
    - All of Java heap is traversed during mark and sweep phases
    - Least Recently Used does not work well for the Java heap

## Infrastructure CPU usage

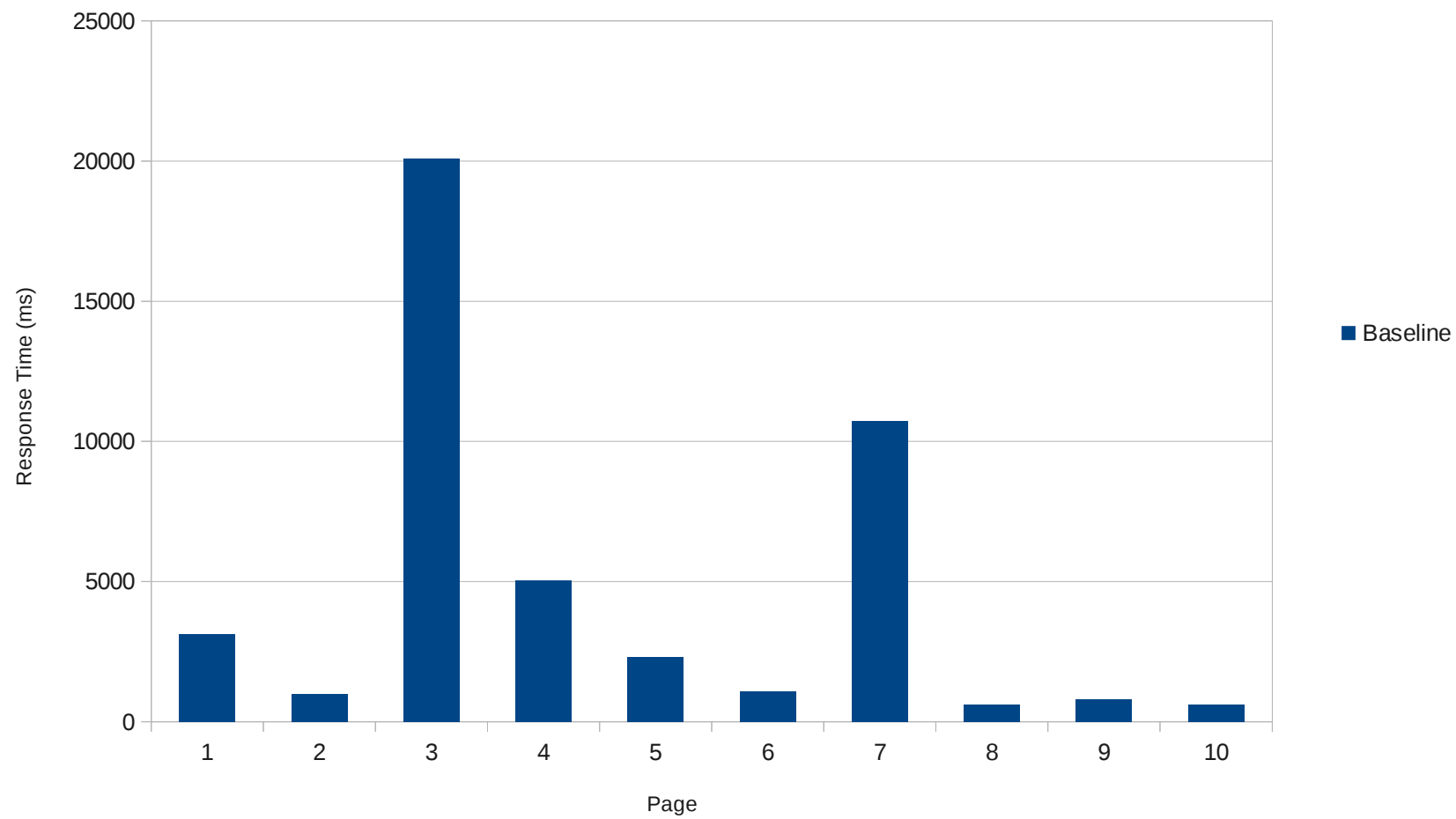
- Insufficient CPU time availability will reduce performance
- Can occur periodically:
  - Cron Jobs running batch applications
  - Database backups
- Or during periods of high load:
  - System becomes CPU bound, limiting performance

## Detecting infrastructure issues

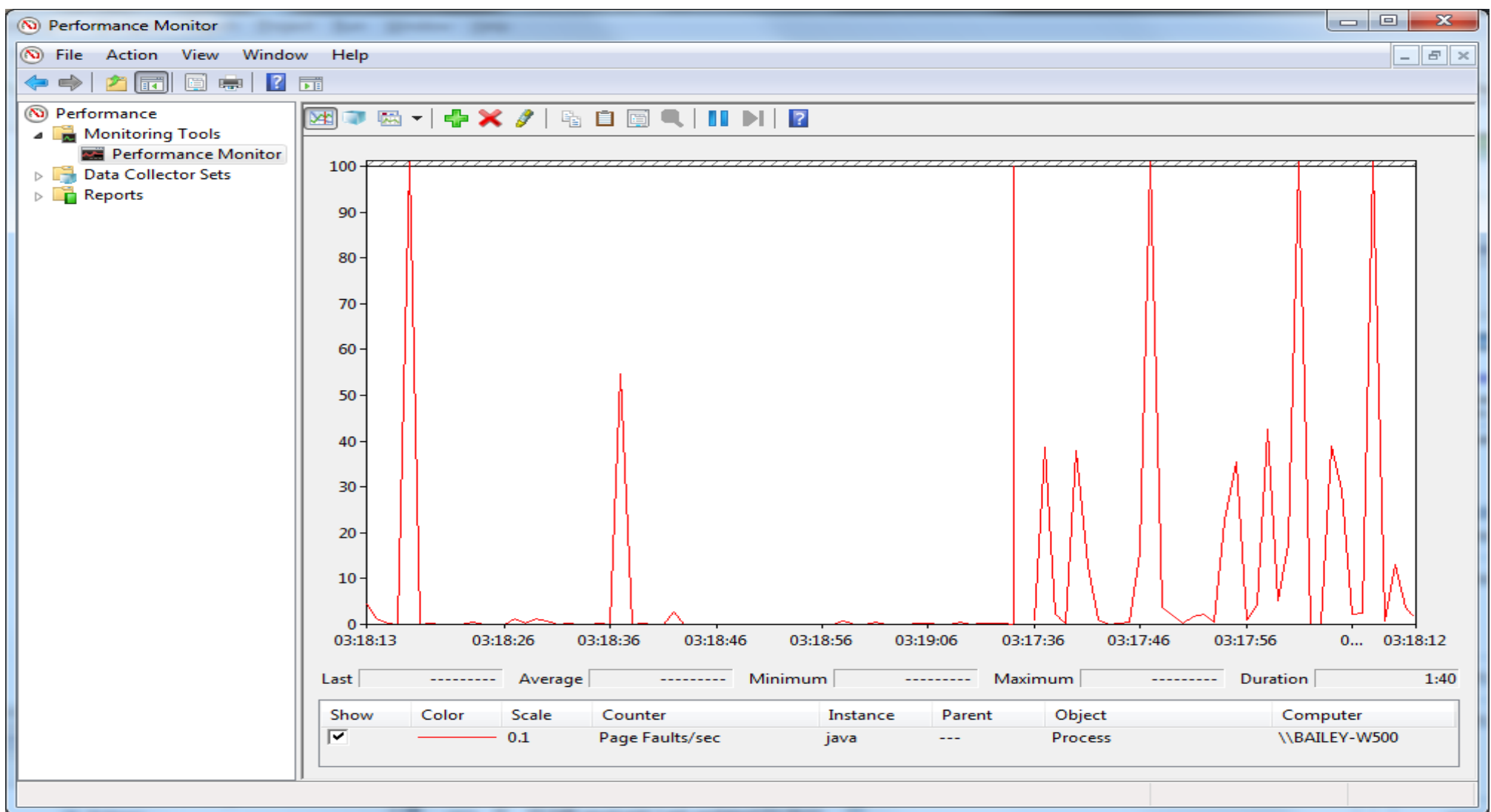
- Detect using Operating System level tools
- Memory on Windows:
  - Paging: using “perfmon” with “Process” counter for “Page Faults/sec”
  - File Cache: using “perfmon” with “Memory” counter for “System Cache Resident Bytes”
- CPU on Windows:
  - Per process: using “perfmon” with “Process” counter for “% Processor Time”
  - Per machine: using “perfmon” with “Processor” counter for “% Processor Time”
- IO on Windows:
  - Network: using “perfmon” with “Network Interface” counter for “Output Queue Length”
  - Disk: using “perform” with “Physical Disk” counter for “Current Disk Queue Length”

