CONTENTS



# Java Performance Optimization

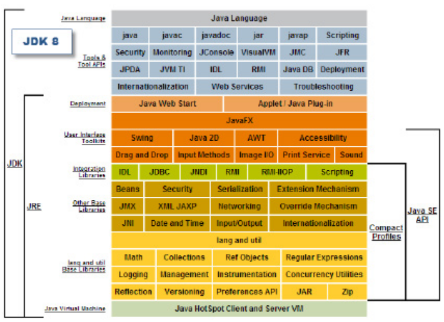
BY PIERRE-HUGUES CHARBONNEAU

Java is among the most widely used programming languages in the software development world today. Java applications are used within many verticals (banking, telecommunications, healthcare, etc.), and in some cases each vertical suggests a particular set of design optimizations. Many performance-related best practices are common to applications of all kinds. The purpose of this Refcard is to help developers improve application performance in as many business contexts as possible by focusing on the JVM internals, performance tuning principles and best practices, and how to make use of available monitoring and troubleshooting tools.

It is possible to define "optimal performance" in different ways, but the basic elements are: the ability of a Java program to perform its computing tasks within the business response time requirements, and the ability of an application to fulfill its business functions under high volume, in a timely manner, with high reliability and low latency. Sometimes the numbers themselves become patternized: for some major websites, a page response time of 500ms maximum per user function is considered optimal. This Refcard will include target numbers when appropriate, but in most cases you will need to decide these on your own, based on business requirements and existing performance benchmarks.

## JVM INTERNALS

### FOUNDATIONS

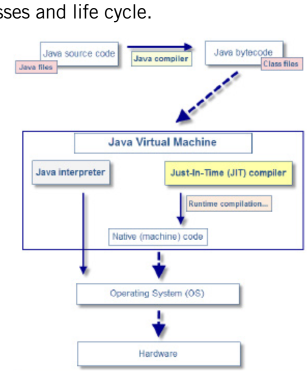



### CODE COMPILATION AND JIT

Java byte code interpretation is clearly not as fast as native code executed directly from the host. In order to improve performance, the HotSpot JVM looks for the busiest areas of byte code and compiles these into native, more efficient, machine code (adaptive optimization). Such native code is then stored in the code cache in non-heap memory.

**NOTE:** most JVM implementations offer ways to disable the JIT compiler (Djava.compiler=NONE). You should only consider disabling such crucial optimization in the event of unexpected JIT problems such as JVM crashes.

The following diagram illustrates the Java source code, just-in-time compilation processes and life cycle.




### MEMORY SPACES

The HotSpot Java Virtual Machine is composed of the following memory spaces.

| MEMORY SPACE | DESCRIPTION |
|---|---|
| **Java Heap** | Primary storage of the Java program class instances and arrays. |
| **Permanent Generation** (JDK 1.7 and older)<br><br>**Metaspace** (JDK 1.8 and later)<br>Java 8 | Primary storage for the Java class metadata.<br><br>**NOTE:** Starting with Java 8, the PermGen space is replaced by the Metaspace and using native memory, similar to the IBM J9 JVM. |
| **Native Heap (C-Heap)** | Native memory storage for the Threads, Stack, code cache including objects such as MMAP files and third party native libraries. |

## CLASS LOADING

Another important feature of Java is its ability to load your compiled Java classes (bytecode) following the start-up of the JVM. Depending on the size of your application, the class loading process can be intrusive and significantly degrade the performance of your application under high load following a fresh restart. This short-term penalty can also be explained by the fact that the internal JIT compiler has to start over its optimization work following a restart.

It is important to note that several improvements were introduced since JDK 1.7, such as the ability for the default JDK class loader to better load classes concurrently.

