Considerations for integrating virtual threads in a Java framework: a Quarkus example in a resource-constrained environment

Arthur Navarro
arnavarr@redhat.com
Red Hat, Univ Lyon, INSA Lyon, Inria, CITI, EA3720
Villeurbanne, France

Julien Ponge
jponge@redhat.com
Red Hat
Lyon, France

Frédéric Le Mouël
frederic.le-mouel@insa-lyon.fr
Univ Lyon, INSA Lyon, Inria, CITI, EA3720
Villeurbanne, France

Clément Escoffier
cescoffi@redhat.com
Red Hat
Valence, France

ABSTRACT

Virtual threads are a highly anticipated feature in the Java world, aiming at improving resource efficiency in the JVM for I/O intensive operations while simplifying developer experience. This feature keeps the traditional thread abstraction and makes it compatible with most of the existing Java applications, allowing developers preferring synchronous imperative abstractions to benefit from better performance without switching to asynchronous and reactive programming models. However, limitations currently hinder the usability of virtual threads. These limitations must be considered when building a piece of software around virtual threads for they might have non-trivial effects. This paper (i) discusses the different strategies envisioned to leverage virtual threads in the *Quarkus* framework, (ii) gives an overview of the final implementation, (iii) presents the benchmark used to characterize the benefits of using virtual threads in a typical container environment where resources are scarce compared to using Quarkus with traditional thread pools and Quarkus with reactive libraries ; (iv) results are interpreted and discussed. Our study reveals that the integration of virtual threads in Quarkus doesn't perform as well as Quarkus-reactive. This seems to be due to a mismatch between the core hypothesis of Netty and virtual threads regarding the amount of threads available.

CCS CONCEPTS

• **Software and its engineering**;

KEYWORDS

reactive programming, java, benchmarking, virtual threads, concurrency

DEBS '23, June 27–30, 2023, Neuchatel, Switzerland

ACM ISBN 979-8-4007-0122-1/23/06…$15.00
https://doi.org/10.1145/3583678.3596895

ACM Reference Format:
Arthur Navarro, Julien Ponge, Frédéric Le Mouël, and Clément Escoffier. 2023. Considerations for integrating virtual threads in a Java framework: a Quarkus example in a resource-constrained environment. In *The 17th ACM International Conference on Distributed and Event-based Systems (DEBS '23), June 27–30, 2023, Neuchatel, Switzerland*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3583678.3596895

1 INTRODUCTION

The emergence of distributed applications and services has brought forth significant challenges that cannot be effectively addressed by traditional monolithic architectures. Unequal and substantial traffic distribution across different parts of an application renders scaling a monolith uniformly inefficient. Additionally, the complexity of applications has increased, necessitating larger teams of developers working on various features and concerns. Furthermore, the process of deploying an application for every update or fix is burdensome and calls for an alternative approach. These challenges, encompassing scalability, modularity, resilience, and agility, have been addressed by decomposing large monoliths into multiple loosely coupled components [6, 7]. Consequently, modern applications are often composed of numerous cooperating microservices [6]. Cloud computing has enabled the hosting of a vast array of services at a lower cost, with the ability to dynamically allocate resources based on traffic demands. However, the pricing model of cloud computing services favors low-resource services due to their finer-grained control over expensive resources, such as memory and CPU time [23]. As such, this study focuses on designing, integrating, evaluating easy-to-use programming abstractions to achieve efficient event processing of distributed asynchronous applications in resource-constrained environments.

Network communications can incur significant costs, making efficient and timely communication between microservices a critical factor. Historically, better performance have been achieved by using non-blocking, asynchronous tools provided by the Operating System [4, 11]. This has led to the development of asynchronous user APIs such as asynchronous *callbacks*, *promises*, and *reactive streams*. However, this paradigm represents a fundamental shift from the traditional synchronous and imperative approach, potentially leading to increased development and maintenance overhead [10, 12]. Consequently, mainstream programming languages like Golang,

C#, Python, JavaScript, and Java are increasingly emphasizing the provision of asynchronous communication primitives that strike a balance between preserving developer experience and facilitating asynchronous communication effectively. Therefore, it is crucial to be able to characterize and compare different libraries or languages when constructing a distributed application in order to ensure that any cost savings achieved through the use of asynchronous programming models are not negated by an increase in maintenance costs. In Java for instance, several approaches exist to build communicating services. Strategies such as *pooling* have spread to alleviate the performance issues of the traditional threading model while experiments with *callbacks*, *futures* and *reactive extensions* were led and entire frameworks such as *Netty* [1] became building blocks for many higher-level pieces of software.

The Java language just introduced *virtual threads* [2]. This solution adds a layer of indirection between the threading model and the underlying non-blocking machinery to offer both a synchronous, imperative user API *and* better performance by leveraging non-blocking asynchronous primitives in the internals of the JVM. It aims at increasing the performance of *existing* Java applications using synchronous primitives simply by replacing traditional threads (referred to as platform threads in the rest of the paper) by virtual ones. This could be an important feature for applications built using frameworks relying on the thread and/or thread pool abstraction, but the Java ecosystem is split between frameworks using thread pools and reactive frameworks built above Netty. While reactive frameworks already provide superior performance, incorporating a synchronous programming model may enhance their development and maintenance.

This paper presents a novel approach for integrating virtual threads into a Java reactive framework, which is the first contribution of the study. The proposed integration is expected to enable users to develop resource-efficient programs in a more straightforward manner, thereby reducing the costs associated with both development and maintenance. The *Quarkus* [3] framework was chosen due to its core reactive design, and the primary focus on cloud-native applications, hence its relevance to low-resource environments.

Our contributions are:

- investigating virtual threads integration in Netty-based frameworks and implementing a solution in Quarkus,
- designing an experimental protocol for testing virtual threads integration in Quarkus,
- evaluating the performance of the implementation in terms of latency, CPU usage, memory footprint, and comparing it with existing programming models.

The structure of this paper is the following. Background information about the impact of message-driven architectures is given in Section 2 along with a description of virtual threads. This section also contains details about the implementation similar constructs in other programming languages as well as our integration of *Quarkus-virtual-threads*. Section 3 describes the experimental protocol used in this study. Results are presented in Section 4. A discussion highlighting the limitations and potential enhancements also appears in this section. A comparison with the existing frameworks and benchmarks is presented in Section 5. Eventually, Section 6 summarizes the findings and discusses broader implications for reactive frameworks and virtual threads.

2 BACKGROUND

This introduction to resource management and modern abstraction aims at clarifying the goals, the constraints and the limitations of the current ecosystem. Through the limitations of both the legacy blocking imperative paradigm and the new non-blocking one, this introduction also presents the motivation for virtual threads.

2.1 Efficient managing of resources in message-driven systems

Processes involved in a distributed system spend most of their time performing I/O operations. This is especially true for microservices architectures where each microservice has a narrow functional scope, and where it must frequently cooperate with other microservices to achieve progress [6, 8]. In this context, efficient management of resources and scalable handling of network I/O is crucial [4]. Efficient resource management is even more important when we consider cloud environments where resources are scarce [23]. Many abstractions to efficiently perform network I/O are embedded in modern Operating Systems: EPoll for Linux, KQueue for macOS, FreeBSD, NetBSD, and I/OCP for Windows.

2.1.1 The old blocking model. The hegemonic model to handle concurrency that led to the C10K problem [11] was to spawn a new *worker* thread per incoming request. Each thread was supposed to execute the request handler logic. If I/O operations had to be performed in the handler, the worker thread would perform a *blocking syscall* that would give control back to the OS scheduler. The thread would then be put in a *wait queue* during the entire operation duration.

