National College of Ireland

# Benchmarking the Performance of Java Virtual Threads in High-Throughput Workloads

MSc Research Project
Cloud Computing


Vishesh Pandita
Student ID: x23184531

School of Computing
National College of Ireland

Supervisor: Yasantha Samarawickrama

**National College of Ireland**
**Project Submission Sheet**
**School of Computing**

| | |
|---|---|
| **Student Name:** | Vishesh Pandita |
| **Student ID:** | x23184531 |
| **Programme:** | Cloud Computing |
| **Year:** | 2024 |
| **Module:** | MSc Research Project |
| **Supervisor:** | Yasantha Samarawickrama |
| **Submission Due Date:** | 20/12/2024 |
| **Project Title:** | Benchmarking the Performance of Java Virtual Threads in High-Throughput Workloads |
| **Word Count:** | 7414 |
| **Page Count:** | 20 |

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

**ALL** internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

| | |
|---|---|
| **Signature:** | Vishesh Pandita |
| **Date:** | 12th December 2024 |

## PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

| | |
|---|---|
| Attach a completed copy of this sheet to each project (including multiple copies). | ☐ |
| **Attach a Moodle submission receipt of the online project submission**, to each project (including multiple copies). | ☐ |
| **You must ensure that you retain a HARD COPY of the project**, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer. | ☐ |

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

| **Office Use Only** | |
|---|---|
| Signature: | |
| Date: | |
| Penalty Applied (if applicable): | |

# Benchmarking the Performance of Java Virtual Threads in High-Throughput Workloads

Vishesh Pandita
x23184531

**Abstract**

This research evaluates the performance of Java Virtual Threads against traditional platform threads in high-throughput workloads and benchmarks the process. Java Virtual Threads were introduced as part of Project Loom in JDK 21 to enable optimal hardware utilization for applications written in a thread-per-request model. Virtual Threads, introduced in JDK 21 in 2023, required in-depth testing to evaluate their contributions to Java's concurrent and asynchronous programming. In this research, attention was given to creating a fair test environment that will not influence the results of benchmarking Java applications under high workloads. The benchmarking was done on both CPU and I/O intensive workloads to test and evaluate the performance of virtual threads compared to platform threads.
In CPU-intensive workloads, Java applications with traditional and virtual threads performed similarly. However, when it comes to I/O intensive workloads, Java virtual thread applications perform better than Java platform thread applications. Applications using Java virtual threads used less memory and had better latency compared to traditional platform threads. Throughput was increased by 60.79% while Latency was decreased by 28.8% while comparing virtual threads with platform threads. Memory usage and CPU utilization also saw an improvement of 36.36% and 14.29% respectively.
Both virtual and platform threads have their strengths and weaknesses. It can not be said that virtual threads are better than platform threads. However, it highly depends on the work scenarios and which metrics one needs to prioritize. Virtual threads performed better in blocking operations but Platform threads are more stable.

## 1 Introduction

With every day the complexity and scale of applications are increasing. This results in increasing the need for concurrency and parallelism to support the growing demand and limited resources. Efficient use of concurrency and parallelism is essential to utilize the limited resources efficiently and to their full potential Sodian et al. (2022). Modern applications are designed to handle millions of requests simultaneously. Examples of such applications are web servers and high-throughput data processing pipelines. These applications rely on efficient concurrency models to handle high-throughput workloads distributively. Java, a leading programming language that is used to create these kinds of applications relied heavily on platform threads till now. This model for parallelism provided a robust but resource-intensive method of mapping directly to Operating System

threads. This limitation is highly evident in blocking operations and high concurrency scenarios which has led to a need to innovate Goetz (2006).
With the introduction of Java Virtual Threads in JDK 21, as part of Project Loom, It brings a significant change in the imperative style asynchronous programming model Ron Pressler (2023). As Figure 1 shows, Virtual Threads are lightweight compared to Platform Threads as they are decoupled from OS threads. The main aim of introducing virtual threads is to improve the Thread-per-request programming model in Java. Although theoretically there is a lot of potential in Java Virtual Threads, robust benchmarking to evaluate the performance of Virtual Threads in high throughput workloads is missing. This research addresses this gap by benchmarking Virtual Threads and Platform Threads in both CPU and I/O intensive workloads. This offers insights into their performance in real-world applications.

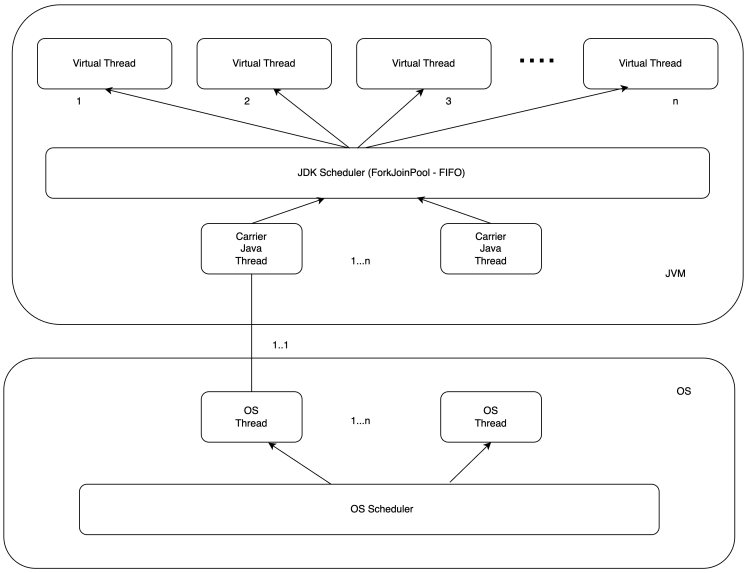


Figure 1: Java Virtual Threads

## 1.1 Background and Motivation

Concurrency has been an essential tool in the field of computing which enabled application to handle multiple tasks in parallel. Java from its inception has implemented concurrency through Platform Threads which matured over time Chen et al. (2010). As Rosà et al. (2023) highlights in paper, Platform Threads have incured high memory overhead and context-switching has been expensive. Which resulted in Java being less suitable for workloads requiring massive concurrency. These issues are often highlighted in I/O bound application in which thread spends most of its time staying idle while waiting for external service such as network or disk operation as described in Navarro et al. (2023).
From its inception Java has introduced parallel programming. With imperative-style of parallel programming supported by Java Thread class it has been in forefront of parallel programming. This programming paradigm implies to the composition of a sequence of code that changes the state of the program. This allowed developers to interact with parallel programming with the pattern they are already familiar with. However this was different from imperative style sequential code in which developers do not have to worry

