JCSI 33 (2024) 306–312
Received: 28 June 2024
Accepted: 10 July 2024

Performance analysis of coroutines and other concurrency techniques in Kotlin language for I/O operations

Analiza wydajności współprogramów i innych metod przetwarzania współbieżnego w języku Kotlin dla operacji wejścia/wyjścia

Michał Grabowiec*, Sebastian Wiktor, Jakub Smołka

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract

This article focuses on analyzing the performance of coroutines and other concurrent processing techniques in Kotlin language for input/output operations. For this purpose, coroutines, traditional threads, thread pool and virtual threads were put together. An appropriate application was created and test scenarios were developed. A series of tests were conducted, followed by an analysis of the obtained results. These results indicate that coroutines and thread pool exhibit the highest performance, highlighting their importance in optimizing concurrent processing in the Kotlin language.

Keywords: Kotlin; coroutines; concurrent processing

Streszczenie

Artykuł skupia się na analizie wydajności współprogramów i innych metod przetwarzania współbieżnego w języku Kotlin dla operacji wejścia/wyjścia. W tym celu zestawiono ze sobą współprogramy, tradycyjne wątki, pulę wątków oraz wątki wirtualne. Stworzono odpowiednią aplikację i opracowano scenariusze badawcze. Przeprowadzona została seria testów, a następnie analiza otrzymanych wyników. Otrzymane wyniki wskazują, że współprogramy i pula wątków cechują się największą wydajnością, co stanowi istotne zagadnienie w kontekście optymalizacji przetwarzania współbieżnego w języku Kotlin.

Słowa kluczowe: Kotlin; współprogramy; przetwarzanie współbieżne

*Corresponding author

Email address: s97060@pollub.edu.pl (M. Grabowiec)

1. Introduction

The rapid advancement of technology elevates expectations for both computer hardware performance as well as the efficiency of software running on it. In response to these needs programming languages keep introducing and improving various concurrent processing techniques that allow different operations to be performed at the same time, making full use of available resources and reducing program execution time. Processing a huge number of input-output operations or handling multiple queries to an external system at the same time requires appropriate approach and choosing the best possible solution. In the case of the Kotlin language, one of the innovative, although not entirely new approaches to concurrent programming are coroutines.

The concept of coroutines has already appeared in the literature in 1958, described by Melvin Conway [1], but it is not as popular and widely used as other methods of concurrent processing. In addition to the aforementioned coroutines, Kotlin being interoperable with multiple languages running on JVM (Java Virtual Machine) offers other traditional approaches for concurrent data processing, such as traditional threads, thread pools or virtual threads. Due to rapidly growing popularity of the Kotlin language [2] and the limited amount of scientific work on coroutines, it is important to examine their performance compared to the other concurrent processing techniques. Evaluating the effectiveness and benefits of these approaches is crucial for software developers in a dynamic, fast-paced environment, as they must choose the most efficient solutions and optimize their applications performance This analysis can provide valuable insights into selecting best concurrent programming technique in the context of Kotlin language.

2. Literature review

Since its introduction in 2011 [3], Kotlin remains a relatively young programming language. The concept of Kotlin coroutines, introduced even later, has not yet been extensively studied in terms of performance compared to other solutions within the language.

Given that Kotlin is directly derived from Java and runs on the Java Virtual Machine [4]. It is reasonable to review studies comparing the performance of coroutines with solutions implemented in Java or Scala. Importantly, a 2020 paper by Everlönn and Gakis demonstrates that Kotlin and Java exhibit very similar performance, with no significant differences in code execution speed. This result can be expected due to highly optimized nature of Java Virtual Machine [5]. In a 2021 paper, Chauhan, Kumar, Sethia, and Alam conducted a performance analysis of Kotlin coroutines by comparing them with the RxJava library. The results of their study clearly indicate that coroutines are a more efficient approach [6]. Researchers Koval, Alistarh and Elizarov in their 2022 paper implemented coroutines into their buffer channel algorithm achieving a tenfold improvement in performance [7]. Given the versatility of the coroutines concept across different programming languages a 2020 paper by Shafi, Hashmi, Subramoni and Panda demonstrated that using coroutines in Python to implement an RDMA-based communication library achieved better performance than using alternative approaches [8]. A 2010 article by Stadler, Wurthinger and Wimmer demonstrated that their proposed implementation of coroutines on the Java Virtual Machine manages resources better, offering higher performance than competing JRuby Fibers-based threads [9]. The results of Beronić, Modric, Mihaljević and Radovan's work indicated that Kotlin-based coroutines are more efficient than traditional Java threads [10].

Summarizing the review of the scientific literature on the topic of coroutines, it is reasonable to hypothesize that Kotlin coroutines may offer higher performance and greater efficiency than other available concurrent processing techniques.

