Comparison of Structured Concurrency Constructs in Java and Kotlin – Virtual Threads and Coroutines

D. Beronić, L. Modrić, B. Mihaljević and A. Radovan
Rochester Institute of Technology Croatia (RIT Croatia), Zagreb, Croatia
dora.beronic@mail.rit.edu, lara.modric@mail.rit.edu,
branko.mihaljevic@croatia.rit.edu, aleksander.radovan@croatia.rit.edu

Abstract - **Ubiquitous multi-core processors with a significant increase in computing power resulted in an omnipresent expansion of concurrent server applications. However, modern multithreaded applications exposed a substantial number of efficiency-related challenges. Consequently, lightweight structured concurrency constructs emerged in various multithreaded applications, as the traditional heavyweight threads model is expensive in regards to memory due to its high dependency on OS kernel threads. Modern programming languages such as Kotlin and Java are both built on the Java Virtual Machine (JVM) and are commonly used in mobile application development and server-side applications, offering, by default, the traditional threads approach. However, Kotlin also includes support for a lightweight concurrency model with coroutines, while Java's virtual threads, announced in the OpenJDK's Project Loom, are still experimental. Such contemporary concurrency implementations are primarily enabling an increase in application performance and efficiency. This paper presents an overview of different approaches to structured concurrency and explores their implementation in programming languages Java and Kotlin. It provides a comparative analysis of traditional threads with coroutines in Kotlin and virtual threads in Java. Based on the conducted testing using benchmarks, we analyzed their performance, described implementation differences, and explored their utilization possibilities and adaptation to real-world use-case scenarios.**

Keywords - Concurrent Programing, Virtual Threads, Coroutines, Structured Concurrency, Java, Kotlin

I. INTRODUCTION

The expansion of multi-core processors brought with it a significant increase in computing power and the need for a broader implementation of concurrent server applications. Moreover, popular and widely used programming languages with strong concurrency support, such as Java and Kotlin, enable software developers to optimize server applications' resource usage without wasting valuable resources such as CPU and memory or development time.

Since the performance of applications is often heavily dependent on the background technology and the programming language in which they are implemented, it is unceasingly worth exploring which languages are more suitable for specific contemporary applications. Some research projects, such as [1], examined the performance of different programming languages according to program length, program effort, runtime efficiency, memory consumption and readability. However, such comparisons often lacked to analyze different modern concurrency constructs and multithreading capabilities. Multithreaded applications, commonly implemented with a thread per client approach, are typically applications in server-side programming that need to perform a large number of input-output (I/O) operations in a short period of time, while serving multiple concurrent users. We considered it would be worth exploring such applications in the context of our research with regard to contemporary structured concurrency constructs.

Most often, software developers were forced to choose between asynchronous applications, which could be expensive in regard to development time, and synchronous applications, which could be costly in relation to memory usage. As a solution to this potentially inefficient synchronization and resource usage in synchronous programming solutions, a new approach to structured concurrency was introduced within some contemporary programming languages. In general, structured concurrency could enable software developers to achieve efficiency similar to the one in asynchronous programming through the simplicity of synchronous programming.

The focus of this paper is the exploration of structured concurrency constructs in selected programming languages Java and Kotlin, while also discussing concurrency in some other widely used programming languages, such as Go, PHP, Python, and C#, and comparing it to some single-threaded asynchronous programming languages, such as Node.js. The emphasis is on the performance analysis of the implementations of structured concurrency in the form of coroutines implemented in Kotlin [1] and the new structured concurrency model of virtual threads introduced as a Preview Feature in Java[2], as part of Project Loom's experimental Java Virtual Machine (JVM) [2].

[1] Coroutines, https://kotlinlang.org/docs/coroutines-overview.html

[2] JEP draft: Virtual Threads (Preview), https://openjdk.java.net/jeps/8277131

II. Related Work

The wide availability of multi-core processors resulted in the development and implementation of many multithreaded programming constructs, which are currently considered a standard for performing concurrent computations. Furthermore, this created a need for the further development of existing approaches to threading, followed by implementation and support for asynchronous programming as well as the extended support for synchronous programming by many existing programming languages. When considering scalability, development time, and concurrency level that those applications can support, the question arises of which approach and programming languages would have performed better.