about thread lifecycle, synchronization or other complexities that are introduced in multi-threading Friesen and Pal (2015).
With the release of Java 8, java released new APIs to support functional-style parallel programming. These APIs provided a intuitive way to write parallel code in functional style. Carvalho and Fialho (2023) explores with the introduction of lambda expressions and Stream API concurrent programming became simplified. Java 8 also released CompletableFuture interface which made functional-style asynchronous programming much more intuitive. These new APIs allowed developers to write non-blocking and asynchronous code easily with declarative methods.
With the introduction of Virtual Threads, Project Loom tried to provide an answer for this problem. New et al. (n.d.) book talks about unlike Platform Threads, Virtual Threads are managed by JVM and this allows creation of millions of Virtual Threads which can coexist efficiently. Virtual Threads simplify writing Thread-per-task model concurrent code without worrying about the OS thread limitations.
Java's concurrency and asynchronous programming paradigms have matured through many iterations which enable developers to choose the model they are comfortable in and that suits their system needs Subramaniam (2014). While functional programming leads to concise and high-level code it often makes less intuitive sense compared to imperative style programming model. The imperative-style programming model with the introduction of Virtual Threads provide a familiar way of coding highly concurrent applications that most developers are familiar with. But understanding the strengths and weaknesses of each paradigm is crucial to make a educated choice of what is right fit of their needs.
The motivation for this study comes from the potential of Virtual Threads to change the concurrency model of Java and lack of research in the field of performance of Virtual Threads in real world applications. This research's aim is to do a rigorous evaluation on performance of Virtual Threads which will help inform developers, architects, and academics about the real world trade-offs between Virtual Threads and Platform Threads. The decision of what programming model to choose should depend on concert knowledge and data, but right now developers are making this choice based on theoretical knowledge. Those choosing this programming model should be aware of the impact of Virtual Threads on the performance and scalability of their systems and this research aims to achieve just that.

## 1.2 Research Questions and Objectives

Keeping the above research problem in mind, the primary research question for this study is: **How do Java Virtual Threads perform compared to traditional Platform Threads in terms of scalability, resource utilization, and latency under high-throughput workloads?**
To address this research question specific objectives were defined:

- Evaluate the performance of Virtual Threads in CPU-bound and I/O-bound workloads.
- Benchmark memory consumption and latency of Virtual Threads compared to Platform Threads.
- Identify scenarios where Virtual Threads excel and situations where Platform Threads remain advantageous.

- Provide actionable insights into the trade-offs and best practices for adopting Virtual Threads in production environments.

This study focuses on CPU-intensive and I/O-intensive workloads to simulate real world applications. Benchmarking is done under similar conditions for Virtual Threads and Platform Threads to ensure fairness and reproducibility.

### 1.3 Report Structure

The rest of the research proposal is arranged as follows: **Section 2, Related Work**, introduces previous state-of-the-art work in the field of Java concurrency models. This will include Platform Threads and Virtual Threads and will highlight their contributions in high-throughput workloads with their limitations also. This section also includes main objectives and findings of previous work which will set foundation for this study. **Section 3, Methodology**, outlines the research approach. This section details the design of benchmarks, workload simulations and metrics used to compare Virtual Threads and Platform Threads. **Section 4, Design Specification**, outlines the technical framework. This will also explain about configurations that were done for controlled and fair testing of applications. **Section 5, Implementation**, provides a detailed overview on how benchmarking was implemented. This will include all the tools, libraries and code structure used to create a fair benchmarking environment. **Section 6, Evaluation**, this section presents the benchmarking results and analysis on the performance of Virtual Threads and Platform Threads across CPU-intensive and I/O-intensive workloads. This section also discusses the trade-offs between the two models. Finally, **Section 7, Conclusion and Future Work**, summarizes all the findings in the research and discuss on the implications for both academic and industrial benefits of this research. It also identifies the potential areas for further research to enhance Java concurrency models.

## 2 Related Work

With the introduction of Java Virtual Threads in JDK 21 as part of Project Loom Ron Pressler (2023), the Java concurrency model has seen a significant evolution. It offers a lightweight threading model which is managed by the JVM itself and sits of top of platform threads. Navarro et al. (2023) explains in the paper that it is much cheaper to create, destroy and switch from than platform threads which makes it much better at handling high-concurrency scenarios. To better understand the context and reasons behind this evolution it is essential to understand the broader aspect of Java concurrency and related studies that exist on traditional threading models, lightweight thread alternatives, and performance benchmarking. This section consists of prior work in these areas, and identifies gaps and provides a foundation for the present research.

### 2.1 Traditional Concurrency in Java: Platform Threads

From the inception of Java, concurrency has relied on Platform Threads, which map directly to Operating System threads. With each Platform Thread involved, there is a significant amount of memory associated with it in the stack and context switching is costly because it is managed by the Operating System Goetz (2006). Even with these issues, Platform Threads are considered stable and are known for their robustness and

ease of use. They have proved to be essential with the modern concurrency demands. However, as workloads grew with time, their limitations became a blocking point.
Numerous studies have highlighted the scalability challenges of Platform Threads in high-throughput applications. Reinders (2007) highlights this issues of Platform Threads that when application requires thousands or millions of concurrent tasks, the memory overhead and context-switching costs that are associated with Platform Threads makes it less suitable for the applications. Similarly, King (2014) focuses on thread starvation and contention that become significant issues in highly concurrent environments. This is even more evident in thread blocks on I/O operations. These findings have motivated the search for an alternative method which is capable of better resource utilization.