In Java, the class **java.lang.Thread** is a wrapper around OS threads. Hence, creating a Java thread implicitly creates an OS-thread. An operation over the network can last several seconds, during which the worker would remain idle. It is a problem due to the following:

(1) spawning a new thread takes time,
(2) it is impossible to accumulate an arbitrary number of idle threads in memory at any given time,
(3) performing a context-switch (loading the state of another thread in CPU registers) is not free.

2.1.2 Leveraging non-blocking primitives brought by modern operating systems. In the early 2000s, the limitations of the threading model led to the formalization of the C10K problem [11], non-blocking communications were put forward as the replacement for the previous model based on concurrent threads. Non-blocking I/O were made available in the JDK in 2002 [4] and leveraged by Netty in 2004. The assumption of non-blocking network I/O is that no CPU work is necessary to wait for an operation to complete (no active

[1] https://netty.io/
[2] https://openjdk.org/jeps/425
[3] https://quarkus.io/
[4] https://docs.oracle.com/javase/8/docs/api/java/nio/package-summary.html

waiting, the NIC will notify the OS when an event happens). What we describe here is a simplified vision of the *Reactor Pattern* [20].

In Linux for instance, *EPoll* is the mainstream non-blocking I/O facility although the recent *io_uring* model is getting more and more traction [5]. With EPoll, processes interested in performing I/O operations register what events on what file descriptors they are interested in to the *epoll instance* and can then interrogate the EPoll instance through certain syscalls. This primitive is often used by the *event-loop model* where a thread *loops* over a queue of tasks and checks the completion of pending I/O operations when it has time to do so. Thus, **an event-loop must never be blocked**: if long-running tasks are performed on the event-loop then it won't be able to check the completion of I/O operations and make progress. This has direct effects on the programming model: asynchronous callbacks, promises, futures, etc. emerge since **a function to call asynchronously must be called** when an event happens, thus leaving the imperative paradigm.

However, the Java ecosystem didn't entirely shift to this programming model. Blocking, imperative paradigm is still widely used and strategies such as pooling have been refined to mitigate the cost of creating new threads as well as limiting the memory consumption by capping the size of the pool (i.e. the number of threads within). Since 2005, frameworks relying on blocking communications in thread pools and frameworks relying on non-blocking communications coexist.

2.2 Non-blocking mechanisms and imperative code

The available options were either to write inefficient imperative, synchronous, blocking code, which is generally considered simpler by most programmers, or to write asynchronous, non-blocking, sometimes declarative code (particularly when using libraries such as reactive streams) code, which is generally perceived as more complex by most programmers. Experiments were led to design abstractions that would benefit from both paradigms: non-blocking synchronous coding style.

2.2.1 Non-blocking imperative communications designs. As mentioned in the introduction 1, various programming languages have implemented non-blocking imperative abstractions with different levels of consistency. Here, we present an overview of the most popular approaches.

On one hand, Python, JavaScript, C# and Kotlin employ the design of `async` *functions*, where non-blocking imperative code restricted to special functions marked with a specific keyword. This implementation introduces some transparency limitations, as the code can only reside within these designated functions. Consequently, using third-party libraries asynchronously can become a complex task, often necessitating the conversion of numerous functions into `async` functions. Developers refer to this issue as the "two colors functions" problem[6] These languages share a similar implementation approach: `async` undergo compile-time modifications and are rewritten using state machines in the case of C# and Kotlin, and generators and promises in the case of Python and JavaScript.

[5]https://unixism.net/loti/what_is_io_uring.html

[6]https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/

In contrast, Golang revolves around the concept of *goroutines*, where every unit of execution is represented by a goroutine. It does not differentiate between `async` and non-`async` code. Golang was purposefully designed with this idea in mind, enabling the language and runtime to be built seamlessly around it. Java virtual threads, by modifying the runtime rather than the compiler, bear closer resemblance to goroutines.

2.2.2 Virtual threads. With the introduction of new features in JDK 19, it is promised that the resource efficiency and scalability of non-blocking asynchronous programming can be utilized while still displaying a synchronous imperative API. This is achieved through the concept of *virtual threads*, which are JVM-managed threads that are not bound to OS threads in a one-to-one relationship [16].

Virtual threads are threads: the JVM and debugging tools consider them as such. However, they are not related to OS threads in the same way platform threads are. Creating a virtual thread doesn't require the creation of an OS thread. Instead, virtual threads will be dynamically mounted on and unmounted from platform threads. Just like a platform thread can execute a function, it can now execute a virtual thread. When a platform thread is used to run a virtual thread we call it a *carrier thread*. Since they must be executed by platform threads, virtual threads can also be seen as *Runnable* or *Callable* (Java reification of the concept of task). However, virtual threads are unique in that they can *suspend and resume their execution*. Runnable and Callable must be executed fully, whereas virtual threads have the capacity to store their progress in memory, suspend their execution and resume it later. To do so, virtual threads possess a *Continuation* instance that has the capacity of performing stack manipulation to store/restore its progress in/from memory. This mechanism of suspension/resumption is triggered when a blocking call is issued by the application. The JDK has been partially rewritten such that every blocking call will eventually hit a specific method that saves the virtual thread progress in memory and unmount it from its carrier thread. In the case of I/O, the JDK leverages non-blocking facilities brought by the OS to be notified when the operation completes in order to resume the virtual thread execution.

We consider virtual threads as a promising feature, their introduction is an opportunity to study how they could interact with other non-blocking abstractions. If virtual threads perform reasonably well compared to other non-blocking asynchronous abstractions already available, Java would offer a wide variety of programming paradigms.

The code below illustrates the difference between reactive streams and traditional array manipulation using virtual threads. A list of names and a list of quotes are both fetched from a remote service or from a database in a non-blocking fashion. The list of quotes must be fetched after the request for the list of names completed since it needs to know its size. A quote is then appended to each name, and the resulting list is returned.

2.3 Quarkus-virtual-threads

In this section we summarize the different options envisioned to leverage virtual threads in the Quarkus framework. Quarkus was

```
public Uni<List<Name>> getAllQuoted() {
  var names = getAll().memoize().indefinitely();
  var quotes = names.onItem().transformToUni(list ->
       getQuotes(list.size()));
  return Uni.combine().all().unis(names,quotes).asTuple()
    .onItem().transform(tuple -> {
      var namesList=tuple.getItem1();
      //can await it since it is already resolved
      var quotesList = tuple.getItem2();
      for(int i=0; i < namesList.size();i ++){
         namesList.get(i).firstName += " -
              "+quotesList.get(i);
      }
      return namesList;
    });
}
```

Figure 1: Concatenation of asynchronous results using reactive streams

```
public List<Name> getAllQuotedBlocking() {
  var names = getAll();
  var quotes = getQuotes(names.size());
  for(int i=0; i < names.size();i ++){
     names.get(i).firstName+= " - "+quotes.get(i);
  }
  return names;
}
```

Figure 2: Concatenation of asynchronous results using virtual threads

designed around an ecosystem of modular extensions, and we implemented virtual threads support in the *resteasy-reactive extension* [7] which is used for exposing HTTP endpoints. Quarkus relies on *Eclipse Vertx* [8][14], which itself relies on *Netty* [9] for non-blocking I/O. It is designed to work with an ecosystem of extensions, most of them leveraging the Vert.x non-blocking primitives. Its HTTP routing layer also uses the Vert.x router. Thus, an entire non-blocking stack is in charge of:

- listening to incoming HTTP requests, and
- routing the requests to correct endpoints and application code.

