IMPACT 2021

JANUARY 2021 VIRTUAL CONFERENCE EVENT

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

CMG
YOUR TECHNOLOGY COMMUNITY

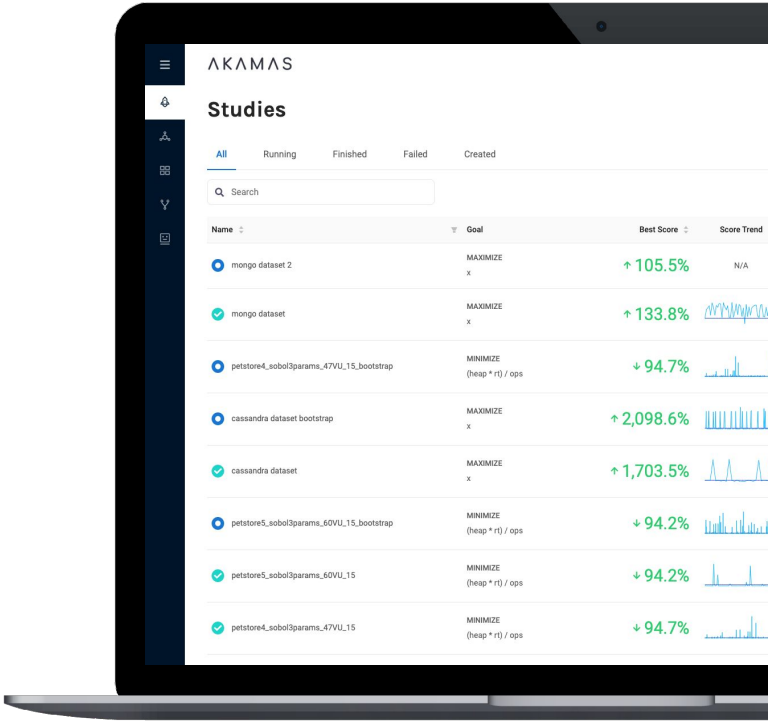
# Who Am I

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






AKAMAS

# Why Java Tuning? Why AI?

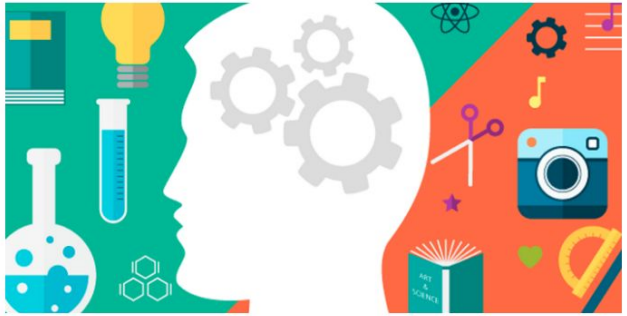


AKAMAS

# JVM Tuning Is a Very Hot Topic!

## Insight on JVM Tuning

Sumit Nagal Aug 23, 2020 · 11 min read



MuleSoft