## 2.2 Historical Context of Java Concurrency

Traditionally parallel programming in Java was achieved by the use of Java Thread class which used Platform Threads. Various improvements were done upon this paradigm which makes it more robust and stable. In Java 5, java.util.concurrent package was introduced which brought functionalities like thread pools, ExecutorService and other synchronization mechanisms in the picture Horstmann (2007). This really improved the experience of developers while interacting with concurrency. Chen et al. (2011) is a paper that explores these advantages of thread pooling and ExecutorService. While the implementations of Platform Threads improved with time, using platform threads was still a challenge due to its high resource utilization which was even more evident in high-concurrency scenarios. Thread starvation was still a major issue with platform threads in asynchronous programming. Ahmad (2016) is a paper which explores the limitations of threads pool. This paper aimd to find the most optimal thread pool size based on the application which is still considered a difficult task.
With Java 8 main focus was given to functional-style parallel and asynchronous programming. It added new functionalities like Stream API and CompleteableFuture interface. Hagl (n.d.) is a paper that examines completeable future and asynchronous pipeline. It explores "Future" and "Chainable Fututre" and Java 8 features like Executors, CompletableFutures, Streams API, and Lambda Expressions. While these changes made functional-style concurrent programming achievable in java, the issue of platform threads was still a bottleneck for high-concurrency scenarios.

## 2.3 The Case for Lightweight Threads

The concept of lightweight threads or "green threads" is not noble to Java. It has been explored in several programming languages as an alternative to OS threads. Begel et al. (1999) is a paper which explores the idea with lightweight thread is that it is managed by the runtime environment rather than the OS. This enables the programming environment to do more efficient context switching and lower the memory overhead. Erlang has developed these lightweight processes to achieve massive concurrency. Similarly, one such notable implementation is coroutines in Go programming language Togashi and Klyuev (2014). They allow millions of concurrent tasks to run with minimal overhead. Goroutines highly leverage cooperative scheduling.
In Java ecosystem, continuous efforts have been put to improve the concurrency model with various new API's like CompletableFuture and reactive libraries like RxJava Ponge et al. (2021). These models addressed some scalability issues but introduced various

complexities with them. These asynchronous models often require callback-based programming approach which made code harder to read and maintain. With the advent of Project Loom java has finally stepped foot in implementing lightweight concurrency model. Virtual Threads aims to address the issues of previous concurrency models by implementing Lightweight threads with familiar synchronous programming model.

## 2.4 Project Loom and Virtual Threads

Project Loom which is initiated by Oracle has introduced Virtual Threads as a lightweight threading model in Java. Virtual Threads are completely managed by Java Virtual Machine and interact with OS threads by the use of Platform Threads. This has enabled creation of millions of threads which can exists efficiently in JVM. By allowing each task to be represented as an individual virtual thread, thread-per-request programming model has largely simplified this concurrent model, particularly in server-side applications.
Pufek et al. (2020) describes the design of Virtual Threads in Java. Emphasis was put in Virtual Threads being compatible with existing Java APIs and the ability to run blocking I/O operations efficiently. Virtual Threads provide a more intuitive way of writing efficient blocking code in sequential programming paradigm. It is different from reactive programming model which is based on callback approach which is less intuitive and harder to debug. With these theoretical advantages, it becomes a lucrative choice to implement blocking high concurrency applications but studies evaluating the performance of Virtual Threads in real-world remain limited.

## 2.5 Benchmarking and Performance Analysis

To evaluate concurrency models proper benchmarking is critical. Several tools and frameworks have been developed to measure the performance of Java applications. One such notable tool is Apache Jmeter, it typically puts load on the application and asses metrics such as throughput, latency, error rates etc. Agnihotri and Phalnikar (2018) is a conference paper that explores efficient creation of performance testing suits in Apache Jmeter. Lenka et al. (2018) explored the importance of designing fair benchmarking that accounts for the factors such as thread scheduling, hardware variability, and workload characteristics. With there study on Platform Threads, they noted that CPU-bound workload generally don't benefit from increasing concurrency due to limited parallelism available in processors. It can also suffer if the concurrency is increased as more overhead work have to be managed. But with I/O-bound workloads the performance improves significantly though efficient thread management.

## 2.6 Virtual Threads in Practice

Since Java has introduced Virtual Threads there has been significant interest by both academic and industrial communities. There are several case studies that have shows the potential of Virtual Threads to simplify concurrency in server-side applications. Haneklint and Joo (2023) compared the performance of Java Virtual Threads with reactive programming in java in a microservice architecture. This research showed Virtual Threads achieved similar scalability compared to reactive streams with significantly simplifying the code complexity.
However most studies focus on specific use cases and do not take into account diverse

workloads. This leads to bias in highlighting the features of Virtual Threads but does not highlight the demerits of Virtual Threads. There have been several concerns about the stability and maturity of Virtual Threads with some researches pointing out the potential challenges in adopting them for production applications. This research aims to address this issue by performing benchmarking in a stable and controlled environment and in diverse workloads.

### 2.7 Gaps in the Literature

Existing studies provide detailed insights into design and benefits of Virtual Threads but several gaps remain in the literature of Java Virtual Threads. Most studies focus on benchmarking specific use cases which leave a gap in evaluation across CPU-bound and I/O-bound workloads. Study on trade-offs between Virtual Threads and Platform Threads have been limited in terms of resource utilization and latency. There is a lack of studies that highlight the challenges and best practices for adoption of Virtual Threads in production environment. By addressing these gaps this research contributes in the existing literature. This research provide empirical evidence on performance of Virtual Threads under diverse workloads and offers actionable insights for its adoption.