Many challenges regarding synchronous programming occur when considering the performance and scalability of applications in the thread-per-request model. For example, web applications tend to heavily consume hardware resources when supporting many concurrent users and meeting the demand of I/O-intensive real-time operations. Regarding response time, research [3] presented that PHP and Python-Web applications exhibited a significant increase in response latency in comparison to Node.js applications. In this case, Node.js had fewer issues as the number of requests increased, while PHP and Python-Web became more unstable under heavier load. Furthermore, Node.js's non-blocking I/O model performed efficiently in retrieving data, although it was, along with Python-Web, less effective in compute-intensive examples, while PHP performed well on middle-scale applications with fewer concurrent users.

Moreover, we can presume that platform-specific programming languages, such as C#, could outperform platform-agnostic programming languages well-suited for different operating systems, such as Java. For example, it has been proven that the .NET Common Language Runtime (CLR) support for multithreading is well adapted to the thread scheduler within Windows operating system (OS). Presumably, .NET programming languages, such as C#, Visual C++, and Visual Basic, could score better in terms of multithreading since they are better adapted to the specifics of the underlying Windows OS. However, this is not always the case, and Java threads in the JVM outperformed the .NET's CLR threads in terms of creation speed, server latency period, and thread synchronization, in some cases with a significant difference of 99.17% [4], thus indicating a potential need for reflection of the current concurrency implementation within the .NET framework.

Go is a statically compiled programming language with a simpler concurrency model, which recently gained popularity, primarily because of applications such as Kubernetes and Ethereum. Research about comparing threads in Java and Goroutines as Go's main concurrency constructs [5] presented that applications in Go can achieve faster compilation. However, Java exhibited more efficient thread allocation handling on a large scale, but it was less efficient in the creation of a large number of threads, meaning that Go could be a better choice for applications with continuous production of threads [6].

Programming languages Java and Kotlin present their concurrency support through high-level APIs. For example, a comparison of libraries such as MvRx and RxJava, has shown that Kotlin's coroutines can outperform JVM threads in regards to concurrency in some cases [7]. Since Kotlin has two implementations of concurrency in the forms of threads and coroutines, software developers can choose which construct they will use for a particular case through the support of high-level APIs. Kotlin also supports asynchronous programming through Callbacks and Futures; however, this type of code could not combine well with synchronous code. The lack of distinction between synchronous and asynchronous code can lead to difficulties in code comprehensives, pushing the developers towards synchronous programming because of its simplicity.

Considering the early access release of JDK within Project Loom [2], Java and Kotlin are currently among a few platform-independent programming languages that have a high-level API call for both concurrency constructs. Based on the previous research, where coroutines outperformed threads in Java [7], and our past research in which virtual threads outperformed standard threads in Java [8], we assumed that it would be beneficial to compare coroutines directly to virtual threads, and explore which of those could present a better performance in regards to scalability, supported concurrency level, and synchronization speed.

III. Introduction to Concurrency in Java and Kotlin

A. Threads in Java and Kotlin

Although efficient, Java's thread implementation is considered relatively heavyweight due to its strong dependency on operating system (OS) kernel threads. In modern concurrent applications, heavy usage of OS kernel threads can cause performance issues, as each newly created Java's thread is run on top of an OS kernel thread. Threads in Java are considered to be only "thin wrappers" around OS kernel threads, as their scheduling is managed by the OS. Furthermore, the OS kernel thread's scheduling is not platform-specific, as they need to provide support for several programming languages. Due to the generic implementation of Java threads (Fig. 1), scheduling could result in the creation of a larger contiguous stack, resulting in low performance and inefficient usage of resources [8].

