Java Performance in 2021: How AI Optimization Will Debunk 4 Long-standing Java Tuning Myths
Who Am I

Obsessed with Performance Optimization
16 years of capacity & performance work
CMG speaker since 2014, Best Paper on Java performance & efficiency in 2015
Co-founder and CTO @ Akamas, a software platform for Autonomous Performance Optimization powered by AI
Why Java Tuning? Why AI?
JVM Tuning Is a Very Hot Topic!

Insight on JVM Tuning

Everything I Ever Learned About JVM Performance Tuning at Twitter

Home > Resources > Articles > Apache Tomcat > Tomcat JVM - What you need to know

Tomcat JVM - What you need to know

Garbage Collection Tuning for Apache Cassandra

Tricks of the Trade: Tuning JVM Memory for Large-scale Services

Xini Shang, Yi Zhang, Fengnan Li, Amruth Saripath, and Girish Baliga
February 26, 2020
JVM Tuning Provides Real Performance and Reliability Benefits

**Tuned** configuration reaches higher transactions/sec and remains **stable** with increasing load

**Default** configuration slows down after peak load then **crash**
“Because Java is so often deployed on servers, this kind of performance tuning is an **essential activity** for many organizations.

The JVM is highly configurable with literally **hundreds of command-line options** and switches. These switches provide performance engineers a **gold mine** of possibilities to explore in the pursuit of the **optimal configuration** for a given workload on a given platform.”

December 4, 2020

Number of OpenJDK HotSpot JVM Options

- JDK6: 691 options in 2013
- JDK11: 846 options in 2018
- JDK15: 731 options in 2020
How JVM Tuning Is Done Today

JVM Tuning is done by performance engineers with an iterative, trial-and-error process.

To guide the tuning process, we rely on industry best practices and hard-won lessons learned on the obscure internals of modern JVMs.

But... do they actually work?
AI quickly explores the complex parameter space much more quickly and smartly than humans.

The performance engineer defines the goal of the optimization in application/business terms e.g. maximize application throughput, minimize cost within performance SLO.

The entire process is automated (run test, compare results, pick new options, configure them, etc.)
Debunking Java Performance Myths
or...
what the human experts learned from the intelligent machines
Myth #1 - Tuning JVM garbage collection performance leads to faster applications
Garbage Collection Performance 101

The key GC performance metrics are:

- **GC Overhead %** aka *GC Time %* is the percentage of time spent in garbage collection (GC pauses)
- **Throughput** is the percentage of time spent running application threads
- **Footprint** is the amount of resources (CPU and memory) required by the JVM
The Golden Rule of GC Performance: Lower GC Time!

The graph models using Amdhal’s Law an ideal system that’s perfectly scalable with the exception of garbage collection.

“It's very important to keep the overhead of doing garbage collection as low as possible”, Oracle GC Tuning Guide *

Fact: App Runs Faster While Having Higher GC Overhead

Application
Spark job

Optimization Goal
Minimize execution time

JVM
OpenJDK 11

Baseline
Default config (G1, 2 GB heap)
Why Is That? JVM GC Threads Can Steal CPUs From Application Threads

<table>
<thead>
<tr>
<th>Application Threads</th>
<th>G1 GC Concurrent Threads</th>
<th>G1 GC Parallel Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>driver-heartbeat 865249</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Executor task 1 865377</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Executor task 1 865378</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Executor task 1 865379</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Executor task 1 865380</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Executor task 1 865381</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Executor task 1 865382</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Executor task 1 865383</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Executor task 1 865384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G1 Concurrent 0 865195</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G1 Concurrent 1 865216</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G1 Main Marker 865194</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G1 Refine 0 865196</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G1 Refine 1 865260</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G1 Young RemSet 865197</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GC Thread 0 865193</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GC Thread 1 865211</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GC Thread 2 865212</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GC Thread 3 865213</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GC Thread 4 865214</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GC Thread 5 865256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GC Thread 6 865257</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GC Thread 7 865258</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Running threads, i.e. on CPU**
- **Runnable threads, i.e. app threads are waiting to be scheduled onto CPU due to competition with G1 GC concurrent threads**
- **Stop-the-world pause**

