Prepared by: [A. Pramathi,A. Likitha nalini],11239a004,11239a001
Date: 2025-12-05

Improving Java Microservice Performance: Virtual Threads and GraalVM

Abstract with Keywords

Abstract

This article investigates contemporary approaches to improving performance and resource efficiency of Java-based microservices by combining two recent advances in the Java ecosystem: virtual threads (Project Loom) and the GraalVM toolchain. We present the research objectives, which are to compare concurrency models (traditional platform threads vs virtual threads), evaluate runtime options (HotSpot JVM vs GraalVM native image), and propose practical recommendations for deploying high-throughput, low-latency microservices. The methodology includes a literature-based survey of recent studies, a reproducible experimental setup using a simple HTTP microservice, and a performance evaluation across metrics such as throughput, latency (p50/p95), memory consumption, and startup time. Results summarized from recent literature and small-scale experiments show that virtual threads simplify concurrency and improve throughput for I/O-bound workloads, while GraalVM native images substantially reduce startup time and memory footprint at the cost of some peak throughput in certain scenarios. We conclude with best-practice guidelines and future research directions exploring hybrid deployments and energy-efficient JVM variants.

Keywords: Virtual Threads, Project Loom, GraalVM, Java Microservices, Concurrency, Performance

---

Introduction

Background

Java remains a dominant language for server-side applications, especially microservices. Recent platform advances—most notably the introduction of virtual threads (JEP 444, finalized in Java 21) and broader adoption of GraalVM tooling—offer new trade-offs for building cloud-native services. Virtual threads provide a lightweight concurrency model that decouples application-level threads from OS carrier threads, while GraalVM offers ahead-of-time compilation into native images and an optimizing runtime (the Graal compiler) that can reduce startup time and memory use.

Research problem and objectives

Prepared by: [A. Pramathi,A. Likitha nalini],11239a004,11239a001
Date: 2025-12-05

While both virtual threads and GraalVM promise operational benefits, it is unclear how they interact in practical microservice deployments and what combinations yield the best trade-offs for common cloud workloads. This study aims to:

1. Compare traditional platform-thread-based concurrency vs virtual threads for typical I/O-bound microservice patterns.

2. Evaluate runtime effects of HotSpot JVM vs GraalVM native images with respect to startup latency, memory footprint, and runtime throughput.

3. Provide actionable guidelines for developers choosing concurrency and runtime strategies for Java microservices.

Literature Survey

Recent work has concentrated on evaluating GraalVM's performance profile and the impact of virtual threads on concurrency simplification and scalability. Studies have reported that GraalVM (and graal-based JVM distributions) can reduce energy consumption and improve peak performance in some benchmarks, while other experimental artifacts provide long-term measurement datasets for the Graal compiler development. The finalization of virtual threads in JDK 21 triggered several analyses and tutorials demonstrating significant simplification of asynchronous code and promising throughput gains for I/O-heavy workloads. However, the literature also highlights trade-offs: GraalVM native images reduce startup and memory at the potential cost of some peak throughput and restrictions on dynamic features (e.g., reflection) unless explicitly configured. The research gap remains in systematically comparing the concurrency model (virtual vs platform threads) and runtime (HotSpot vs GraalVM native image) in a single, reproducible microservice context.

Methodology

Research design

This research follows a mixed approach: (1) a targeted literature synthesis of recent (last 3 years) studies and platform release notes, and (2) an empirical evaluation using an open, reproducible microservice benchmark.

Tools and environment

- Java Development Kit 21 or later (for virtual threads support)

- GraalVM (native-image tool) for AOT compilation

Prepared by: [A. Pramathi,A. Likitha nalini],11239a004,11239a001
Date: 2025-12-05

- Microservice: minimal HTTP server (using Java's `HttpServer` or a lightweight framework)
- Load generator: `wrk` or `hey`
- Measurements: throughput (requests/sec), latency distribution (p50, p95), memory (RSS), and startup time
- Containerization: Docker for packaging experiments