## 3 Methodology

This research takes a systematic approach for benchmarking for Virtual Threads and Platform Threads in high-throughput workloads. The benchmarking is done keeping in mind the alignment of research objectives. The methodology broadly includes contextual analysis, implementation, workload design and testing. The methods chosen are done by keeping in mind the research questions and objectives and to ensure that the outcome is robust and insightful for comparing performance of Virtual Threads and Platform Threads.

### 3.1 Requirements and Contextual Analysis

The primary objective of this research was to benchmark and analyze the performance of Java Virtual Threads against traditional Platform Threads. To achieve this objective, the benchmarking was done under two different types of workloads, CPU-intensive and I/O-intensive workloads. To do a robust benchmarking which generates meaningful results deep understanding of Java's concurrency models was required. A focus on metrics that highlighted the important aspects in high-throughput scenarios was also required.
**Key Considerations:**

- **Threading Models**: Traditional Platform Threads are well-established and stable. But Virtual Threads are relatively new and were introduced in JDK 21. To find out the strengths and weaknesses both threading models needed to be thoroughly and fairly tested. It was essential to use JDK 21 or above for testing because even though Virtual Threads were available to use from JDK 19, they were in beta phase and release only in JDK 21 for production use.

- **Workload Types**: CPU-intensive workloads generally test raw computational efficiency. Adding more concurrency generally does not lead to a good result. Some-

times adding more concurrency can degrade the results which needed to be thoroughly tested. While on the other hand I/O-intensive workloads shows the handling of blocking operations which greatly benefits from increasing concurrency. Theoretically this is the case in which Virtual Threads should shine.

- **Benchmark Metrics**: Metrics such as throughput, latency, CPU utilization and memory usage are calculated. The choice to compare these metrics was done because there are some of the most important metrics that dictate the scalability of an application. Throughput and latency are the most crustal factors that define the speed of the application and its response. While CPU utilization and memory usage are the most important factors that indicate how efficient a system is. In an ideal system high throughput and CPU utilization is desired while low latency and memory usage is desired.

## 3.2 Data Gathering and Workload Design

Two types of workloads were designed to test the threading models:

**CPU-Bound Workload**

- **Objective**: To measure the performance of Virtual and Platform Threads in computationally intensive tasks.
- **Design**: Task include calculation of Prime numbers from lower limit to upper limit. Arguments for lower and upper limits are received from request.
- **Justification**: CPU-bound tasks include processor cores. Testing the threads ability to efficiently use hardware is essential. CPU-bound are processor intensive tasks and increasing concurrency does not generally increase efficiency. But if concurrency is increased it can affect negatively in the performance of the application. It is important to find out what will the negative effect with Virtual and Platform Threads.

**I/O-Bound Workload**

- **Objective**: To measure the performance of Virtual and Platform Threads in tasks involving significant waiting periods.
- **Design**: Simulated blocking operation by creating two java applications. One application GETs the request and fetches the results from other java application which take a specific blocking time and produces the response.
- **Justification**: This design creates ideal I/O-bound tasks which evaluate how threads manage the blocking operations and scalability under high concurrency.

## 3.3 Implementation

The implementation involves developing four Java applications to simulate different workloads and trading model combinations:

- **Platform Threads - CPU-Bound Workload**

- **Platform Threads - I/O-Bound Workload**
- **Virtual Threads - CPU-Bound Workload**
- **Virtual Threads - I/O-Bound Workload**

**Development Setup**:

- **Programming Language**: Java 21 was used because it comes with Virtual Threads enabled.
- **Development Tools**L IntelliJ IDEA and Maven was used for development and dependency management.
- **Frameworks**: The java.util.concurrent package for thread management and basic I/O operations.

Each application was designed to perform a fixed number of tasks, which makes it consistent for using both Virtual Threads and Platform Threads and compare the outcomes.

## 3.4 Testing and Evaluation

Testing was focused on high-concurrency scenarios using simulated workloads to evaluate scalability and efficiency.
**Tools Used**:

- **Apache JMeter**: Jmeter was used to generate concurrent requests. This simulates real-world high-throughput conditions.
- **VisualVM**: VisualVM was use to profile JVM resource utilization. The profiling includes thread activity, CPU utilization and memory usage.
- **AWS EC2 Instances**: Deployed on t2.micro and t5.large instance to ensure consistent computing environments.

**Test Configuration**:

- **Concurrency Levels**: Requests from 100 concurrent users for 10 minutes with 1 minute of ramp-up period.
- **Repetition**: Each test was conducted three times. The results with best-represented values are selected. Results with network errors were ignored and errors related to threads and application are highlighted.
- **Metrics Captured**:
  - Throughput (request per second)
  - Latency (average time per request)
  - CPU usage (% utilization)
  - Memory usage

**Warm-Up Runs**: Before the final testing, each application underwent a warm-up phase of 10 concurrent users for 60 seconds to allow the JVM to optimize code execution.

## 3.5 Data Cleaning and Validation

After the benchmarking was done data was collected from Apache JMeter and VisualVM. The data was analyzed for inconsistencies. Results that were outliers due to external factors like AWS etc were discarded. Requests that were failed or connection error occurred during load testing due to limited number of TCP ports were discarded. Cleaned data was then organized and converted to CSV files for further analysis using statistical tools.

## 3.6 Limitations

Despite creating a robust benchmarking environment this methodology has the following limitations:

- **Hardware Dependency**: Benchmarks done on AWS EC2 instances may not reflect the same performance on other hardware configurations.
- **Scenario Specificity**: Research was done on CPU-bound and I/O-bound workloads which is sufficient to demonstrate the comparison but does not cover all the real-world use cases. Different complex systems have different levels of CPU and I/O bound operations and readers have to make an educated choice based on the results which concurrency method is more beneficial for them.
- **New Technology Maturity**: Virutal Threads are relatively new and have number of undiscovered performance and stability issues.

This methodology for the research integrates workload design, implementation, testing and detailed analysis to benchmark the performance of Java Virtual Threads and Platform Threads. This is done by leveraging robust tools like Apache Jmeter and VisualVM, and creating a controlled environment on AWS EC2. This research provides valuable insights into the threading models trade-offs. This study will be critical for considering adoption of Java Virtual Threads in high-throughput Java applications.