# Page response performance benchmark: baseline

Page Performance  
Average Page Response



# Paging in perfmon



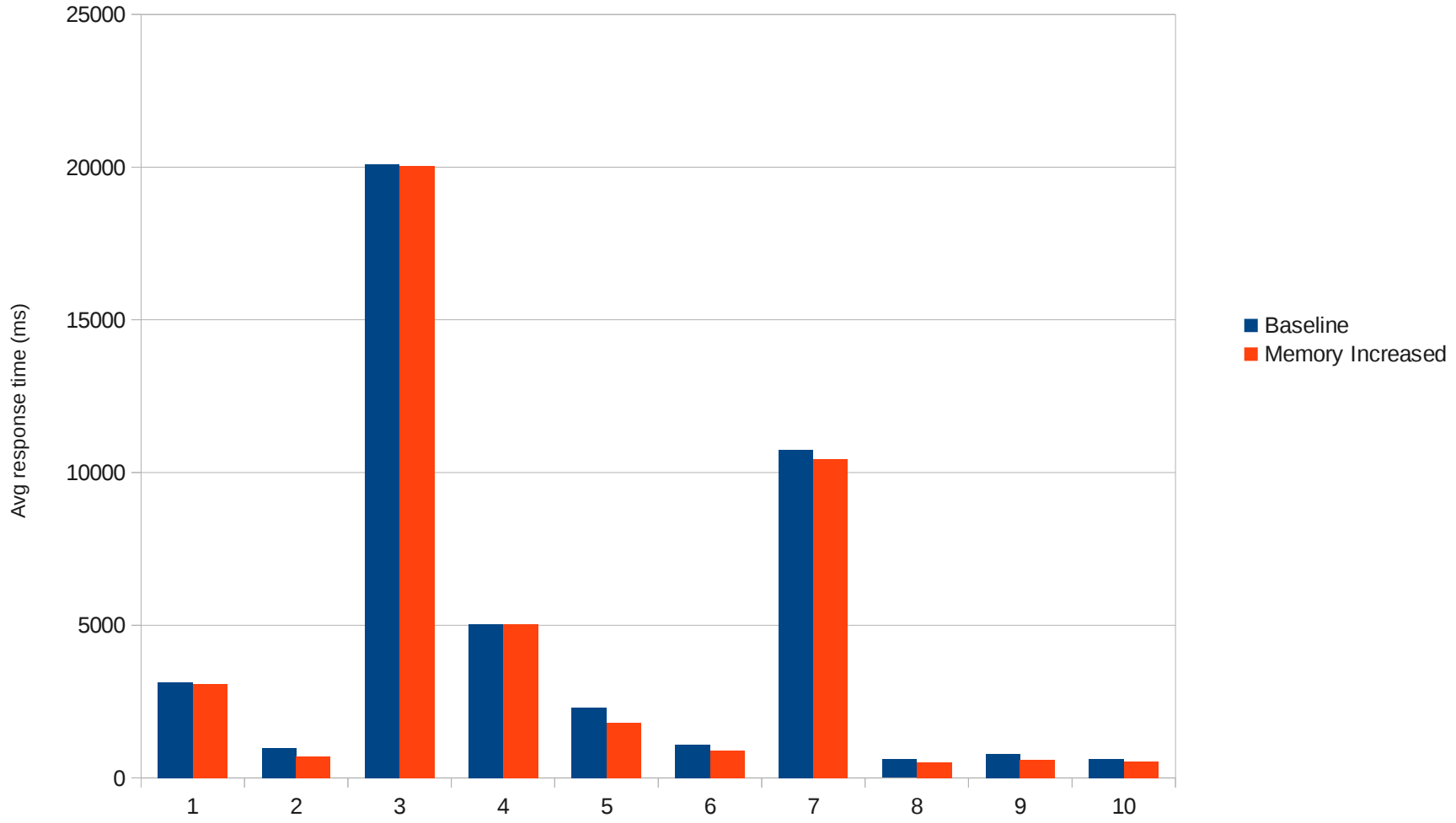
## Resolving infrastructure issues

- Add more physical resources to the process
  - Assign more to the: Machine, Guest OS, LPAR, Zone, etc
  
- Reduce the physical resource requirements
  - Reduce the application footprint
  - Reduce the application CPU usage
  - Reduce the IO

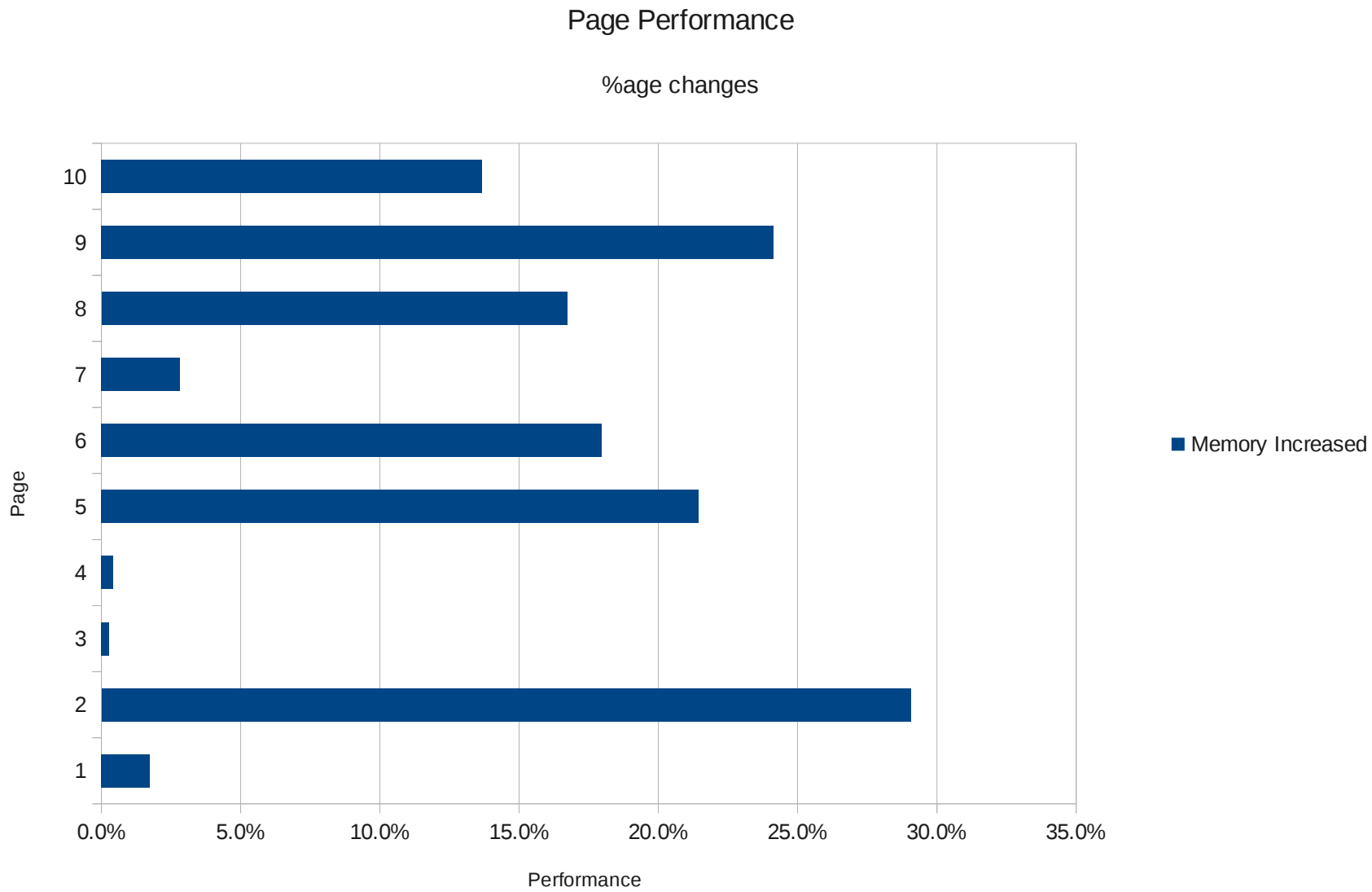
# Page response performance benchmark: memory increased