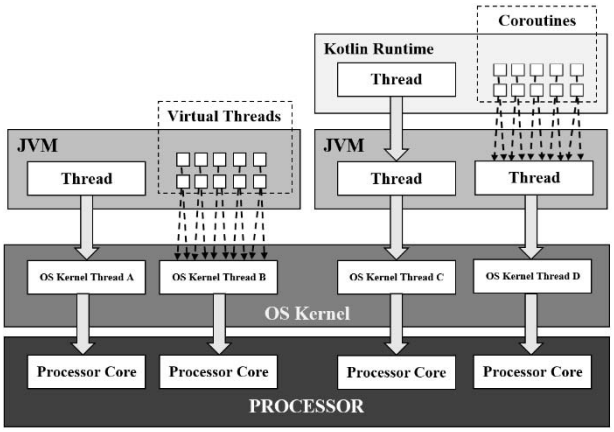


Figure 1. Representation of Thread Mapping in the JVM and Kotlin Runtime

Kotlin was also built on the Java Virtual Machine (JVM), as presented in Fig.1, and initially designed to provide interoperability with Java, thus offering the possibility to use Java's code in Kotlin and vice versa. Therefore, Kotlin's threads work based on the same principle as Java's threads, as Kotlin's *kotlin.concurrent* library is bound to java.lang.Thread class. Creating a thread in Kotlin is practically the same as in Java; the thread can be created using the Thread class or Runnable interface and started with the *start()* function. Kotlin's API contains *thread()*[3] function, which can both create and start a new thread, thus offering a simple implementation of thread creation.

B. Virtual Threads in Java

Announced within the OpenJDK's Project Loom [2], virtual threads (previously called fibers) are considered a new approach to efficient and lightweight structured concurrency in Java. As still in experimental status, virtual threads' current API is not a standard part of the JDK, but it reveals simple-to-use methods adapted to the java.lang.Thread API. From a software developer's point of view, starting a new virtual thread with *startVirtualThread()* method is as straightforward as with regular threads. Therefore, virtual threads offer a similar synchronous code writing technique, with a high level of effectiveness that is manifested in notably lower runtime compared to regular Java's threads [9].

In contrast to Java's standard threads model, virtual thread implementation aims not to be as highly dependent on the OS kernel threads. As the JVM entirely manages the virtual threads' scheduler, it uses fewer OS kernel threads to finish the task execution by releasing the *carrier thread* as soon as thread blocking occurs [10]. Upon unblocking, virtual threads can continue running on another OS kernel thread, and that way, during execution, one virtual thread can be run on a few nonidentical OS kernel threads (Fig. 2). Because of such JVM-specific scheduling, virtual threads show great potential for performance improvement in heavy-load concurrent applications, as they can achieve better scalability and faster synchronization [8].

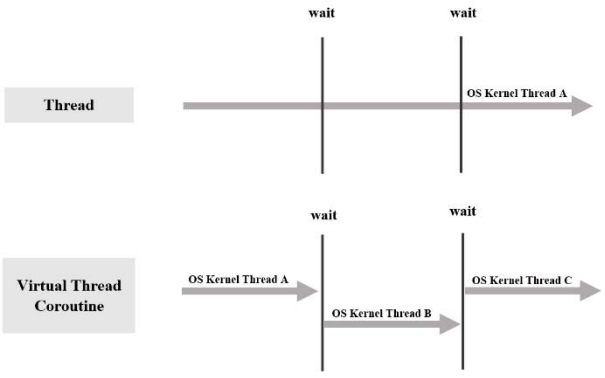


Figure 2. Representation of OS Kernel Thread usage during blocking operations with Threads and Virtual Threads/Coroutines

C. Coroutines in Kotlin

Lightweight concurrency constructs called coroutines have been a standard part of Kotlin since 2017. Coroutines were implemented with the goal of creating concurrency structures that are independent of platform-specific implementations yet consider the importance of code readability and performance improvements [11]. They are not bound to the OS kernel threads, as a single coroutine can execute on the same OS kernel thread or several of them, correspondingly to Java's virtual threads. Their implementation follows the main principle of structured concurrency and is available through the *kotlinx.coroutine* library. The library offers easy-to-use functions, such as *launch()* method used to start a new coroutine.