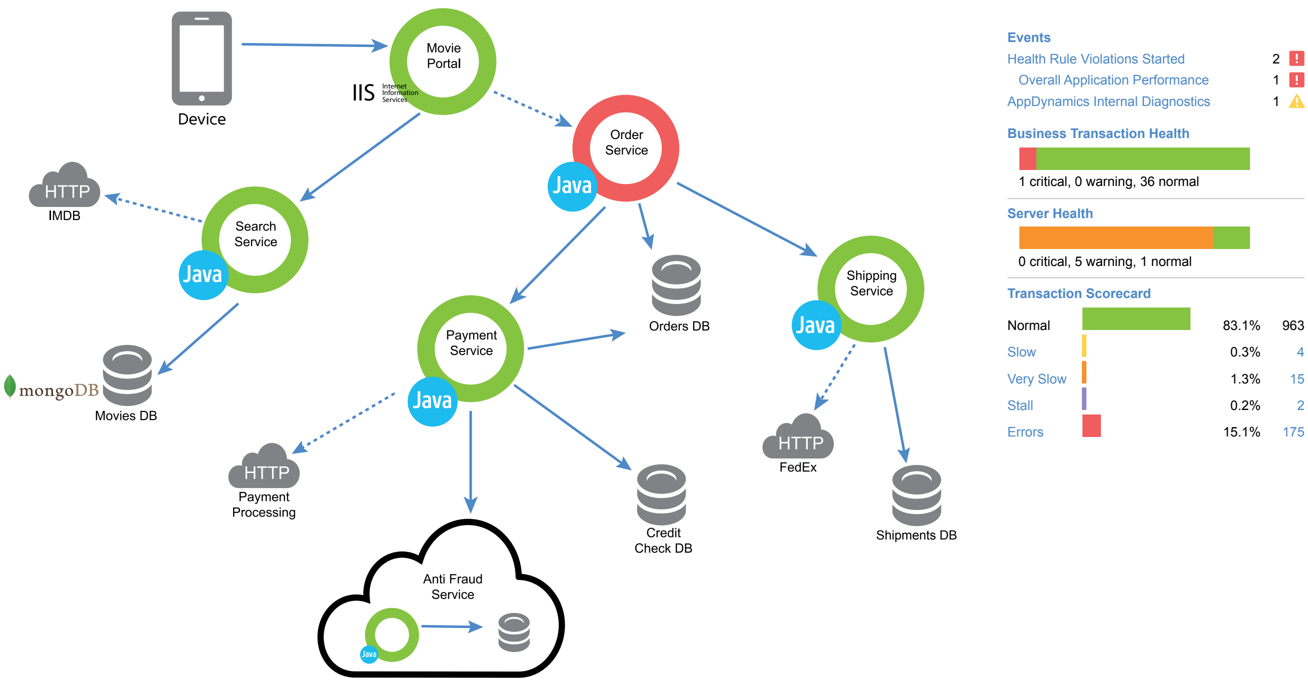
### HOT SPOTS

| AREA OF CONCERN | RECOMMENDATION |
|---|---|
| **Performance degradation following a JVM restart.** | Avoid deploying an excessive amount of Java classes to a single application classloader (ex: very large WAR file). |

APPDYNAMICS

# There's nothing about Java that AppDynamics doesn't see.

AppDynamics gives you the visibility to take command of your Java application's performance, no matter how complicated or distributed your environment is.



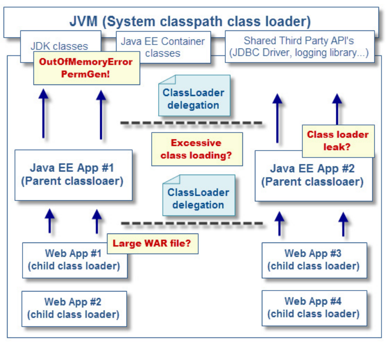

**Start your Free Trial**

When your business runs on Java, count on AppDynamics to give you the complete visibility you need to be sure they are delivering the performance and business results you need — no matter how complex, distributed or asynchronous your environment, live 'in production' or during development.

See every line of code. Get a complete view of your environment with deep code diagnostics and auto-discovery. Understand performance trends with dynamic baselining. And drastically reduce time to root cause and remediation.

See why the world's largest Java deployments rely on the AppDynamics Application Intelligence Platform. Sign up for a FREE trial today at www.appdynamics.com/java.

| | |
|---|---|
| **Excessive class loading contention (thread lock, JAR file searches…) observed at runtime, degrading the overall performance.** | Profile your application and identify code modules performing dynamic class loading operations too frequently. Look aggressively for non-stop class loading errors such as ClassNotFoundException and NoClassDefFoundError.<br><br>Revisit any excessive usage of the Java Reflection API and optimize where applicable. |
| **java.lang.OutOfMemoryError: PermGen space (JDK 1.7 and older)**<br><br>**java.lang.OutOfMemoryError: Metaspace (JDK 1.8 and later)**<br><br>**Native memory leak observed.** | Revisit the sizing of your JVM Permanent Generation, Metaspace (MaxMetaSpaceSize) and / or native memory capacity, where applicable.<br><br>Analyze your application class loaders and identify any source of metadata memory leak. |




## TROUBLESHOOTING & MONITORING

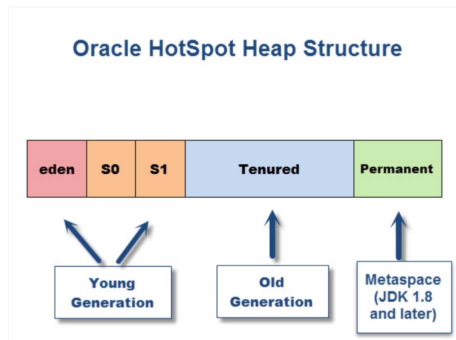
| GOAL | RECOMMENDATION |
|---|---|
| **Keep track of the Java classes loaded to the different class loaders.** | Profile your application using a Java profiler of your choice such as JProfiler or Java VisualVM . Focus on class loader operations and memory footprint.<br><br>Enable class loading details via –verbose:class. For the IBM JVM, generate multiple Java core snapshots and keep track of the active class loaders and loaded classes. |
| **Investigate suspected source(s) of class metadata memory leak(s).** | Profile your application and identify the possible culprit(s).<br><br>Generate and analyze JVM heap dump snapshots with a primary focus on ClassLoader and java.lang.Class instances.<br><br> |
| **Ensure a proper Permanent Generation / Metaspace and native memory sizing.** | Closely monitor your PermGen, Metaspace and native memory utilization, and adjust the maximum capacity where applicable.<br><br>Analyze your application class loaders size and identify opportunities to reduce the metadata footprint of your applications, where possible. |