## Page Performance

### Average Page Response



# Page response performance benchmark: memory increased



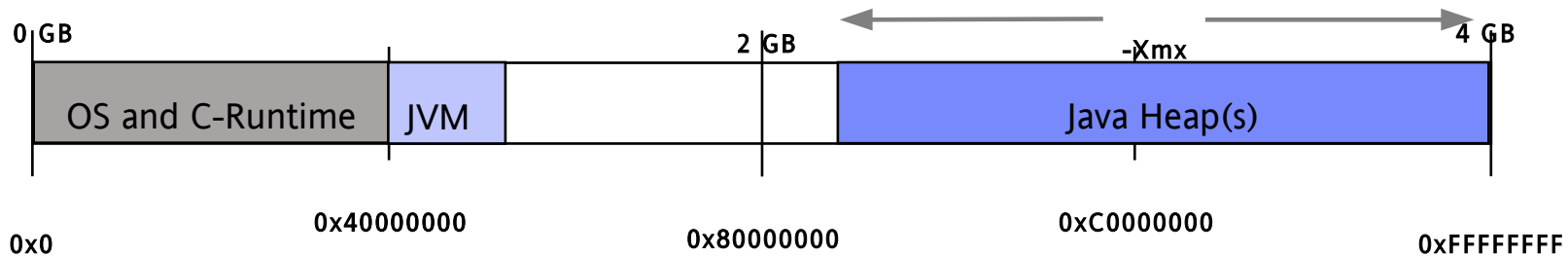


## Java runtime

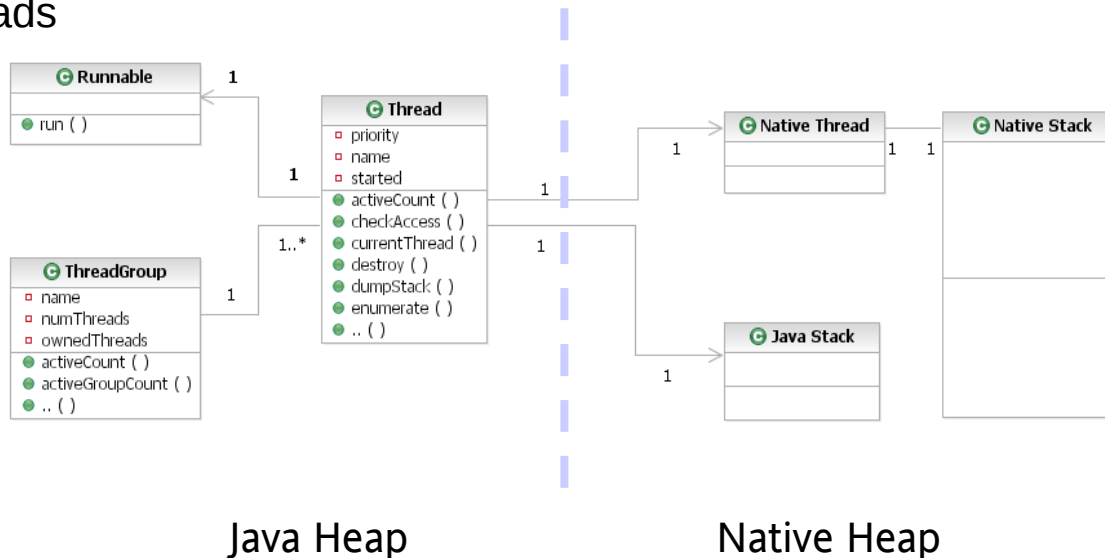
- Typical resource constraints:
  - Memory: insufficient Java heap results in OutOfMemory or high GC overhead
  - CPU Garbage Collection overhead, or driven by Java application
  - Synchronization driven by Java application
  - IO driven by Java application
- Easy to diagnose
- Easy to resolve (relatively)

## Java runtime memory

- Java runtime uses memory for:
  - Java Heap(s), Java Virtual Machine (JVM), “Native” heap, OS and C-language runtime



- Java heap(s) are managed using Garbage Collection
- Other memory usage can be indirectly driven by application usage and garbage collection
  - eg. Java Threads



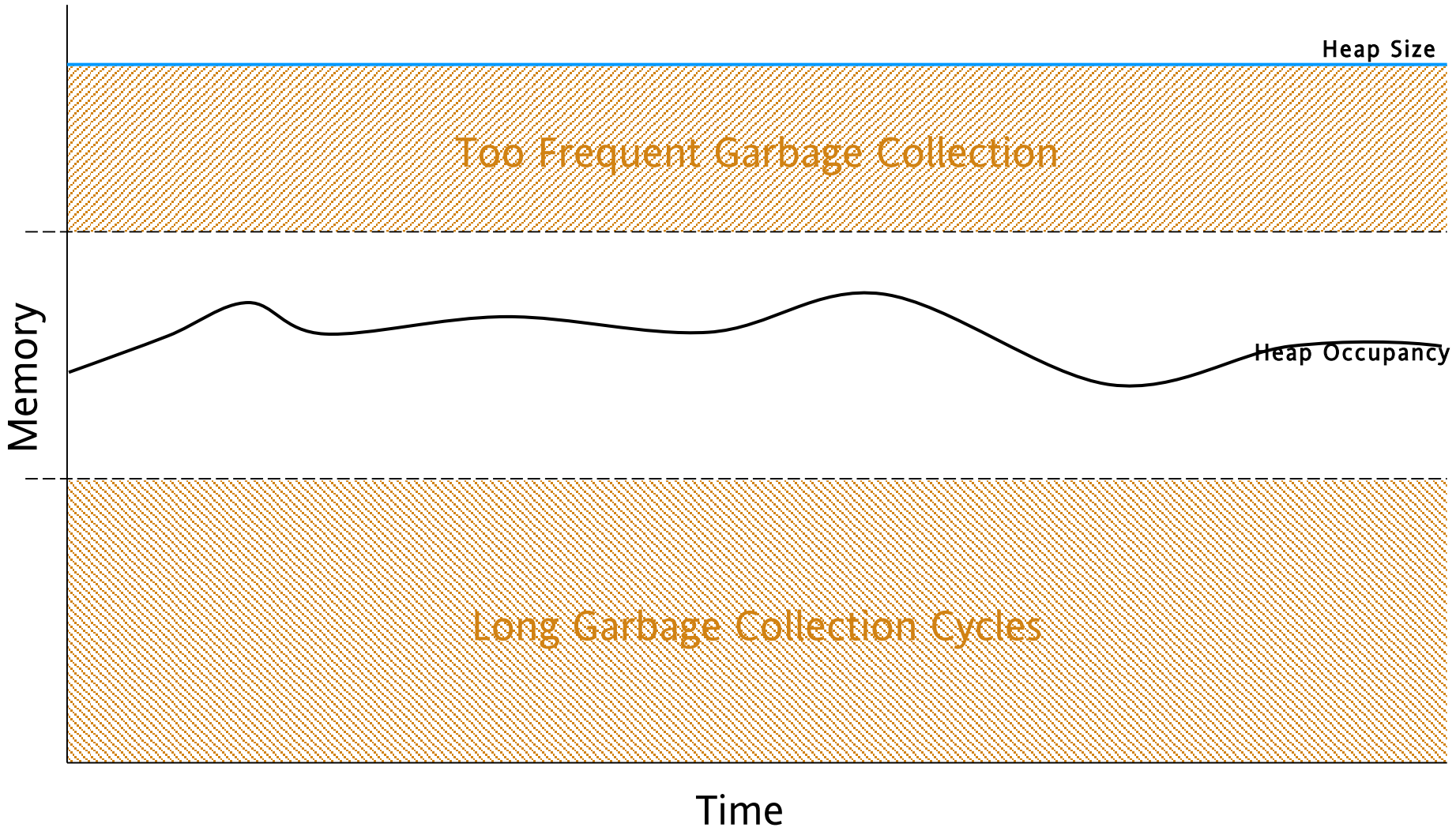
## Java runtime problems

- Insufficient Java heap memory leads to:
  - OutOfMemoryError due to Java heap exhaustion
  - Garbage Collection running excessively, increasing CPU and affecting performance
- Insufficient non-Java (“native”) heap leads to:
  - OutOfMemoryError due to process address space exhaustion
  - Driver for Java heap garbage collection (DirectByteBuffer cleaners)

## Detecting Java runtime problems

- Log and trace analysis:
  - “Native” heap: OS level logs (ps, svmon, perfmon)
  - Java heap: verbose:gc output
  
  - Post processed using:  
IBM Monitoring and Diagnostic Tools for Java - Garbage Collection and Memory Visualizer (GCMV )
  
- Live monitoring:
  - “Native” heap IBM Monitoring and Diagnostic Tools for Java - Health Center
  - Java heap: IBM Monitoring and Diagnostic Tools for Java - Health Center  
Visual VM, Mission Control

# Garbage collection performance



# Too Frequent Garbage Collection

**Status**

- Classes i
- Environment x
- Garbage Collection x
- I/O i
- Locking ✓
- Native Memory ✓
- Profiling i

**Analysis and Recommendations**

- x The mean occupancy is 81%. This is high, so you may improve application performance by increasing your heap size.
- ! The application seems to be using some quite large objects. The largest request which triggered an allocation failure was for 10000 KB.
- i The recommended command line is `-Xmx336m`.
- ✓ The memory usage of the application does not indicate any obvious leaks.

**Used heap (after collection)**

**Used heap (after collection) and Heap size**

size (MB)

elapsed time (minutes)

Heap size

Used heap (after collection)

**Summary** | **Object Allocations**

GC Mode	Default (optthruput)
Largest memory request	10000 KB
Mean garbage collection pause	239 ms
Mean heap unusable due to fragmentation	1.6 MB
Mean interval between collections	2924 ms
Number of collections	76
Number of collections triggered by allocation failure	75
Proportion of time spent in Garbage Collection pauses	8.16%
Proportion of time spent unpaused	91.8%
Rate of garbage collection	437 MB/minute
System (forced) garbage collection count	0

# Garbage Collection Pause Times

**Status**

- Classes i
- Environment x
- Garbage Collection !
- I/O i
- Locking ✓
- Native Memory ✓
- Profiling ✓

**Analysis and Recommendations**

! The application seems to be using some quite large objects. The largest request which triggered an allocation failure was for 10000 KB.

! The mean occupancy is 80%. This is high, so you may improve application performance by increasing your heap size. Increasing the heap size should reduce the garbage collection overhead from its current reported level of 6%.

i The recommended command line is -Xmx343m -Xmaxf0.9.