3. Test application

This section details the development of a test application designed to evaluate the performance of Kotlin coroutines. The application compares different methods of performing I/O (input/output) operations on files and processing HTTP requests to an external REST API. These methods were implemented using various dispatchers provided by Kotlin coroutines implementation as well as thread pool, virtual threads and traditional threads.

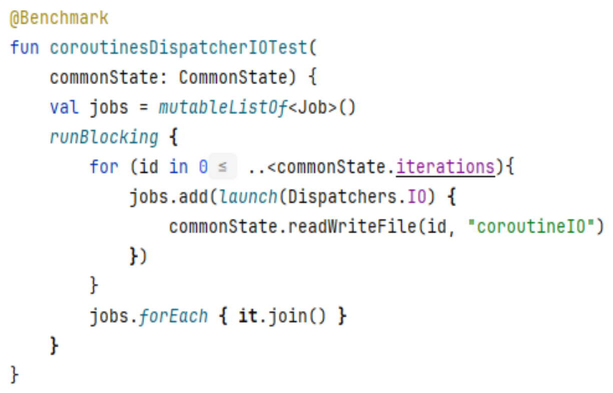
3.1. Application structure

The application consists of two primary classes: FileReadWriteTest and RestApiTest. Each class has an associated State class, which contains the variables and methods needed to set up the benchmarks. This class includes variables such as the number of operations to be executed simultaneously, the path of the directory containing files for the read/write benchmark and the endpoint URL for the network I/O benchmark. It also includes methods to initialize the required data for the read/write benchmark and to set up the HTTP client for the network I/O one. This design approach effectively separates the code being measured from the code responsible for setting up the benchmark. Both the FileReadWriteTest and RestApiTest classes contain 5 test methods, which are detailed in section 3.2. The API used in the tests was implemented locally, on the same computer. It was a simple Rest API with single endpoint handling GET type requests that retrieved a simple, dynamically generated list of 5 tasks to do and returned it as a JSON (JavaScript Object Notation) object. API was created using the .NET 6.0 framework and Visual Studio 2022.

3.2. Test methods

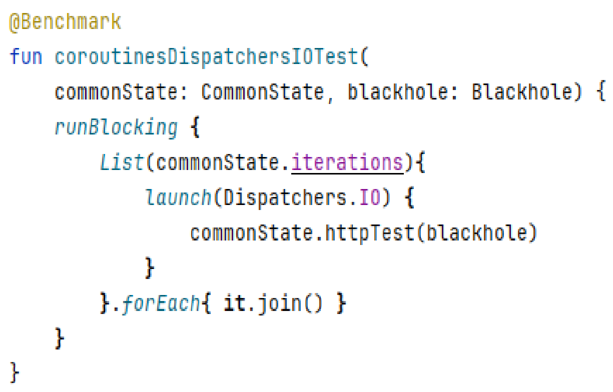
Each test method implements a different approach to performing I/O operations:

1. coroutinesDispatcherIOTest: This method utilizes coroutines running on an IO dispatcher specifically designed to handle input/output operations such as file handling or network operations in the most efficient matter. These coroutines are executed by a shared pool of threads (consisting of 64 threads by default) running inside the dispatcher [11]. The code of this method in both variants is shown in Figure 1 and Figure 2.
2. coroutinesDispacherDefaultTest: This method employs coroutines running on the default dispatcher which is backed by a shared pool of threads. The maximum number of threads in this pool is equal to the number of CPU cores, but it is ensured to have at least two threads. This setup ensures efficient utilization of available resources, particularly for CPU-intensive operations [12].
3. threadPoolTest: This method creates a fixed pool of threads internally to perform I/O operations. The number of threads can be specified by the developer and be any value, but in the test application the number is set to match the available CPU cores.
4. threadsTest: This method initiates a new thread for each I/O operation. In the benchmark, the number of threads created equals the number of operations performed.
5. virtualThreadsTest: This method utilizes lightweight virtual threads, implemented by the Java runtime rather than the operating system. By leveraging optimizations provided by the JVM, virtual threads can help in reducing memory and CPU usage while facilitating the concurrent execution of multiple operations [13].



```
@Benchmark
fun coroutinesDispatcherIOTest(
    commonState: CommonState) {
    val jobs = mutableListOf<Job>()
    runBlocking {
        for (id in 0 ≤ ..<commonState.iterations){
            jobs.add(launch(Dispatchers.IO) {
                commonState.readWriteFile(id, "coroutineIO")
            })
        }
        jobs.forEach { it.join() }
    }
}
```

Figure 1: Code of coroutinesDispatcherIOTest method in FileReadWriteTest class.



```
@Benchmark
fun coroutinesDispatchersIOTest(
    commonState: CommonState, blackhole: Blackhole) {
    runBlocking {
        List(commonState.iterations){
            launch(Dispatchers.IO) {
                commonState.httpTest(blackhole)
            }
        }.forEach{ it.join() }
    }
}
```

Figure 2: Code for the coroutinesDispatcherIOTest method in the RestApiTest class.