# 4 Design Specification

This section of the research outlines the framework, techniques, and requirements for benchmarking Java Virtual Threads and Platform Threads. The architecture and implementation choices have been made such that it give a fair, robust, and reproducible comparison under high-throughput workloads. This section is structured intro three parts: the architecture and tools used, the design of workloads, and the associated requirements.

## 4.1 Architecture and Tools

The architecture employed for this research is modular in nature which ensures flexibility and efficiency in benchmarking the threading models. Each aspect is isolated and collects independent results to ensure minimal interference between components.

### 4.1.1 Threading Model Implementation

This research is done upon two threading models - Platform Threads and Virtual Threads. Applications implementing Platform Threads and Virtual Threads were implemented standalone java applications to ensure robust results. Platform threads utilize the traditional thread-per-request model where threads are mapped directly to operating system threads. Virtual Threads leverage Java's lightweight user-mode threads introduced in JDK 21. They are managed by the JVM and allows thousands of threads to operate concurrently with minimal overhead.

### 4.1.2 Application Modules

- **Workload Execution**: Apache Jmeter is used to simulate high-concurrency requests to stress-test the applications.
- **Request Handling**: Java applications are made using Spring Boot 3.2 and Java 21. The application manages the thread allocation and ensures efficient processing under various concurrency levels. APIs used for Virtual Threads and Platform Threads is java.util.concurrent.
- **Performance Monitoring**: Performance monitoring is essential to record CPU and memory utilization. VisualVM is used to profile the JVM.

## 4.2 Workload Design

To study the performance of threading models two types of workloads were designed: CPU-bound and I/O-bound. Focus while choosing these workloads was given to close depiction to real world scenarios and to highlight the performance differences between Virtual and Platform Threads.

### 4.2.1 CPU-Bound Workload

The objective to create this workload was to measure raw computational efficiency under high-concurrency scenarios. The workload is designed in such a way that it computes all the prime numbers between the given range. The range is decided by the incoming request which was load coming from Apache Jmeter in this case. The implementation was done using ExecutorService API of Java concurrent package. The tasks are evenly distributed by the ExecutorService and couple with CPU cores. These tasks test how efficiently each threading model utilize the hardware's computational resources as well as what drop in performance is observed when concurrency is increased but tasks are CPU bound.

### 4.2.2 I/O-Bound Workload

The objective to create this workload was to measure the performance of managing blocking operations by Virtual and Platform Threads in high-concurrency scenarios. This workload receives a request and calls a external service which has a fixed delay so the thread is in waiting state and it is a blocking operation. Concurrent requests are sent by Apache Jmeter for load testing. The implementation is done using ExecutorService API of Java concurrent package. Attention was given to distribute tasks equally with each

thread. This type of blocking operation highlighted the lightweight threading capabilities of Virtual Threads compared to Platform Threads.

# 5 Implementation

For the implementation of a robust, controlled environment to perform benchmarking on the performance of Java Virtual Threads and Platform Threads under high-throughput workloads a number of individual components with specific tasks were made. This section explains the development process of these components with the setup of the applications, workload design, deployment environment, and monitoring mechanisms as shown in Figure 2.

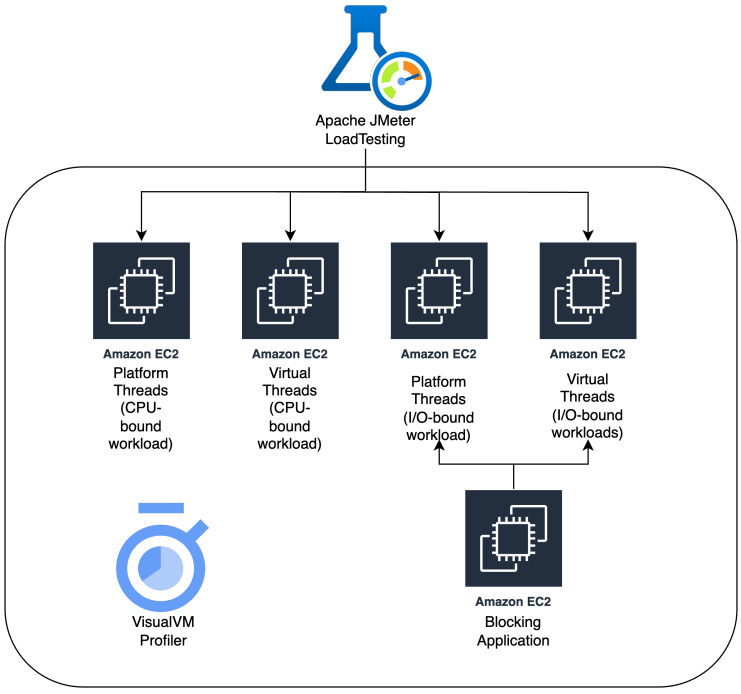


Figure 2: System Design

## 5.1 Development of Applications

Five applications in total were implemented for testing of Virtual Threads and Platform Threads. This design ensures the elimination of biases from the benchmarking process.

### 5.1.1 CPU-Bound Application with Platform Threads

The objective to create this application is to simulate computationally intensive tasks using Platform Threads in Java. The implementation of this application is done using Spring Boot 2.3 and JDK 21. Task involved is finding all the prime numbers between a given range. This is implemented with java.util.concurrent package in Java. Each thread is given the task to detect if the number is prime or not. This compute intensive task makes sure that the thread is busy and not idle.

### 5.1.2 CPU-Bound Application with Virtual Threads

The objective of creating this application is to compare this with CPU-Bound application with Platform Threads. The tasks it does is same as the previous application as we need to benchmark the performance of Threads. But the difference is implementation of that is done with Virtual Threads instead of Platform Threads. Everything else is kept intentionally to remove any biases from the results.