## 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

Uber Engineering

JON HADDAD

## Garbage Collection Tuning for Apache Cassandra

General Engineering

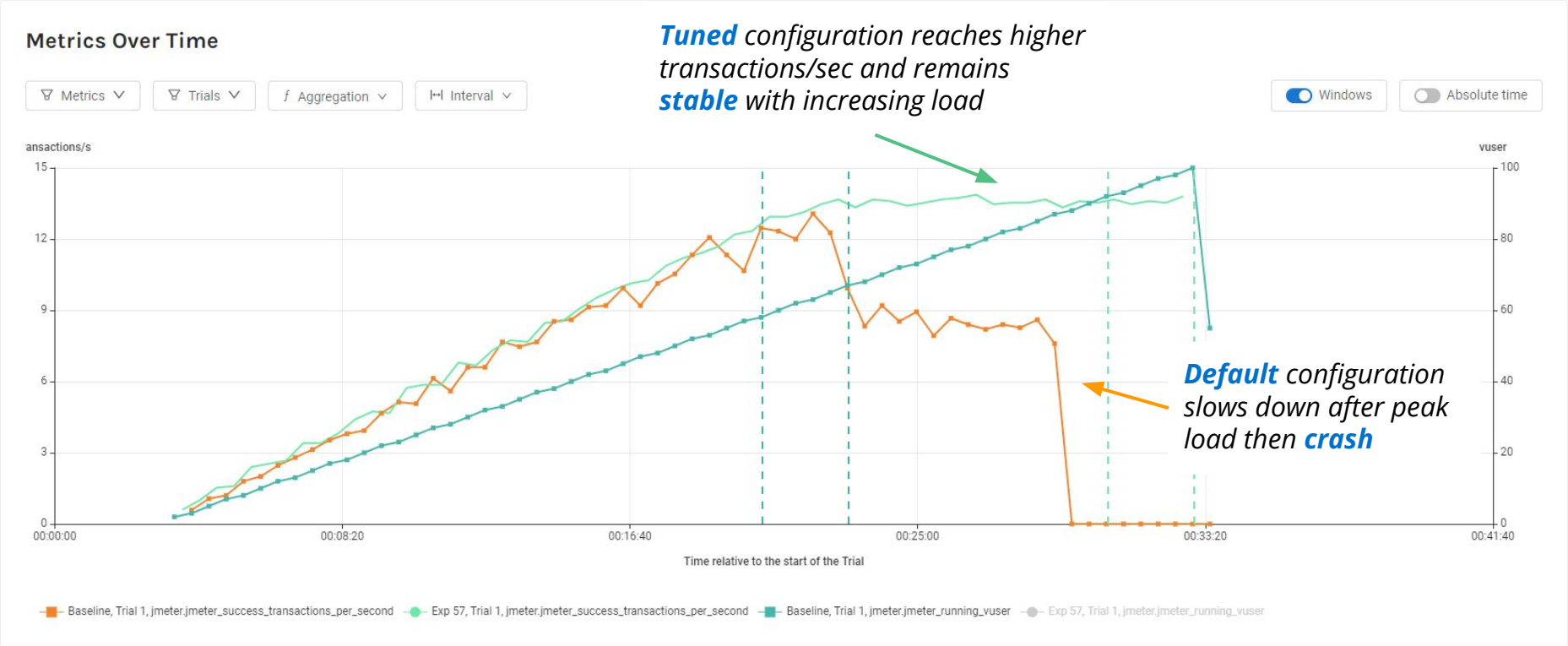
General Engineering

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

Xinli Shang, Yi Zhang, Fengnan Li, Amruth Sampath, and Girish Baliga

February 26, 2020

# JVM Tuning Provides Real Performance and Reliability Benefits



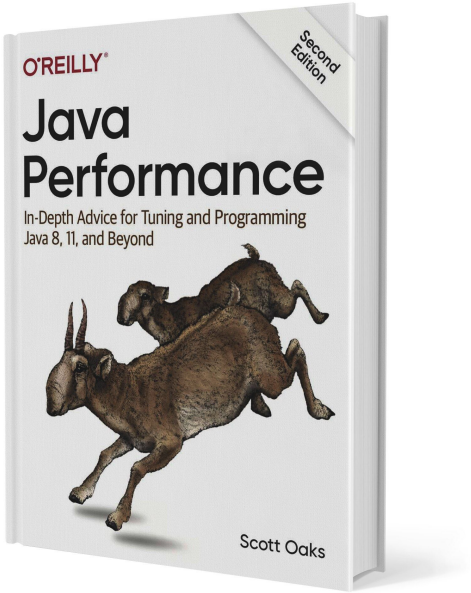