## GARBAGE COLLECTION

The Java garbage collection process is one of the most important contributing factors for optimal application performance. In order to provide efficient garbage collection, the Heap is essentially divided into sub areas.

### HEAP AREAS

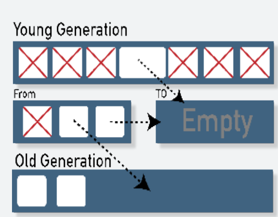
| AREA | DESCRIPTION |
|---|---|
| Young Generation (nursery space) | Part of the heap reserved for allocation of new or short-lived objects.<br><br>Garbage is collected by a fast but stop-the-world YG collector. Objects that have lived long enough in the young space are promoted to the old space<br><br>**NOTE:** It is important to realize that an excessive size and / or GC frequency of the YG space can significantly affect the application response time due to increased JVM pause time. |
| Old Generation (tenured space) | Part of the heap reserved for long-lived objects.<br><br>Garbage is usually collected by a parallel or mostly concurrent collector such as CMS or gencon (IBM JVM).<br><br>**PERFORMANCE TIP:** It is very important to choose and test the optimal GC policy for your application needs. For example, switching to a "mostly" concurrent GC collector such as CMS or G1 may significantly improve your application average response time (reduced latency). |

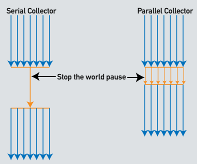
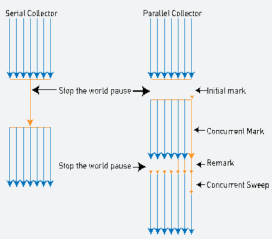



### GC COLLECTORS

Choosing the right collector or GC policy for your application is a determinant factor for optimal application **performance**, **scalability** and **reliability**. Many applications are very sensible to response time latencies, requiring the use of mostly **concurrent** collectors such as the HotSpot `CMS` or the IBM GC policy `balanced`.

As a general best practice, it is highly recommended that you determine most suitable GC policy through proper performance and load testing. A comprehensive monitoring strategy should also be implemented in your production environment in order to keep track of the overall JVM performance and identify future areas for improvement.

| GC | ARGUMENTS | DESCRIPTION |
|---|---|---|
| Serial Collector | -XX:+UseSerialGC<br><br>(Oracle HotSpot) | Both Young and Old collections are done serially, using a single CPU and in a stop-the-world fashion.<br><br>**NOTE:** This policy should only be used by client-side applications not sensitive to JVM pauses.<br><br> |

| | | |
|---|---|---|
| **Parallel Collector** (throughput collector) | **-XX:+UseParallelGC**<br><br>**-XX:+UseParallelOldGC**<br><br>**(Oracle HotSpot)**<br><br>**-Xgcpolicy:optthruput**<br><br>**(IBM J9, single space, stop-the-world)** | Designed to take advantage of available CPU cores. Both Young and Old collections are done using multiple GC threads (via `–XX:ParallelGCThreads=n`), thus better leveraging the available CPU cores from the host.<br><br>**NOTE:** While the collection time can be reduced significantly, applications with large heap size are still exposed to large and stop-the-world old collections and affecting the response time<br><br><br> |
| **Mostly concurrent collectors (low-latency collectors)** | **Concurrent Mark-Sweep**<br><br>**-XX:+UseConcMarkSweepGC**<br>**Garbage First (G1), JDK 1.7u4+**<br>**-XX:+UseG1GC**<br>**(Oracle HotSpot)**<br><br>**-Xgcpolicy:balanced**<br>**(IBM J9 1.7+, region-based layout for the Java heap, designed for Java heap space greater than 4 GB)**<br><br>**-XX:UseConcMarkSweepGC**<br>**-XX:+UseConcMarkSweepGC**<br>**Garbage First (G1), JDK 1.7u4+**<br>**-XX:+UseG1GC**<br>**(Oracle HotSpot)**<br><br>**-Xgcpolicy:balanced**<br>**(IBM J9 1.7+, region-based layout for the Java heap, designed for Java heap space greater than 4 GB)** | Designed to minimize impact on application response time associated with Old generation stop-the-world collections.<br><br>Most of the collection of the old generation using the CMS collector is done *concurrently* with the execution of the application.<br><br>**NOTE:** The YoungGen collections are still stop-the-world events, thus requiring proper fine-tuning in order to reduce the overall JVM pause time.<br><br><br> |