### 5.1.3 Blocking Application

The main objective of this application is to get a request and create a delay of 100 milliseconds and return the response. This application is also created using Spring Boot 2.3 and Java 21. The delay is created by making the thread sleep for 100 millisecond before sending the response. This application is essential to benchmark I/O-Bound Applications.

### 5.1.4 I/O-Bound Application with Platform Threads

The objective of this application is to simulate blocking operations which are I/O-bound in nature. The implementation of this application is done using Spring Boot 3.2 and Java 21. Task involved handling request by each thread which is blocking I/O in nature. This application gets the request and makes a request to Blocking Application stated above and waits for the response. Implementation is done such that each thread handles a single request. This request will always be blocked for at least 100 milliseconds as that is the blocking time for Blocking Application. The implementation is done using java.util.concurrent and java.util.stream packages in Java. It uses platform threads for every operations.

### 5.1.5 CPU-Bound Application with Virtual Threads

The objective of this application is same as the one with Platform Threads but with using Virtual Threads. The implementation is done using Spring Boot 3.2, Java 21 and Virtual Threads. Tasks involved is same as the Platform Thread application to avoid any biases for the benchmarking. The implementation is also done using similar packages but uses Executors.newVirtualThreadPerTaskExecutor() rather than Executors.newFixedThreadPool() which was done in Platform Thread application.

## 5.2 Workload Simulation

Workload simulation is done using Apache Jmeter. Two types of Test Plans were created, one for CPU-Bound applications and one for I/O-Bound applications. In the Thread Group of Jmeter, Number of Threads is set to 100. Ramp-up period is set to 60 seconds and total duration for each tests is 10 minutes. Then stress testing is done on each application while recording the results. It collected results like total number of requests processed, throughput, average/99th/95th percentile latencies, response time graphs etc. At the same time Profiling of the application was done using VisualVM which tracked activities like thread activity, heap usage, CPU usage while the stress-testing was going on.

## 5.3 Testing Environment

The applications were converted into JAR files with the help of Maven. These files were then deployed to AWS EC2 instances. To create a uniform testing environment which ensures consistency and reproducibility same configurations of AWS EC2 instances were used. AWS EC2 configurations that were used were of t3.xlarge instant type with 4 vCPUs, 16 GB RAM and Ubuntu 22.04 operating system. JDK 21 was used for all the benchmarking.

## 5.4 Benchmarking Process

For the benchmarking process, first came the warm-up phase where each application underwent a load of 10 threads for 5 minutes to allow the JVM's Just-In-Time compiler to optimize code execution. Then in the execution phase the full load testing was done with 100 threads for 10 minutes and all the metrics were measured. Each test was repeated three times and average was taken to minimize anomalies.

# 6 Evaluation

This section shows the benchmarking results for the comparisons of Java Virtual Threads and Platform Threads. The primary metrics evaluated in this study were throughput, latency, memory usage and CPU utilization. The results were achieved in a robust and bias-free benchmarking environment. The results are combination of multiple iterations of testing done in the benchmarking environment. The results of the benchmarks are presented in a comparative way between Platform Threads and Virtual Threads in each workloads.

## 6.1 Performance metrics 1: Throughput

Throughput is a very essential metic for any web application. It represents the number of requests that are processed per second. With the help of Apache Jmeter, the throughput of the benchmarks were noted. As shown in Figure 3, For CPU intensive workloads, while using the traditional Platform Threads the throughput came out to be 8204.1 req/sec. While using the Virtual Threads the throughput decreased 0.63% to 8153.2 req/sec. But for the I/O intensive workloads the throughput for Platform Threads came out to be 5410.1 req/sec while the throughput for Virtual Threads came out to be 8898.3 req/sec which is a 60.79% increase as shown in Figure 4.
For CPU intensive workloads throughput was comparable between Platform Threads and Virtual Threads which is expected as these tasks are computationally heavy which primarily depends on hardware. While in I/O intensive workloads throughput achieved by Virtual Threads was significantly higher than throughput achieved by Platform Threads due to their efficient way of handling the blocking operations which minimizes the idle time.

## 6.2 Performance metrics 2: Latency

Latency is another crucial performance metric for web application as it represent the amount of time taken to process a request. This was also measured with the help of

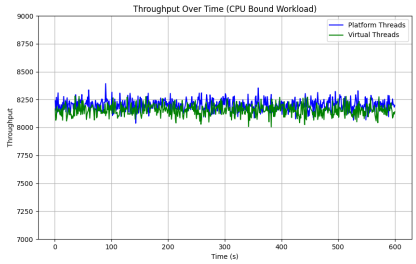


Figure 3: Throughput Over Time (CPU-Bound workload)

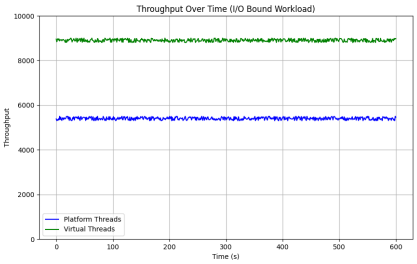


Figure 4: Throughput Over Time (I/O-Bound workload)

Apache Jmeter. As shown in Figure 5, for CPU intensive workload the average latency for Platform Threads was 121 milliseconds while the average latency for Virtual Threads was around 118 milliseconds. It is a 2.48% improvement in the latency for CPU intensive workloads. But for I/O intensive workloads the latency for Platform Threads was 448 milliseconds while for Virtual Threads it was 319 milliseconds. This shows a promising 28.8% improvement in latency when application was switched to Virtual Threads as shown in Figure 6.
Latency difference for CPU intensive workloads was minimal which is expected because most of the time is consumed by computations and not by threads. But for I/O intensive workloads the latency got significantly decreased which highlights the efficiency of Virtual Threads.