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

**Pause Times**

Pause times (not including exclusive access)

elapsed time (ms)

elapsed time (minutes)

**Summary** | **Object Allocations**

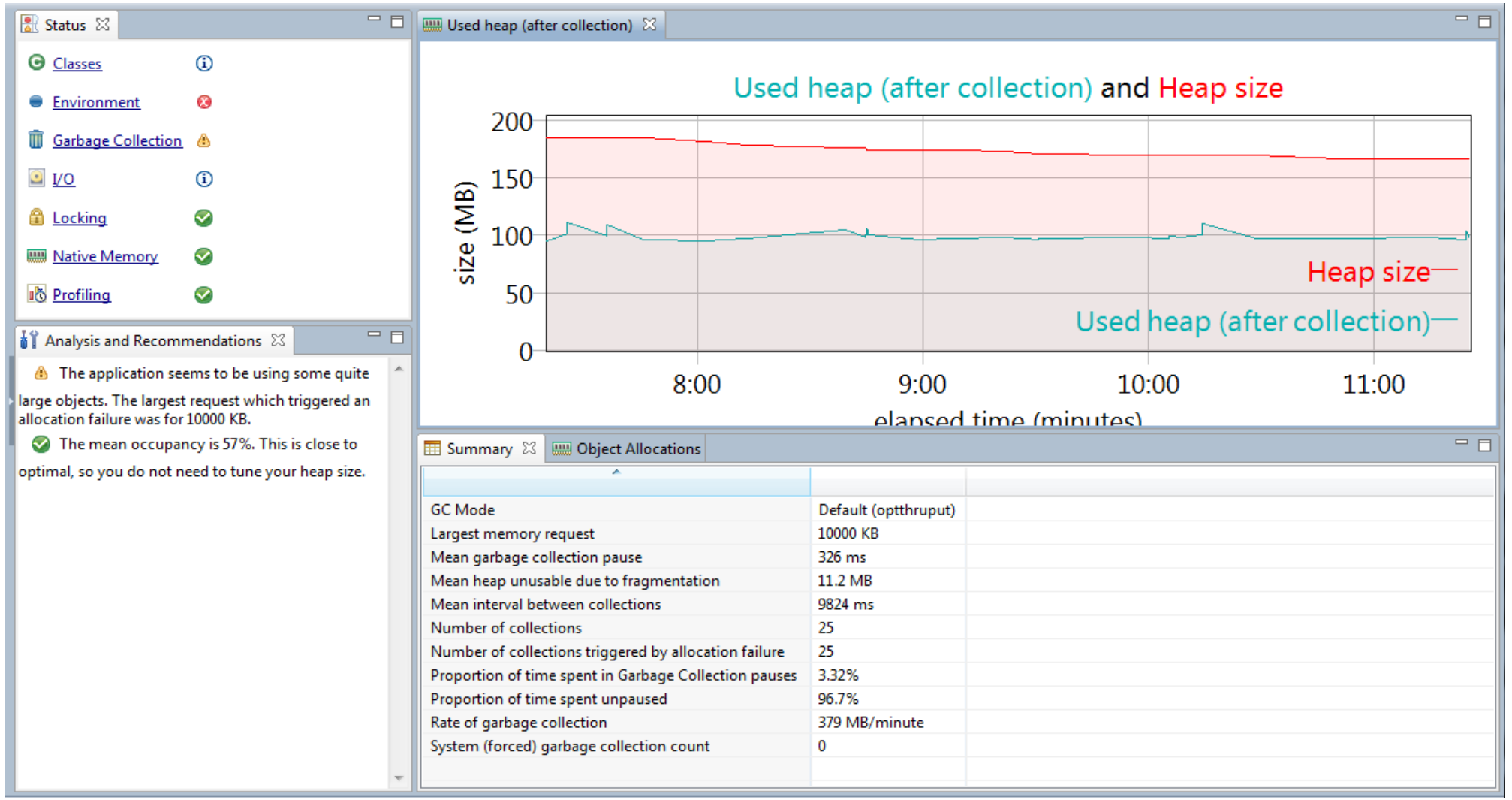
GC Mode	Default (optthruput)
Largest memory request	10000 KB
Mean garbage collection pause	250 ms
Mean heap unusable due to fragmentation	2.99 MB
Mean interval between collections	3961 ms
Number of collections	269
Number of collections triggered by allocation failure	264
Proportion of time spent in Garbage Collection pauses	6.32%
Proportion of time spent unpaused	93.7%
Rate of garbage collection	326 MB/minute
System (forced) garbage collection count	0

## Resolving Java runtime problems

- Add more resources to the Java runtime
  - Java heap: Increase Java heap size
  - Native heap: Move to 64bit or reduce Java heap size
- Reduce the memory requirements
  - Reduce the Java application footprint



# Increased Java heap size



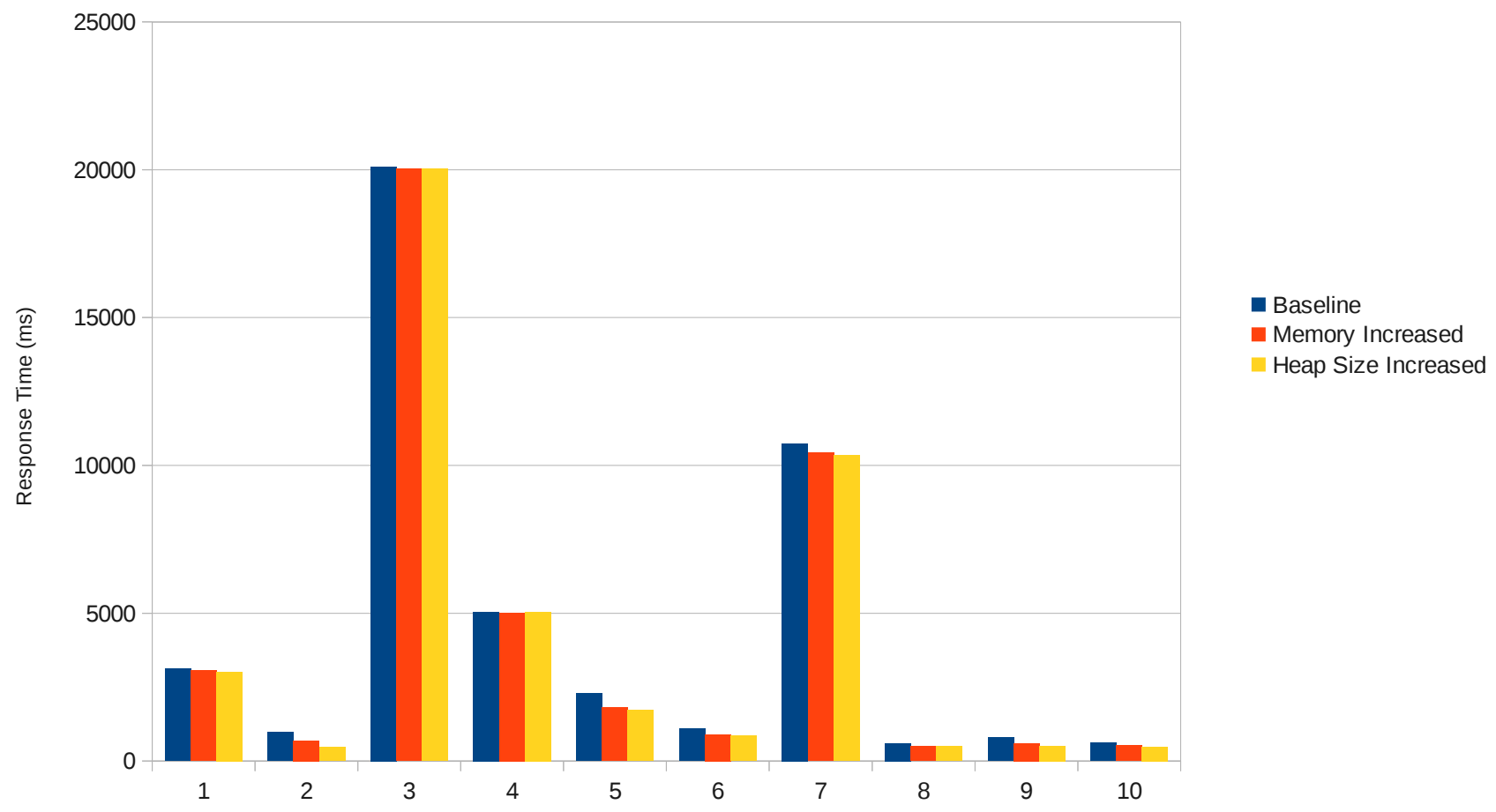
## Effect on Garbage Collection Pause Times

- Reduction in:
  - Time spent in GC 59%
- However this is only 4.84% of total time

