MIPRO 2020, September 28 - October 02, 2020, Opatija, Croatia

Achieving Efficient Structured Concurrency through Lightweight Fibers in Java Virtual Machine

P. Pufek, D. Beronić, B. Mihaljević and A. Radovan
[*] Rochester Institute of Technology Croatia, Zagreb, Croatia
paula.pufek@mail.rit.edu, dora.beronic@mail.rit.edu,
branko.mihaljevic@croatia.rit.edu, aleksander.radovan@croatia.rit.edu

Abstract - **Contemporary concurrent server applications, commonly built of smaller and independent services, are using concurrent threads to serve many incoming requests and often have to perform under excessive load. Those applications are relatively easy to develop in general-purpose, imperative programming languages such as Java, and have great tooling support. However, such applications are not easily scalable, mostly due to relying on oversized OS kernel threads, which can be created only in a limited number on finite hardware resources. Furthermore, heavyweight OS threads are implemented so robustly and generically to support various usage scenarios, and are, therefore, wasteful in resources and often inefficient in addressing specific application demands.**

Those challenges in software development resulted in various asynchronous programming techniques. This paper presents an exploration of a novel structured concurrency model in the Java Virtual Machine (JVM), introduced within OpenJDK's Project Loom. It is focused on the exploitation of fibers, new lightweight implementation of virtual threads within the JVM depending on delimited continuations. Furthermore, on several applications' benchmark cases, we analyzed performance with traditional threads and new fibers in different configurations. Finally, we discussed the current challenges of implementing fibers as a feasible approach for the more efficient future of Java.

Keywords - Structured Concurrency, Fibers, Java, Java Virtual Machine, Threads, Delimited Continuations

I. INTRODUCTION

One of the most important aspects of writing applications that concurrently respond to a large number of incoming requests is the achievement of optimal efficiency. Applications that provide such services could be developed with simple, blocking, synchronous code or might use asynchronous libraries that are more scalable but harder to implement, profile, and debug. In both cases, kernel threads are assigned for the services and are frequently wasteful in resources. To increase the extent of optimization in terms of scalability and preserving the utilization of external memory, software developers should not depend on the underlying system's memory allocation. Moreover, to demonstrate more convincing correctness and manageability, the internal structure of a program should not be treated as a black box [1].

Several programming languages and runtimes started exploring *Structured Concurrency* through independent mechanisms for asynchronous programming. Structured Concurrency might be defined as an organization of computations in a way that the limited hardware powers are sufficient to guarantee that computations will establish the desired effect [2]. For example, in Python, asynchronous I/O library named Trio[1] implements scopes of nurseries that launch coroutines, and in C programming language scopes that use bundles of coroutines are implemented within *libdill* tool[2].

The concept of Structured Concurrency implements continuations that represent a form of control flow that can be captured and manipulated [3] within a given scope with the aim of reducing the limited number of OS threads allocated for each control flow in a program. Moreover, modern programming languages such as Java, Scala, and Python currently support different systems for achieving concurrency.

In contrast to the common threads model, the *Actor* model in Scala implements asynchronous communication and enables structuring controls with Actors [4]. In this case, Actors are treated as primitives of concurrent digital computation, can send messages to other Actors, create new Actors, and decide how to handle the next method call. All of these actions might be executed concurrently, as the order of the system's tasks is not determined in advance. Another example is the *coroutine* concept thoroughly explored and used in Python, which supports the persistence of the coroutine's data between calls while temporarily suspending the execution flow and continuing where it was stopped when it is required for later use. However, to be able to implement a similar approach for modeling concurrency in Java, each coroutine might be assigned to a heavyweight thread [5]. That could consequently make a system less scalable as more operating system-native threads are required to efficiently perform the operation that might be expensive with regard to memory usage. With the intention of returning back to a simple, synchronous way of writing the code, without requiring such a great amount of power, a novel concept

[1] Trio: a friendly Python library for async concurrency and I/O, https://trio.readthedocs.io/en/stable/
[2] libdill: Structured Concurrency for C, http://libdill.org/

of *Continuations* might promise efficient flow of controls with trivial synchronization [6].

II. Novel Concurrency Concept in Java

To solve the problem of heavy-load thread utilization in Java, an early prototype is proposed within OpenJDK's Project Loom[3], which investigates Java Virtual Machine's (JVM's) features and APIs with the intention to support high-throughput lightweight concurrency within the Java platform. It suggests the increase of the number of lightweight threads, so-called *fibers*, scheduled in the JVM runtime so that the program does not rely directly on the availability and sufficiency of heavyweight OS kernel threads. In contrast to threads, fibers allow organizing concurrency in a coherent manner as efficient higher-level constructs built on top of continuations, which are implemented within the JVM.