4. Research methods

To analyze the performance of coroutines and other concurrent processing techniques in the context of input/output operations, two test cases were defined and implemented:

- sending requests to an external Rest API,
- reading from and writing to 1 MB files on the device.

Both test cases were implemented using different approaches described in the section 3.2. A series of benchmarks was conducted to measure execution time as well as resource consumption of the application code. These benchmarks are detailed in the following section. The application code was developed using Kotlin version 2.0.0 in the IntelliJ IDEA 2024.1.2 integrated development environment. The kotlinx.coroutines library version 1.9.0-RC was utilized to implement coroutines-based solutions, and the application was executed using the Java Development Kit version 22.0.1. Benchmarks were conducted on a computer with the specifications shown in Table 1.

Table 1: Test computer configuration

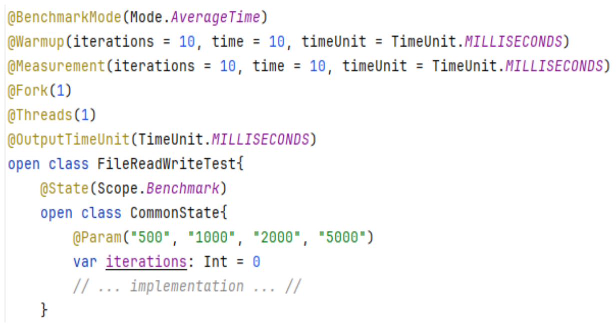
Component	Specification
Operating system	Windows 11 Pro 23H2
Processor	AMD Ryzen 5 5600X 6 cores, 12 threads
Memory	16 GB, DDR4, 3600MHz
Hard drive	Kingston KC3000 M.2 PCIe 4.0 NVMe

4.1. Usage of resources

To gauge resource consumption accurately, the emphasis was placed on measuring CPU utilization and machine memory usage throughout the execution of the code implementing the test cases outlined in section 4. For this purpose, a software called VisualVM was used. This is an advanced tool designed to monitor and analyze applications running on the JVM platform. Its capabilities include tracking heap memory consumption, monitoring CPU utilization or tracking the number of threads being used. The tool measures these metrics in in real time and by using appropriate plugins allows them to be exported, enabling reliable analysis of the results.

4.2. Execution time

In the context of an application performance, one of the most crucial factors is the program execution time, which serves as a key evaluation criterion, particularly for applications requiring responsiveness, high throughput and efficient use of system resources. In order to conduct execution time benchmarks, the JMH (Java Microbenchmark Harness) library developed by Oracle was employed. Developers of this tool considered the intricate process of code optimization by the Java Virtual Machine's compiler, ensuring a series of repeatable and reliable tests. Furthermore, the JMH's maintenance and development by the same team as the JVM offers added reliability, as the tool's developers possess deep insights into the JVM's internal operations, which ensures that the user gets the most reliable results. The benchmark was configured to measure the average execution time of each tested method enabling the assessment of input/output operation performance. Prior to actual measurement, the benchmark undergoes a warm-up phase consisting of 10 iterations that allows JVM to perform its JiT (Just in Time) compilation and other optimization techniques, stabilizing the state of the Java Virtual Machine before the benchmark begins. Subsequently, the measurement phase, also consisting of 10 iterations, follows. Results are reported in milliseconds. The configuration of the performance test is shown in Figure 3.



```
@BenchmarkMode(Mode.AverageTime)
@Warmup(iterations = 10, time = 10, timeUnit = TimeUnit.MILLISECONDS)
@Measurement(iterations = 10, time = 10, timeUnit = TimeUnit.MILLISECONDS)
@Fork(1)
@Threads(1)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
open class FileReadWriteTest{
    @State(Scope.Benchmark)
    open class CommonState{
        @Param("500", "1000", "2000", "5000")
        var iterations: Int = 0
        // ... implementation ... //
    }
```

Figure 3: JMH benchmark configuration.

5. Results

This section presents the results of the conducted tests obtained through the research methods and scenarios described previously. The results of the execution time measurement for each concurrent processing approach are depicted in the form of bar charts containing information about the average execution time of operations. The graphs are shown in Figures 4-11. Additionally, Table 2 illustrates resource consumption during file read and write operations for 5000 iterations. During the second scenario, tests involving sending requests to an external indicated negligible resource consumption across all methods. Given their minimal relevance to the overall research, these findings were omitted from the results. The absence of tests for the method based on virtual threads within the same test scenario stemmed from the constraints imposed by the HttpClient library utilized for handling the API calls.