# Page response performance benchmark: Java Heap Size Increased

## Page Performance

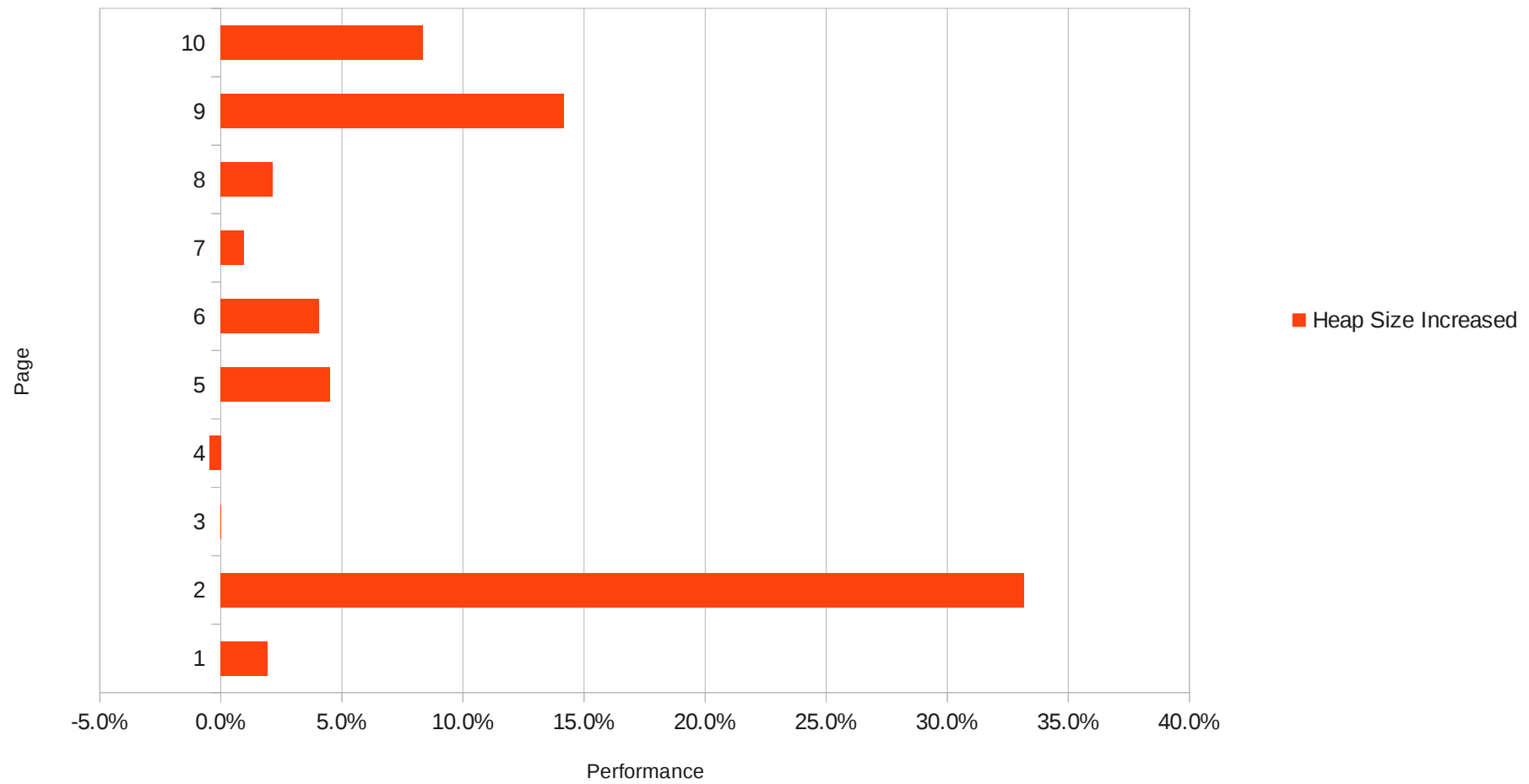
Average Page Response Time



# Page response performance benchmark: Java Heap Size increased

Page Performance

%age changes



## Java application

- Typical resource constraints:
  - Memory: insufficient caching affects application throughput and responsiveness
  - CPU: insufficient threading causes limits on scalability
  - Synchronisation: synchronized resources limits scalability and throughput of the application
  - I/O: blocking on I/O limits throughput and responsiveness
- Hard to diagnose
- Can be expensive (or impossible!) to resolve

## Java application CPU usage

- High CPU usage by Java methods highlight areas of potential optimization
  - 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 if performance is not the focus at development time
  
- 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 time being spent in this trivial method?”

## Java application synchronization

- Throughput does not increase linearly with load
- At limit of throughput the CPU is still low
  - Inability to scale
  - Not all CPU can be utilized
  - Limit on throughput and responsiveness
- Bottleneck where threads need to synchronize with each other for application correctness
  - Caused by large numbers of threads requiring synchronized resource at the same time
  - Caused by long hold time by thread that owns resource
  - Or a mixture of both

# Health Center: application method CPU usage

**Status**

- Classes i
- Environment x
- Garbage Collection !
- I/O i
- Locking ✓
- Native Memory ✓
- Profiling !

**Analysis and Recommendations**

! The method `ShoppingServlet.deliberateSlowMethod()` is consuming approximately 80% of the CPU cycles. It may be a good candidate for optimization.

i The monitored JVM generated more data than the client could consume, and so some samples have been lost. Profile accuracy should not be significantly affected.

**Method profile**

Filter methods:  Apply Clear

Samples	Self (%)	Self	Tree (%)	Tree	Method
29692	80.5	<div style="width: 80.5%; height: 10px; background-color: red;"></div>	80.5	<div style="width: 80.5%; height: 10px; background-color: red;"></div>	<code>com.ibm.websphere.samples.plantsbywebspherewar.ShoppingServlet.deliberateSlowMethod()</code>
252	0.68		3.99	<div style="width: 3.99%; height: 10px; background-color: green;"></div>	<code>java.io.ObjectInputStream.defaultReadFields(java.lang.Object, java.io.ObjectStreamClass)</code>
231	0.63		0.98		<code>com.ibm.rmi.iiop.CDRReader.availableData(int, int, int)</code>
225	0.61		4.12	<div style="width: 4.12%; height: 10px; background-color: green;"></div>	<code>java.io.ObjectInputStream.readObject0(boolean)</code>
136	0.37		4.06	<div style="width: 4.06%; height: 10px; background-color: green;"></div>	<code>java.io.ObjectInputStream.readOrdinaryObject(boolean)</code>
118	0.32		0.58		<code>java.io.ObjectInputStream.readHandle(boolean)</code>
88	0.24		4.07	<div style="width: 4.07%; height: 10px; background-color: green;"></div>	<code>java.io.ObjectInputStream.readArray(boolean)</code>
83	0.22		4.01	<div style="width: 4.01%; height: 10px; background-color: green;"></div>	<code>java.io.ObjectInputStream.readSerialData(java.lang.Object, java.io.ObjectStreamClass)</code>
72	0.2		1.18		<code>com.ibm.rmi.iiop.CDRReader.alignAndCheck(com.ibm.jtc.orb.nio.Aligner, int, int)</code>
70	0.19		0.27		<code>java.io.ObjectInputStream\$BlockDataInputStream.readInt()</code>
65	0.18		0.23		<code>com.ibm.rmi.iiop.ColocatedInputStream.mark()</code>
64	0.17		1.01		<code>com.ibm.rmi.iiop.IIOPOutputStream.sendFragment()</code>
63	0.17		0.29		<code>java.io.ObjectInputStream\$BlockDataInputStream.readUTFBody(long)</code>
53	0.14		0.2		<code>com.ibm.rmi.iiop.CDRReader.read_wstring()</code>
53	0.14		0.14		<code>com.ibm.rmi.iiop.ObjectCopierFactory\$AbstractCopier.run()</code>
50	0.14		1.1		<code>com.ibm.rmi.iiop.CDRInputStream.read_octet_array(byte[], int, int)</code>
49	0.13		0.5		<code>java.io.ObjectInputStream.readString(boolean)</code>
48	0.13		0.28		<code>org.apache.derby.iapi.types.SQLBinary.readFromStream(java.io.InputStream)</code>
37	0.1		0.14		<code>java.io.ObjectInputStream\$HandleTable.markDependency(int, int)</code>
34	0.092		0.13		<code>java.lang.ClassLoader.defineClassImpl(java.lang.String, byte[], int, int, java.lang.Object)</code>

**Invocation paths** **Called methods** **Timeline**

Methods that call `ShoppingServlet.deliberateSlowMethod()`

- M `ShoppingServlet.deliberateSlowMethod`
  - M `ShoppingServlet.performTask (100%)`
    - M `ShoppingServlet.doGet (100%)`
      - M `HttpServlet.service (100%)`



# Health Center: application synchronization

Status

**Classes**

- Environment
- Garbage Collection
- I/O
- Locking
- Native Memory
- Profiling

Analysis and Recommendations

✓ No problems detected.

Monitors bar chart

Inflated Java Monitors

Slow (height) and % miss (color)

Monitor	Slow lock count (number)
[18DD27F8] com/ibm/ws/util/BoundedBuffer\$GetQueueLock@015EC270 (Object)	1
[18F86388] com/ibm/ws/util/BoundedBuffer\$GetQueueLock@0211A0C0 (Object)	29
[187D21C0] com/ibm/ws/util/BoundedBuffer\$GetQueueLock@01596600 (Object)	10
[18F61618] com/ibm/ws/util/BoundedBuffer\$GetQueueLock@015EC480 (Object)	1
[18F61500] com/ibm/ws/util/BoundedBuffer\$GetQueueLock@015C7928 (Object)	16
[18CC5148] java/misc/Launcher\$AppClassLoader@0110B428 (Object)	7
[18F86388] com/ibm/ws/util/BoundedBuffer\$GetQueueLock@02124C00 (Object)	7
[18F61500] com/ibm/ws/util/BoundedBuffer\$GetQueueLock@015C7928 (Object)	9

Monitors

Inflated Java Monitors

% miss	Gets	Slow	Recursive	% util	Average hold time	Name
0	7126	1	0	0	94238	[18DD27F8] com/ibm/ws/util/BoundedBuffer\$GetQueueLock@015EC270 (Object)
0	5845	0	3605	0	79453	[16C88A74] com/ibm/rmi/iiop/WorkQueue@013DB808 (Object)
0	3761	0	0	0	125672	[18DD27F8] com/ibm/ws/util/BoundedBuffer\$GetQueueLock@015EC290 (Object)
1	3536	29	0	0	14120	[18F86388] com/ibm/ws/util/BoundedBuffer\$GetQueueLock@0211A0C0 (Object)
0	3186	0	1970	0	81191	[16C88A74] com/ibm/rmi/iiop/WorkQueue@013DBC00 (Object)
0	2879	0	0	0	93496	[16CB7130] java/util/concurrent/ConcurrentSkipListMap@015EBD28 (Object)
0	2760	0	0	0	55120	[16CB7424] com/ibm/ws/util/BoundedBuffer\$GetQueueLock@0158B008 (Object)

## ShoppingServlet.deliberateSlowMethod()

```
private void deliberateSlowMethod() {  
  
    // -----  
    // User clicked on the Tulips, let's tip toe through a  
    // slow method  
    // -----  
  
    System.out.println("==> STARTING SLOW METHOD");  
  
    long timestamp = System.currentTimeMillis();  
    long target = timestamp + SLOWTIME;  
  
    System.out.println("timestamp="+timestamp);  
    System.out.println("resume at="+target);  
    while(timestamp < target) {  
  
        timestamp = System.currentTimeMillis();  
  
    }  
  
    System.out.println("==> ENDING SLOW METHOD");  
}
```

## Health Center: application method CPU usage

Status

- Classes i
- Environment x
- Garbage Collection !
- I/O i
- Locking ✓
- Native Memory ✓
- Profiling ✓

Analysis and Recommendations

✓ Execution time was relatively evenly balanced between methods. No obvious candidates for optimization were found.