AKAMAS

# JVM Tuning Is Essential, but It's Hard

*"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."*

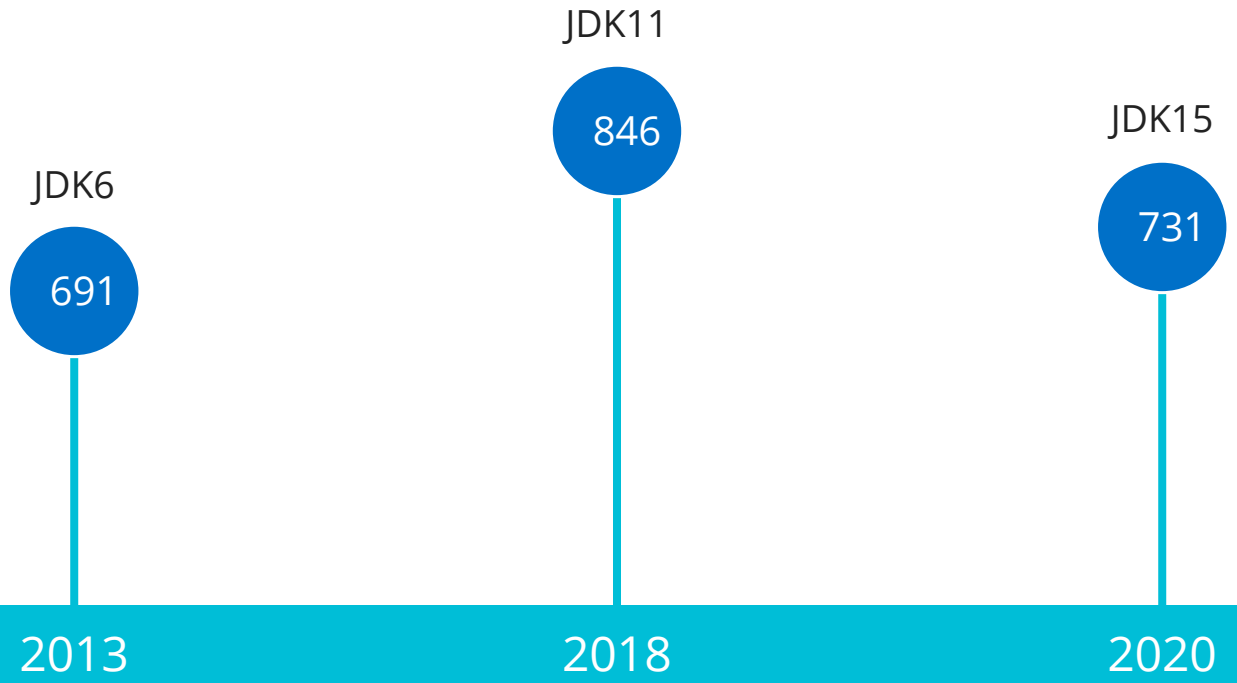



December 4, 2020
https://blogs.oracle.com/javamagazine/java-performance-2nd-edition



AKAMAS

# Number of OpenJDK HotSpot JVM Options

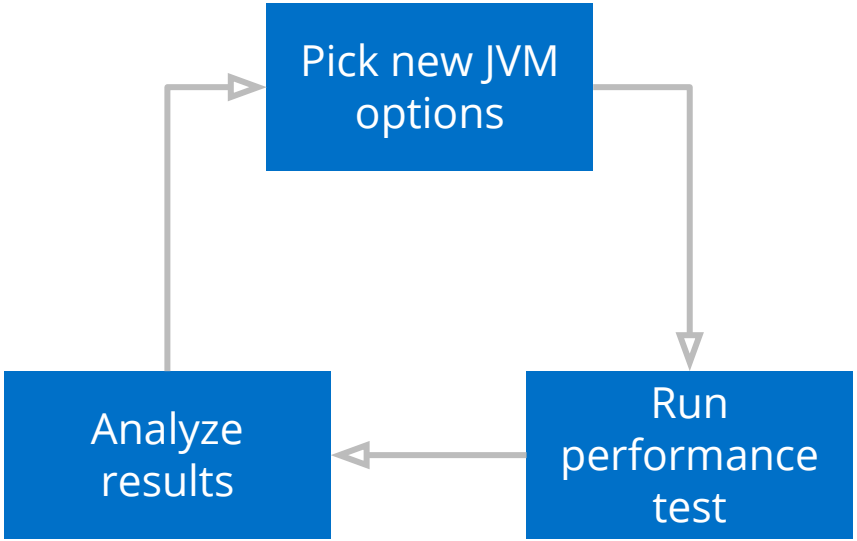


# How JVM Tuning Is Done Today

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?