Table 2: Resource consumption during execution of 5000 iterations of reading and writing a file

Technique	CPU usage [%]	Memory usage [MB]
Dispatcher Default	46,6	765
Dispatcher IO	13,1	751
Threads	41,1	1575
Thread Pool	8,8	392
Virtual Threads	54	1162

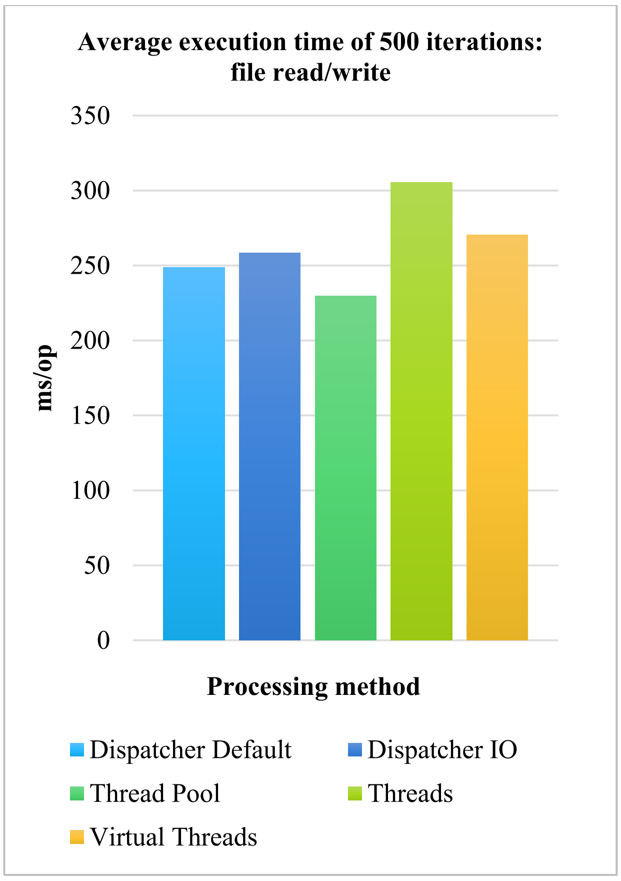


Figure 4: Average execution time of 500 iterations: file read/write.

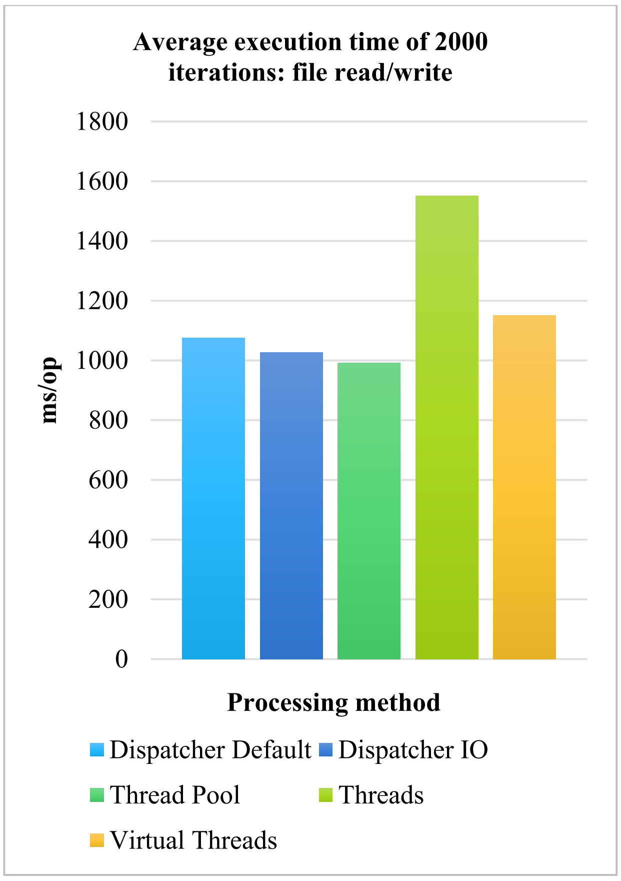


Figure 6: Average execution time of 2000 iterations: file read/write.