On the other hand, user code such as endpoint handlers can be executed either in a reactive way on the event-loops (by using asynchronous APIs and *not performing blocking calls*), or in a blocking way (by offloading the computation to a pool of worker-threads). Three options were available in order to leverage virtual threads. They are discussed in the next section Table 1 summarizes the pros and cons of each strategy.

Forking the model of the worker thread. This is the easiest answer. Since Quarkus already provides a blocking execution model with a dedicated pool of threads, it requires minimal modification of the existing code base to add an unbounded pool of virtual threads and use them as workers. To do so, it is possible to use an *Executor* provided by the JDK, designed to create a new virtual thread every time a task is submitted; then submit the execution of this virtual thread to a pool of carrier threads [10]. However, this approach has two major drawbacks. There is now a separate pool of threads that will be used as workers. Hence, the overall number of threads, and thus the memory footprint of the application, is increased. Virtual threads use a FIFO variant of the *ForkJoinPool* executor to schedule virtual threads on carrier threads, it is hard to strictly bound the number of *ForkJoinWorkers* since it is only possible to specify a target level of parallelism (the number of *running* threads, the number of blocked and running worker threads can be bigger). Furthermore, since the routing layer is executed on the event-loop and the endpoint handler on a worker thread, a context-switch is required: remove the event-loop from the CPU to put the carrier in its stead (operation performed by the OS) which could harm performance.

Modifying Netty, the underlying non-blocking machinery. Netty uses a fixed, limited set of threads called *event-loops* and leverages non-blocking facilities of the OS. It also provides a blocking mode (now discarded) where the small set of event-loops is replaced with a bigger pool of worker threads. Modifying the executor of the blocking mode to spawn virtual worker threads instead or platform worker threads was envisioned, but this idea was quickly discarded on the basis that (i) modifying Netty was not feasible for our team, and (ii) the blocking mode is now discarded, using it would require its integration into the supported features of Netty.

Using the event-loops as carrier-threads. Virtual threads require an Executor [11] to be dispatched among their carriers threads. It is possible to specify an executor, including an executor that would use existing event-loops as workers. This idea has the advantage of minimizing the number of threads in the application as well as the number of context switches (no need for the OS to unschedule the event-loop to put a carrier in its place). This strategy is the most promising of the three since (i) it requires little modification of the code base, (ii) it doesn't depend on any external project, and (iii) it minimizes the costs associated with context-switching between event-loops and carrier threads.

Deadlock issue. In this model the event-loop has three functions:

- it must fulfill its function as an event-loop for the entire non-blocking inner-workings of the framework,
- it must fulfill its function as an event-loop for the non-blocking endpoint handlers declared and implemented by the developer,
- it must work as a carrier thread for virtual threads on which certain endpoint handlers (declared and implemented by the developer) will be executed.

It appeared that when a carrier thread shares locks with virtual threads it might cause a deadlock situation. To manage access to

[7] https://quarkus.io/blog/resteasy-reactive/
[8] https://vertx.io
[9] https://netty.io

[10] https://download.java.net/java/early_access/loom/docs/api/java.base/java/util/concurrent/Executors.html#newVirtualThreadPerTaskExecutor()
[11] https://download.java.net/java/early_access/loom/docs/api/java.base/java/util/concurrent/Executors.html

locks, the JDK uses a queue [12]: a thread that can't access a lock will enqueue itself and wait to be notified by the previous member of the queue that the lock has been released. If both the virtual thread and its carrier are waiting for the same lock and the virtual thread is notified first, it will wake up and will want to resume its computation. Its continuation will be submitted to its carrier *that can't execute it since it is already in the queue for the lock*. The virtual thread will be forced to wait for the carrier thread, itself waiting for the virtual thread. This is a deadlock situation, and fixing it would require implementing intricate scheduling strategies in the executor to be able to detect such situations and avoid them. Moreover, the current state of the *JDK* makes it impossible to implement such a strategy without (i) performing bytecode manipulation at compile time, or (ii) opening the access to restricted modules. Although it would be an interesting perspective for future work, it was decided to discard this strategy.

Final choice. The Netty option being practically infeasible for us and the event-loop option presenting a high risk of deadlocks or an intricate executor implementation to solve it, we were led to opt for the worker model.

Strategy	Pros	Cons
Forking worker model	Simple, fits virtual threads model	Context switches
Using event-loop as carrier	No context-switch, Fewer threads overall	Potential deadlocks
Modifying Netty event-loops to be virtual threads	Integration done at the Netty level, Netty-based frameworks would benefit from it	Can't modify Netty upstream, unpredictable effects

Table 1: Comparison of the different Quarkus-virtual-threads options

3 EXPERIMENTAL PROTOCOL

The objective of this experiment is to evaluate the performance of Quarkus services in resource-constrained environments. To achieve this, the services were executed inside Docker containers running Red Hat ubi8 [13] operating system. Each container was allocated 512 MB of RAM and 0.5 CPU cores. The JDK version 19.0.1 was used in this experiment. The host machine used for running these containers was a 64-bit Ubuntu 22.04.2 LTS with 32 GB of memory and equipped with 12 Intel® Core™ i7-10850H CPUs @ 2.70GHz.

Details about how the experiment was designed as well as how each component of the experiment (load generator, database, service) was integrated are given in this section.

3.1 Discriminating tests on a large functional scope

Before designing the benchmark, the *Techempower benchmark suite* [14] was used to rapidly cover a wide spectrum of behaviors. Three versions of the same Quarkus application were compared: (i) the first one was powered by the non-blocking engine built above Netty, (ii) the second one was powered by virtual threads, (iii) the last one was powered by the blocking multithreading model. The initial hypothesis was that Quarkus-reactive was going to perform best for IO-intensive tasks, followed by Quarkus-virtual-threads, and finally Quarkus-blocking. For CPU-intensive tasks, we didn't expect much difference between the three implementations.

This suite measures how a framework responds to different kinds of load: some CPU-bound, some IO bound. We select two tests of the suite, a CPU-bound one and an IO-bound one here.

Test Name	quarkus-reactive	quarkus-v-thread	quarkus-blocking
JSON	43 955	10 682	43 863
FORTUNE	15 607	11 227	10 890

Table 2: Results of the benchmark suite for JSON and FORTUNE tests for each technology (number of requests processed)

Table 2 summarizes the findings. JSON is a CPU-bound test where the application only has to encode a static object in JSON and return it to the client, FORTUNE is an IO-bound, more involved test where the application has to query a database, sort the results and return them as a JSON object to the client. The results were appalling: Quarkus-virtual-threads performed worse than Quarkus-reactive *and* Quarkus-blocking for simply encoding an answer. It offered almost no improvement compared to the blocking version, both being about 33% slower than the reactive version. This benchmark revealed that the way Netty was using *ThreadLocals* put much pressure on the Garbage Collector, especially during serialization. Time spent in GC is time lost for the application, hence the significant difference between Quarkus-reactive and Quarkus-virtual-threads. This stems from the underlying assumption of Netty: the application uses a small set of event-loops (a few threads). The Quarkus-virtual-threads on the other hand, created *thousands* of virtual threads, leading to the creation of thousands of ThreadLocals instances.

