A journey to JVM Excellence: Navigating JDKs and JVMs for Peak Performance

Ionut Balosin

www.ionutbalosin.com

@ionutbalosin

ionutbalosin@mastodon.social

Florin Blanaru

@gigiblender

gigiblender@mastodon.online

Agenda

Introduction

01

Ionut Balosin

Software Architect @ Raiffeisen Bank International
Technical Trainer | Security Champion | Blogger | Speaker

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

Florin Blanaru

Senior Software Engineer @ Axelera AI
TornadoVM - ex contributor
Student of the year award from RISC-V foundation - 2019

Interested in

- Language Runtimes
- Compilers
- Performance Analysis & Tuning

https://ionutbalosin.com/2023/03/jvm-performance-comparison-for-jdk-17

JVM Performance Comparison for JDK 17

Content

Context

The current article describes a series of Java Virtual Machine (JVM) benchmarks targeting the Just-In-Time (JIT) Compilers to assess different JIT Compiler optimizations by following specific code patterns. At a first glance, even though some of these patterns might rarely appear directly in the user programs, they could occur after a few optimizations (e.g., inlining of high-level operations).

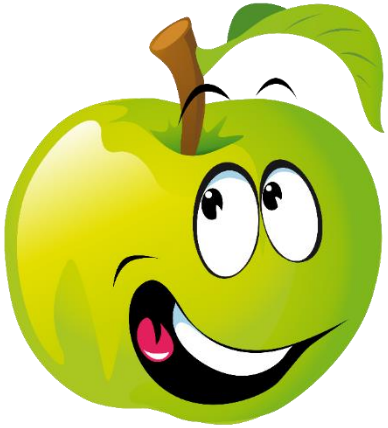
In addition, there is a small set of benchmarks (i.e., a macro category) covering larger programs (e.g., Fibonacci, Huffman coding/encoding, factorial, palindrome, etc.) using some high-level Java APIs (e.g., streams, lambdas, fork-join, etc.). Nevertheless, this is only complementary but not the main purpose of this work.

For a few benchmarks (i.e., the most representative, in our opinion) we provide an in-depth analysis (i.e., optimized generated assembly code, flame graphs, etc.), as well as the normalized geometric mean.

The list of included JIT compilers is:

- C2 (Server) JIT

JMVs / JIT Compilers from JDK 17



vs

vs

OpenJDK 17.0.6	GraalVM EE 22.3.0	GraalVM CE 22.3.0
C2 JIT	Graal JIT	Graal JIT

Note: Meanwhile, GraalVM EE was renamed to Oracle GraalVM

Configuration

Dell XPS 15 7590 (x86_64)	
CPU	Intel Core i7-9750H 6-Core
MEMORY	32GB RAM
OS / Kernel	Ubuntu 20.04 LTS

Apple MacBook Pro (arm64)	
CPU	M1 Chip 10-Core, 16-Core Neural Engine
MEMORY	32GB RAM
OS / Kernel	macOS Monterey 12.6.1

Benchmarking Tool	
JMH v1.36	5x10s warm-up iterations, 5x10s measurement iterations, 5 JVM forks

Note: please check jvm-performance-benchmarks GitHub repo for the full config

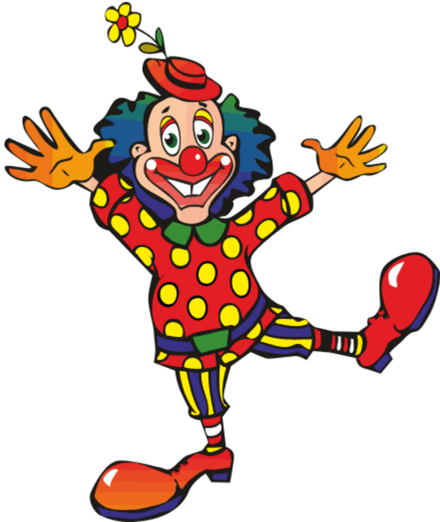
Benchmarks

02

Infrastructure Baseline Benchmark

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

References: [article][code source]