Kotlin's coroutines are built upon the concept of suspendable functions, based on the Continuation Passing Style (CPS) [7]. When a coroutine is suspended, a state-machine is generated for it, holding details about the so-called suspension point. Execution will proceed in state-machine from the last suspension point when the coroutine is no longer suspended and can continue execution. Through such design, coroutines achieve wide flexibility in use and have an inexpensive creation. To adhere to the principle of structured concurrency, coroutine builders in Kotlin are extensions of the *CoroutineScope*[4] interface, which establishes parent-child relationships between coroutines with the propagation of cancelation.

IV. Testing Environment

A. Testing Environment Setup

Our academic research testing environment setup consisted of a 64-bit machine with Ubuntu operating system version 20.04.3 LTS, with 16 GB memory and Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz processor. It included the OpenJDK's early access build of 64-bit JDK-19, which was used to run Project Loom features and to use virtual threads and coroutines for our benchmark testing.

B. Test Cases

Our benchmark consisted of a multithreaded HTTP Server, waiting for incoming standard requests at a port. The same benchmark program was written in both Java and Kotlin, using different concurrency constructs. There were four test cases – for Java using regular threads and virtual threads and for Kotlin using regular threads and coroutines. A task is executed at each incoming request, consisting of creating an object, writing the created object's information to a file, and returning the object's data in a response. Two types of software were used to record the results – Vegeta[5], a tool for application load testing, performing request attacks to an HTTP server at a defined constant rate, and VisualVM[6], a tool that provides a visual representation of information about, among others, threads, heap, CPU, and memory usage for applications running on the JVM.

[3] thread, https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.concurrent/thread.html

[4] CoroutineScope, https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/index.html

[5] Vegeta, https://github.com/tsenart/vegeta

[6] VisualVM, https://visualvm.github.io/

V. PRELIMINARY RESULTS

Within the scope of our research, we recorded results in regards to latency, the number of total OS kernel threads started, and used heap memory space. Vegeta software was used to perform request attacks at two selected rates – 2000 and 8000, each having a duration of 10 seconds. After three pre-runs used to warm up the test, we recorded latency twice for each test case at each rate. From these results, average latencies were calculated. Information from VisualVM was used to analyze the number of total threads started and used heap after 5 consecutive runs of request attacks using the Vegeta tool.

A. Total threads started

Concurrency constructs in both Java and Kotlin use and depend on OS kernel threads, and the number of such threads they use affects application performance. Given results should be observed taking into consideration that 5 consecutive runs of attacks have been performed, each making 2000 requests per second during 10 seconds. Therefore, we can expect the creation of approx. 100 000 threads, coroutines, or virtual threads (depending on what is tested). Following the same calculation, about 400 000 of them are expected to be created at a request rate of 8000.

Our findings also answer the question of how many OS kernel threads are required to execute tasks for the given number of requests performed by threads, virtual threads, or coroutines. Our preliminary results, presented in Table I, have shown significant differences in OS kernel thread usage between concurrency constructs in Java and Kotlin.

TABLE I. THE AVERAGE NUMBER OF THREADS STARTED AFTER 5 CONSECUTIVE ATTACK RUNS TO OUR TESTING PROGRAM

Rate	2000	8000
Java - threads	100 017.33	388 749.67
Java - virtual threads	35.67	268.33
Kotlin - threads	100 016.00	382 669.67
Kotlin - coroutines	23.67	23.67

In more detail, Java's threads create and require more than 2803 times more OS kernel threads than virtual threads at an attack rate 2000. As expected, virtual threads also notably outperform threads at a rate 8000. Threads in Kotlin create more than 4225 times more OS kernel threads than coroutines at a rate 2000, and more than 16166 times more of them at a rate 8000. While there are no significant differences between concurrency constructs in Java and Kotlin at both rates, more significant differences are visible at rate 8000, where coroutines outperform virtual threads by creating 11.34 times fewer OS kernel threads. Based on these results, we can conclude that both virtual threads and coroutines are more efficient when taking into consideration OS kernel threads usage, which means that they have a significantly better optimization of scheduling between OS kernel thread swapping than the current implementation of threads. A lower number of created OS kernel threads could also lead to lower latency and lower memory usage results. Furthermore, based on these results, we can also confirm that, as previously mentioned, for almost every new thread in either Java or Kotlin, a new OS kernel thread is created.