Test Name	quarkus-reactive	quarkus-v-thread	quarkus-block
JSON	43 450	42 736	40,437
FORTUNE	17 673	16 011	11 461

Table 3: Results of the benchmark suite for JSON and FORTUNE tests for each technology after correction (number of requests processed)

Table 3 describes the results after fixing Quarkus-virtual-threads. The behavior of Netty was modified and doesn't rely as much on the creation of ThreadLocals. The improvement is significant: the performance of Quarkus-virtual-threads is multiplied by 4 for CPU-intensive tasks and by 1.5 for IO-intensive tasks, confirming our initial hypothesis.

These preliminary tests highlighted major mismatches between virtual threads and Netty. These are likely to happen *for every reactive framework assuming a low number of event-loops*, making the integration of virtual threads in existing reactive frameworks challenging. This low number of event-loops is to be understood in the context of constrained resources. As discussed in 1, microservices have limited resources and aim at using them as efficiently as possible.

[12] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/locks/AbstractQueuedSynchronizer.html
[13] https://hub.docker.com/r/redhat/ubi8
[14] https://www.techempower.com/benchmarks/#section=data-r20&hw=ph&test=fortune

3.2 Characterizing the load and the goals of the test

Based on a benchmark survey of 2015 [9], a *load test* can be of one of these kinds :

- load testing : *"assessing the behavior of a system under load in order to detect load-related problems."*
- performance testing : *"measuring and/or evaluating performance related aspects of a software system (response time, throughput, resource utilization)"*
- stress testing : *"extreme conditions to verify the robustness of the system and/or to detect various load-related problems (e.g., memory leaks and deadlocks)"*

Our benchmark is an instance of performance testing since metrics such as CPU usage, memory footprint, latency, throughput are measured. These measures are conducted under a given load, so the benchmark can also be characterized as load testing. Furthermore, resources of the service are constrained through containerization, delay is added using chaos engineering tools, and the load is increased to values where the service displays behavior judged to be "failing". The benchmark can hence also be described as being an instance of stress testing. The benchmark is thus at the intersection of load testing, performance testing and stress testing.

Questions about the design of the load must also be addressed [9]. Typically, the difference is between (i) realistic load and (ii) fault inducing load. The goal of this experiment is to see how an application will react to an IO-bound workload that will force it to create thousands of virtual threads. Putting the application under heavy load will lead to the creation of numerous virtual threads. It is then possible to see how much time is spent in creating, mounting and unmounting them from their carriers. Delay is added between the database and the application to force the virtual threads to remain idle while waiting for the results, thus making it possible to see the impact of thousands of idle virtual threads on memory footprint. In this regard, the load is mostly *fault inducing* since it is specifically designed to maximize memory footprint as well as the number of virtual threads switches. To load/stress the service, a scenario inspired by the Fortunes test of the Techempower benchmark suite [15] is used. In this benchmark, the service must :

(1) fetch all "Fortunes" objects in a Database
(2) add a Fortune object to the list it fetched from the Database
(3) sort the list
(4) return it using a template engine. In our test we simply serialize the list in JSON, and return it

The service was deployed in a controlled environment to see the evolution of CPU usage, memory footprint, latency and footprint on different specific configurations.

3.3 Integrating different components into the experiment

In this section, the role of each component as well as the different constraints put on each of them will be discussed. The Figure 3 is an overview of the different components involved in the benchmark, their relation and the different variables we can tweak.

[15]https://www.techempower.com/benchmarks/#section=data-r20&hw=ph&test=fortune

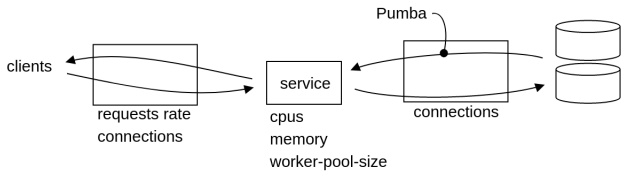

Figure 3: Diagram of the different components involved in the benchmark

3.3.1 Load generator. To drive the load, Hyperfoil [16] is used. It was selected because it satisfied this set of requirements:

- avoids COF (coordinated-omission fallacy) [22]
- properly dimensions the number of connections
- gives us control on sharing or not sharing connections
- support the *Open System Model* [22]

Pitfalls. Hyperfoil builds a report of what happened during the test. This report can indicate that something went wrong with an aspect that we do not measure, that isn't supposed to be the limiting factor. For instance, the number of connections required for the test are specified before running the benchmark. This number of connections needs to be properly tuned in order to have enough of them to support the requested rate. For every run, Hyperfoil generates a report, making it possible to check that no connection is blocked.

3.3.2 Service. The service is the subject of the benchmark.

response-time/latency: The longest response-time registered in the run is considered. The infrastructure provides a detailed list of every request, so it is possible to check that the number plotted is not an outlier by comparing it with the longest 0.1% requests for instance.

memory consumption: Memory consumption is measured by frequently probing the Resident Set Size of the service.

throughput: Throughput offers a different perspective on how the service responds to high loads. Maybe the latency will explode at N req/sec, but the service will still be able to deliver $2 * N$ req/sec. We hence need to measure both.

CPU usage: CPU usage can be an indication of how efficiently resources are used. Moreover, it is useful to know if the bottleneck is the service itself or if it is located somewhere else (bad performance coupled with very low CPU usage might mean that the problem is somewhere else). Just like the RSS, CPU usage is probed several times during the test.

worker-pool size : when in Blocking mode, the worker-pool needs to be configured at the right size(too small : too few threads, performance won't be optimal ; too big : too many threads, might get an *OutOfMemoryError*).

DB connection-pool size : the throughput (and the latency) depends greatly on how fast data can move from the database to the service and from the service to the database. This is directly linked to the number of connections available to handle the traffic.

Pitfalls. For the measures to be correct, the service needs not only to be the bottleneck of the benchmark but to be so for the right reasons. Unequal tuning of one of the application versions

[16]https://hyperfoil.io/

(reactive, virtual threads, blocking) might result in an erroneous comparison. Three different versions of the application are tested: Quarkus-reactive, Quarkus-blocking, and Quarkus-virtual-threads. Quarkus-reactive and Quarkus-virtual-threads have their own strategy regarding the number of IO threads (event-loops) they use but Quarkus-blocking doesn't set a worker-pool size depending on the available memory, it only has a default value that can be overridden. The correlation between the size of the worker-pool W and the maximum concurrency C is obvious : $C = W/d$ with d the average request service-time. The bigger the worker-pool the better the results. However, if the limit is set too high, worker threads will occupy too much space in memory and pressure the garbage collector, leading the application to spend enough time in GC to actually downgrade the performance. This could even result in an *OutOfMemoryError* and kill the application. Thus, a preliminary test iterating over a range of worker-pool sizes was performed to find the optimal one. The conclusion was that setting the worker-pool size to the amount of MB in memory (if the container has 512 MB of memory available, use a worker-pool of 512 worker threads) works best. This seems to be a good compromise between avoiding OOME and getting good performance.

Since the application runs in a *Java Virtual Machine (JVM)*, many things can be tuned in order to optimize performance. Properly tuning the JVM for a specific application is hard and time-consuming. We decided to use the Parallel Garbage collector and to set the maximum heap size of the VM to half the available memory. Finally, when a Java process is started, the beginning of its lifetime will be significantly impacted by the *JVM warm-up*. According to [13], the warm-up time is "commonly the bottleneck of short running jobs, even when the job is I/O-intensive". Hence, the first 20s of a run are discarded to make sure all the needed classes have been loaded, and the hot methods have been optimized by the JIT. Since the JIT can take a few minutes to find and optimize the major hot methods, we run each test three times in the same VM. Some chaos introduced by the GC and the JIT might lead some tests to sporadically exhibit terrible performance under load that are near their supported limit. To avoid it, we took the decision to choose between the best of the three runs.