Experimental variables

- Concurrency model: platform threads (ThreadPoolExecutor) vs virtual threads (`Thread.ofVirtual().start()`/Executors.newVirtualThreadPerTaskExecutor())
- Runtime: HotSpot JVM (JDK 21) vs GraalVM native image
- Workload type: I/O-bound (simulated blocking calls) vs CPU-bound

---

Implementation

Microservice implementation overview

The microservice is intentionally minimal to reduce noise from frameworks. It exposes a single endpoint `/process` that simulates work:

- **I/O-bound variant:** performs a non-blocking simulated I/O by calling `Thread.sleep()` to emulate blocking I/O (or, preferably, uses real I/O such as reading from a local socket) and returns a JSON payload.
- **CPU-bound variant:** performs a small CPU-bound computation (e.g., computing a cryptographic hash over a small buffer).

Two builds are produced:

1. **HotSpot build** — compiled and run on JDK 21 with either a platform-thread executor or a virtual-thread executor.
2. **GraalVM native-image** — the same code compiled to a native executable with required reflection/configuration options.

Prepared by: [A. Pramathi,A. Likitha nalini],11239a004,11239a001
Date: 2025-12-05

Flowchart (conceptual)

1. Client -> HTTP /process -> Dispatcher (virtual or platform threads) -> Handler (I/O or CPU work) -> Response

(Include a simple flowchart in the full paper when preparing final PDF.)

Results

Note: The numbers below summarize trends observed across multiple recent studies and small-scale reproducible tests rather than a single large-scale experiment. They are presented to illustrate typical behavior reported in the literature and our sample runs.

Observed trends

- **Throughput & Latency (I/O-bound):** Virtual threads often achieve higher throughput and lower p95 latency compared to platform threads under high concurrent load, because virtual threads multiplex more efficiently over OS carriers for blocking operations.

- **CPU-bound workloads:** Platform threads or tuned thread pools remain competitive for CPU-heavy tasks because scheduling overhead and contention become dominant.

- **Startup time & memory:** GraalVM native images dramatically reduce startup time and baseline memory usage, which benefits short-lived services or serverless deployments. However, native images may require additional configuration for reflection and dynamic proxies.

- **Operational complexity:** Using virtual threads simplifies concurrency code (fewer callbacks, easier reasoning). GraalVM adds complexity in build-time configuration but yields operational benefits in resource-constrained environments.

Example table (suggested for final article)

Configuration	Workload Type	Throughput (rel.)	p95 Latency (rel.)	Startup Time	Memory Footprint
HotSpot + Platform Threads	I/O-bound	1.0 (baseline)	1.0	medium	medium

Prepared by: [A. Pramathi,A. Likitha nalini],11239a004,11239a001
Date: 2025-12-05

HotSpot + Virtual Threads	I/O-bound	1.4x	0.8x	medium	medium
GraalVM Native + Platform	I/O-bound	0.9x	0.9x	0.2x	0.4x
GraalVM Native + Virtual	I/O-bound	1.2x	0.7x	0.2x	0.4x

(rel. = relative to baseline in typical runs — illustrative only)

Conclusion

This paper examined how virtual threads and GraalVM affect Java microservice performance. Virtual threads simplify concurrent programming and provide significant benefits for I/O-bound services. GraalVM native images reduce startup time and memory footprint, making them attractive for serverless or containerized deployments. A hybrid approach—using virtual threads on HotSpot for heavy throughput services and GraalVM native images for latency-sensitive or resource-constrained functions—may offer practical benefits.

Future scope

- Large-scale, multi-node benchmarking in cloud environments (Kubernetes)
- Energy efficiency and cost-per-request analyses across JVM variants
- Tooling improvements for GraalVM reflection/config generation to ease migration

---

References (recent, suggested for the last 3 years)

1. Oracle. *Java SE 21 Release Notes* (2023).
2. Oracle / Java Magazine. *Exploring the design of Java's new virtual threads* (2023).
3. Vergilio, T.G. et al. *Comparative Performance and Energy Efficiency Analysis of JVM Variants and GraalVM in Java Applications* (2023).

4. Bulej, L. et al. *GraalVM Compiler Benchmark Results Dataset* (2023).

5. Additional resources: GraalVM academic publications and community artifacts (2022–2025).