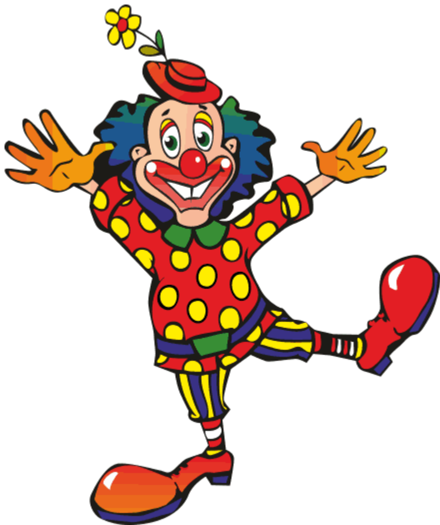


Enum Value Lookup Benchmark

Iterates through the enum values and returns the value that matches a lookup value

Pattern of seen in business applications where microservices RESTful APIs defined in OpenAPI/Swagger use enums

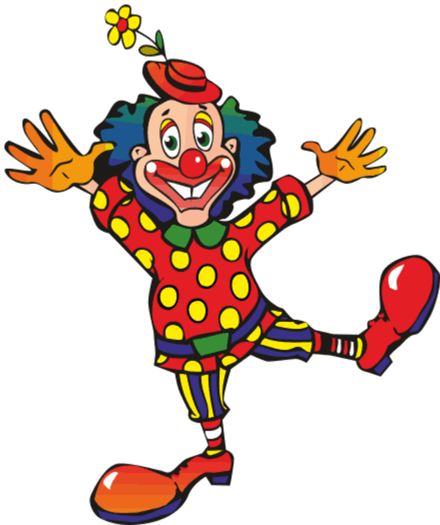
References: [article][code source]



Lock Coarsening Benchmark

Tests how the compiler can effectively coarsen/merge adjacent locks
Optimization useful to reduce the overhead of object locking/unlocking
Biased locking - used to optimize locking - is now proposed for deprecation

References: [article][code source]

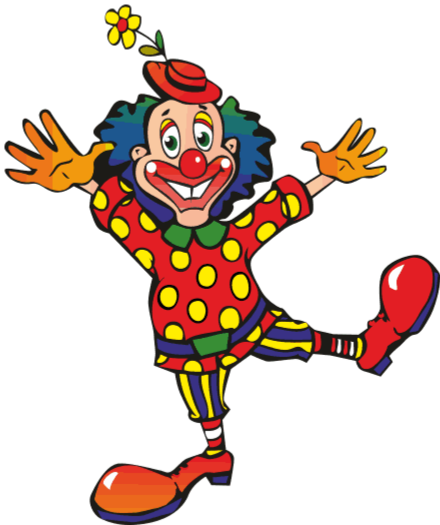


Dead Local Allocation Store Benchmark

Checks how the compiler handles dead allocations

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

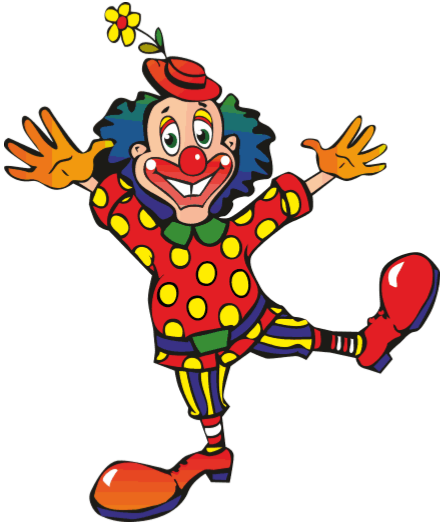
References: [article][code source]



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]

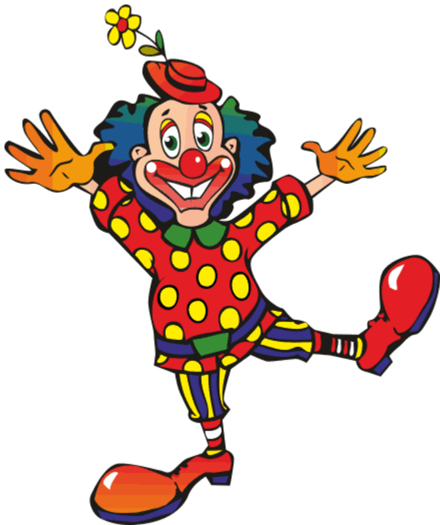


Recursive Method Call Benchmark

Tests the performance of recursive method calls in classes, interfaces and lambda functions