3.3.3 Database. The following points must be considered:

- properly dimensioniong the number of connections,
- using a Chaos Engineering Tool, *Pumba* [17] to add delay (only outgoing packets),
- making sure that the Chaos Engineering tool can handle the rate.

The use of a Chaos Engineering tool makes it harder to characterize the service: it could be the source of an overhead that could skew the results.

Pitfalls. Pumba makes it possible to specify what delay to apply to the outgoing packets of a certain containerized process.

The base hypothesis is that no matter the delay, it shouldn't impact the server since it doesn't add any load to it. Thus, for a given request-rate (one that is low enough for the server to keep low response-time), a linear relation between the response-time and the latency should be observed. If the maximum response-time is t without Pumba, it should be $t + 100$ ms when Pumba adds 100ms of delay, and $t + d$ ms when we add d ms of delay. This hypothesis has been proven wrong. After comparing several runs using different delays to compare the behaviors of the servers, it appeared that although latency was linearly correlated to the delay at first there was a point at which latency increased exponentially.

In order to make sure that Pumba is not the bottleneck of the benchmark, a baseline was required. This baseline is used to induce what a measure of an overloaded service should look like in terms of CPU usage, RSS, packet drop rates. Every run done using Pumba is compared to the results of the baseline to check if the bottleneck comes from the service or from Pumba.

4 EXPERIMENT RESULTS

As mentioned earlier, delay is added between the database and the service to force virtual threads to stay in memory in order to see how latency will impact the implementation. These results are compared to a baseline without any delay to see how latency (inevitable in a distributed application) will impact the performance of Quarkus-blocking, Quarkus-virtual-threads and Quarkus-reactive. To better understand the results, an investigation is led using the *Java Flight Recorder (JFR)* for the capture of JVM event.

4.1 No delay induced by Pumba, baseline test

Figure 4 shows the evolution of Resident Set Size (RSS) [18], CPU usage, latency and throughput of the service as the number of requests per second generated by Hyperfoil increases. This run is performed without Pumba, it is thus the baseline. Since the time it takes for the request to complete is in the order of 100µs and doesn't involve any I/O-operation, Quarkus-blocking is expected to be the best option, or close to the best option.

System load areas. The goal of this experiment is to correlate different metrics. Ideally, we would like to be able to explain high latency and/or low throughput using CPU usage or memory consumption. The Table 4 summarizes the different limits for each working condition in terms of latency l, throughput T, memory M and CPU CU.

	Latency	Throughput	RSS	CPU
Normal	$l < 2s$	$T = R$	$M < 330$	$CU < 80\%$
Critical	$l >= 2s$	$T = R$	$M > 330$	$CU < 80\%$
Overload	$l >> 2s$	$T < R$	$M > 330$	$CU > 80\%$

Table 4: Different working conditions

(1) **normal conditions** : the throughput T is the same as the requested rate R, resource utilization isn't maxed out.
(2) **critical conditions** : latency starts increasing although the throughput is still the same as the requested rate. Messages might pile up and cause long latencies after a long time. Memory consumption or CPU usage should near the limits
(3) **overload conditions** : the latency is high, the service can't process as many messages as requested. CPU usage should

[17] https://github.com/alexei-led/pumba

[18] https://docs.oracle.com/cd/E29584_01/webhelp/endeca_glossary/src/gend_resident_set_size_dgraph.html

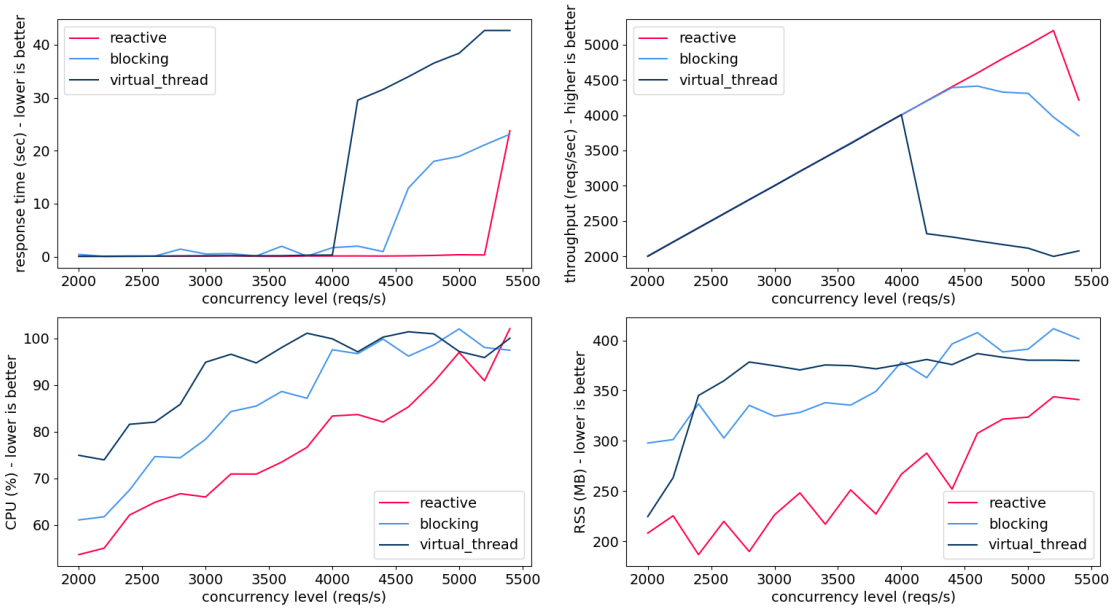

Figure 4: Evolution of the latency, CPU usage, RSS and throughput in function of the request rate, no delay, use the default virtual threads scheduler

have reached the limit, memory consumption might have also reached its limit.

Although the goal is to characterize the latency of a system, throughput is a critical measure that gives insights about the ability of the system to support a certain load. Memory consumption measured with the RSS might not be an indicator as reliable as CPU usage to determine if the service is overloaded or nearing overload. Although it steadily increases as the load increases, the Figure 4 shows that systems in a critical state all have a CPU usage nearing 100% whereas a discrepancy of 50 MB can be observed in terms of memory usage.

Comparing the three systems.

- Quarkus-virtual-threads rapidly reaches a plateau in terms of memory consumption at 2500-3000 req/s and still continues to keep a small response time and a satisfying throughput.
- Quarkus-blocking also reaches high memory footprint early.
- Their respective CPU usages remain low and seem to increase linearly with the request rate.
- Quarkus-reactive has the lowest memory footprint during the entire experiment as well as the lowest CPU usage.

We can derive a certain number of conclusions from this run:

(1) A high memory footprint alone doesn't seem to be enough to cause an increase in latency or a stagnation of throughput.
(2) When threads are blocked for a small amount of time (as low as tens of µs), Quarkus-blocking is less expensive than Quarkus-virtual-threads in terms of CPU-usage.
(3) Quarkus-reactive is the most efficient system, especially in terms of memory consumption.