Method profile

Filter methods:  Apply Clear

Samples	Self (%)	Self	Tree (%)	Tree	Method
794	4.14		6.66		com.ibm.rmi.iiop.CDRReader.availableData(int, int, int)
593	3.09		16.5	■	java.io.ObjectInputStream.defaultReadFields(java.lang.Object, java.io.ObjectStreamClass)
497	2.59		16.9	■	java.io.ObjectInputStream.readObject0(boolean)
264	1.38		16.8	■	java.io.ObjectInputStream.readOrdinaryObject(boolean)
250	1.3		2.44		java.io.ObjectInputStream.readHandle(boolean)
232	1.21		7.9		com.ibm.rmi.iiop.CDRReader.alignAndCheck(com.ibm.jtc.orb.nio.Aligner, int, int)
232	1.21		1.44		com.ibm.rmi.iiop.CDRReader.read_wstring()
223	1.16		1.97		java.util.Properties.loadImpl(java.io.Reader)
204	1.06		1.42		com.ibm.rmi.iiop.ColocatedInputStream.mark()
194	1.01		1.2		com.ibm.rmi.util.buffer.ColocatedByteBuffer.write(byte[], int, int)
186	0.97		16.8	■	java.io.ObjectInputStream.readArray(boolean)
123	0.64		1.03		java.io.ObjectInputStream\$BlockDataInputStream.readInt()
120	0.63		16.6	■	java.io.ObjectInputStream.readSerialData(java.lang.Object, java.io.ObjectStreamClass)
118	0.61		0.73		java.io.ObjectInputStream\$HandleTable.markDependency(int, int)
108	0.56		1.39		org.apache.derby.impl.store.raw.data.StoredPage.readRecordFromArray(j.l.Object[], int, int[], int)
107	0.56		0.96		java.io.ObjectInputStream\$BlockDataInputStream.readUTFBody(long)
104	0.54		0.54		sun.io.ByteToCharSingleByte.convert(byte[], int, int, char[], int, int)
93	0.48		1.84		java.io.ObjectInputStream.readString(boolean)

Invocation paths | Called methods | Timeline

Methods that call CDRReader.availableData()

- CDRReader.availableData
  - CDRReader.alignAndCheck (100%)
    - CDRInputStream.read\_octet\_array (85.0%)
      - IIOPInputStream.readPrimArray (99.4%)
        - CDRReader.readBytesForString (0.44%)
          - CDRInputStream.<init> (0.15%)
            - CDRInputStream.read\_long (10.6%)

# Health Center: application synchronization

Status

- Classes i
- Environment x
- Garbage Collection !
- I/O i
- Locking ✓
- Native Memory ✓
- Profiling ✓

Analysis and Recommendations

✓ No problems detected.

Monitors bar chart

Inflated Java Monitors

Slow (height) and % miss (color)

Monitor	Slow lock count (number)
[18D66884] com/ibm/ws/util/BoundedBuffer\$GetQueueLock@015C9960 (Object)	7
[1B6743C4] com/ibm/ws/util/BoundedBuffer\$GetQueueLock@0207FE78 (Object)	107
[16D0E218] com/ibm/rmi/iiop/WorkQueue@013CA928 (Object)	0
[16D0E218] com/ibm/rmi/iiop/WorkQueue@013CA970 (Object)	0
[18E07BD4] java/util/concurrent/ConcurrentSkipListMap@015C74B8 (Object)	0
[18D66884] com/ibm/ws/util/BoundedBuffer\$GetQueueLock@015C9BC0 (Object)	0
[1900C100] com/ibm/ws/util/BoundedBuffer\$GetQueueLock@0158E028 (Object)	0

Monitors

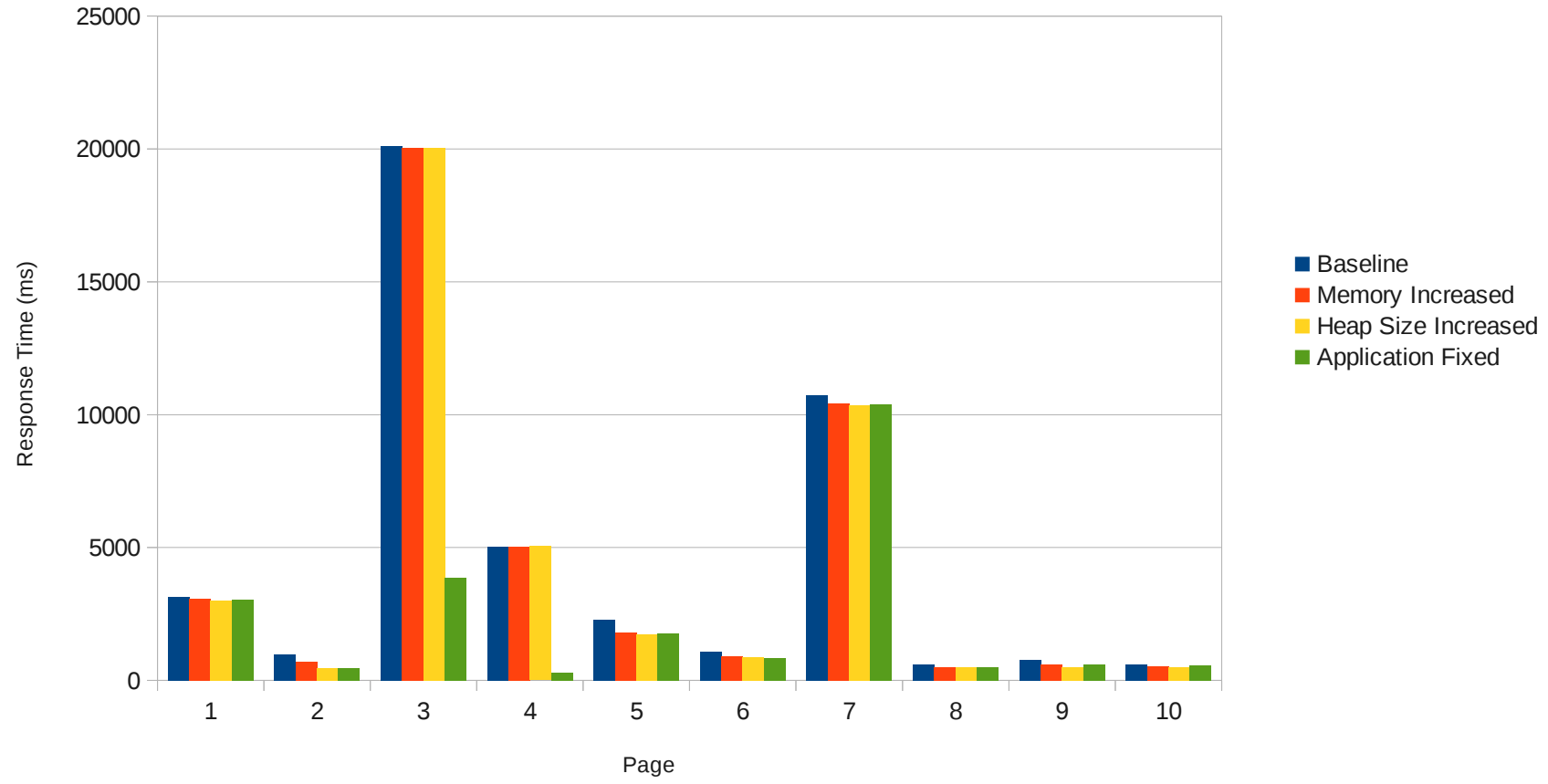
Inflated Java Monitors

% miss	Gets	Slow	Recursive	% util	Average hold time	Name
0	151...	7	0	0	43818	[18D66884] com/ibm/ws/util/BoundedBuffer\$GetQueueLock@015C9960 (Object)
1	105...	107	0	0	43857	[1B6743C4] com/ibm/ws/util/BoundedBuffer\$GetQueueLock@0207FE78 (Object)
0	9808	0	5864	0	84006	[16D0E218] com/ibm/rmi/iiop/WorkQueue@013CA928 (Object)
0	6161	0	3758	0	82741	[16D0E218] com/ibm/rmi/iiop/WorkQueue@013CA970 (Object)
0	6000	0	0	0	124049	[18E07BD4] java/util/concurrent/ConcurrentSkipListMap@015C74B8 (Object)
0	4930	0	0	0	55780	[18D66884] com/ibm/ws/util/BoundedBuffer\$GetQueueLock@015C9BC0 (Object)
0	4887	8	0	0	42706	[1900C100] com/ibm/ws/util/BoundedBuffer\$GetQueueLock@0158E028 (Object)