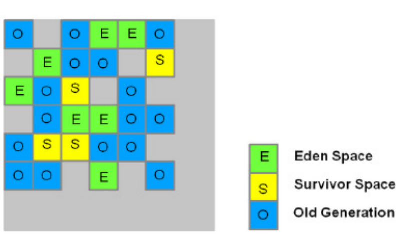
## GARBAGE FIRST (G1) COLLECTOR

The HotSpot G1 collector is designed to meet user-defined garbage collection (GC) pause time goals with high probability, while achieving high throughput. This latest HotSpot collector essentially partitions the heap into a set of equal-sized heap regions, each a contiguous range of virtual memory. It concentrates its collection and compaction activity on the areas of the heap that are likely to be full of reclaimable objects (garbage first), or in other words on areas with the least amount of "live" objects.

Oracle recommends the following use cases or candidates for using the G1 collector, especially for existing applications currently using either the CMS or parallel collectors:

- Designed for applications that require large heaps (>= 6 GB) with limited GC latency (pause time <= 0.5 second).
- More than 50% of the Java heap is occupied with live data (objects that cannot be reclaimed by the GC).
- The rate of object allocation rate or promotion varies significantly.
- Undesired long garbage collection or compaction pauses (longer than 0.5 to 1 second).

**G1 Heap Allocation**




## JAVA HEAP SIZING

It is important to realize that no GC policy can save your application from an inadequate Java heap sizing. Such exercise involves configuring the minimum and maximum capacity for the various memory spaces such as the Young and Old generations, including the metadata and native memory capacity. As a starting point, here are some recommended guidelines:

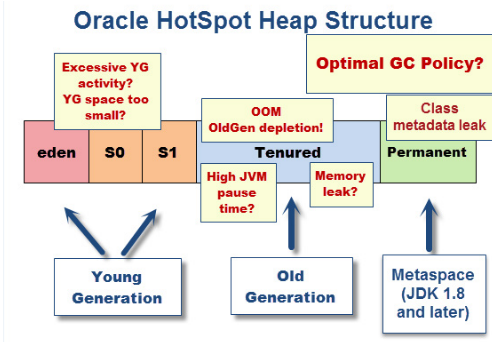
- ✔ Choose wisely between a 32-bit or 64-bit JVM. If your application needs more than 2 GB to run with acceptable JVM pause time due to a large live data footprint, consider using a 64-bit JVM.
- ✔ Remember that the application is king: make sure that you profile it and adjust the heap sizing based on our application memory footprint. It is always recommended to measure the live data footprint through performance and load testing.
- ✔ A larger heap is not always better or faster: do not over-tune the Java heap. In parallel of JVM tuning, identify opportunities to reduce or "spread" your application memory footprint in order to keep the average JVM pause time < 1 %.
- ✔ For a 32-bit JVM, consider a maximum heap size of 2 GB in order to leave some memory from the address space to the metadata and native heap.
- ✔ For 64-bit JVMs, explore vertical and horizontal scaling strategies instead of simply attempting to expand the Java heap size beyond 15 GB. Such an approach very often provides better throughput, better leverages the hardware, and increases your application fail-over capabilities.
- ✔ Do not re-invent the wheel: take advantage of the multiple open source and commercials troubleshooting and monitoring tools available. The APM (Application Performance Management products have evolved significantly over the past decade.