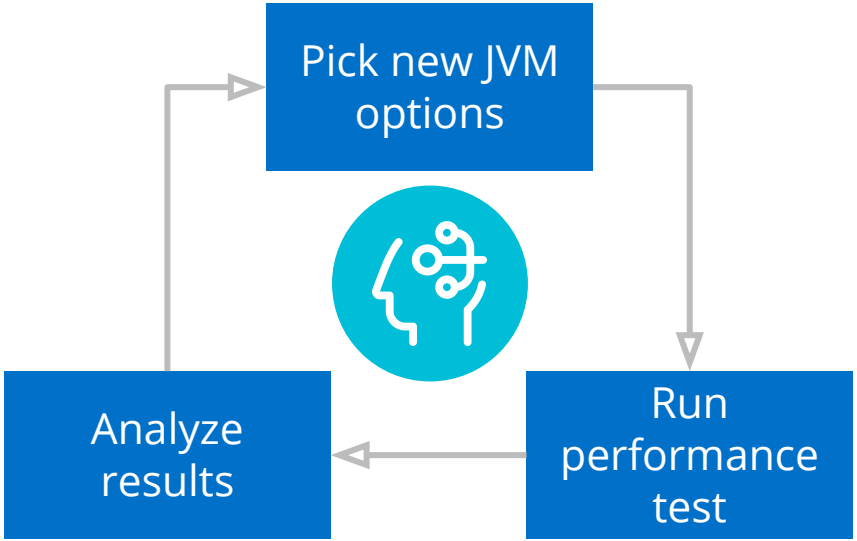

AKAMAS

# How AI-driven Performance Tuning Looks Like

**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.)






AKAMAS

# Debunking Java Performance Myths

or...