B. Used heap

Fig. 3 shows heap space usage measured in megabytes (MB) for different concurrency constructs in Java and Kotlin. All concurrency constructs occupy more heap space at a bigger attack rate, due to more incoming requests and the creation of a greater number of objects, as previously explained. At both rates, virtual threads perform better than regular threads in Java by using notably less heap space. In Kotlin, coroutines also use less heap space than regular threads at both request attack rates. Virtual threads demonstrate better results than coroutines, having 69.31% smaller heap usage at attack rate 2000 and 35.29% smaller heap usage at rate 8000. Based on our preliminary results, we can conclude that in our test case, Java's virtual threads can outperform all the other observed concurrency models by having the lowest heap usage at both rates.

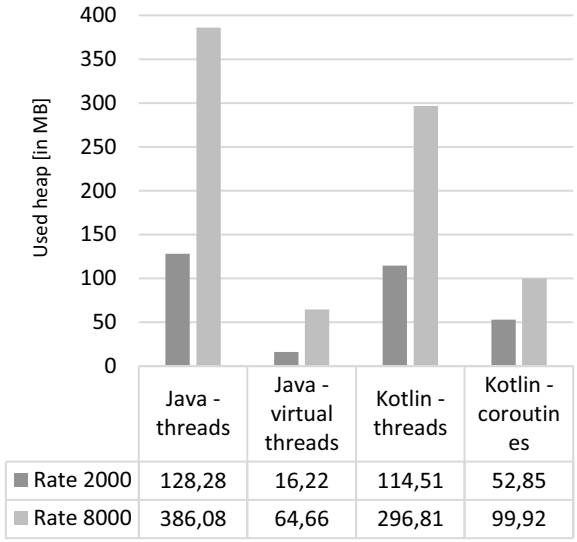


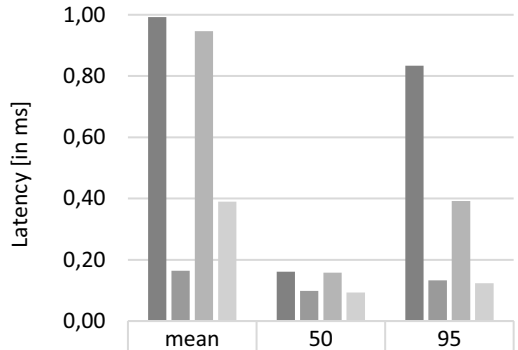
Figure 3. Used heap space after 5 consecutive attack runs to our testing programs in Java and Kotlin

On the contrary, in our test case, Java's threads show the worst results in regards to heap usage, with the highest heap usage values and the biggest differences between lower and higher attack rates. As threads are wrappers around OS kernel threads, it is harder to remove them from the heap, causing them to eventually "clog" the heap. Since a significant number of OS kernel threads are used to handle the high request rate, they are occupying the resources needed for the efficient memory management process and garbage collection, disabling them from clearing the heap and removing unnecessary objects created by the previous requests.

In comparison to threads, mostly because of better schedulers, virtual threads and coroutines both release their OS kernel thread from the heap as soon as they are no longer needed, without unnecessary hold. With virtual threads' schedulers being even more specific and directly connected to the OS kernel threads than coroutine's one, virtual threads are released from the heap even faster than coroutines which are dependent on the java.lang.Threads.

