Automated Runtime Transition between Virtual and Platform Threads in the Java Virtual Machine

Andrea Rosà, Matteo Basso, Leonardo Bohnhoff, Walter Binder
Università della Svizzera italiana (USI), Switzerland
Email: andrea.rosa@usi.ch, matteo.basso@usi.ch, leonardo.bohnhoff@usi.ch, walter.binder@usi.ch

2023 30th Asia-Pacific Software Engineering Conference (APSEC) | 979-8-3503-4417-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/APSEC60848.2023.00080

***Abstract*—*Virtual threads* are a new feature of the Java Virtual Machine (JVM) complementing the regular Java threads (called *platform threads*). Virtual threads promise a significant throughput improvement in workloads that make intensive use of blocking operations, but incur performance overheads in CPU-bound workloads, due to the cost of creating and maintaining the data structures associated with them. Hence, the optimal choice between platform and virtual threads depends on application behavior (which can change significantly during the execution) and on the system running the application.**

This work-in-progress paper proposes a new framework to automatically and adaptively select the type of threads created by an application, switching between platform and virtual threads as the application's behavior changes over time. The framework monitors several metrics of the running application. Based on these metrics, users can specify criteria for deciding when virtual threads should be preferred over platform threads and vice versa. We present a use case on the Eclipse Jetty web server. Our preliminary evaluation shows that the framework selects the thread type yielding the highest throughput depending on the application behavior, without introducing performance drawbacks.

***Index Terms*—Virtual threads, platform threads, multithreading, adaptive system, Java Virtual Machine.**

I. Introduction

Virtual threads are a new feature of the Java Virtual Machine (JVM), introduced in JDK 19 [1]. They aim at improving the scalability of high-throughput concurrent server applications involving the intensive use of blocking operations. Virtual threads exhibit the same interface of regular Java threads (called *platform threads*) and can be used interchangeably with them.

Virtual threads are implemented as lightweight threads running on top of platform threads, which are in turn wrappers around operating system (OS) threads. When a platform thread executes a *blocking operation*, the OS puts the thread into the WAITING state [2], scheduling the thread out of execution and performing an expensive context switch. Instead, when a virtual thread executes a blocking operation, the underlying platform thread is not blocked: the virtual thread is suspended, but the platform thread continues the execution by starting a new virtual thread or resuming a previously suspended one.

This design leads to a better resource utilization particularly in the case of applications performing many blocking operations, as demonstrated by the increasing adoption of virtual threads in web servers (such as Eclipse Jetty [3] and Helidon [4]) and frameworks facilitating the development of web, cloud, and microservice-oriented applications (including Spring Framework [5], Spring Boot [6], and Quarkus [7]). On the other hand, virtual threads incur overheads related to the creation and maintenance of the data structures used to support them, and they may result in performance penalties if the cost of using them is not counterbalanced by the avoidance of many expensive context switches. Deciding which *thread type* (i.e., platform or virtual threads) leads to the highest throughput is generally very hard [8], [9], [10], [11], as it greatly depends on the application behavior and on the underlying system. Particularly in web, cloud, and microservice-oriented applications, the application behavior depends heavily on the rate and nature of the requests made by the users, which are very likely to change over time. Hence, virtual threads may perform better in some phases of the execution, while platform threads may be preferred in other phases.

Our work aims at supporting this class of applications. In this work-in-progress paper, we propose a new framework for the JVM able to automatically adapt the type of threads employed as the application's behavior changes over time (§III). The framework periodically monitors the activity of each spawned thread, tracking several relevant metrics. Based on them, users can define criteria to classify one thread type as *preferred*. Periodically, the framework determines which is the current preferred thread type and adapts threads with non-preferred type to the preferred one. To do so, the framework requires only few small code modifications to support a simple transition protocol. New threads created via the framework will always be of the preferred type. We implement a prototype of the proposed framework as an extension of JDK 20 (i.e., the latest Java release at the time of writing).

We demonstrate the benefits of using our framework on Jetty [3], a production-standard widely adopted web server [12], [13], [14] (§IV). Our preliminary evaluation results show that the framework applies transitions to the thread type yielding the highest throughput depending on the application's behavior, without introducing performance penalties.

We complement the paper with necessary background information (§II), a discussion of related work (§V), and our concluding remarks (§VI).

II. Background

Virtual threads are implemented in the class java.lang.VirtualThread, a subclass of java.lang.Thread (which represents a platform thread). As such, virtual threads can run any code which can be run by platform threads. When needed, the virtual thread *scheduler* selects a platform thread

p on top of which a virtual thread v will run. This operation is called *mounting*, while p is denoted as the *carrier* of v. Each virtual thread is associated with a *continuation*, a structure able to capture, suspend (called *freezing*), and resume (called *thawing*) the current execution context of a thread. When a virtual thread executes a blocking operation, the scheduler *unmounts* the virtual thread from its carrier and freezes the corresponding continuation. Then, it selects another virtual thread ready to be executed (whose corresponding continuation is thawed), mounting it on the platform thread, thus avoiding blocking the carrier.[1] Once the blocking operation is completed, the scheduler assigns the virtual thread to a possibly different carrier to resume execution. Internally, the scheduler is implemented as a dedicated ForkJoinPool [15] using platform threads as workers. In contrast, when a blocking operation is performed on a platform thread, the thread will be *parked*, i.e., it is put into the WAITING state [2] and becomes inactive, which will lead to an OS context switch, assigning the core to another thread ready to execute.

III. The Proposed Framework

Here we describe the proposed framework using a running example. Fig. 1 shows the (very simplified) code of a custom thread pool. Black code denotes the original implementation of the thread pool, red code the modifications that we introduce to use our framework, and gray code original lines removed by our modifications. Workers in the pool are created and started via method startThread (lines 25–36), which specifies a Runnable (l. 26–33) dictating the behavior of each worker. In the original implementation, workers continuously take one task out of a non-exclusive queue (l. 30) and execute it until the pool is active (l. 28), after which point they terminate.

```
1  public class Pool {
2   private Set<Thread> threads;
3   private AtomicBoolean active;
4   private AdaptiveThreadFactory atf =
5      createATF();
6
7   private ThreadTypeSelector createSelector() {
8    return (rBlocking, CPU, rCreated, nActive,
9       currentType) -> {
10    //Enum name omitted for brevity
11    if (rBlocking < 600) return PLATFORM;
12    if (rBlocking >= 1100) return VIRTUAL;
13    if (CPU < 0.75) return VIRTUAL;
14    return currentType;};
15  }
16
17  private AdaptiveThreadFactory createATF() {
18   atf = new AdaptiveThreadFactory();
19   atf.setThreadTypeSelector(createSelector());
20   atf.setThreadCreationHandler(
21      this::startThread);
22   return atf;
23  }
24
25  private void startThread() {
26   Thread thread = new Thread( () -> {
27   Thread thread = atf.newThread( () -> {
28    while(active.get() && !AdaptiveThreadFactory
29       .isMarkedForTransition(Thread.currentThread())){
30     pollAndExecuteNewTask();
31    }
32    threads.remove(Thread.currentThread());
33   });
34   threads.add(thread);
35   thread.start();
36  }
37 }
```

Fig. 1. Example of a class making use of the proposed framework.

```
1  public class AdaptiveThreadFactory implements
2      ThreadFactory, AutoCloseable {
3
4    public enum ThreadType {PLATFORM, VIRTUAL,
5        NONE}
6
7    @FunctionalInterface
8    public interface ThreadTypeSelector {
9      ThreadType select(int rBlocking,
10       double CPU, int rCreated, int nActive,
11       ThreadType currentType);
12   }
13
14   public Thread newThread(Runnable
15       runnableToExecute) {...}
16   public void setThreadTypeSelector(
17       ThreadTypeSelector selector) {...}
18   public void setThreadCreationHandler(Runnable
19       threadCreationHandler) {...}
20   public static boolean
21       isMarkedForTransition(Thread thread) {...}
22   ...
23 }
```

Fig. 2. Relevant components of the AdaptiveThreadFactory API.

A. *The* AdaptiveThreadFactory *API*

The framework exposes a new class AdaptiveThreadFactory (shown in Fig. 2), which implements the interfaces java.util.concurrent.ThreadFactory and java.lang.AutoCloseable offered by the Java Class Library. ThreadFactory exposes the method newThread, returning a new instance of java.lang.Thread. Calling newThread (l. 14–15) on an instance of AdaptiveThreadFactory returns either a platform or virtual thread, depending on the decision made by the framework. An AdaptiveThreadFactory maintains a set of references to the threads that it has created and are currently running. An application may instantiate multiple instances of AdaptiveThreadFactory; each of the latter monitors only the activity of the threads it has created.

B. *Monitored Metrics*

An AdaptiveThreadFactory continuously monitors and records several metrics on the activity of the threads it has created or the system state. Our framework prototype currently supports the monitoring of the following metrics, each of which can selectively be disabled if not needed:

- *Rate of blocking operations* (rBlocking). We measure the number of blocking operations performed by a thread in a time interval T_{blocking}, regardless of whether they result in a thread park or a continuation freeze.
- *CPU utilization* (CPU). We track the average system CPU utilization, provided by a

[1]Unless there is no virtual thread ready to be executed, in which case the platform thread will be blocked. For simplicity, we omit the case in which a virtual thread is *pinned* to its carrier [1].

com.sun.management.OperatingSystemMXBean. We empirically observed that the value returned by the bean can fluctuate significantly. For this reason, we record the last N_{CPU} measurements, each performed at interval T_{CPU}, using the average of these samples as a metric.
- *Rate of created threads* (rCreated). We record the amount of threads created in a time window T_{creation}.
- *Amount of active threads* (nActive). We also measure the number of active (i.e., non-terminated) threads associated with an AdaptiveThreadFactory.

C. Declaring the Preferred Thread Type

Users can specify criteria to determine when one thread type should be classified as *preferred*. To this end, the API exposes the ThreadTypeSelector interface (Fig. 2, l. 8–12). The select method allows one to specify the chosen criteria based on the metrics collected by the framework (§III-B) and the current thread type. Once created, a ThreadTypeSelector can be linked to an AdaptiveThreadFactory via method setThreadTypeSelector (l. 16–17). The ThreadTypeSelector is used to determine the preferred thread type when creating a new thread—in which case the current ThreadType is NONE (l. 4–5)—or periodically (as explained in §III-D). It should be noted that by returning currentType or null from select, the user can denote that there is no preferred thread type under a given condition (the preferred thread type is *undefined*). In this case, the framework will not change thread type. In the example of Fig. 1, the selector is implemented in lines 7–15, and it is set in the AdaptiveThreadFactory in l. 19.

D. Transitioning to the Preferred Thread Type

When an application requests the creation of a new thread via AdaptiveThreadFactory.newThread, the framework creates a thread of the type that is currently preferred depending on the user-defined criteria. In the example shown in Fig. 1, regular thread creation (l. 26) is replaced with a call to newThread (l. 27) to make use of the framework. In case of already executing threads (as long as they were created via an AdaptiveThreadFactory), the framework periodically determines whether their type is classified as preferred. This is conceptually performed by checking, on each active thread, whether the ThreadType returned by the set ThreadTypeSelector differs from currentType (and is not null). In this case, the framework initiates a *transition* where all thread types are adapted to the preferred one, if not already of such type. A transition involves the termination of the currently executing threads with non-preferred type associated with an AdaptiveThreadFactory as well as the creation of a new set of threads of the other type, following the process described below.

Thread creation can be performed in two ways. In the first way, after having terminated a thread, the factory creates a new thread (of the current preferred type) by invoking a *thread creation handler*, i.e., a user-defined Runnable specifying code to be performed for creating the new thread. The handler needs to include at least the invocation of the factory's newThread method and is provided to the factory via method setThreadCreationHandler (Fig. 2, l. 18–19). In Fig. 1, the handler is set to Pool.startThread (l. 20–21). This means that, after the termination of a thread, the framework creates and starts a new thread (of the preferred type) following the same logic that the pool would use when a new worker is needed. In the second way (applied if no handler is specified), the factory does not invoke any handler after the termination of a thread and leaves the creation of a new thread to the client (which must occur via newThread). This variant can be useful to clients already implementing a mechanism for detecting the need for new threads and handling their creation.

To support the termination of the threads associated to a factory, class AdaptiveThreadFactory defines method isMarked-ForTransition, which returns true when a factory has determined that the thread passed as argument is of a non-preferred type and would like to terminate it (Fig. 2, l. 20–21). The user is required to identify *safe locations* in the thread's flow of execution in which the thread can be terminated when marked for a transition. These safe locations need to be chosen such that the application is not left in an inconsistent state. At each such location, the user is required to specify code leading to the regular termination of the thread if isMarkedForTransition returns true. In Fig. 1, a safe location is at l. 28, when the thread evaluates whether the pool is active. Here, if the thread is interrupted due to a transition, the application is not left in an inconsistent state, because the thread has finished processing the current task and the queue is not exclusive to the worker. To support the transition, we simply add a check to isMarkedForTransition in the while condition, which will cause the termination of the loop (and of the thread) if true.

The transition is carried out by a separate thread, which periodically (with time interval $T_{\text{transition}}$) determines whether some threads have non-preferred type. If a type is determined to be non-preferred for $N_{\text{transition}}$ times in a row, the framework initiates a transition. As there could be a minor performance penalty associated with a transition, we implement this mechanism to avoid changing thread type too often, particularly for transient application phases where the tracked metrics could significantly fluctuate but only for a short time.

IV. Use Case: Jetty Web Server

Here, we apply the proposed framework to the Jetty web server, showing preliminary evaluation results.

A. The Jetty Web Server

Jetty relies on a thread pool [16] to execute tasks (i.e., requests for web pages submitted by users). We exercise Jetty via a publicly available benchmark [17], composed of the web server itself and a load generator, which sends requests for retrieving web pages to the server. Both components can be extensively parametrized to simulate user-defined workloads. The important parameters for our work are the following (we modified the benchmark to introduce parameters not included in the original version).
- minThreads and maxThreads: minimum (resp. maximum) number of workers in the thread pool.

- requestRate: the number of requests sent to the web server by the load generator every second.
- blockingOps: number of blocking operations required to process an user request.
- blockingTime: the expected time a blocking operation needs to complete.
- processingTime: the expected time required to process the request, excluding the time spent in blocking operations.

On average, each request is expected to take processingTime $+$ blockingOps $\cdot$ blockingTime to be processed by the server. We modified Jetty's thread pool such that the workers can benefit from the proposed framework. The code shown in Fig. 1 actually shows the core of Jetty's pool (although simplified). The modifications performed to integrate our framework in Jetty are very similar to those shown in the figure.

B. Evaluation Methodology

We execute the benchmark in multiple settings, varying the parameters maxThreads, requestRate, and blockingOps to generate different workloads. Table I reports the chosen values (including those of the other customizable parameters of the framework and Jetty). Each experiment is carried out as follows. When a new resourceRate is set, the load generator takes some time to reach the target resourceRate. We let the rate stabilize to the target value, then we measure the number of requests processed by the server in a period of 60 seconds. This number divided by the period provides the *throughput* of the execution, which we use as the main evaluation metric in this section.

We use the ThreadTypeSelector shown in Fig. 1 (l. 7–15). The criteria were derived through extensive experimentation, after measuring the throughput of both platform and virtual threads under different workloads on the target machine.[2]

We explain the criteria as follows. Virtual threads may avoid expensive context switches, but they incur overheads related to the creation and maintenance of the data structures (i.e., continuations) used to support them. When the rate of blocking operations is limited ($<$600), the overhead of using them is higher than the performance gain in avoiding context switches. In contrast, their cost pays well when a significant amount of blocking operations ($\geq$1100) is performed. We observed that when the rate is between 600 and 1100, the thread type yielding the highest throughput fluctuates greatly. However, we noticed that platform threads tend to underutilize the CPU significantly more than virtual threads (due to the higher number of context switches). In cases where the CPU utilization drops below 75%, we observed that virtual threads consistently ensure higher throughput than platform threads, and hence we classify them as the preferred type. For a higher CPU utilization we did not observe a clear trend, and we leave the preferred type undefined.

For each setting, we execute three variants of the web server: 1) using our framework; 2) using only platform threads; 3) using only virtual threads. The latter two are evaluated to compare their throughput with the first one and to verify which thread type is *optimal* or *suboptimal* (i.e., leads to the best or worst throughput, respectively) in a given scenario. We execute each experiment five times to account for possible variance in the performance. The values reported in the next section are the average of these five runs.

For some settings, the preferred thread type is unspecified and the framework does not apply any transition. We manually verified that for all these settings, the difference between the throughput when running platform and virtual threads is lower than 10%, which confirms that both thread types provide similar performance. The following section focuses on the settings where the framework determines a preferred thread type.

[2]All experiments are carried out on a machine equipped with an Intel Xeon CPU E5-2680 (2.70 GHz) processor, supporting two NUMA nodes, each with 8 cores and 64 GB of RAM. The web server is exclusively executed in one NUMA node, while the load generator in the other, to limit interference and perturbation between the two processes. We use Ubuntu 20.04.5 LTS and set the CPU governor to “performance” to disable frequency scaling.

TABLE I
CONSIDERED VALUES OF THE FRAMEWORK'S AND JETTY'S PARAMETERS.

Parameter	Values	Parameter	Values
T_{blocking}	200 ms	T_{creation}	200 ms
T_{CPU}	100 ms	N_{CPU}	5
$T_{\text{transition}}$	1500 ms	$N_{\text{transition}}$	5
minThreads	10	blockingOps	0, 1, 2, 3, ..., 8
requestRate	1200, 1600, 2000, 2400, ..., 3600	maxThreads	16, 32, 48, 64, 80
blockingTime	0.5 ms	processingTime	5 ms

C. Preliminary Results

The goal of our preliminary evaluation is to 1) verify that the preferred thread type determined by the framework is the one resulting in the highest throughput and 2) assess whether the underlying metric-collection and transition logic of the proposed framework result in a small or negligible performance penalty. To evaluate the latter point, we check whether the throughput using our framework is close to the one obtainable by employing the optimal thread type directly (without using the proposed framework).

Fig. 3 compares the throughput of the three web-server versions in all the settings (30) where the framework determined a preferred thread type. We manually validated that in all settings the framework always applies transitions to the thread type yielding the highest throughput. From the figure, we notice that the throughput of the framework is close to the one obtainable by the optimal thread type in all settings. The highest slowdown factor (i.e., the ratio of the two throughputs, using the one of our framework as denominator) is 1.016$\times$ (setting 19), while the smallest is 0.923$\times$ (setting 1). The average slowdown factor is 0.994$\times \pm$ 0.02 (actually a speedup), which demonstrates that, on average, using the proposed framework incurs in negligible or no overhead and can be used in production-standard applications without performance penalties.

V. RELATED WORK

Virtual threads are a recent addition to the JVM; as such, the research community has little focused on them so far. Pufek

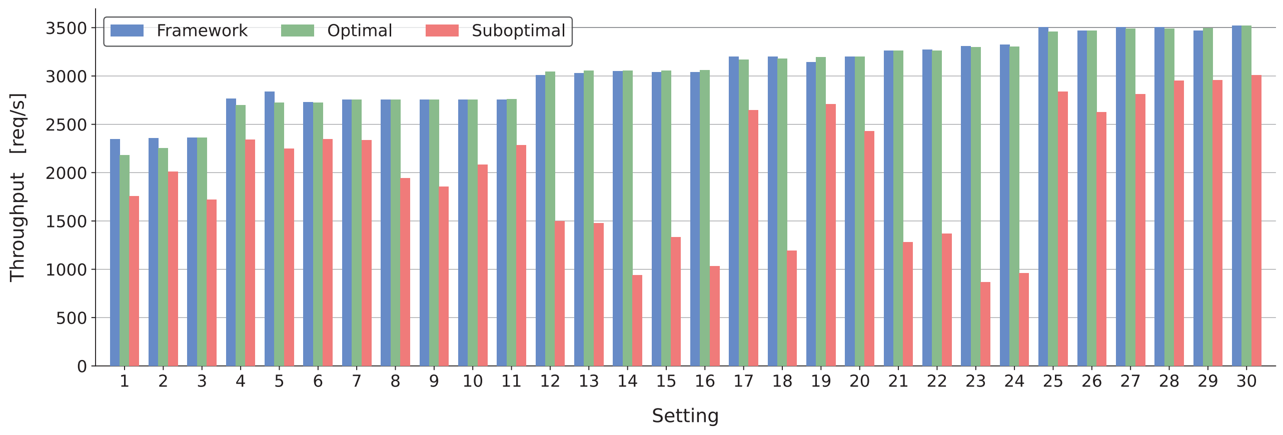

Fig. 3. Comparison of throughput obtainable by the proposed framework, the optimal and the suboptimal thread type.

et al. [18] and Beronić et al. [19] explore the performance of platform and virtual threads in several scenarios, reporting lower latency for virtual threads (especially for long-running applications) and a lower execution time of parallel sorting algorithms when virtual threads are employed. In another work, Beronić et al. [20] compare the performance of virtual threads with Kotlin's coroutines, showing that they are superior to both platform threads and Kotlin's regular threads in terms of latency and heap-space consumption. Navarro et al. [21] report their experience in integrating virtual threads into Quarkus [7]. Differently from the above work, we propose a framework able to adapt thread type based on dynamic application- and system-level metrics. We are not aware of other work proposing a similar technology for virtual threads on the JVM.

VI. Conclusions

In this work-in-progress paper, we presented a novel framework to automatically adapt the type of threads used by the application over time, according to user-defined criteria discriminating the preferred thread type. These criteria are based on application- and system-level dynamic metrics monitored by the framework. We also presented a use case where we use our framework in the Jetty web server, showing that the framework applies transitions to the thread type yielding the highest throughput without introducing performance drawbacks.

One limitation of our work is that the criteria used for the evaluation have been obtained experimenting on a single application and machine and are therefore workload- and platform-dependent. This paper does not aim at proposing platform-independent criteria, which requires much more extensive experimentation. It is also likely that a more general criteria should consider hardware characteristics (e.g., the number of CPU cores) that are currently not supported by the ThreadTypeSelector API and therefore cannot be used to determine the preferred thread type. We are actively working to address these open issues.

As part of our future activities, we plan to expand the metrics tracked by the framework (and the ThreadTypeSelector API) to support more flexible criteria. We also plan to evaluate the framework on additional use cases.

Acknowledgments

The research presented in this paper was supported by Oracle (ERO project 1332) and by the Swiss National Science Foundation (project 200020_188688).

References

[1] "JEP 444," https://openjdk.org/jeps/444, 2023.
[2] "Class Thread.State," https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/lang/Thread.State.html#WAITING, 2023.
[3] "Eclipse Jetty," https://eclipse.dev/jetty/, 2023.
[4] "Helidon," https://helidon.io/, 2023.
[5] "Spring Framework," https://spring.io/, 2023.
[6] "Spring Boot," https://spring.io/projects/spring-boot, 2023.
[7] "Quarkus," https://quarkus.io/, 2023.
[8] "JEP 425," https://openjdk.org/jeps/425, 2022.
[9] "Jetty 12 – Virtual Threads Support," https://webtide.com/jetty-12-virtual-threads-support/, 2023.
[10] "Embracing Virtual Threads," https://spring.io/blog/2022/10/11/embracing-virtual-threads, 2022.
[11] "Writing Simpler Reactive REST Services with Quarkus Virtual Thread Support," https://quarkus.io/guides/virtual-threads, 2023.
[12] "Usage of Eclipse Jetty in Spring Boot," https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started.html, 2023.
[13] "Usage of Eclipse Jetty in Eclipse IDE," https://eclipse-jetty.github.io/, 2023.
[14] "Usage of Eclipse Jetty in Apache Hadoop," https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/dependency-analysis.html, 2023.
[15] "Class ForkJoinPool," https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/concurrent/ForkJoinPool.html, 2023.
[16] "Class QueuedThreadPool – Jetty 10," https://eclipse.dev/jetty/javadoc/jetty-10/org/eclipse/jetty/util/thread/QueuedThreadPool.html, 2023.
[17] "jetty-load-base," https://github.com/jetty-project/jetty-load-base, 2023.
[18] P. Pufek, D. Beronić, B. Mihaljević, and A. Radovan, "Achieving Efficient Structured Concurrency through Lightweight Fibers in Java Virtual Machine," in *MIPRO*, 2020, pp. 1752–1757.
[19] D. Beronić, P. Pufek, B. Mihaljević, and A. Radovan, "On Analyzing Virtual Threads – a Structured Concurrency Model for Scalable Applications on the JVM," in *MIPRO*, 2021, pp. 1684–1689.
[20] D. Beronić, L. Modrić, B. Mihaljević, and A. Radovan, "Comparison of Structured Concurrency Constructs in Java and Kotlin – Virtual Threads and Coroutines," in *MIPRO*, 2022, pp. 1466–1471.
[21] A. Navarro, F. L. Mouël, J. Ponge, and C. Escoffier, "Considerations for Integrating Virtual Threads in a Java Framework: a Quarkus Example in a Resource-constrained Environment," in *DEBS*, 2023, p. 103–114.