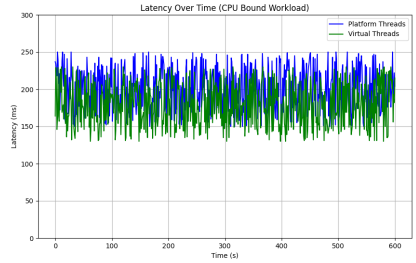


Figure 5: Latency Over Time (CPU-Bound workload)

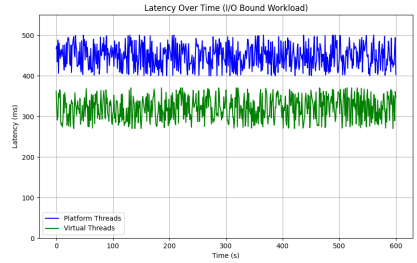


Figure 6: Latency Over Time (I/O-Bound workload)

## 6.3 Performance metrics 3: Memory Usage

Memory usage is a important metric that needs to be carefully evaluated. It evaluates the efficiency of Java application. This was measured by VisualVM by profiling the applications. As shown in Figure 7, for CPU intensive workloads the Platform Threads used 1.5 GB of memory on average while Virtual Threads used 1.3 GB. It is a 13.33% improvement over traditional Platform Threads. For I/O bound workloads the Platform Threads used 2.2 GB of memory while Virtual Threads used 1.4 GB of memory which is 36.36% improvement as shown in Figure 8.
This demonstrate that Virtual Threads consume lower memory compared to Platform Threads due to their lightweight architecture. The reduction was more evident in I/O intensive workloads because it had more involvement of concurrent threading and management of blocking threads.

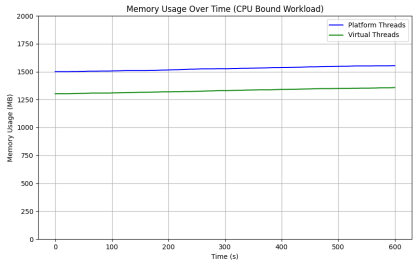


Figure 7: Memory Usage Over Time (CPU-Bound workload)

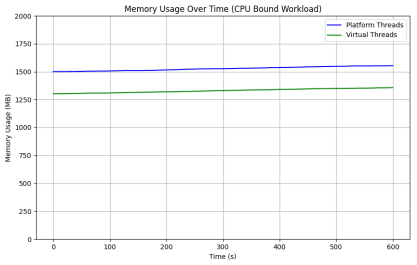


Figure 8: Memory Usage Over Time (I/O-Bound workload)

## 6.4 Performance metrics 4: CPU Utilization

CPU utilization is one of the most important metric because it directly affects how much computation power is needed to do a certain amount of work. Also applications that utilize the CPU more efficiently give better performance. This metric was measured by VisualVM by profiling the Java applications. As shown in figure 9, for CPU intensive workloads, Platform threads showed CPU utilization of 94% while for Virtual Threads it was 93%. So Platform Threads performed 1.05% better than Virtual Threads in terms of utilising the CPU power. But for I/O intensive workloads, Platform Threads showed 72% of CPU utilization while Virtual Threads showed 84% of CPU utilization. This is a 14.29% improvement over traditional Platform Threads as shown in Figure 10.
The CPU utilization for CPU intensive workloads was nearly identical but Platform Threads performed better because they are more stable and Virtual Threads are very new compared to them. While in I/O intensive workloads Virtual Threads made better use of CPU resources which was done by reducing the idle times with the help of lightweight threads.

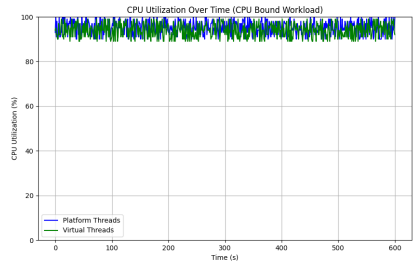


Figure 9: CPU Utilization Over Time (CPU-Bound workload)

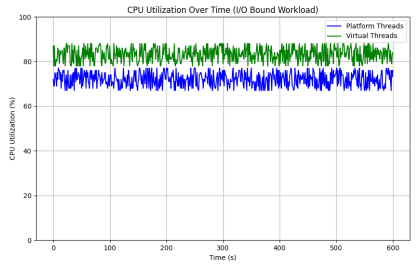


Figure 10: CPU Utilization Over Time (I/O-Bound workload)

## 6.5 Discussion

With the release of Java Virtual Threads the way to achieve concurrency for high-throughput workloads is simplified. They offer a promising way to achieve concurrency through a simplified way without the use of callbacks. This study did robust benchmarking of Java Virtual Threads and compared with Traditional Platform Threads. Results of four main metrics were collected which were Throughput, Latency, Memory usage and CPU Utilization.

Throughput achieved in CPU intensive workload was similar for both Platform Threads and Virtual Threads because most of the tasks done are computational and there is no need for higher number of threads or high concurrency. That is why even with the help of Virtual Threads no significant improvement was observed. But that changed in the case of I/O intensive workloads in which high number of threads were used. And most of the times the thread is in waiting state where it is waiting for a response for I/O operation. It is at this position that Virtual Threads shine and because they are lightweight and managed by JVM, there is no issue for millions of virtual threads to exist at the same time in the waiting state are therefore the throughput saw a increase.
For Latency also not much difference was achieved in CPU intensive workloads. That is due to the fact that most of the time taken by the request is the computation that is done which was calculating prime numbers in this case. So, while using lightweight thread should be beneficial for other metrics it did not affect latency. While in I/O intensive workloads most of the time is spent in thread management and changing its states from waiting to active to stop. Therefore by using Virtual Threads we see a significant drop in latency which is critical for applications.
Memory usage is another critical metric which creates a significant effect on other metics if not managed properly. Virtual Threads showed improvement in both cases for this metric. More so in I/O bound than CPU bound just because of the high number of threads used in I/O tasks. This reflects on the lightweight nature of the Virtual Threads as it takes significantly less space to create a Virtual Threads than to create a Platform Thread.
CPU utilization is one of the most important metric as it can significantly change the over all system performance and cost. Virtual Threads showed no improvement in CPU Utilization in CPU-bound workloads which is expected as most of the utilization was because of the computation. While it showed improvements in I/O-bound workloads which shows lightweight threads utilized the CPU better. Main factor for this is because virtual threads reduce the idle time for the CPU because there is less or easy context-switching compared to Platform Threads.