Linux CPU scheduler tracing visualization by perfetto.dev
Myth #1 - Tuning JVM garbage collection performance leads to faster applications

**BUSTED**

Key Takeaway

**JVM Performance != Application Performance**

If we blindly follow JVM performance **best practices**, we can make our apps run **slower** or consume **more resources**.
Myth #2 - You cannot escape the Throughput - Latency - Footprint trade-off
2 of 3 principle

“Improving one or two of these performance attributes, (throughput, latency or footprint) results in sacrificing some performance in the other”

Charlie Hunt
former Oracle JVM Performance Lead

Fact: App Runs Faster, With Higher Throughput and Lower Footprint

Application
Enterprise CRM (Tomcat 7.0)

Optimization Goal
Minimize footprint with response time constraints

JVM
OpenJDK 8, running on RHEL 7 (4 CPUs, 8 GB RAM)

Baseline
Default (1.7 GB heap)

<table>
<thead>
<tr>
<th>Heap size (goal)</th>
<th>Baseline</th>
<th>Best</th>
<th>Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.7 GB</td>
<td>252.56 MB</td>
<td>-85.5%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Throughput</th>
<th>Response time (AVG)</th>
<th>Response time (90pct)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>6.95 req/s</td>
<td>3.5 s</td>
</tr>
<tr>
<td>Best</td>
<td>8.59 req/s</td>
<td>2.72 s</td>
</tr>
<tr>
<td>Δ</td>
<td>+23.6%</td>
<td>-22%</td>
</tr>
</tbody>
</table>
Fact: 20% Faster App With 36% Lower CPU Usage

Java batch, OpenJDK 11

- 20% lower response time
- 36% lower CPU used
- 20% lower memory used

Some optimal JVM options AI identified
Myth #2 - You cannot escape the Throughput - Latency - Footprint trade-off.

Key Takeaway

If we look at the broader application performance picture, it is possible to tune the JVM to achieve all three and exploit the trade-offs to our advantage.
Myth #3 - Let the JVM do it!
JVMs May Trade-Off the Wrong Performance Metric

Renaissance benchmark, OpenJDK 11 default config (G1 GC)

App runs faster with more memory

GC time increases with more memory

<table>
<thead>
<tr>
<th>Heap (Xmx)</th>
<th>Execution time</th>
<th>GC time</th>
<th>Max heap used</th>
<th>GC count</th>
<th>GC CPU time</th>
<th>GC pause avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 GB</td>
<td>282 s</td>
<td>16.1 %</td>
<td>2022 MB</td>
<td>855</td>
<td>176 s</td>
<td>53 ms</td>
</tr>
<tr>
<td>3 GB</td>
<td>256 s</td>
<td>17.8 %</td>
<td>2478 MB</td>
<td>636</td>
<td>178 s</td>
<td>72 ms</td>
</tr>
<tr>
<td>4 GB</td>
<td>250 s</td>
<td>18.4 %</td>
<td>2766 MB</td>
<td>585</td>
<td>176 s</td>
<td>78 ms</td>
</tr>
<tr>
<td>5 GB</td>
<td>223 s</td>
<td>19.7 %</td>
<td>3400 MB</td>
<td>507</td>
<td>172 s</td>
<td>87 ms</td>
</tr>
</tbody>
</table>

G1 preferred not to use the available memory, severely impacting throughput and CPU usage
What’s the best GC for my application?

“GC is the core focus of much Java performance tuning. And with the advent of new GC models, it is difficult to assess which one works best for a given type of workload.”

December 4, 2020
The Evolution of Garbage Collectors

The OpenJDK community works hard to improve current GCs and create new “pause-less” collectors.