what the human experts learned from the intelligent machines



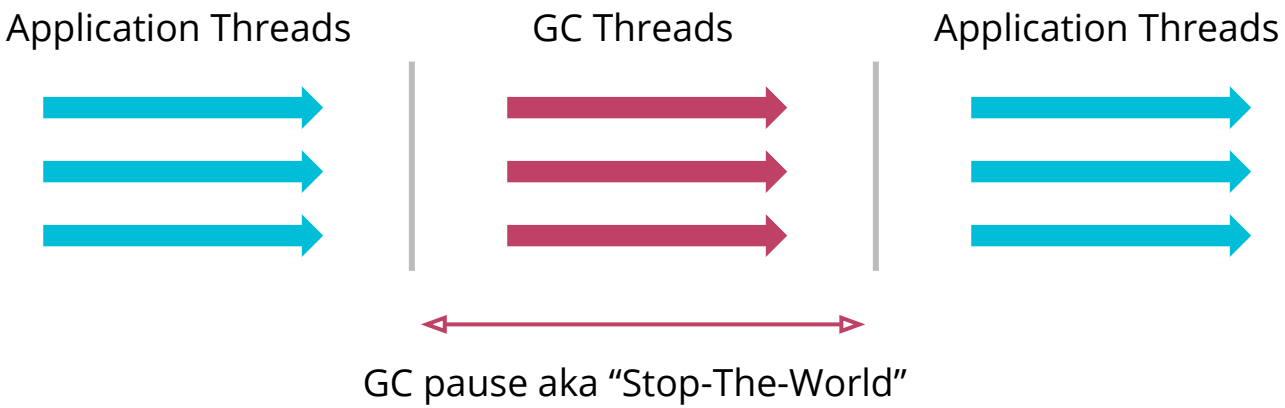
AKAMAS

# Myth #1 - Tuning JVM garbage collection performance leads to faster applications



AKAMAS

# 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

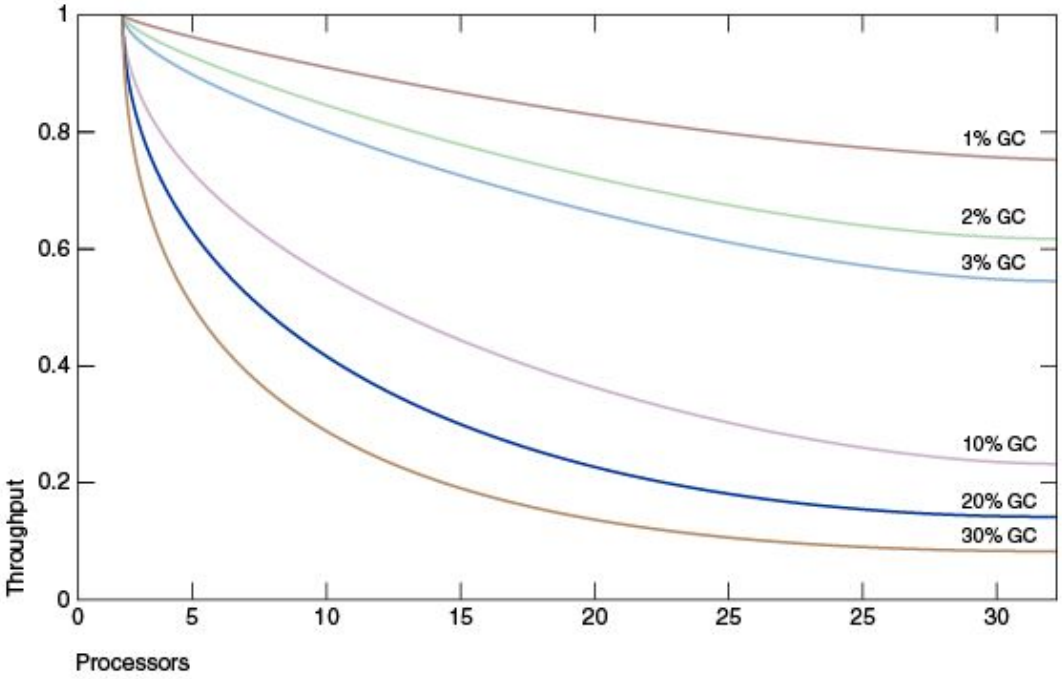


AKAMAS

# 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* *




* https://docs.oracle.com/en/java/javase/11/gctuning/introduction-garbage-collection-tuning.html