# 7 Conclusion and Future Work

## 7.1 Conclusion

The goal of this research was to evaluate the performance of Java Virtual Threads against traditional Platform Threads under high-throughput workloads. By benchmarking applications with two different workloads, CPU-Bound and I/O-Bound, this study prides insights into how the two threading models handle concurrency and where Virtual Threads shine. The results demonstrated that:

- **For CPU-Bound Workloads**: Both Virtual Threads and Platform Threads showed comparable performance in throughput, latency and CPU utilization. This signifies that CPU-intensive tasks are bottlenecked by computational resources and increasing concurrency will not have much significance.
- **For I/O-Bound Workloads**: In this case Virtual Threads outperformed Platform Threads in throughput and latency. With the lightweight architecture they

enabled higher concurrency, reduced memory consumption and led to better CPU utilization.

These findings achieved the initial objectives of the study. It provides clear evidence for the conditions under which Virtual Threads can outperform traditional threading models. This research contributes to both academia and practice by providing concrete data on threading model performance, addressing gaps in literature and offering guidance for developers. The findings are constrained by controlled environment and workload types tested which makes it necessary to explore those ideas in the future.

## 7.2 Future Work

This research has several avenues for future work to be done:

- **Mixed Workload Scenarios**: Further studies should analyze mixed workloads as they depict a better picture of real world scenarios. The workloads should be a combination of CPU-bound and I/O-bound tasks.
- **Integration with Cloud-Native Applications**: Extending this testing in cloud-native architecture such as Kubernetes or serverless architecture can produce interesting results. It will help developers to better understand the place of Virtual Threads in the threading models.
- **Reactive and Asynchronous Models**: Combining Java Virtual Threads with reactive frameworks like Project Reactor could offer additional insights into concurrency and asynchronous approaches.

# References


Agnihotri, J. and Phalnikar, R. (2018). Development of performance testing suite using apache jmeter, *Intelligent Computing and Information and Communication: Proceedings of 2nd International Conference, ICICC 2017*, Springer, pp. 317–326.

Ahmad, S. (2016). Load balancing in distributed framework for frequency based thread pools, *Computational Ecology and Software* **6**(4): 150.

Begel, A., MacDonald, J. and Shilman, M. (1999). Picothreads: Lightweight threads in java.

Carvalho, F. M. and Fialho, P. (2023). Enhancing ssr in low-thread web servers: A comprehensive approach for progressive server-side rendering with any asynchronous api and multiple data models., *WEBIST*, pp. 37–48.

Chen, K.-Y., Chang, J. M. and Hou, T.-W. (2010). Multithreading in java: Performance and scalability on multicore systems, *IEEE Transactions on Computers* **60**(11): 1521–1534.

Chen, K.-Y., Chang, J. M. and Hou, T.-W. (2011). Multithreading in java: Performance and scalability on multicore systems, *IEEE Transactions on Computers* **60**(11): 1521–1534.

Friesen, J. and Pal, S. (2015). *Java threads and the concurrency utilities*, Springer.

Goetz, B. (2006). *Java concurrency in practice*, Pearson Education.

Hagl, R. (n.d.). Programming: Concepts and languages.

Haneklint, C. and Joo, Y. H. (2023). *Comparing Virtual Threads and Reactive Webflux in Spring: A Comparative Performance Analysis of Concurrency Solutions in Spring*, PhD thesis.
**URL:** *https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-328247*

Horstmann, C. S. (2007). *Java Concepts for Java 5 and 6*, John Wiley & Sons, Inc.

King, I. E. (2014). Understanding thread interactions.

Lenka, R. K., Mamgain, S., Kumar, S. and Barik, R. K. (2018). Performance analysis of automated testing tools: Jmeter and testcomplete, *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, IEEE, pp. 399–407.

Navarro, A., Ponge, J., Le Mouël, F. and Escoffier, C. (2023). Considerations for integrating virtual threads in a java framework: a quarkus example in a resource-constrained environment, *Proceedings of the 17th ACM International Conference on Distributed and Event-based Systems*, pp. 103–114.

New, E. J., Model, T., Veen, R. and Vlijmincx, D. (n.d.). Virtual threads, structured concurrency, and scoped values.

Ponge, J., Navarro, A., Escoffier, C. and Le Mouël, F. (2021). Analysing the performance and costs of reactive programming libraries in java, *Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, pp. 51–60.

Pufek, P., Beronić, D., Mihaljević, B. and Radovan, A. (2020). Achieving efficient structured concurrency through lightweight fibers in java virtual machine, *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, IEEE, pp. 1752–1757.

Reinders, J. (2007). *Intel Threading Building Blocks, Outfitting C++ for Multi-core Processor Parallelism*, O'Reilly Media, Inc.

Ron Pressler, A. B. (2023). JEP 444: Virtual Threads, `https://openjdk.org/jeps/444`. [Accessed 11-12-2024].

Rosà, A., Basso, M., Bohnhoff, L. and Binder, W. (2023). Automated runtime transition between virtual and platform threads in the java virtual machine, *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, pp. 607–611.

Sodian, L., Wen, J. P., Davidson, L. and Loskot, P. (2022). Concurrency and parallelism in speeding up i/o and cpu-bound tasks in python 3.10, *2022 2nd International Conference on Computer Science, Electronic Information Engineering and Intelligent Control Technology (CEI)*, pp. 560–564.

Subramaniam, V. (2014). Functional programming in java: harnessing the power of java 8 lambda expressions.

Togashi, N. and Klyuev, V. (2014). Concurrency in go and java: performance analysis, *2014 4th IEEE international conference on information science and technology*, IEEE, pp. 213–216.