# Page response performance benchmark: deliberateSlowMethod() changed

Page Performance

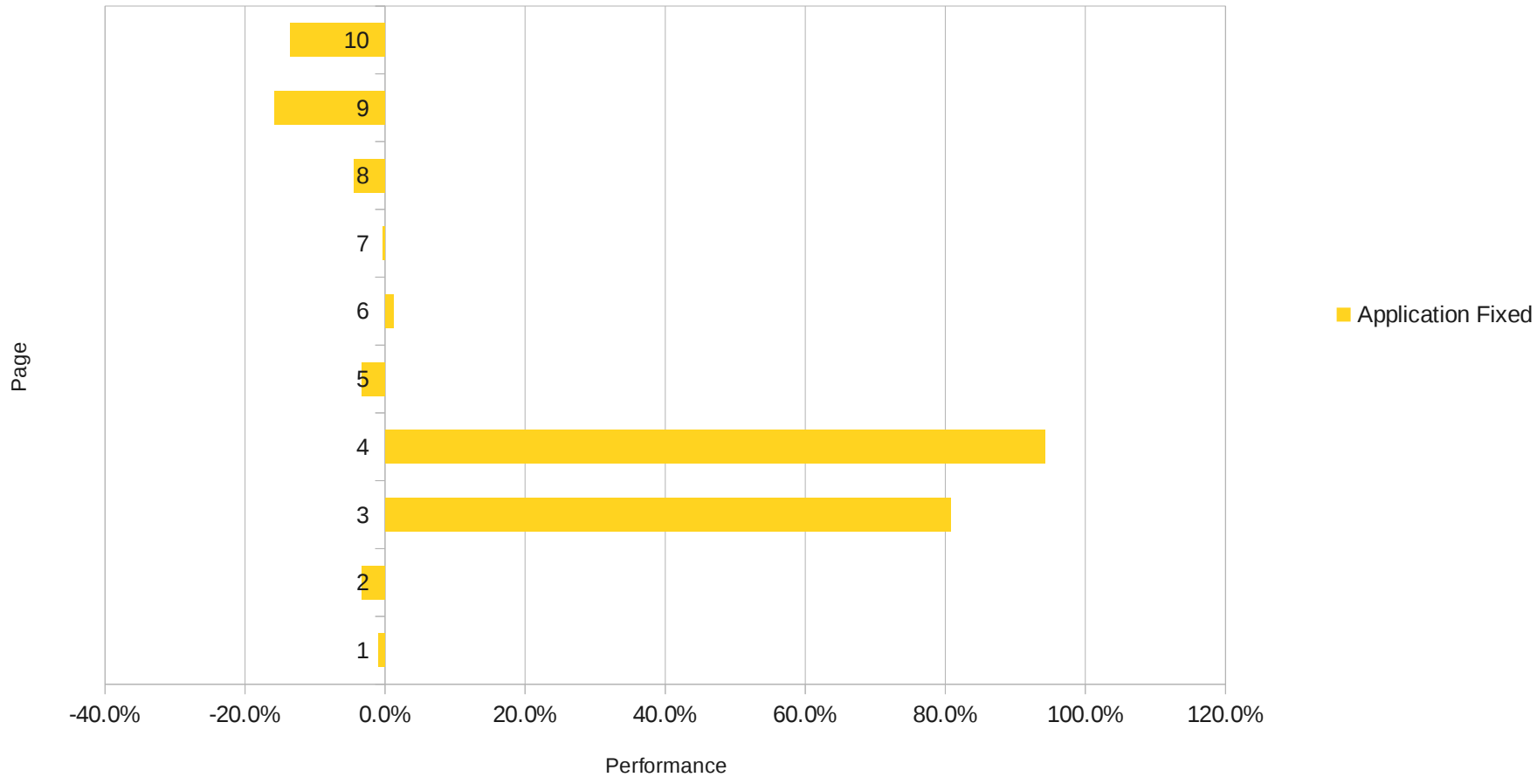
Average Page Response



# Page response performance benchmark: deliberateSlowMethod() changed

Page Performance

%age changes



## Java application memory usage

- Used for “in-flight” work
  - eg. Currently active transactions in a messaging system
- Used for caching data
  - Reduce volume of IO and improve responsiveness

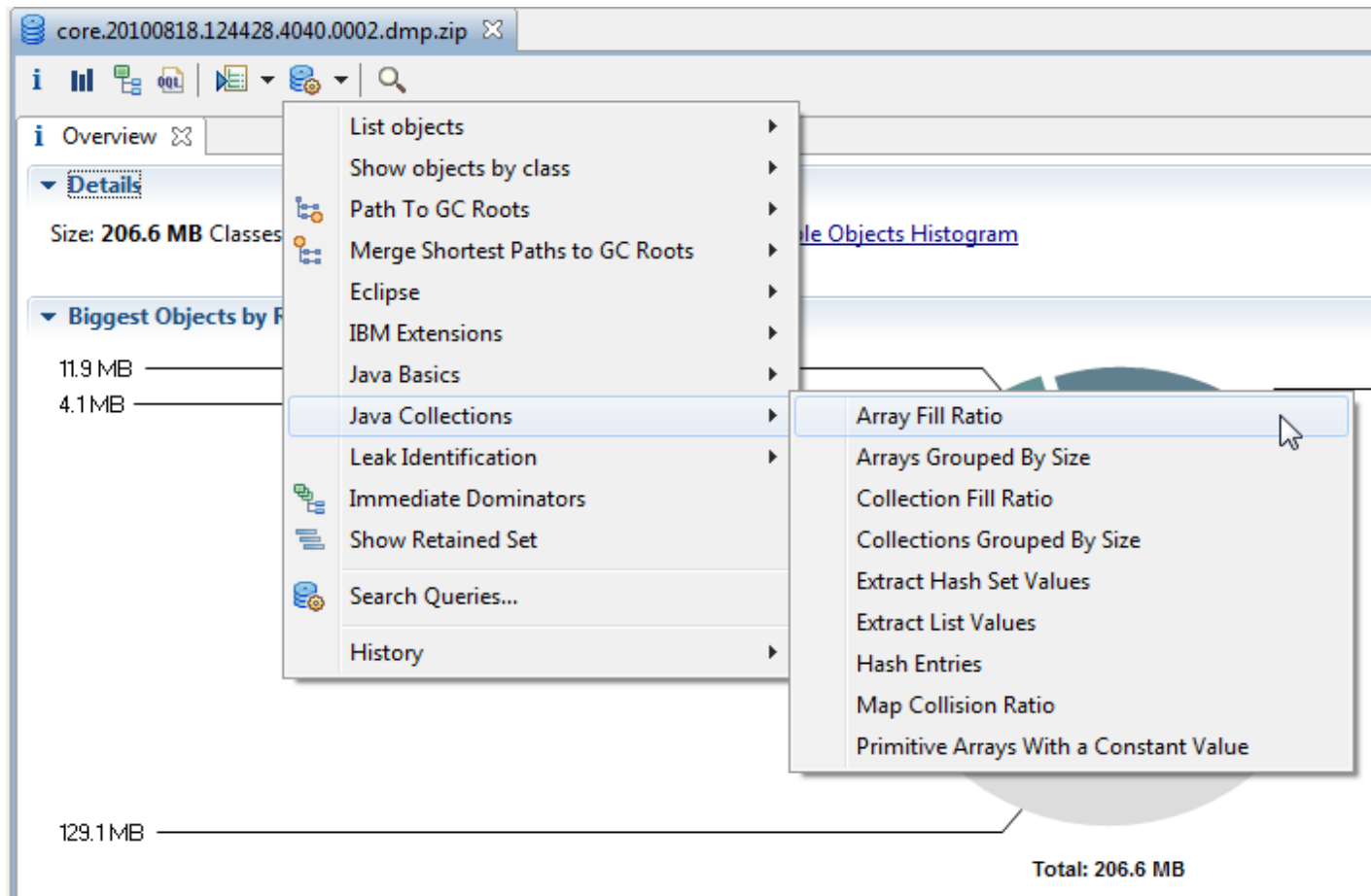
## Java application memory problems

- Memory Leaks
  - Unbounded growth of collections
  - OutOfMemoryErrors
  
- Memory Footprint
  - Incorrectly sized caches
  - Inefficient collection selection
  - Leads to lower performance
  
- Garbage generation
  - Creation/destruction of large amounts of data
  - Leads to lower performance