AKAMAS

# Fact: App Runs Faster While Having Higher GC Overhead

## Application

Spark job

## Optimization Goal

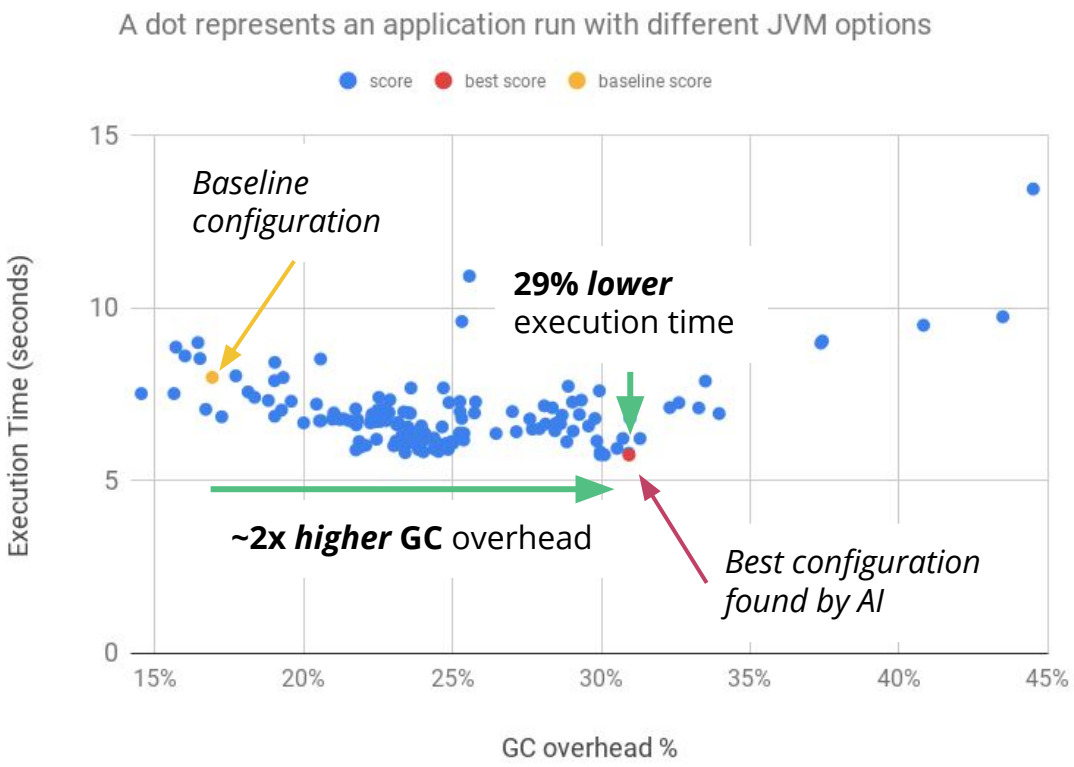
Minimize execution time

## JVM

OpenJDK 11

## Baseline

Default config (G1, 2 GB heap)






AKAMAS

# Why Is That? JVM GC Threads Can Steal CPUs From Application Threads

*Application Threads*

*G1 GC Concurrent Threads*

*G1 GC Parallel Threads*


*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*



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.**



AKAMAS

# Myth #2 - You cannot escape the Throughput - Latency - Footprint trade-off



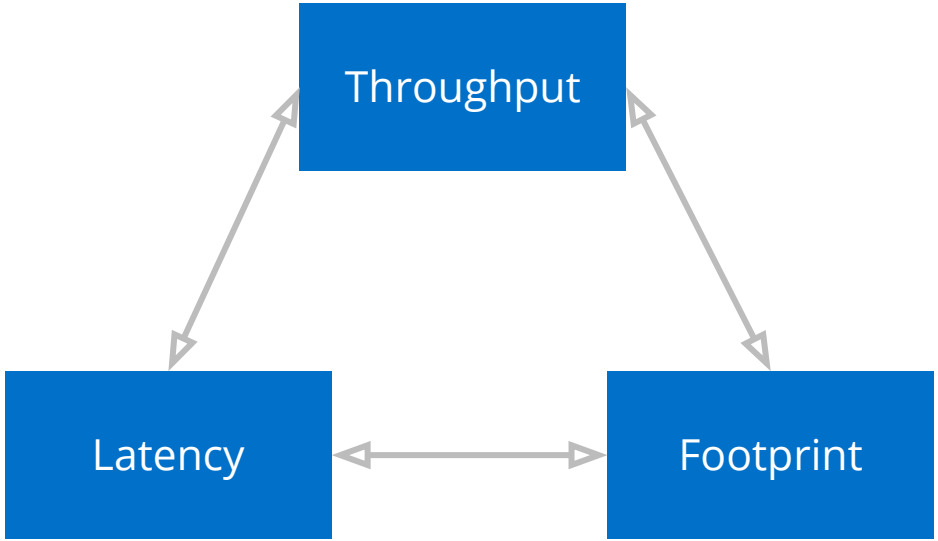
AKAMAS

# The Java Performance Tradeoffs

**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*




* http://gotocon.com/dl/goto-chicago-2014/slides/CharlieHunt_TheFundamentalsOfGCPerformance.pdf



AKAMAS

# 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)

| | **Heap size (goal)** |
|---|---|
| Baseline | 1.7 GB |
| Best | 252.56 MB |
| Δ | **-85.5%** |

| | **Throughput** | **Response time (AVG)** | **Response time (90pct)** |
|---|---|---|---|
| Baseline | 6.95 req/s | 3.5 s | 5.35 s |
| Best | 8.59 req/s | 2.72 s | 4.42 s |
| Δ | **+23.6%** | **-22%** | **-17%** |