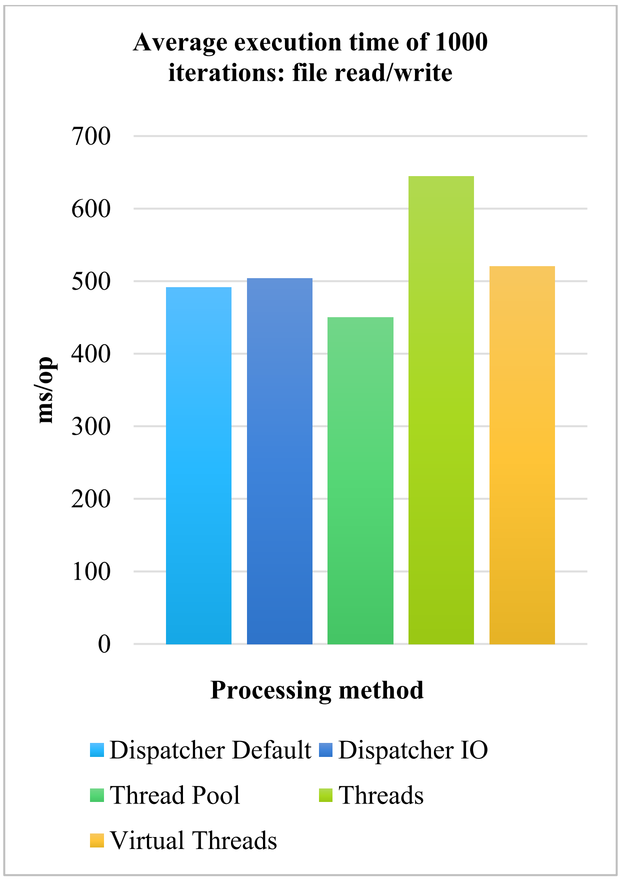


Figure 5: Average execution time of 1000 iterations: file read/write.

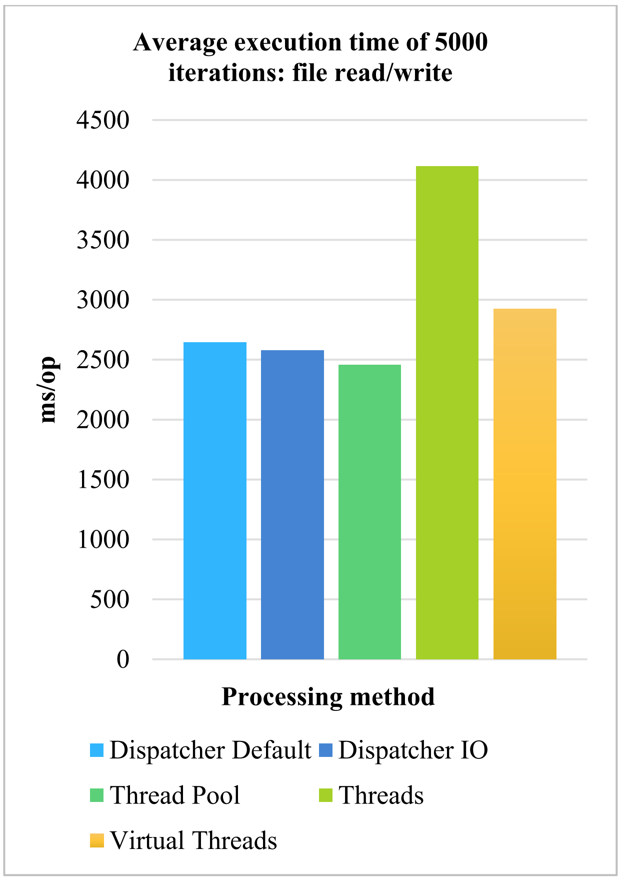


Figure 7: Average execution time of 5000 iterations: file read/write.

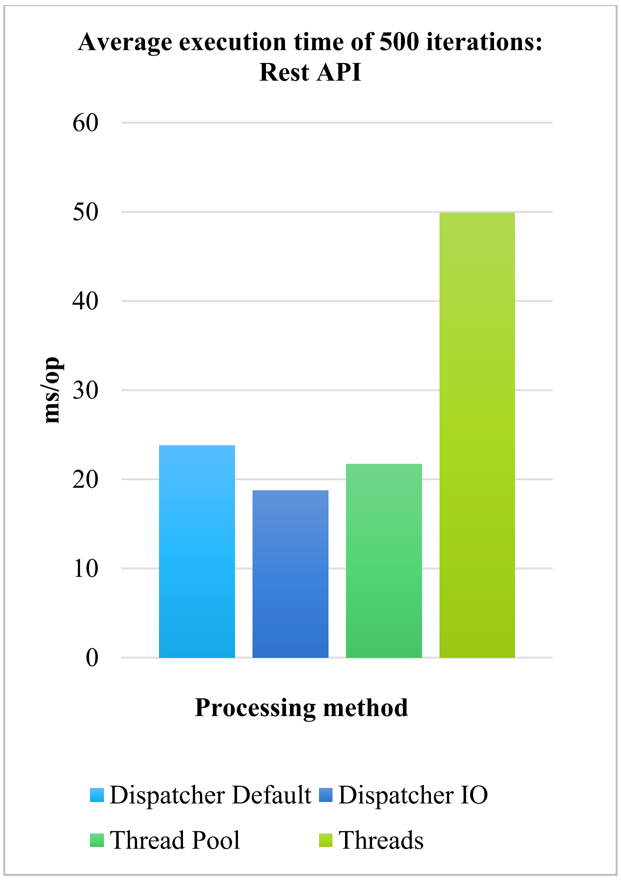


Figure 8: Average execution time of 500 iterations: Rest API.