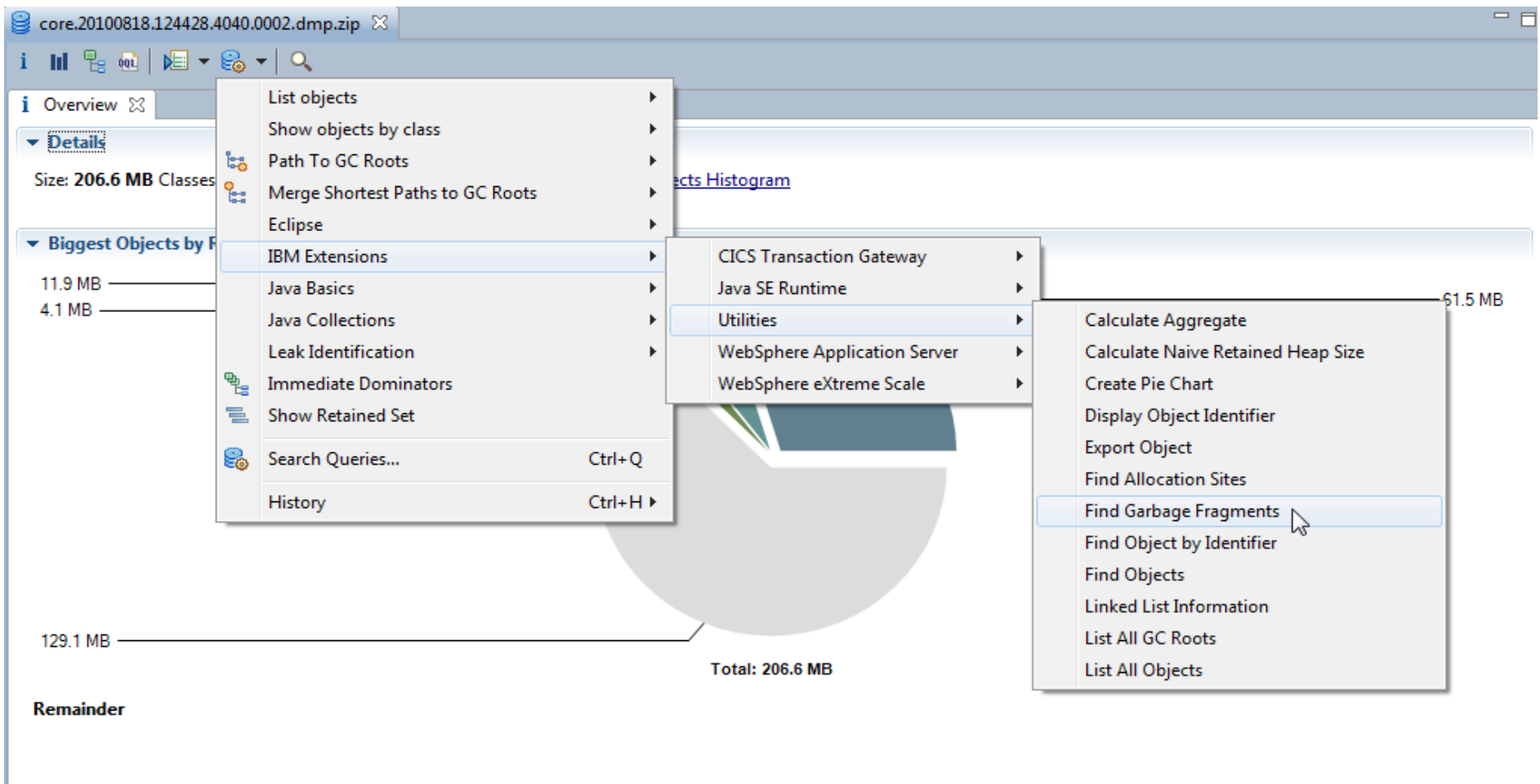
## Analyzing your Collections

- Eclipse Memory Analyzer Tool (MAT) provides Collection analysis:



## Analyzing your garbage

- Eclipse Memory Analyzer Tool (MAT) with the IBM Extensions for Memory Analyzer provides garbage analysis:



## Memory Footprint Summary

- Collections exist in large numbers in many Java applications
  
- Example: IBM WebSphere Application Server running PlantsByWebSphere:
  - HashSet 1,551 instances
  - HashMap 12,151 instances
  - ArrayList 10,600 instances (excluding HashSets)
  - LinkedList 1,148 instances
  - ArrayList 9,530 instances
  - **22,829** total collection instances
  
  - When running a 5 user test load, and using 206MB of Java heap

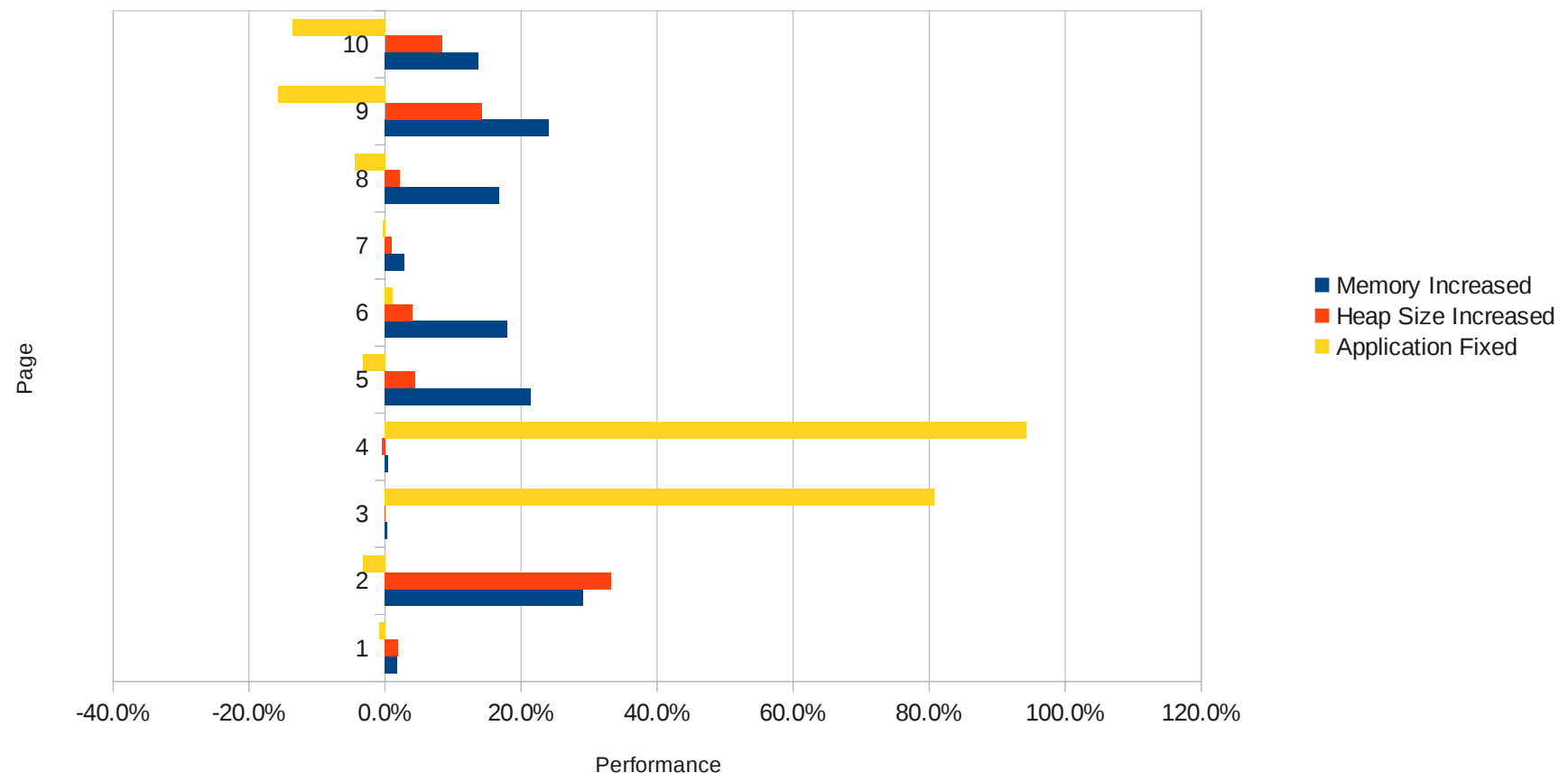
## Summary

- Importance of:
  - Repeatable benchmark
  - Incremental measurements as changes are made
  
- Tools are available to help you see what's going on:
  - Garbage Collection and Memory Visualizer (all vendors)
  - HealthCenter (IBM only)
  - Other profilers (eg. YourKit) (all vendors)
  - Memory Analyzer (all vendors)

# Page response performance benchmark: Summary of Changes

Page Performance

%age changes



## Summary

- Infrastructure resources affect performance
  - Paging and Garbage Collection much less than you might expect
  - However, beware of CPU “starvation” from other processes!
- Vast majority of performance gains are in the application!

## References

- **Get Products and Technologies:**

- IBM Monitoring and Diagnostic Tools for Java:
  - <https://www.ibm.com/developerworks/java/jdk/tools/>

- **Learn:**

- Health Center InfoCenter:
  - <http://publib.boulder.ibm.com/infocenter/hctool/v1r0/index.jsp>

- **Discuss:**

- IBM on Troubleshooting Java Applications Blog:
  - <https://www.ibm.com/developerworks/mydeveloperworks/blogs/troubleshootingjava/>
- Health Center Forum:
  - <http://www.ibm.com/developerworks/forums/forum.jspa?forumID=1461>
- IBM Java Runtimes and SDKs Forum:
  - <http://www.ibm.com/developerworks/forums/forum.jspa?forumID=367&start=0>

## Copyright and Trademarks

© IBM Corporation 2012. All Rights Reserved.

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., and registered in many jurisdictions worldwide.

Other product and service names might be trademarks of IBM or other companies.

A current list of IBM trademarks is available on the Web – see the IBM “Copyright and trademark information” page at URL: [www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml)