Unlocking Efficiency Gains Using Java Virtual Threads

Nilesh Jagnik

Los Angeles, USA
nileshjagnik@gmail.com

Abstract

Concurrency has always been desirable in software systems, especially in applications that are I/O bound, because it can drastically improve the throughput and scalability of the system. A common way to achieve concurrency is multithreading. In the past, multithreading had scalability limits due to threads being expensive. To achieve higher scalability developers had to use asynchronous programming instead.However, asynchronous programming has its own drawbacks and has limited language support. Virtual threads, released in Java 21, are cheap and lightweight threads.Multithreading with the use of virtual threads can remove the scalability limits that it previously used to have. In this paper, we take a look at virtual threads, their benefits, uses and the caveats associated them.

Keywords: Concurrency, Multithreading, Asynchronous Programming, Software Efficiency, Scalability

I. INTRODUCTION

Efficiency and throughput are very important in software applications, especially for servers that need to serve requests at a high rate. Several techniques can used for improving the efficiency of programs. The most common ones are multithreading,and the use of frameworks that allow asynchronous and event-based programming. Both of these techniques come with some benefits and drawbacks. There is a limit up to which multithreading can provide benefits. This is because of threads being expensive. On the other hand, asynchronous programming can improve scalability beyond multithreading, but it is difficult to properly implement. Even when properly implemented, it still has limitations due to the support for asynchronous programming not being natively present in programming languages. Developers usually have to rely on frameworks for providing asynchronous programming support, and frameworks are also limited in functionality by what the programming language allows.

Virtual threads are a feature introduced in Java 21 that make multithreading a much more appealing option for Java servers. In this paper, we discuss the benefits of virtual threads and how they can help improve scalability without any of the drawbacks of asynchronous programming.

II. CONCURRENCY TECHNIQUES

Most programs have I/O operations which need to wait for importing input from or exporting output to external sources. For example, external RPCs, reading from disk or making database calls are all I/O operations. Code that is blocked waiting for I/O cannot proceed in execution. If no other code is executed at this time, the CPU will remain idle. To avoid waste of CPU resources, other code must be run by the OS to while this code is performing blocking I/O. This is the reason we need concurrency. Concurrency is highly

desirable in software programs. Concurrency allows parts of code to run independently of each other. This in turn allows better use of CPU resources and improves efficiency and throughput of servers running code.

A. *Multithreading*

The most common way to introduce concurrency in code is by making it multithreaded. Multithreading relates to operating system (OS) threads. Threads are entities that can track the state in which a program is. The OS allows programs to create and allocate work on threads. These threads are then managed by the OS and executed on CPU cores. The OS switches the thread currently executing on a CPU core if it is performing blocking I/O. In multithreading, code is written such that is can be run independently of other code. Then threads are spawned which can run this code concurrently on CPU cores.

The main problem with multithreading (without virtual threads)is that OS threads are quite expensive because the OS has to track a lot of state related to program execution. So, there are an upper limit on how many threads can be spawned without adding too much overhead on the OS (OS may also fail to create new threads due to lack of resources). And thus, there is an upper limit on efficiency gain by multithreading, especially for applications that perform at lot of blocking I/O. Most server applications lie in this category.

B. *Asynchronous Programming*

Asynchronous programming is offered by frameworks that allow developers to create tasks that execute pieces of code. The framework then handles scheduling of these tasks in a concurrent manner. A set pool of threads is allotted to the framework to schedule these tasks with. The framework switches out the task executing on threads when it starts performing blocking I/O. The catch here is that the framework can not detect blocking I/O automatically. To circumvent this issue, programmers need to specify the exact statements in code where blocking I/O is being performed. Developers need to be educated about this programming paradigm. This approach is quite prone to errors. In some cases, developers may not even know whether some API they are calling contains blocking I/O or not. In addition, most frameworks that provide async support, do not provide good error handling and debugging facilities. This is due to the disconnected nature of tasks and threads. For example, thread dump does not print all the tasks that the framework was executing because the runtime is unaware of them.

III. PLATFORM AND VIRTUAL THREADS

Let us dive a bit deeper into changes related to threads in Java 21. Platform threads are traditional threads. There is a 1:1 mapping between platform threads and OS threads. Platform threads in Java are thin wrappers around OS threads. Platform thread is an instance of java.lang.Thread which is implemented in the traditional way.

Virtual threads are a new implementation of the java.lang.Thread. Virtual threads are not tied to any specific OS threads. Instead, they are lightweight thread implementations. These are analogous to the tasks created when using asynchronous frameworks. However, they have the same interface as traditional threads.

The Java runtime is responsible for scheduling virtual threads on platform threads. From the developer's perspective, virtual threads work the same way as traditional threads (apart from thread creation). Virtual threads are lightweight because they don't store as much state as traditional threads and keep a shallow call stack. So many more virtual threads can be spawned due to being cheap

```
try(ExecutorService executor =
Executors.newVirtualThreadPerTaskExecutor()){
  Future<?> future = executor.submit(task);
                future.get();
  System.out.println("Completed task.");
```

Fig. 1. Using the Thread Builder interface for creating virtual threads

IV. Benefits of Virtual Threads

Due to virtual threads being cheap and plentiful, traditional multithreading can be used with them to get much better efficiency and throughput. These gains can be observed with needing to use asynchronous programming. Asynchronous programming can really increase code complexity, which can be avoided with the use of virtual threads. Additionally, since virtual threads behave similarly to traditional threads, thread dumps contain info on virtual threads, which can be immensely helpful for debugging issues. To solve the problem of error handling, Java also has a preview feature called structured concurrency, which adds better handling of concurrent tasks that rely on each other.

V. Using Virtual Threads

There are two main ways to create and use virtual threads.

A. *Using Thread.Builder Interface*

Fig. 1 shows how virtual threads can be created by using the Thread.Builderinterface. Once a builder is created, a Runnable can be scheduled on the thread using the start() method. The Thread.Builder interface also allows setting some properties like name.

```
Thread.Builder builder =
Thread.ofVirtual().name("VThread");

Runnable task =()->{
Thread.sleep(2000);
System.out.println("Performed work for 2s.");
};

Thread vThread = builder.start(task);
vThread.join();
```

Fig. 2. An ExecutorService that creates new virtual threads every time a task is submitted to it

In addition to this, it is also possible to create a thread with starting the execution using the unstarted() method. And calling start() on the Thread object itself.

B. *Creating an ExecutorService*

In addition to creating a Thread object using Thread.Builder, it is possible to create an ExecutorService withthe newVirtualThreadPerTaskExecutor() method. This ExecutorService creates new virtual threads anytime ExecutorService.submit() is invoked. Fig. 2 shows an example of how this can be used in practice.

Note that this method returns an instance of theFuture type.

VI. Caveats Of Virtual Threads

Virtual threads are a great tool for solving scalability problems, but users should take care before onboarding to virtual threads. Let us discuss some things to be aware of.

A. *CPU Intensive Tasks*

Virtual threads are not really suited to CPU intensive tasks since there is no blocking I/O in them. Creating too many virtual threads will only lead to increase in thread management overhead.

B. *Compatibility*

Older libraries in existing codebases may not be optimized for virtual threads (due to other factors listed), which may lead to issues. This is also true for some of Java's own libraries.

C. *Pinning*

In some cases, namely, when a synchronized block is run inside a virtual thread, and when calling a native method or foreign function inside a virtual thread, the platform thread currently being used is rendered unable to switch to executing other virtual threads. Such cases need to be carefully handled.

D. *Fairness and Starvation*

Creating too many virtual threads with a limited number of platform threads may lead to scheduling issues like fairness and starvation, where some threads may not get a fair chance to execute.

E. *High Memory Usage*

ThreadLocal variables are compatible with virtual threads. However, spawning a large number of virtual threads with ThreadLocal variables may lead to high memory consumption and unforeseen issues.

CONCLUSION

Virtual threads are an excellent way to improve application throughput for I/O bound workloads. They follow the conventional Thread interface and can be used the same way as traditional threads for the most part. They also remove the need to adopt asynchronous programming patterns which are known to have several problems such as improper error propagation, debuggability, etc. However, special care must be taken when using virtual threads so that undesirable side effects can be avoided. In addition, the codebase needs to be updated to use Java 21, which may require some migration effort.

REFERENCES

1. Ram Lakshmanan, “Java Virtual Threads — Easy introduction (Feb 2023),”https://medium.com/@RamLakshmanan/java-virtual-threads-easy-introduction-44d96b8270f8
2. Ron Pressler, Alan Bateman, “JEP 444: Virtual Threads (Mar 2023),”https://openjdk.org/jeps/444
3. “Virtual Threads (Sep 2023),”https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html
4. Mark Paluch, “Embracing Virtual Threads(Oct 2022),”https://spring.io/blog/2022/10/11/embracing-virtual-threads
5. “Threads and Virtual Threads: Demystifying the World of Concurrency In Modern Times (Jun 2024),”https://www.zymr.com/blog/threads-and-virtual-threads-demystifying-the-world-of-concurrency-in-modern-times
6. “Virtual Thread support reference (Mar 2024),”https://quarkus.io/guides/virtual-threads