C. Latency

Results of our benchmark testing presented that at both performed request attack rates, virtual threads in Java and coroutines in Kotlin have better latency results than regular threads in both languages. Fig. 4 presents the mean values, 50 percentiles, and 95 percentiles average latency for concurrency constructs at an attack rate of 8000. At a higher attack rate, threads in both Java and Kotlin have a similar mean and 50 percentile latency, while Kotlin's threads have 2.13 times smaller 95 percentile latency. Virtual threads presented better performance compared to coroutines at mean latency by being 2.37 times faster, while coroutines show slightly better 95 percentile results.



	mean	50	95
Java - threads	0,9921	0,1615	0,8331
Java - virtual threads	0,1642	0,0988	0,1329
Kotlin - threads	0,9464	0,1582	0,3915
Kotlin - coroutines	0,3896	0,0931	0,1234

Figure 4. The average latency for concurrency constructs in Java and Kotlin at a request attack rate of 8000

At a lower attack rate of 2000, both Java's threads and virtual threads somewhat outperform Kotlin's threads and coroutines. The average maximum latency recorded during research was reached by Java's threads at a higher attack rate of 51.56 milliseconds. Java's virtual threads have the lowest average maximum latency with 42.98 milliseconds.

The concurrency level that the application can support is based on the number of concurrent requests that the application can handle in a specific time period. Based on our preliminary test results, virtual threads and coroutines can support significantly higher concurrency levels with lower latency periods, making them more efficient concurrency models regarding thread creation, server response time, and heap usage.

VI. Discussion and Future Research

One of the primary goals of concurrency is to optimize the usage of multi-core processors and increase the application performance with the same amount of resources. However, the release of new concurrency approaches, including the development of new implementations, such as virtual threads, might also bring other benefits.

Recently, virtual machines with the ability to automate different areas of program execution, such as memory management and garbage collection, have gained popularity within the JVM and .NET Framework's CLR ecosystems. Moreover, contemporary polyglot VMs enable the developers to take advantage of the polyglottic combination and synergy of different programming languages. One such example is Project Metropolis' open-source polyglot virtual machine GraalVM, mainly written in Java. Based on the previous comparison of GraalVM with the standard VM in JDK 11, it exhibited better performance on several concurrent benchmarks [12]. This means that further development of the VMs and the addition of new languages, such as Kotlin, could bring the benefits of structured concurrency into polyglot programming, enabling the developers to combine different programming languages, such as Java and Kotlin, to achieve high performance in a simpler manner.

Furthermore, based on our test results and previous research in automatic memory management, since virtual threads and coroutines are releasing their OS kernel threads significantly faster, they could improve the memory management process. One of our previous research projects explored the performance of different JDKs in combination with several Garbage Collectors (GCs), presenting the differences in application performance based on the usage of different garbage collector algorithms [13]. Because all JVM threads are able to access the heap, the improvement in the synchronization speed between threads leads to the better management of the objects stored in a heap and their removal by GC. Project Loom is currently using the standard G1 GC, and it is yet to be seen how other GCs could perform with virtual threads. Therefore, our future research will focus on a more detailed analysis of certain aspects of automatic memory management and various garbage collection algorithms. More specifically, exploring the differences between virtual threads' and coroutines' behavior and performance in that regard. We are also considering more general research on the optimization of the automatic memory management process within the project Loom's JDK, which could lead to better usage of resources and further improvement of application performance.

VII. Conclusion

By the comparison of Kotlin's coroutines and original JVM-based threads, it is evident in our benchmark's preliminary test results that structured concurrency has shown a significant improvement in the performance of concurrent applications written in Kotlin. Analogously, based on our preliminary results, a similar conclusion can be drawn when comparing new virtual threads and standard threads in Java. We have a reason to believe that similar outcomes could be expected of virtual threads after being released within the standard JDK release, presumably within the following year. The final release of Project Loom-extended JDK, with embedded virtual threads as a standard option, would demonstrate the benefits of structured concurrency usage in real-life applications. It could be the anticipated improvement of performance to existing applications, entirely or partially (polyglot) written in Java, without significant changes to the application source code.