AKAMAS

# Fact: 20% Faster App With 36% Lower CPU Usage

## Java batch, OpenJDK 11

*Some optimal JVM options AI identified*

| Score | METRIC benchmark benchmark_cpu_used | | METRIC benchmark benchmark_cpu_util | | METRIC benchmark benchmark_duration | | METRIC jvm jvm_memory_used | | METRIC linux cpu_load_avg | | PARAMETER jvm jvm_gcType | PARAMETER jvm jvm_maxTenuringThreshold | | PARAMETER jvm jvm_newSize | | PARAMETER jvm jvm_parallelGCThreads | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ↓ 20.0% | 2.35 CPUs | -36.8% | 58.87 % | -36.8% | 6.12 s | -20.0% | 1.27 GB | -20.2% | 2.17 tasks | -54.3% | Parallel | 6.00 | -60.0% | 454.00 MB | +51.3% | 1.00 threads | -75.0% |
| 6.12 s | 3.73 CPUs | | 93.16 % | | 7.64 s | | 1.59 GB | | 4.74 tasks | | G1 | 15.00 | | 300.00 MB | | 4.00 threads | |

Total 553 items < 1 ••• 6 7 8 9 10 ••• 19 > 30 / page

*20% lower response time*

*36% lower CPU used*

*20% lower memory used*



AKAMAS

# Myth #2 - You cannot escape the Throughput - Latency - Footprint trade-off

BUSTED

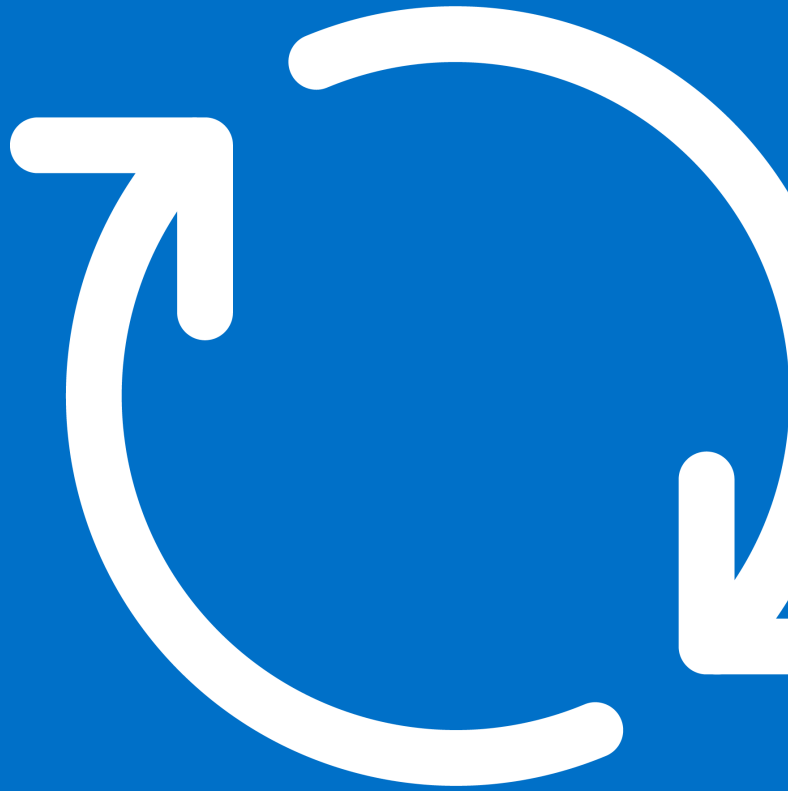
## Key Takeaway

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



AKAMAS

# Myth #3 - Let the JVM do it!





AKAMAS

# JVMs May Trade-Off the Wrong Performance Metric

## Renaissance benchmark, OpenJDK 11 default config (G1 GC)

*App runs faster with more memory*

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

*GC time increases with more memory*

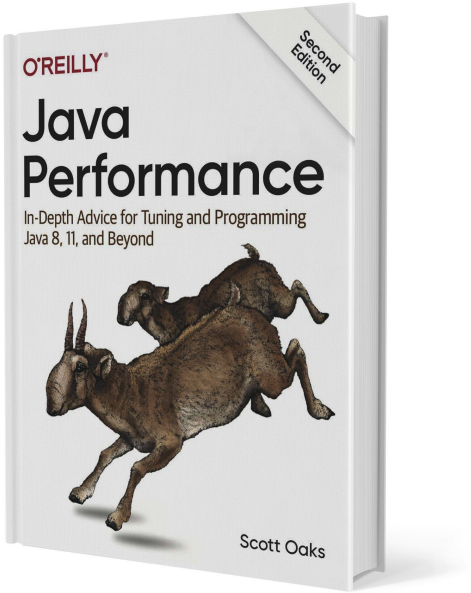
*G1 preferred not to use the available memory, severely impacting throughput and CPU usage*



AKAMAS

# 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
https://blogs.oracle.com/javamagazine/java-performance-2nd-edition



AKAMAS

# 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)*

| Score | METRIC jvm cpu_used | | METRIC jvm jvm_heap_size | | METRIC jvm jvm_memory_used | | METRIC Linux cpu_load_avg | | METRIC Linux cpu_util | | METRIC renaissance duration | | PARAMETER jvm jvm_gcType |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0%<br>6.11 s | 3.63 CPUs | | 8 GB | | 3.66 GB | | 3.53 tasks | | 79.36% | | 6.11 s | | G1 |