## JDK 1.8 METASPACE GUIDELINES

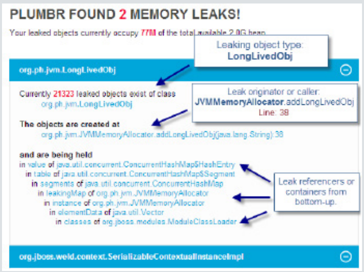
Reference: Oracle Java 8 – GC tuning

| GOAL | RECOMMENDATION |
|---|---|
| **Memory Sizing** | By default, the Metaspace memory space is unbounded and will use the available process and/or OS native memory available for dynamic expansions. The memory space is divided into chunks and allocated by the JVM via mmap. We recommend keeping the default, dynamic resize mode as a starting point for simpler sizing combined with close monitoring of your application metadata footprint over time for optimal capacity planning. |
| **GC Tuning** | A new JVM option is available (-XX:MaxMetaspaceSize=<NNN>), allowing you to limit the amount of native memory committed for class metadata. It is recommended to use it as a safeguard mechanism when facing physical resources (RAM) constraints and other scenarios such as the presence of memory leaks.<br><br>For Java applications with larger class metadata footprint and/or dynamic classloading, we recommend to tune the initial Metaspace size via the new JVM option: -XX:MetaspaceSize=<NNN> ex: 1 GB. This tuning approach will help avoid early garbage collections induced for class metadata, especially during the "warm-up" period of your Java application. |
| **Monitoring & Troubleshooting** | Similar to UseCompressedOops for Java object references, UseCompressedClassesPointers can also be used (it is enabled by default) to minimize the memory footprint. NOTE: while this tuning can help contain the "committed" memory footprint of class pointers, the default metaspace memory reservation is 1 GB when using this option. You may observe a larger virtual memory footprint of the Java process vs. JDK 1. 7. In order to monitor the Metaspace usage, Oracle has updated both the Java VisualVM tool and the GC logs. We recommend to analyze the verbose:gc data for a detailed view of the Metaspace memory usage, GC behavior and dynamic resize frequency.<br><br>**PERFORMANCE TIP:** the new Metaspace implementation will not resolve by itself existing class metadata memory leaks. It is recommended to analyze any unexpected class metadata reachable references when such problem is suspected.<br><br>**PERFORMANCE TIP:** there are some risks associated with using the default or unbounded mode for the Metaspace. If left unchecked, a Metaspace memory leak has the potential to deplete the physical RAM of your infrastructure and may lead to excessive disk paging and/or OS hang. It is recommended to monitor closely the Metaspace usage and take preventive (restart) actions of the JVM, when required, as a short-term action. Long-term solutions normally involve resolving memory leaks and tuning your application class metadata footprint The usage of Java profiler tools and JVM Heap Dump analysis will greatly help you achieve such goals. |

### HOT SPOTS




### TROUBLESHOOTING & MONITORING

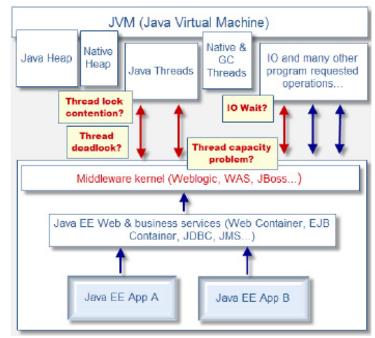
<table>
<tr><th>GOAL</th><th>RECOMMENDATION</th></tr>
<tr><td>Measure and monitor your application YoungGen and OldGen memory footprint, including the GC activity.<br><br>Determine the right GC policy and Java heap size for your application.<br><br>Fine-tune your application memory footprint such as live objects.</td><td>Profile and monitor your application using a Java profiler of your choice such as JProfiler, Java VisualVM, or other commercial APM products.<br><br>Enable the JVM GC activity logging via –verbose:gc. You can also use tools such as GCMV (GC Memory Visualizer) in order to assess your JVM pause time and memory allocation rate.<br><br>PERFORMANCE TIP: an excessive memory allocation rate may indicate a need to perform vertical and/or horizontal scaling, or to decouple your live data across multiple JVM processes.<br><br>For your long-lived objects or long-term live data, consider generating and analyzing JVM heap dump snapshots. Heap dump analysis is also very useful at optimizing your application memory footprint (retention).<br><br>PERFORMANCE TIP: Since going from a 32-bit to a 64-bit machine increases heap requirement for an existing Java application by up to 1.5 times (bigger ordinary object pointers), it is very important to use -XX:+UseCompressedOops in Java version prior to 1.7 (which is now default). This tuning argument greatly alleviates the performance penalty associated with a 64-bit JVM.</td></tr>
<tr><td>Investigate OutOfMemoryError problems and suspected source(s) of OldGen memory leak.</td><td>Profile your application for possible memory leaks using tools such as Java VisualVM or Plumbr (Java memory leak detector).<br><br>PERFORMANCE TIP: Focus your analysis on the biggest Java object accumulation points. It is important to realize that reducing your application memory footprint will translate in improved performance due to reduced GC activity.<br><br><br><br><br>Generate and analyze JVM heap dump snapshots using tools such as Memory Analyzer.<br><br><br>Problem Suspect 1<br><br>The class "oracle.xdb.XMLType", loaded by "java.net.URLClassLoader @ 0x73560fa10", occupies 1,142,909,144 (61.99%) bytes. The memory is accumulated in one instance of "java.util.HashMap$Entry[]" loaded by "<system class loader>".<br><br>Keywords<br>java.net.URLClassLoader @ 0x73560fa10<br>java.util.HashMap$Entry[]<br>oracle.xdb.XMLType<br><br>Details »</td></tr>
</table>

## JAVA CONCURRENCY

Java concurrency can be defined as the ability to execute several tasks of a program in parallel. For large Java EE systems, this means the capability to execute multiple user business functions concurrently while achieving optimal throughput and performance.

Regardless of your hardware capacity or the health of your JVM, Java concurrency problems can bring any application to its knees and severely affect the overall application performance and availability

### THREAD LOCK CONTENTION




Thread lock contention is by far the most common Java concurrency problem that you will observe when assessing the concurrent threads health of your Java application. This problem will manifest itself by the presence of 1…n BLOCKED threads (thread waiting chain) waiting to acquire a lock on a particular object monitor. Depending on the severity of the issue, lock contention can severely affect your application response time and service availability.