JDK 11 HotSpot collectors:

- Serial: single-threaded GC, memory efficient, great for small memory
- Parallel: multi-threaded GC, great for throughput-oriented applications
- Concurrent Mark and Sweep (CMS): deprecated as of JDK 9
- G1: multi-threaded GC, meets pause goals and provides good throughput
- Z: low-latency GC, available from JDK15
- Shenandoah: low-latency GC, available from JDK15
Evaluating GCs performance and resource efficiency

Renaissance benchmark, OpenJDK 15 default config

All delta (%) values are related to the Baseline (G1 GC)

<table>
<thead>
<tr>
<th>Score</th>
<th>SERIAL</th>
<th>Parallel</th>
<th>Z</th>
<th>Shenandoah</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
<td>2.78 CPUs -22.3%</td>
<td>1.68 GB -49.4%</td>
<td>1.68 GB -49.4%</td>
<td>1.68 GB -49.4%</td>
</tr>
<tr>
<td>9.6%</td>
<td>2.87 CPUs -22.3%</td>
<td>2.56 tasks -29.8%</td>
<td>2.56 tasks -29.8%</td>
<td>2.56 tasks -29.8%</td>
</tr>
<tr>
<td>22.3%</td>
<td>2.11 CPUs +0.9%</td>
<td>2.3 GB -30.8%</td>
<td>3.6 tasks -1.1%</td>
<td>3.6 tasks -1.1%</td>
</tr>
<tr>
<td>18.8%</td>
<td>3.58 CPUs -0.1%</td>
<td>5.5 GB +67.5%</td>
<td>4.41 tasks +21.1%</td>
<td>4.41 tasks +21.1%</td>
</tr>
<tr>
<td>23.9%</td>
<td>3.66 CPUs +2.2%</td>
<td>4.25 GB +28%</td>
<td>4.32 tasks +18.5%</td>
<td>4.32 tasks +18.5%</td>
</tr>
</tbody>
</table>

**Serial** is 10% slower, but very efficient on memory (-49%) and CPU (-22%)

**Parallel** is 22% faster, while also being very efficient on memory (-31%)

**Z and Shenandoah** are significantly slower and inefficient on both memory and CPUs
Vendor Defaults Can Make Your App Slower and More Costly

... what about memory consumption? Is there any evidence how much more native memory a compiler thread will use after such a change?

... real data from real applications is of course needed to know for sure, but the evidence we have so far suggests the benefits are significant while potential downsides ... have not manifested in our tests.

... there is surely a risk that on average some specific application will see more large compilations. An open question is if this will have a noticeable effect.

https://bugs.openjdk.java.net/browse/JDK-8234863
Key Takeaway

Modern JVMs constantly **evolve** to support a huge variety of scenarios. **Default** settings may be far from optimal for your specific applications.
Myth #4 - Cargo cult JVM tuning
Some JVM misconfigurations we’ve found

Setting GC threads in K8s containers

“We set the number of GC parallel threads set equal to half the container CPU limits, to avoid impacting application threads”

Problem: confusion about concurrent vs parallel threads *

Setting initial heap size equal to max heap size in k8s containers

“We always do that to avoid unnecessary GC cycles”

Problem: this turns off dynamic memory footprint of the JVM **

** https://www.openshift.com/blog/scaling-java-containers
Myth #4 - Cargo culting JVM tuning

BUSTED

Key Takeaway

You cannot copy & paste JVM configurations as they are specific to the application, environment, and optimization goal.
Wrap up
So, What Did We Learn?

JVM tuning is a hot topic, and rightly so - it can provide significant application performance and reliability benefits - but it’s hard

By tuning hundreds of JVMs, we got some groundbreaking evidence showing that some long-standing Java performance best practices may not actually work.

Performance engineers need to level up the game and leverage new tools like AI to exploit the significant optimization potential - the future is bright!
Contacts

Italy HQ
Via Schiaffino 11
Milan, 20158
+39-02-4951-7001

USA East
211 Congress Street
Boston, MA 02110
+1-617-936-0212

USA West
12655 W. Jefferson Blvd
Los Angeles, CA 90066
+1-323-524-0524

Singapore
5 Temasek Blvd
Singapore 038985

@Akamas
@AkamasLabs
@akamaslabs
@movirigroup

© 2021 Akamas • All Rights Reserved • Confidential