| ↑ 9.6%<br>6.61 s | 2.78 CPUs<br>3.58 CPUs | -22.3% | 7.73 GB<br>8 GB | -3.3% | 1.68 GB<br>3.32 GB | -49.4% | 2.56 tasks<br>3.65 tasks | -29.8% | 65.62%<br>84.39% | -22.2% | 6.61 s<br>6.03 s | +9.6% | Serial<br>G1 |
| ↓ 22.3%<br>4.68 s | 3.61 CPUs<br>3.58 CPUs | +0.9% | 7.11 GB<br>8 GB | -11.1% | 2.3 GB<br>3.32 GB | -30.8% | 3.6 tasks<br>3.65 tasks | -1.1% | 83.06%<br>84.39% | -1.6% | 4.68 s<br>6.03 s | -22.3% | Parallel<br>G1 |
| ↑ 18.8%<br>7.17 s | 3.58 CPUs<br>3.58 CPUs | -0.1% | 8 GB<br>8 GB | 0% | 5.56 GB<br>3.32 GB | +67.5% | 4.41 tasks<br>3.65 tasks | +21.1% | 90.98%<br>84.39% | +7.8% | 7.17 s<br>6.03 s | +18.8% | Z<br>G1 |
| ↑ 23.9%<br>7.47 s | 3.66 CPUs<br>3.58 CPUs | +2.2% | 8 GB<br>8 GB | 0% | 4.25 GB<br>3.32 GB | +28% | 4.32 tasks<br>3.65 tasks | +18.5% | 87.46%<br>84.39% | +3.6% | 7.47 s<br>6.03 s | +23.9% | Shenandoah<br>G1 |

Total 15 items < 1 > 30 / page

***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*

AKAMAS

# Vendor Defaults Can Make Your App Slower and More Costly

JDK / JDK-8234863

Increase default value of MaxInlineLevel

... 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



AKAMAS

# Myth #3 - “Let the JVM do it”

BUSTED

## Key Takeaway

Modern JVMs constantly **evolve** to support a huge variety of scenarios
**Default** settings may be far from optimal for your specific applications



AKAMAS

# Myth #4 - Cargo cult JVM tuning



AKAMAS

# 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://docs.oracle.com/en/java/javase/11/gctuning/garbage-first-garbage-collector-tuning.html
** https://www.openshift.com/blog/scaling-java-containers



AKAMAS

# 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**



AKAMAS

# Wrap up



AKAMAS

# 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



AKAMAS