4.2 Constant 200ms delay, increasing rate

As soon as delay is added, Quarkus-blocking behaves differently. Its maximum level of concurrency is easy to compute, as seen in 3 : $C = W/d$ with d the average request service-time (in seconds) and W the number of workers. In this case, $W = 500$ and $d = 200$, so a maximum concurrency level of $C = 500 * 5 = 2500$ is to be expected. Each of the three systems is expected to be affected by the additional delay, Quarkus-blocking especially due to its hard limit in terms of scalability.

Hypothesis: performance should decrease for the three systems (bigger resource consumption, smaller throughput, earlier latency increase) but Quarkus-blocking should be the most affected by the delay.

This hypothesis is somewhat consistent with what we measure through our experiment in the Figure 5: although every system is impacted by the delay, Quarkus-blocking is especially affected. The rate at which the latency of the system becomes overloaded is now 2500 req/s. As seen in Figure 4, high memory footprint alone is not enough to cause high latency responses. Adding delay shows that high CPU-usage alone can't cause it neither. The CPU-usages of Quarkus-virtual-threads and Quarkus-reactive stabilize at 90% and their latency starts increasing only when the memory footprint becomes high enough.

The most obvious difference between the two runs is the CPU-usage of each system: they increase by about 30%, some specific comparisons are displayed in Table 5. To understand why delaying the database response causes a drop in performance as well as an

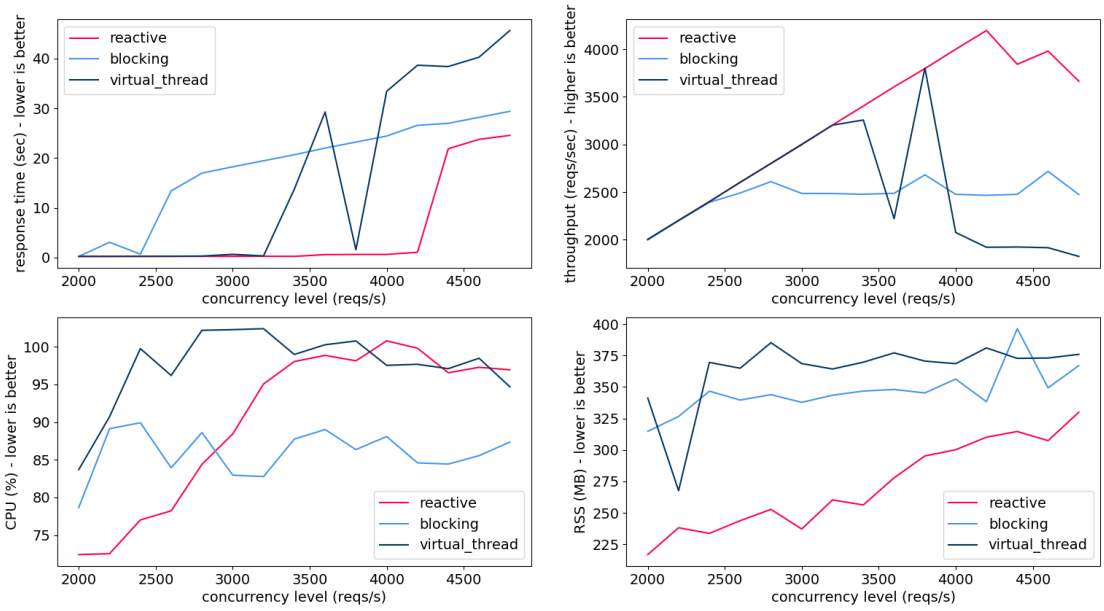

Figure 5: Evolution of the latency, CPU usage, RSS and throughput in function of the request rate, 200ms delay, use the default virtual threads scheduler

System	2200 req/s	3200 req/s	4200 req/s
Quarkus-reactive	57.24	70.9	83.7
Quarkus-reactive-200	75.47	95.0	107.9
Quarkus-blocking	61.7	84.3	98.4
Quarkus-blocking-200	82.8	85.3	84.6
Quarkus-virtual-threads	73.9	99.8	101.3
Quarkus-virtual-threads-200	92.9 (+19.0)	106.0	101.7

Table 5: Comparing CPU-usage (in %) at different concurrency level with and without delay

increase in CPU-usage, an analysis using the *Java Flight Recorder (JFR)* was performed.

4.3 JFR analysis

We decided to analyze two situations :

(1) Quarkus-virtual-threads at 4500 req/s with and without delay to see if JFR can provide insights on why performance is affected by delay
(2) Quarkus-reactive at 4500 req/s with and without delay to see what changes in a service that remains responsive and efficient.

JDK Mission Control (JMC) [19] was used to analyze the recording. Table 6 summarizes the important results. It is an extract of larger set of data [20].

[19]https://www.oracle.com/java/technologies/javase/products-jmc8-downloads.html
[20]https://github.com/anavarr/quarkus_virtual_threads_results

	Quarkus-virt-threads-0	Quarkus-virt-threads-200
Max-latency	1.44 s	31.54 s
GC count	275	709
Avg pause	18.662 ms	92.270 ms
Longest pause	89.723 ms	520.437 ms
Young Collection time	5.132 s	41.685 s
Old Collection time	N/A	23.734 s
Sum of pauses	5.132 s	65 s

	Quarkus-reactive-0	Quarkus-reactive-200
Max-latency	1.08 s	704.64 ms
GC count	192	183
Avg pause	15.169 ms	14.968 ms
Longest pause	32.898 ms	49.312 ms
Young Collection time	3.004	2.739s
Old Collection time	N/A	N/A
Sum of pauses	3.004 s	2.739 s

Table 6: GC information summary

Using Table 6, it is clear that the Garbage Collector (GC) is under much higher pressure in the case of Quarkus-virtual-threads with a 200ms of delay. CPU usage is already fairly high (about 90% according to Figure 5), having to stop the computation to run the GC for tens of milliseconds in average might be the dealbreaker here. Additionally, among the 709 GC performed by Quarkus-virtual-threads (scenario Quarkus-virtual-threads-200), 65 are *Old Generation GC*. The old generation is the set of *long-living* objects. After a certain time alive, the JVM promotes live objects to the Old Generation space to free space in the young Generation [24]. Old Generation GC takes longer because it has to go through every living object.

It is explicitly stated in [24] that "for Responsive applications, major garbage collections should be minimized". In this case, young collection itself takes more than 40s, with an average pause longer than usual (65 ms against 19 ms without delay [21]).

GC seems to be the reason why the performance of Quarkus-virtual-threads is degraded when delay is added. Several hypotheses regarding what objects trigger GC must be proposed:

- since requests take longer to be processed, more requests must be handled concurrently, hence more virtual threads might live in the system at the same time, occupying space;
- Quarkus-virtual-threads relies on Netty (that hasn't been optimized for this use-case), when handling a high number of concurrent requests, buffers and caches might be used in a suboptimal way and occupy more memory than in the Quarkus-reactive implementation.

Conducting a comprehensive analysis of the heap can provide valuable insights into the veracity of these hypotheses. It is imperative to note that these hypotheses have different implications. If virtual threads are indeed the root cause of prolonged garbage collection, then the optimal solution would involve modifying the JDK to reduce their size, if feasible. Conversely, if the increased memory footprint is attributed to the use of Netty in conjunction with virtual threads, then the solution would not impact the JDK. Therefore, the implementation of a solution would likely be more straightforward.

Heap Analysis. Through a comparative analysis of heap dumps obtained from the Quarkus-reactive and Quarkus-virtual-threads implementations of the application, we can ascertain the factors contributing to garbage collection (GC) inefficiencies. The results presented here are extracted from this repository [22].