**EXAMPLE:** Thread lock contention triggered by non-stop attempts to load a missing Java class (ClassNotFoundException) to the default JDK 1.7 ClassLoader.

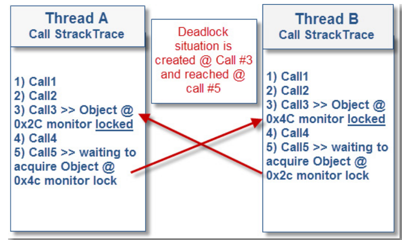
```
ult task-15" prio=6 tid=0x0000000014849800 nid=0x2050 waiting for monitor entry [0x00000(
a.lang.Thread.State: BLOCKED (on object monitor)
java.lang.ClassLoader.loadClass(ClassLoader.java:403)
aiting to lock <0x00000000ab84c0c8> (a java.lang.Object)
sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:308)
java.lang.ClassLoader.loadClass(ClassLoader.java:356)
org.jboss.modules.ConcurrentClassLoader.performLoadClass(ConcurrentClassLoader.java:371)
org.jboss.modules.ConcurrentClassLoader.loadClass(ConcurrentClassLoader.java:119)
java.lang.Class.forName0(Native Method)
java.lang.Class.forName(Class.java:186)
org.jboss.tools.examples.rest.MemberResourceRESTSe                                  rResourceRE:
org.jboss.tools.examples.rest.MemberResourceRESTService$Proxy$_$$_WeldClientProxy.System(
sun.reflect.GeneratedMethodAccessor15.invoke(Unknown Source)
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
java.lang.reflect.Method.invoke(Method.java:601)
..............
```



It is highly recommended that you aggressively assess the presence of such a problem in your environment via proven techniques such as Thread Dump analysis. Typical root causes of this issue can vary from abuse of plain old Java synchronization to legitimate IO blocking or other non-thread safe calls. Lock contention problems are often the "symptoms" of another problem.

### JAVA-LEVEL DEADLOCKS

True Java-level deadlocks, while less common, can also greatly affect the performance and stability of your application. This problem is triggered when two or more threads are blocked forever, waiting for each other. This situation is very different from other more common "day-to-day" thread problems such as lock contention, threads waiting on blocking IO calls etc. A true lock-ordering deadlock can be visualized as per below:



The Oracle HotSpot and IBM JVM implementations provide **deadlock detectors** for most scenarios, allowing you to quickly identify the culprit threads involved in such condition. Similar to lock contention troubleshooting, it is recommended to use techniques such as thread dump analysis as a starting point.

```
Found one Java-level deadlock:              Java-level deadlock detected!
=============================
"pool-1-thread-5":
  waiting to lock monitor 0x04d29604 (object 0x2705cc50, a eu.javaspec:
  which is held by "pool-1-thread-1"
"pool-1-thread-1":
  waiting to lock monitor 0x04d2896c (object 0x2705cc58, a eu.javaspec:
  which is held by "pool-1-thread-2"
"pool-1-thread-2":
  waiting to lock monitor 0x0256e45c (object 0x2705cc60, a eu.javaspec:
  which is held by "pool-1-thread-3"
"pool-1-thread-3":
  waiting to lock monitor 0x0256e3f4 (object 0x2705cc68, a eu.javaspec:
  which is held by "pool-1-thread-4"
"pool-1-thread-4":
  waiting to lock monitor 0x04d2966c (object 0x2705cc70, a eu.javaspec:
  which is held by "pool-1-thread-5"
```

Once the culprit code is identified, solutions involve addressing the lock-ordering conditions and/or using other available concurrency programming techniques from the JDK such as `java.util.concurrent.locks.ReentrantLock`, which provides methods such as `tryLock()`. This approach gives Java developers much more flexibility and ways to prevent deadlock or thread lock "starvation."

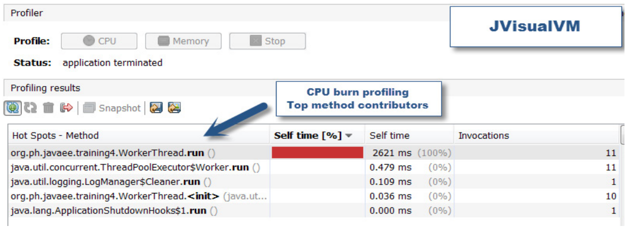
### CLOCK TIME AND CPU BURN

In parallel with the JVM tuning, it is also essential that you review your application behavior, more precisely the highest clock time and CPU burn contributors.

When the Java garbage collection and thread concurrency are no longer a pressure point, it is important to drill down into your application code execution patterns and focus on the top response time contributors, referred as clock time. It is also crucial to review the CPU consumption of your application code and Java threads (CPU burn). High CPU utilization (> 75%) should not be assumed to be "normal" (good physical resource utilization). It is often the symptom of inefficient implementation and/or capacity problems. For large Java EE enterprise applications, it is essential to keep a safe CPU buffer zone in order to deal with unexpected load surges.

