# JVM Performance Comparison for JDK 21

**Ionut Balosin**

www.IonutBalosin.com | ionut.balosin@gmail.com | @ionutbalosin

# Agenda

# Introduction

01

# Ionut Balosin

- Software Architect @ Raiffeisen Bank International
- Technical Trainer
- Security Champion
- Oracle ACE Associate
- Blogger
- Speaker

www.IonutBalosin.com | ionut.balosin@gmail.com | @ionutbalosin

# My Training Catalogue

**Software Architecture Essentials**

**Java Performance Tuning**

**Designing High-Performance, Scalable, and Resilient Applications**

**Application Security for Java Developers**

**Training figures**: 80+ sessions | 900+ trainees | 1300+ hours | 10+ clients | 4+ countries
**Conference figures**: 35+ sessions | 14+ countries

**www.IonutBalosin.com/training**

# Thanks to Florin Blanaru (co-author)

- Senior Software Engineer @ AXELERA ARTIFICIAL INTELLIGENCE
- TornadoVM - ex contributor
- Student of the year Award from the RISC-V Foundation, 2019

Compilers

Language Runtimes

Performance Analysis & Tuning

GitHub: @gigiblender | X: @gigiblender | Mastodon: @gigiblender https://ionutbalosin.com/2024/02/jvm-performance-comparison-for-jdk-21

# JVM Performance Comparison for JDK 21

## Context

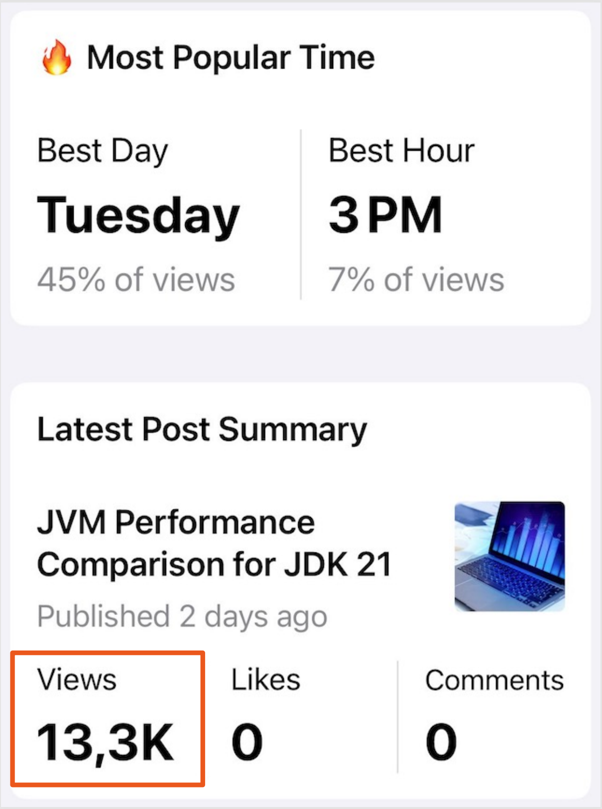
The current article describes a series of Java Virtual Machine (JVM) benchmarks with a primary focus on top-tier Just-In-Time (JIT) compilers, such as C2 JIT, and Graal JIT. The benchmarks are structured in three distinct (artificial) categories:

1. **Compiler**: This category is dedicated to assessing JIT compiler optimizations by following specific handwritten code patterns. It assesses common optimizations found in compilers, including inlining, loop unrolling, escape analysis, devirtualization, null-check elimination, range-check elimination, dead code elimination, vectorization, etc.
2. **Api**: This category includes benchmarks targeting common APIs from both the Java Platform, Standard Edition (Java SE) (e.g., `java.io`, `java.nio`, `java.net`, `java.security`, `java.util`, `java.text`, `java.time`, etc.) and the Java Development Kit (JDK) (e.g., `jdk.incubator.vector`, etc.).
3. **Miscellaneous**: This category covers a broader spectrum of classical programs (e.g., Dijkstra's shortest path, factorial, Fibonacci, Game of Life, image rotation, knapsack problem, N queens, palindrome, Huffman coding/encoding, Lempel-Ziv-Welch compression, etc.) using different techniques (e.g., dynamic programming, greedy algorithms, backtracking, divide and conquer, etc.), various programming styles (e.g., iterative, functional), and high-level Java APIs (e.g., streams, lambdas, fork-join, collections, etc.).

The categorization is for informative purposes to better organize and direct the focus of our benchmarks, ranging from low-level (compiler benchmarks) to high-level (API and Miscellaneous) benchmarks.

For this report we aggregated in total a number of **1112 benchmark runs**, including all three categories.

**The traffic during the first two days** after we launched the article **exploded**



# JMVs / JIT Compilers  from JDK 21

vs.

vs.

| OpenJDK | Oracle GraalVM | GraalVM CE |
|---|---|---|
| **C2 JIT** | **Graal JIT** | **Graal JIT** |

# Hardware/OS Configuration

x86_64

| Dell XPS 15 7590 | |
|---|---|
| CPU | Intel Core i7-9750H 6-Core |
| MEMORY | 32GB RAM |
| OS / kernel | Ubuntu 20.04 LTS / 5.15.0-101-generic |
| | disabled hyper-threading and turbo-boost |

arm64

| Apple MacBook Pro | |
|---|---|
| CPU | M1 Chip 10-Core, 16-Core Neural Engine |
| MEMORY | 32GB RAM |
| OS / kernel | macOS Ventura 13.6.1 / Darwin kernel version 23.4.0 |

# Software Configuration

**1112** benchmark runs (in the entire JDK suite)

**JMH v1.37** (5x10s warm-up it., 5x10s measurement it., 5 JVM forks [*] )

**OpenJDK 21** | **GraalVM CE 21+35.1** | **Oracle GraalVM 21+35.1**

Total execution time (per JDK suite/per JVM): ≈ **6.5 days**

[*] - each JVM fork starts with "*-Xms4g -Xmx4g -XX:+AlwaysPreTouch*"

# Benchmarks

02

# Infrastructure Baseline Benchmark

Used as a baseline to assess the infrastructure overheads.
Should be the same between the JVMs for a fair comparison.