The top consumers for both Quarkus-virtual-threads and Quarkus-reactive are presented in Table 7. Although there are less byte arrays in Quarkus-virtual-threads than in Quarkus-reactive, they weigh about thrice as much. It appears that about 81.3MB is occupied by only 10 147 byte arrays containing the JSON array returned by the Fortune benchmark. The presence of arrays of char instances is explained the same way. After further investigations, it appeared that the *Jackson* library generates thousands of *ThreadLocal*s in its management of buffers. These data structures occupy a lot of space and take time to be collected. In Quarkus-reactive, there are only two of them since they are spawned for each event-loop. On the other hand, in Quarkus-virtual-threads, thousands of them are created every second since a virtual thread is spawned for every incoming request. As seen in Section 3.1, this comes from the mismatch between the core hypothesis of reactive programming and those of pool-based synchronous programming.

4.4 Discussing our results

The strategy to get a clean dataset was to perform three runs and to systematically choose the best one to avoid outliers. A better approach would be to run the tests N times and perform a true statistical analysis on the results to have clear indications on the likelihood of getting extreme latencies at each concurrency level. Running the benchmark takes a long time: about 20 minutes per level of concurrency. Considering that the experiments displayed in this paper range over 30 to 50 levels of concurrency, performing one run of the benchmark takes between 10 to 16-17h. Running them N times such that N be big enough for a statistical analysis to be meaningful would take days if not weeks. Moreover, such results would be meaningful to make accurate predictions on how Quarkus-virtual-threads might behave in some given situations, but this is beyond the scope of this paper. The current experiments already gave directions for future research and optimization.

However, a more reliable characterization of performance is an endeavor that we might undertake in the future.

5 RELATED WORKS

This experiment investigates and evaluates the integration of the Java virtual threads in an existing reactive framework, Quarkus. A review of the state of the art was necessary for (i) existing tools for benchmarking Java microservices, (ii) existing tests of Java virtual threads, and (iii) characterizing existing frameworks to motivate the work on Quarkus.

5.1 Existing benchmarks for testing Java microservices

As outlined in Section 3, this study employs a predominantly fault-inducing workload, and is situated at the confluence of load testing, performance testing, and stress testing. We conducted a review of different existing benchmark suites. Most benchmarks aim at characterizing the application under test, not the JVM itself. Therefore, these benchmarks, such as [5] is limited in our context. While benchmark suites such as DaCapo [3] or SPECs [21] are frequently used to tune components of the JVM (such as JIT compilers, garbage collectors, profilers), they weren't designed for studying concurrency, which is the main objective of our work. Additionally, they were created before atomic operations, Java Lambdas and lock-free data structures were added to the JDK. The Renaissance benchmark suite to enhance the JDK. It consists of 21 different benchmarks. Most of them are useless for our purposes since we are interested in IO-intensive operations. Although it seems to study concurrency and parallelism better than previous benchmark suites, *"all benchmarks run within a single JVM process, and only rely on the JDK as an external dependency. Some benchmarks use network communication, and they are encoded as multiple threads that exercise the network stack within a single process (using the loop-back interface)."* [17]. As explained in Section 3, our experiment requires control over the duration of I/O operations to study how virtual threads compare to threads and reactive constructs in an environment where latency between services is non-negligible. It was implemented via a database running in another container and the use of chaos engineering tools to add delay to the network interface of the database container. This goes against the design of the Renaissance suite. We can however observe similarities between the benchmark suites mentioned earlier and the benchmark developed here: the warm-up phase is typically found everywhere, and the actual measure is always performed on the *steady-state* execution after the warm-up.

[21] https://github.com/anavarr/quarkus_virtual_threads_results
[22] https://github.com/anavarr/quarkus_virtual_threads_results

Quarkus-virtual-thread			
Class name	Number of instances	Shallow heap	Retained heap
byte[]	288 768	98.9MB	98.9MB
char[]	15073	82.8MB	82.8MB
java.lang.Object[]	284030	16.4MB	211.1MB
int[]	13791	10.1MB	10.1MB
java.util.HashMap$Node[]	31603	6.8MB	8.6MB
Quarkus-reactive			
Class name	Number of instances	Shallow heap	Retained heap
byte[]	473026	30.8MB	30.8MB
java.lang.Object[]	493610	18.0MB	59.8MB
java.lang.String	416243	10.0MB	35.8MB
java.util.HashMap$Node[]	48599	8.1MB	10.9MB
io.quarkus.benchmark.model.Fortune	256958	6.2MB	6.2MB

Table 7: Top 5 consumers for each application

5.2 Existing tests of virtual threads in Java

Several studies [1, 2, 18] led by the same team have investigated the performance of virtual threads compared to Java platform threads and Kotlin coroutines. The findings indicate that virtual threads offer superior performance to both Java platform threads and Kotlin coroutines. Specifically, they can achieve better scalability and faster synchronization [2].

The benchmark used in the latest publication [1] is also a multithreaded HTTP server. For each request, an object is created, its information is written to a file and its data is returned. The IO operation performed is a disk I/O operation and the benchmark is run on Linux. However, non-blocking disk IO in Linux is notoriously complex [23], which makes it harder to interpret the results.

Moreover, these studies differ from ours in terms of service size. Specifically, the studies used: (i) an Ubuntu 18.04.3 64-bit virtual machine with a base memory of 9 GB running on Windows 10 64-bit OS with an x64-based processor with 16 GB of RAM [18]; (ii) an Ubuntu-20.04 64-bit machine with 13 GB of RAM, 8-core VM running on Windows 10 64-bit OS with an x64-based processor and 16 GB of RAM [2]; and (iii) an Ubuntu-20.04.3 LTS 64-bit machine with 16 GB memory and Intel® Core™ i7-6700 CPU @ 3.40GHz processor [1]. Notably, virtual threads were not compared to existing reactive programming models in these studies. Instead, they were compared to platform threads on machines with ample resources. In contrast, we evaluate virtual threads against reactive libraries on machines with limited resources.

5.3 Existing Java frameworks

In the experiments of the *Spring* [24] framework, Tomcat's standard thread pool is replaced with a virtual threads based executor [25,26]. Current users of the reactive version of Spring, *Spring WebFlux* [27], won't benefit from virtual threads. In contrast, Quarkus is a relatively newer framework focused on cloud-native applications. Its core is reactive and users can choose to delegate some work to blocking thread pools. As such, the three programming models (Quarkus-blocking, Quarkus-reactive and Quarkus-virtual-threads) can thus be used in the same application depending on the context.

Helidon Níma [28] is an instance of framework that was specifically designed for virtual-threads. It was briefly tested against Netty and performed favorably [29]. However, the benchmark was limited in scope, and its conclusion should be nuanced. It takes place on the loopback interface, without any delay, and no long-running operation was performed. Thus, it is difficult to predict how an application will behave when thousands of virtual threads are kept in memory while waiting for IO completion. Additionally, as Níma is built from the ground up, it needs to reimplement features already implemented in well-established frameworks such as Netty. As a result, it is essential to conduct further benchmarking after incorporating missing features to confirm that the performance gain is still substantial

To the best of our knowledge, Jakarta EE [30] has not released any information regarding its support for virtual threads.

Finally, several options were available for the reactive streams library. The three most widespread libraries propose similar services and display similar performance [15]. Mutiny was picked since it is the default choice for Quarkus. According to prior benchmarks [15], the performance of the application should not be impacted significantly by the choice of library.