Stay away from traditional tracing approaches such as adding response time "logging" in your code. Java profiler tools and APM solutions exist precisely to help you with this type of analysis and in a much more efficient and reliable way. For Java production environments lacking a robust APM solution, you can still rely on tools such Java VisualVM, thread dump analysis (via multiple snapshots) and OS CPU per Thread analysis.

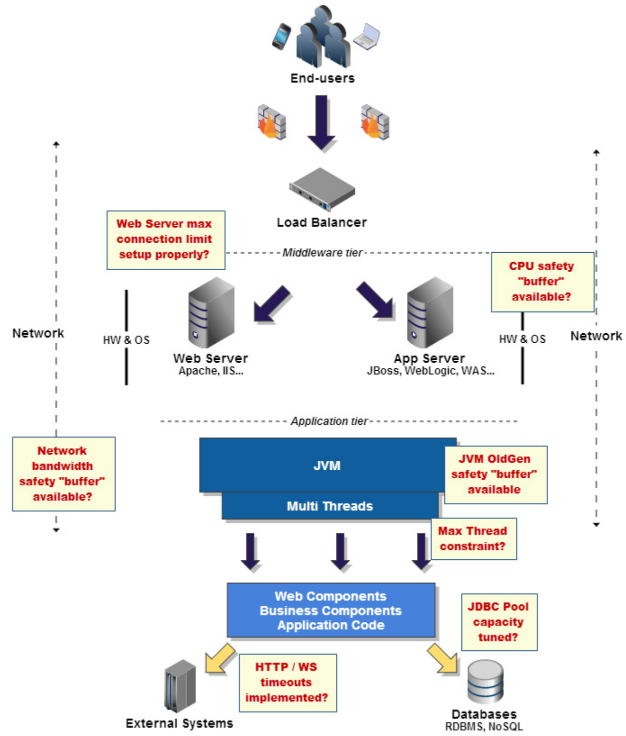
Finally, do not try to address all problems at the same time. Start by building a list of your top five clock time and CPU burn contributors and explore solutions.




## APPLICATION BUDGETING

Other important aspects of your Java applications performance are stability and reliability. This is particularly important for applications operating under a SLA umbrella with typical availability targets of 99.9%. These systems require a high fault-tolerant level, with strict application and resource budgeting in order to prevent domino effect scenarios. This approach prevents for example one business process from using all available physical, middleware, or JVM resources.

### HOT SPOTS




### TIMEOUT MANAGEMENT

Lack of proper HTTP/HTTPS/TCP IP timeouts between your Java application and external systems can lead to severe performance degradation and outage due to middleware and JVM threads depletion (blocking IO calls). Proper timeout implementation will prevent Java threads from waiting for too long in the event of major slowdown of your external service providers.

```
"http-0.0.0.0-8443-102" daemon prio=3 tid=0x022a6400 nid=0x1bd runnable [0x78efb00
   java.lang.Thread.State: RUNNABLE
  at java.net.SocketInputStream.socketRead0(Native Method)
  at java.net.SocketInputStream.read(SocketInputStream.java:129)
  at com.sun.net.ssl.internal.ssl.InputRecord.readFully(InputRecord.java:293)
  at com.sun.net.ssl.internal.ssl.InputRecord.read(InputRecord.java:331)
  at com.sun.net.ssl.internal.ssl.SSLSocketImpl.readRecord(SSLSocketImpl.java:789)
  - locked <0xdd0ed968> (a java.lang.Object)
  at com.sun.net.ssl.internal.ssl.SSLSocketImpl.readD
  at com.sun.net.ssl.internal.ssl.AppInputStream.read
  - locked <0xdd0eda88> (a com.sun.net.ssl.internal.s
  at java.io.BufferedInputStream.fill(BufferedInputSt
  at java.io.BufferedInputStream.read1(BufferedInputS
  at java.io.BufferedInputStream.read(BufferedInputSt
  - locked <0xddb1f6d0> (a java.io.BufferedInputStrea
  at sun.net.www.http.HttpClient.parseHTTPHeader(HttpClient.java:687)
  at sun.net.www.http.HttpClient.parseHTTP(HttpClient.java:632)
  at sun.net.www.http.HttpClient.parseHTTP(HttpClient.java:652)
  ..................
```