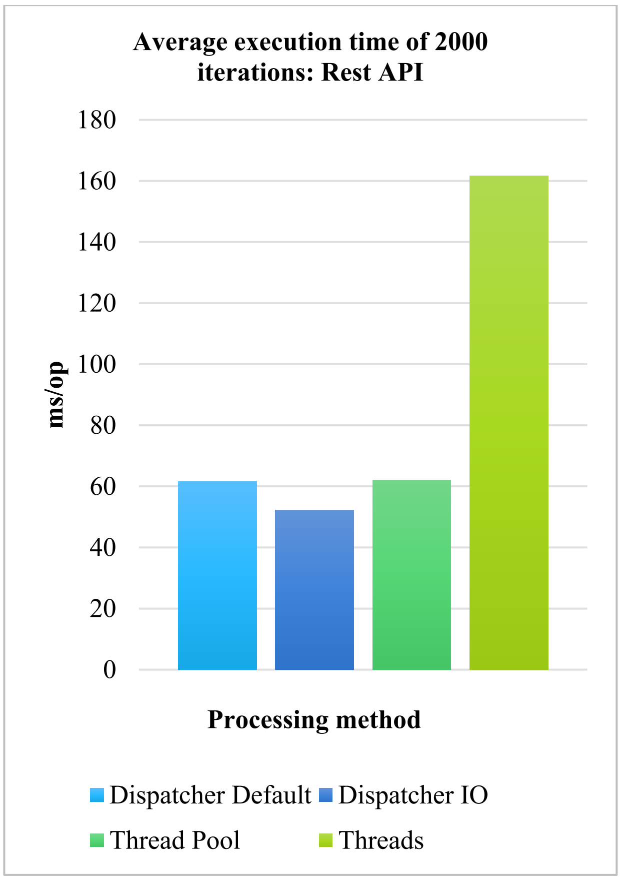


Figure 10: Average execution time of 2000 iterations: Rest API.

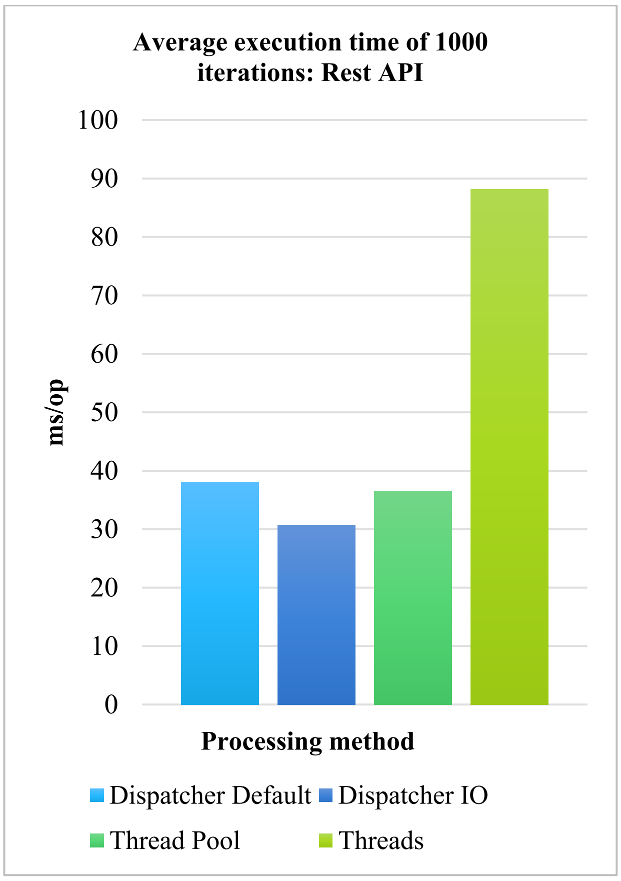


Figure 9: Average execution time of 1000 iterations: Rest API.

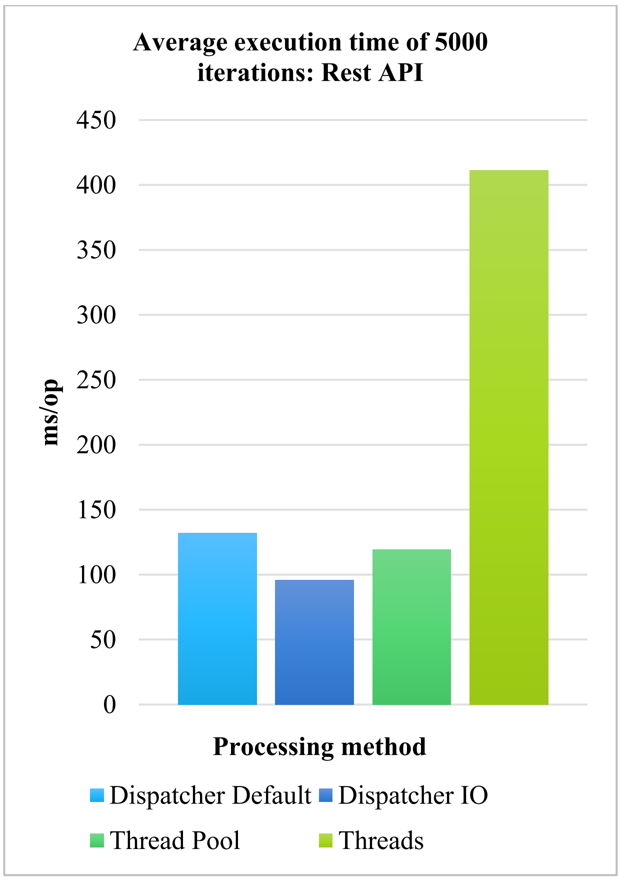


Figure 11: Average execution time of 5000 iterations: Rest API.

6. Discussion of results

The study shows that the most efficient approach for handling input-output operations in Rest API applications is through the use of coroutines using an IO dispatcher. The execution times in this case were consistently the lowest across various numbers of iterations and were, respectively: 18.7 ms/op, 30.6 ms/op, 52.2 ms/op and 95.9 ms/op. Following closely behind is the usage of a thread pool, which yielded execution times of: 21.7 ms/op, 36.5 ms/op, 62.1 ms/op and 119.1 ms/op. On average, the execution time difference between these two approaches was 16.2% in favoring coroutines with the IO dispatcher. Coroutines utilizing the default dispatcher ranked third in terms of execution speed, trailing behind the thread pool by only 6% on average. However, they were considerably slower than coroutines with the input/output dispatcher by 26%. Traditional threads emerged as the least efficient performers across all scenarios, exhibiting the highest execution times for varying numbers of iterations. They lagged significantly behind coroutines with the IO dispatcher, being 2.6 times slower for 500 iterations and a staggering 4.3 times less efficient for 5000 iterations. As mentioned in the previous section, the resource consumption for the operations performed in this test scenario were negligible and their analysis has no bearing on determining which approach is the most efficient.

The research conducted in accordance with the second test scenario revealed that the most efficient method for processing files is the use of thread pool for which the execution times were respectively: 229.7 ms/op, 449.8 ms/op, 991.6 ms/op and for 5000 iterations 2457.4 ms/op. Slightly inferior performance was demonstrated by coroutines based on the IO dispatcher, with execution times of 258.5 ms/op, 503.5 ms/op, 1025.9 ms/op, and 2579.6 ms/op. On average, coroutines in this configuration performed operations 8.2% slower than the thread pool. Coroutines using the default dispatcher yielded results similar to those based on the IO dispatcher, with slight differences of 10 ms and 12 ms faster for 500 and 1000 iterations respectively, but executing operations longer by 50 ms and 66 ms for 2000 and 5000 iterations Virtual threads achieved results similar to coroutines for 500 and 1000 iterations, but at 2000 and 5000 iterations, they were slower than the IO dispatcher-based coroutines by 12.1% and 13.2% respectively. Once again, traditional threads emerged as the least efficient performers, with significantly longer execution times for all iteration variants. These times surpassed those of thread pool operations by 32.9%, 43.2%, 56.3%, and 67.4% respectively.

During performance testing of file processing, resource consumption was measured for each approach. The lowest average CPU usage was observed when processing using a thread pool and coroutines using the default dispatcher, at 8.8% and 13.1% respectively. Traditional threads utilized an average of 41.1% of the CPU, slightly outperforming coroutines based on the default dispatcher (46.6% average CPU usage), with virtual threads utilizing the most resources at an average of 54% of CPU resources. Memory usage exhibited significant variation among the tested techniques. The thread pool consumed the least memory at 392 MB. Both coroutines variants required a similar amount of memory, with 765 MB and 751 MB respectively, while threads and virtual threads consumed the most memory at 1575 MB and 1165 MB respectively.