Although it was not yet possible to perform more comprehensive comparisons of Java's virtual threads and Kotlin's coroutines on a more extensive number of appropriate use cases and benchmarks, our preliminary results already present some interesting findings. Coroutines have proven to be a bit more stable option in regards to OS kernel threads usage. On the contrary, virtual threads have better results from the perspective of heap space usage due to more specific schedulers. In regards to latency, both constructs have demonstrated the ability to endure higher levels of concurrent requests. Therefore, the decision between these concurrency constructs, as well as the selection of the main programming language for implementation, is up to software developers, primarily depending on the characteristics of an application under development. However, when developing modern software applications, it would be recommended to explore all the advantages that new structured concurrency constructs such as coroutines and virtual threads could bring, and, particularly in the case of Java applications, take into account any other potential improvements and optimizations of virtual threads that could happen when they officially become a standard part of the JDK.

REFERENCES

[1] L. Prechelt, "An empirical comparison of seven programming languages," in Computer, vol. 33, no. 10, pp. 23-29, Oct. 2000, doi: 10.1109/2.876288.

[2] R. Pressler, "State of Loom," May 2020. [Online]. Available: https://cr.openjdk.java.net/~rpressler/loom/loom/sol1_part1.html Accessed on: 15-Jan-2022.

[3] K. Lei, Y. Ma and Z. Tan, "Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js," 2014 IEEE 17th International Conference on Computational Science and Engineering, 2014, pp. 661-668, doi: 10.1109/CSE.2014.142.

[4] A. Zentner, "Multithreading in .Net and Java: A Reality Check", Journal of Computers, pp. 426-441, 2018. doi: 10.17706/jcp.13.4.426-441.

[5] N. Togashi and V. Klyuev, "Concurrency in Go and Java: Performance analysis," in Proc. of the 4th IEEE Int. Conference on Information Science and Technology, Shenzhen, 2014, pp. 213-216.

[6] P. Abhinav, A. Bhat, C. Joseph and K. Chandrasekaran, "Concurrency Analysis of Go and Java", in Proc. of the 5th Int. Conference on Computing, Communication and Security (ICCCS), 2020. doi: 10.1109/icccs49678.2020.9277498.

[7] K. Chauhan, S. Kumar, D. Sethia and M. Alam, "Performance Analysis of Kotlin Coroutines on Android in a Model-View-Intent Architecture pattern", in Proc. of the 2nd Int. Conference for Emerging Technology (INCET), 2021. doi: 10.1109/incet51464.2021.9456197.

[8] D. Beronić, P. Pufek, B. Mihaljević and A. Radovan, "On Analyzing Virtual Threads – a Structured Concurrency Model for Scalable Applications on the JVM," in Proc. of the 44th Int. Convention on Information, Communication and Electronic Technology (MIPRO), 2021, pp. 1684-1689, doi: 10.23919/MIPRO52101.2021.9596855.

[9] P. Pufek, D. Beronić, B. Mihaljević and A. Radovan, "Achieving Efficient Structured Concurrency through Lightweight Fibers in Java Virtual Machine," 2020 43rd Int. Convention on Information, Communication and Electronic Technology (MIPRO), 2020, pp. 1752-1757, doi: 10.23919/MIPRO48935.2020.9245253.

[10] B. Evans, "Going inside Java's Project Loom and virtual threads" Java magazine, January, 2021.

[11] R. Elizarov, M. Belyaev, M. Akhin, and I. Usmanov, "Kotlin Coroutines: Design and Implementation", Proc. of the 2021 ACM SIGPLAN Int. Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!' 21), Chicago, IL, USA, ACM, New York, NY, USA, 2021, pp. 68-84, doi: 10.1145/3486607.3486751

[12] M. Šipek, B. Mihaljević, and A. Radovan, "Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM," in Proc. of the 42nd Int. Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2019, pp. 1671-1676, doi: 10.23919/MIPRO.2019.8756917

[13] P. Pufek, H. Grgić and B. Mihaljević, "Analysis of Garbage Collection Algorithms and Memory Management in Java," in Proc. of the 42nd Int. Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2019, pp. 1677-1682, doi: 10.23919/MIPRO.2019.8756844.