References: [article][code source]

# Dead Argument Elimination Benchmark

Assesses the removal of arguments that are directly dead, as well as arguments that are passed into function calls as dead arguments of other functions.

References: [article][code source]

# Dead Local Allocation Store Benchmark

Checks how the compiler handles dead allocations.

Dead allocation is an allocation that is not used by subsequent instructions.

References: [article][code source]

# Lock Coarsening Benchmark

Tests how the compiler can effectively coarsen/merge adjacent locks.
Optimization useful to reduce the overhead of object locking/unlocking.

References: [article][code source]

# NPE Throw Benchmark

Tests the implicit vs explicit throw and catch of NPE in a hot loop.
The callee is never inlined into the caller.

References: [article][code source]

# Scalar Replacement Benchmark

Tests the ability of the compiler for perform escape analysis and scalar replacement.

References: [article][code source]

# String Concatenation Benchmark

Checks measuring the performance of various concatenation methods using different data types (e.g., String, int, float, char, long, double, boolean, Object): *StringBuilder*, *StringBuffer*, *String.concat()*, *plus* operator, *StringTemplate*

References: [article][code source]

# Enum Values Lookup Benchmark

Iterates through the *enum* values and returns the value that matches a lookup value. This pattern is commonly seen in business applications, where microservices with RESTful APIs defined in *OpenAPI/Swagger* use *enums*.

References: [article][code source]

# Conclusions

03

# Geometric Mean

$$\left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}} = \sqrt[n]{x_1 x_2 \dots x_n}$$

**"How to not lie with statistics: the correct way to summarize benchmark results"**

- Philip J Fleming, John J Wallace

# JIT Geometric Mean x86_64

**Note:** This is purely informative to have a high-level understanding of the overall benchmark scores.

| Rank | JIT | Normalized Geometric Mean | Number of Benchmarks | Unit |
|---|---|---|---|---|
| 1 | Oracle GraalVM | 0.77 | 1112 | ns/op |
| 2 | C2 | 1 | 1112 | ns/op |
| 3 | GraalVM CE | 1.03 | 1112 | ns/op |

*The first in the row is the fastest, and the last in the row is the slowest.*

# JIT Geometric Mean arm64

**Note:** This is purely informative to have a high-level understanding of the overall benchmark scores.

| Rank | JIT | Normalized Geometric Mean | Number of Benchmarks | Unit |
|---|---|---|---|---|
| 1 | Oracle GraalVM | 0.83 | 1112 | ns/op |
| 2 | C2 | 1 | 1112 | ns/op |
| 3 | GraalVM CE | 1.08 | 1112 | ns/op |

*The first in the row is the fastest, and the last in the row is the slowest.*

# Oracle GraalVM JIT

| Key Strengths |
| --- |
| ✓ improved partial escape analysis |
| ✓ more aggressive inlining heuristics (including polymorphic inlining, recursive method inlining, constructor inlining, etc.) |
| ✓ more compact TLAB allocation code for grouped allocations |
| ✓ extended vectorization support |
| ✓ loop unrolling |
| ✓ able to devirtualize and inline call sites up to four different targets |
| ✓ (in general) cleaner and more compact/efficient CPU assembly instructions |

# C2 JIT

| Key Strengths |
|---|
| ✓ vectorization support |
| ✓ extended intrinsics support |
| ✓ efficient exceptions handling mechanism when the same implicit exception is thrown multiple times in the hot path |
| ✓ loop unrolling |
| ✓ able to devirtualize and inline call sites up to two different targets |

# GraalVM CE JIT

| Key Strengths |
| --- |
| ✓ improved partial escape analysis |
| ✓ (in general) better inlining heuristics than C2 JIT |
| ✓ able to devirtualize and inline call sites with a higher number of targets(, even exceeding Oracle GraalVM JIT) |

# Challenges & Lessons Learned

04

1. **Microbenchmarking is not about numbers**, without a proper understanding of what happens, the benchmark has no value.

2. Microbenchmarking is **not always a good predictor** for large scale applications.

3. There are **differences between different architectures** (e.g., x86_64, arm64).

4. **Microbenchmarking for GC is (in general) misleading**. We initially had a few GC-targeted benchmarks, but we dropped them.

# Future Work

05

- Keep on enhancing the completeness of the benchmark suite.
- Release the report for the next JDK LTS release(s). We have already successfully done so for JDK 17 and JDK 21.
- Try to include the Falcon JIT Compiler (from Azul Prime VM).
- Potentially extend it to RISC-V architecture, besides x86_64 and arm64.
- **If you want to contribute to this project, please feel free to reach out to us.**

# Thank You

# Additional Resources

Code Source: [https://github.com/ionutbalosin/jvm-performance-benchmarks]

JDK 21 article: [https://ionutbalosin.com/2024/02/jvm-performance-comparison-for-jdk-21]

JDK 17 article : [https://ionutbalosin.com/2023/03/jvm-performance-comparison-for-jdk-17]

Skill Up Now: [https://ionutbalosin.com/training]