7. Conclusions

The purpose of this paper was to analyze the performance of coroutines and other concurrent processing techniques within the Kotlin language for input/output operations.

The literature review highlighted that coroutines could be among the most efficient techniques for concurrent processing of IO operations in JVM environment.

To comprehensively explore the paper's topic, two test scenarios were developed and prepared. These scenarios consisted of implementing test methods, developing test methodology as well as selecting appropriate tools to measure each concurrent processing technique accurately. The chosen tools were specifically selected for their precision in measurement, an essential aspect given the context of the Java Virtual Machine. This approach ensured that the evaluation process was robust and capable of providing reliable insights into the performance of various concurrent processing techniques.

The analysis of the test results revealed that IO dispatcher-based coroutines achieved the second-best performance in file processing scenario and emerged as the top performer in testing methods involving sending HTTP requests to the Rest API. Meanwhile, the thread pool-based method showed equally impressive results, excelling as the most efficient approach for file processing and ranking second in handling the Rest API requests. Notably, both methods exhibited the lowest utilization of CPU and memory resources. Coroutines based on the default dispatcher also demonstrated satisfactory performance, trailing slightly behind the two top-performing techniques in each test scenario. Traditional threads and virtual threads delivered the poorest results, significantly lagging behind other solutions. These methods required the most CPU and memory resources, with the performance gap widening with each increase in the number of iterations.

Additionally, the flexibility afforded by the ability to use different dispatchers positions coroutines as the most versatile among all concurrent processing methods. This inherent flexibility enables effective utilization across a wide range of tasks, further solidifying their appeal as a preferred approach in various programming scenarios.

Based on the research conducted and the analysis of the results, it was demonstrated that coroutines, along with thread pools, stand out as the most efficient methods of concurrent processing in the Kotlin language, particularly for input/output operations. Future research could explore the performance of coroutines and other concurrent processing techniques in more diverse and complex scenarios, such as real-time data processing or large-scale distributed systems. Expanding the scope to include these areas could provide deeper insights into the scalability and robustness of coroutines under different workloads.

References

[1] R. Elizarov, M. Belyaev, M. Akhin, I. Usmanov, Kotlin coroutines: design and implementation, Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '21) (2021) 68–84, https://doi.org/10.1145/3486607.3486751.

[2] Octoverse: The state of open source and rise of AI in 2023, https://github.blog/2023-11-08-the-state-of-open-source-and-ai/, [23.05.2024].

[3] D. Gotseva, Y. Tomov, P. Danov, Comparative study Java vs Kotlin, In 27th National Conference With International Participation (2019) 86–89, https://doi.org/10.1109/TELECOM48729.2019.8994896.

[4] D. Jemerov, S. Isakova, Kotlin in Action, Manning Publications, New York, 2016.

[5] N. Everlönn, S. Gakis, Java and Kotlin, a performance comparison, Bachelor thesis, Kristianstad University, Kristianstad, 2020.

[6] K. Chauhan, S. Kumar, D. Sethia, M. N. Alam, Performance Analysis of Kotlin Coroutines on Android in a Model-View-Intent Architecture pattern, In 2021 2nd International Conference for Emerging Technology (INCET) (2021) 1–6, http://dx.doi.org/10.1109/INCET51464.2021.9456197.

[7] N. Koval, D. Alistarh, R. Elizarov, Fast And Scalable Channels In Kotlin Coroutines, Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (2023) 107–118, https://doi.org/10.48550/arXiv.2211.04986.

[8] A. Shafi, J. M. Hashmi, H. Subramoni, D. K. Panda, Blink: Towards Efficient RDMA-based Communication Coroutines for Parallel Python Applications, In 2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC) (2020) 111–120, http://dx.doi.org/10.1109/HiPC50609.2020.00025.

[9] L. Stadler, T. Wurthinger, C. Wimmer, Efficient Coroutines for the Java Platform, Proceedings of the 8th International Conference on Principles and Practice of Programming in Java (2010) 20–28, http://dx.doi.org/10.1145/1852761.1852765.

[10] D. Beronić, L. Modrić, B. Mihaljević, A. Radovan, Comparison of Structured Concurrency Constructs in Java and Kotlin – Virtual Threads and Coroutines, In 2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO) (2022) 1466–1471, https://doi.org/10.23919/MIPRO55190.2022.9803765.

[11] Kotlin language documentation: Dispatcher IO, https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-dispatchers/-i-o.html, [29.05.2024].

[12] Kotlin language documentation: Default dispatcher, https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-dispatchers/-default.html, [29.05.2024].

[13] Power of Java Virtual Threads: A Deep Dive into Scalable Concurrency, https://kiranukamath.medium.com/power-of-java-virtual-threads-a-deep-dive-into-scalable-concurrency-18fa4d382f9c, [29.05.2024].