6 CONCLUSION AND PERSPECTIVES

This paper aims at describing and motivating the current integration of an easy-to-use programming abstraction, virtual threads, in the Quarkus framework. An experimental protocol was developed to evaluate the performance of the implementation in the context of resource-constrained environments with a fault-inducing load. The study involved a thorough examination of the tools used to obtain meaningful performance metrics. It reveals that although Quarkus-virtual-threads outperformed Quarkus-blocking *in the context of the experiment*, it still does not scale to the concurrency

[23] https://unixism.net/loti/async_intro.html
[24] https://spring.io/
[25] https://spring.io/blog/2023/02/27/web-applications-and-project-loom
[26] https://spring.io/blog/2022/10/11/embracing-virtual-threads
[27] https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html
[28] https://helidon.io/nima
[29] https://medium.com/helidon/helidon-n%C3%ADma-helidon-on-virtual-threads-130bb2ea2088
[30] https://jakarta.ee/

levels that Quarkus-reactive is capable of achieving. Additionally, resource management is less efficient than Quarkus-reactive, with a particularly high memory footprint. However, analysis indicates that *virtual threads are not directly responsible for the high memory consumption*, and that other data structures generated by Quarkus-Netty are the main memory consumers. The study suggests that it may be possible to optimize the way virtual threads are integrated into the Quarkus framework to reduce the heap size, GC pressure, and improve scalability and resource management efficiency. It should be noted that the findings may not necessarily generalize to other scenarios, and further investigations with realistic loads may provide additional insights into the performance of the three variants (Quarkus-blocking, Quarkus-reactive and Quarkus-virtual-threads). However, the study highlights the mismatch between the core assumptions of Netty, which assumes a limited number of event-loop threads, and the core assumptions of virtual threads, which assume that they are cheap to create and can be spun up as needed. The findings suggest that *the integration of virtual threads in frameworks built on top of Netty or that rely heavily on ThreadLocals, presuming a small thread count, is likely to experience significant garbage collection (GC) pressure.*

Future investigations should explore the performance of the three variants under different architectures with more CPU cores and memory. It should also be compared with similar programming abstractions in other languages such as Golang goroutines, Python coroutines, Kotlin coroutines, C# tasks, JavaScript asynchronous functions. Additionally, it suggests that performance degradation should be evaluated alongside productivity gains. If the utilization of virtual threads significantly simplifies the implementation of a particular feature and the accompanying performance impact is minimal, it can prove advantageous. Finally, since garbage collection appears to be the main problem, different garbage collectors should be tested in order to characterize their impact on the performance of virtual threads integration. For example, using more optimist hypothesis could enable a quicker collection of unused ThreadLocals and improve performance. As mentioned in [1], work was started in 2019 [19] to compare different garbage collection algorithms and could be extended to virtual threads.

ACKNOWLEDGMENTS

This work is partially supported by Red Hat Research.

REFERENCES

[1] D. Beronić, L. Modrić, B. Mihaljević, and A. Radovan. 2022. 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)*. 1466–1471. https://doi.org/10.23919/MIPRO55190.2022.9803765

[2] D. Beronić, P. Pufek, B. Mihaljević, and A. Radovan. 2021. On Analyzing Virtual Threads – a Structured Concurrency Model for Scalable Applications on the JVM. In *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*. 1684–1689. https://doi.org/10.23919/MIPRO52101.2021.9596855

[3] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[4] Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. 2014. The Reactive Manifesto v2.0. https://www.reactivemanifesto.org/.

[5] Emmanuel Cecchet, Veena Udayabhanu, Timothy Wood, and Prashant Shenoy. 2011. BenchLab: An Open Testbed for Realistic Benchmarking of Web Applications. In *Proceedings of the 2nd USENIX Conference on Web Application Development*. USENIX Association, USA, 12.

[6] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*, Manuel Mazzara and Bertrand Meyer (Eds.). Springer International Publishing, Cham, 195–216. https://doi.org/10.1007/978-3-319-67425-4_12

[7] Patrik Fortier, Frédéric Le Mouël, and Julien Ponge. 2021. Dyninka: a FaaS framework for distributed dataflow applications. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2021)*. Association for Computing Machinery, New York, NY, USA, 2–13. https://doi.org/10.1145/3486605.3486789

[8] Kasun Indrasiri and Prabath Siriwardena. 2018. *Integrating Microservices*. Apress, Berkeley, CA, 167–217. https://doi.org/10.1007/978-1-4842-3858-5_7

[9] Zhen Ming Jiang and Ahmed E. Hassan. 2015. A Survey on Load Testing of Large-Scale Software Systems. *IEEE Transactions on Software Engineering* 41, 11 (Nov. 2015), 1091–1118. https://doi.org/10.1109/TSE.2015.2445340

[10] Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter. 2013. An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications (DYLA '13)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/2489798.2489802

[11] Dan Kegel. 1999. The C10K Problem. http://www.kegel.com/c10k.html.

[12] Sirojiddin Komolov, Nursultan Askarbekuly, and Manuel Mazzara. 2020. An Empirical Study of Multi-Threading Paradigms Reactive Programming vs Continuation-Passing Style. In *2020 the 3rd International Conference on Computing and Big Data (ICCBD '20)*. Association for Computing Machinery, New York, NY, USA, 37–41. https://doi.org/10.1145/3418688.3418695

[13] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. 2016. Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 383–400.

[14] Julien Ponge. 2020. *Vertx in Action*. Manning Publications, USA.

[15] Julien Ponge, Arthur Navarro, Clément Escoffier, and Frédéric Le Mouël. 2021. Analysing the Performance and Costs of Reactive Programming Libraries in Java. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2021)*. Association for Computing Machinery, New York, NY, USA, 51–60. https://doi.org/10.1145/3486605.3486788

[16] Ron Pressler and Alan Bateman. 2021. JEP 425: Virtual Threads (Preview). https://openjdk.java.net/jeps/425.

[17] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 31–47. https://doi.org/10.1145/3314221.3314637

[18] P. Pufek, D. Beronić, B. Mihaljević, and A. Radovan. 2020. Achieving Efficient Structured Concurrency through Lightweight Fibers in Java Virtual Machine. In *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*. 1752–1757. https://doi.org/10.23919/MIPRO48935.2020.9245253

[19] P. Pufek, H. Grgić, and B. Mihaljević. 2019. Analysis of Garbage Collection Algorithms and Memory Management in Java. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 1677–1682. https://doi.org/10.23919/MIPRO.2019.8756844

[20] Douglas C. Schmidt. 1995. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In *Pattern Languages of Program Design*. Vol. 1. ACM Press/Addison-Wesley Publishing Co., USA, 529–545.

[21] Kumar Shiv, Kingsum Chow, Yanping Wang, and Dmitry Petrochenko. 2009. SPECjvm2008 Performance Characterization. In *Computer Performance Evaluation and Benchmarking (Lecture Notes in Computer Science)*, David Kaeli and Kai Sachs (Eds.). Springer, Berlin, Heidelberg, 17–35. https://doi.org/10.1007/978-3-540-93799-9_2

[22] Gil Tene. 2013. How Not to Measure Latency.

[23] Bill Williams. 2012. *The Economics of Cloud Computing: An Overview For Decision Makers [Book]*. Cisco Press, USA.

[24] Michael Williams, Krishnanjani Chitta, Marcie Young, and Glenn Stokol. 2012. Java Garbage Collection Basics. https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html.