**STUCK thread due to lack of HTTPS timeout with a remote Web Service provider (blocking IO)**

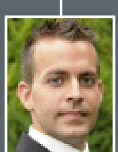
## TOOLS

| GOALS | RECOMMENDED TOOLS |
|---|---|
| Pro-active and real-time performance monitoring, tuning, alerting, trending, capacity management and more | Enterprise APM solutions<br><br>**NOTE**: APM solutions provide tools allowing you to achieve most of the following Java performance goals out-of-the-box |
| Performance and load testing | Commercial performance testing solutions<br><br>Apache JMeter (jmeter.apache.org) |

| | |
|---|---|
| **JVM garbage collection assessment, memory allocation rate and troubleshooting** | **Oracle Java VisualVM**<br>docs.oracle.com/javase/8/docs/technotes/guides/visualvm/intro.html<br><br>java.dzone.com/articles/profile-your-applications-java<br><br>**Oracle Java Mission Control**<br>oracle.com/technetwork/java/javaseproducts/mission-control/java-mission-control-wp-2008279.pdf<br><br>oracle.com/technetwork/java/javase/jmc53-release-notes-2157171.html<br><br>**IBM Monitoring and Diagnostic Tools for Java** (via IBM Support Assistant tool)<br>www-01.ibm.com/software/support/isa<br><br>**JVM verbose:gc logs**<br>JVM argument : `-verbose:gc`<br>docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html<br><br>**IBM GCMV**<br>ibm.com/developerworks/java/jdk/tools/gcmv |
| **JVM heap and class metadata memory leak analysis** | **Oracle Java VisualVM** and **Oracle Java Mission Control**<br><br>**IBM Monitoring and Diagnostic Tools for Java**<br><br>**Memory Analyzer** (heap dump analysis, hprof and phd formats)<br>eclipse.org/mat<br>ibm.com/developerworks/java/jdk/tools/memoryanalyzer<br><br>**Plumbr** (Java memory leak detector)<br>plumbr.eu<br><br>**jmap** (heap histogram and heap dump generation)<br>oracle.com/technetwork/java/javase/tooldescr-136044.html#gbdid<br><br>**JVM verbose:class logs**<br>JVM argument : `-verbose:gc & -verbose:class`<br><br>**IBM Java core file analysis** (via kill -3 <PID>)) |
| **JVM memory profiling and heap capacity sizing** | **Oracle Java VisualVM and Java Mission Control**<br><br>**IBM Monitoring and Diagnostic Tools for Java**<br><br>**Java profilers** (JProfiler, YourKit)<br>en.wikipedia.org/wiki/JProfiler<br><br>www.yourkit.com<br><br>**Memory Analyzer** (heap dump and application memory footprint analysis) |

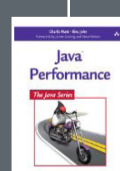
| | |
|---|---|
| **JVM and middleware concurrency troubleshooting such as thread lock contention and deadlocks** | Oracle Java VisualVM and Oracle Java Mission Control (threads monitoring, thread dump snapshots)<br><br>jstack, native OS signal such as kill -3 (thread dump snapshots)<br>oracle.com/technetwork/java/javase/tooldescr-136044.html#gblfh<br><br>**IBM Monitoring and Diagnostic Tools for Java**<br><br>**NOTE:** Proper knowledge on how to perform a JVM thread dump analysis is highly recommended |
| **Java application clock time analysis and profiling** | **Oracle Java VisualVM** and **Oracle Java Mission Control** (build-in profiler, sampler and recorder)<br><br>**Java profilers** (JProfiler, YourKit) |
| **Java application and threads CPU burn analysis** | **Oracle Java VisualVM** and **Oracle Java Mission Control** (CPU profiler)<br><br>**Java profilers** (JProfiler, YourKit)<br><br>**NOTE:** You can also fall back on JVM thread dump and OS CPU per Thread analysis, if necessary |
| **Java IO and remoting contention analysis, including timeout management assessment and tuning** | **Oracle Java VisualVM** and Oracle Java Mission Control (threads monitoring, thread dump snapshots)<br><br>**jstack**, native OS signal such as **kill -3** (thread dump snapshots)<br><br>**IBM Monitoring and Diagnostic Tools for Java**<br><br>**NOTE:** Proper knowledge on how to perform a JVM thread dump analysis is highly recommended |
| **Middleware, Java EE container tuning such as threads, JDBC data sources and more.** | **Oracle Java VisualVM** and **Oracle Java Mission Control** (extra focus on exposed Java EE container runtime MBeans)<br><br>**Java EE container** administration and **management console** |

## ABOUT THE AUTHOR



**PIERRE-HUGUES ['P-H'] CHARBONNEAU** has worked as an IT architect for CGI Canada for the last 15 years. He specializes in production system troubleshooting, middleware, JVM tuning and capacity analysis. P-H is the creator and primary author of Java EE Support Patterns, a technology blog dedicated to Java EE and Java technologies, focusing on memory leaks, coding best practices, anti-patterns, and troubleshooting techniques. He also runs a YouTube channel offering interactive videos and tutorials on Java and middleware. In his free time he enjoys cinema, sports, nutrition and spending quality time with his family.

## RECOMMENDED BOOK




Java Performance is the most comprehensive book on the subject, covering all aspects of performance tuning at every stage. Includes sections on JVM performance monitoring, systematic profiling, HotSpot tuning, bencharking, web application and services performance, Enterprise Java Beans performance, tips and tricks, and more.

BUY NOW



DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513
888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776

VERSION 1.0