The ability to inline recursive calls is essential

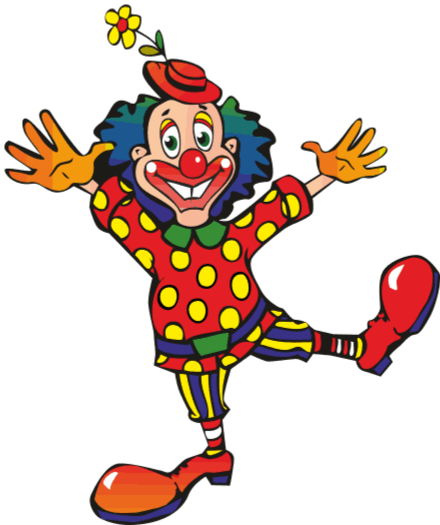
References: [article][code source]



Scalar Replacement Benchmark

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

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

JIT	Normalized Geometric Mean	Unit
GraalVM EE JIT	0.72	ns/op
C2 JIT	1	ns/op
GraalVM CE JIT	1.28	ns/op

Note: this is purely informative to have a high-level understanding of the overall benchmark scores (in total 273 benchmarks)

JIT Geometric Mean

arm64

JIT	Normalized Geometric Mean	Unit
GraalVM EE JIT	0.83	ns/op
C2 JIT	1	ns/op
GraalVM CE JIT	1.57	ns/op

Note: this is purely informative to have a high-level understanding of the overall benchmark scores (in total 273 benchmarks)

Application Developer Guidelines

	Graal EE JIT	C2 JIT
a lot of objects created	✓	
high degree of polymorphic calls	✓	
myriad of tiny nested/recursive calls	✓	
optimized exception handling [1]		✓
extended intrinsic set		✓

Note: please take these guidelines with precaution

[1] - C2 JIT optimizes exceptions that are frequently thrown (e.g., -XX:-OmitStackTraceInFastThrow)

Challenges & Lessons Learned

04

Microbenchmarking is not trivial

Microbenchmarking is not just about numbers; without a proper understanding of what is happening, the benchmark has no value

Microbenchmarking is not always a reliable predictor for large-scale applications

There might be differences between various architectures (e.g., x86_64, arm64)

Microbenchmarking for Garbage Collectors can be misleading

Future Work

05

Future work

At the moment, we are in the process of collecting results for **JDK 21**

We always strive to enhance the completeness of the benchmark suite by adding complementary benchmarks

In case you want to contribute to this project, feel free to reach out to us

Thank You

Resources

Code Source

https://github.com/ionutbalosin/jvm-performance-benchmarks

Article

https://ionutbalosin.com/2023/03/jvm-performance-comparison-for-jdk-17

Appendix

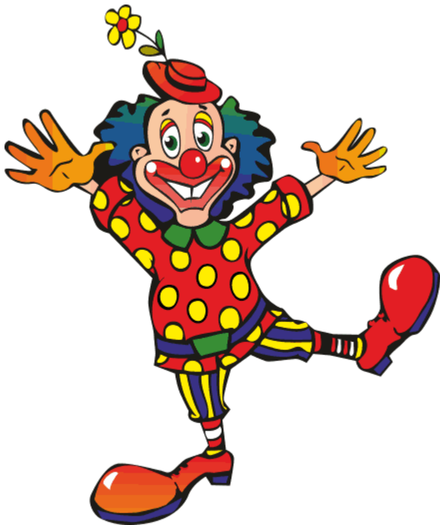
Mandelbrot Vector Api Benchmark

Tests the performance of Project Panama's Vector API for computing the Mandelbrot Set

Still an incubator module in the JDK

Subject to change between releases

References: [article][code source]



Megamorphic Method Call Benchmark

Compares virtual calls with different number of targets
Checks the performance of manually splitting the call sites into monomorphic call sites

References: [article][code source]