In Java, the Continuation object commonly is associated with a Runnable object that represents the body of continuation, and a *ContinuationScope,* which allows nesting different continuations inside each other. When Continuation, which has its own stack, is executed, it either runs until it is not completed or yields. If it yields, it is returned to the caller, but since it remembers its state the next time it is run it continues from where it left off. Continuations are not supposed to be used directly, thus FiberScope API has been implemented to enable utilization of continuations.

III. Structured Concurrency

The main idea behind structured concurrency [2] is the ability to ensure that every time an application concurrently runs any number of tasks required by a developer, they all execute successfully and join up together after completion of each. With structuring the layout of the code to reflect the structure of execution flow, software developers might be able to make assertions about the next computations and control all concurrently running tasks that are working on a process.

Similarly to structured programming, structured concurrency structures the code as a set of nested code blocks that do not overlap [7]. To achieve more efficient control flow, the code is structured so that the tasks are bound to a scope with predefined start and end of operation's execution. If any of the tasks within the container (scope) are unsuccessful, the whole scope is canceled, and the stack frame where the scope was opened reports an exception. If several threads are performing a single operation, a developer should have an ability to know where each part of the task is executed within the code – which thread performs which operation, with the aim to enhance overall performance and decrease the possibility of exceptions.

A. Coroutines

Thus far, several variants of coroutines have been implemented in programming languages, under different names such as generators, fibers, and green threads. As they do not have to be synchronized and they explicitly pass control to each other, typically by invoking *yield* method [8], several high-level abstractions have been developed to make use of these low-level features.

The Abstract Behavioral Specification (ABS) modeling language, presented in [9], integrates actor-based programming with futures and coroutines. In the ABS model, the future is generated when a method is called asynchronously. It can be passed via actors that hold a reference to it and can check whether the corresponding method has completed and get a return value. In the case of the method's failure, the coroutines allow explicit suspension statements that enable multiple control flows in only one actor. Java library JAAC[4], which generates programs from ABS sources and provides a scalable implementation of asynchronous programming, currently supports coroutines that operate on the source and bytecode level in the JVM. At the source-code level, every thread could be allocated to each coroutine, which might not be sufficiently scalable. However, the architecture of this system allows an actor to run multiple tasks within its stack by storing its tasks in a queue. Each thread – actor, is able to select a task from its queue and run it as a separate process, concurrently with already running processes.

To adapt to JVM's specification of automatic memory management [10], the Garbage Collection algorithms had to be modified to support the implementation of lightweight threads. This process requires searching through the whole heap and finding all root references of live objects when freeing up the unused memory [11]. Thus, when a coroutine is located on the thread's stack, coroutine's stack is searched in exactly the same manner as threads' stacks are, and is investigated to check if there are parts of the heap ready for reclamation [8].

B. Continuations

The Continuation can be interpreted as the state of a running thread that can be captured and altered. It is commonly referred to as "*the rest of the computation*" [12] as it runs from the point in time when it was instantiated. According to [3], there are three categorizations of continuations:

- *First-class continuations* – fully store computation's state and can be reinstated many times
- *Delimited continuations* – part of continuation used to perform a function that can be invoked multiple times
- *One-shot continuations* – act as exit statements and can exist at multiple method scopes at once

Delimited continuations might be considered as functions that can be called multiple times. Furthermore, they are only paused for a particular amount of time and do not span throughout the entire life span of a program [14].

The two main functions of continuations are capturing (creating a continuation and storing the dynamic

[3] Project Loom, https://openjdk.java.net/projects/loom/

[4] JAAC, https://github.com/JaacRepo/JAAC

computation context of the currently running thread) and resuming (restoring the stack to exactly the same state as when the continuation was captured). The capture method associates the continuation with the currently running thread and multiple method invocations might be made on the same continuation object. The method resume is allowing a continuation to be reinstated in case if the capture method has already been invoked on that particular continuation object. If not, a runtime exception is thrown.

C. *Fibers*

Currently, Fibers are experimentally implemented as the JDK library on top of delimited Continuations combined with Java Schedulers in the Project Loom. Fibers are structured as they are constrained to FiberScope API; thus, every fiber is alive as long as the scope that created it, and multiple Fibers can be started that will be managed by the JVM. As soon as scope is exited, the close method is automatically called on FiberScope and it blocks every other method call until all fibers that are inside the block are terminated.

This core concept of Fibers allows the usage of structured concurrency in a way that all next operations would have to wait until Fibers scheduled in a particular scope have all finished their tasks. As expected, after each fiber has executed its assigned task, the following operations can proceed.

However, there are plenty of differences between Threads and Fibers. In the implementation of lightweight threads in the Quasar library[5], each fiber occupies approx. 400 bytes of RAM and decreases the use of CPU when switching the tasks. A fiber might have a return value, but implementation in the Project Loom also supports *Fiber<Void>* for type *V* which returns *null*. Furthermore, the *run* method is allowed to throw an exception, either InterruptedException or SuspendException; however, thread's run method does not throw any exceptions. Just as a thread might have a higher or lower priority, we are able to decide the order and importance of fiber's tasks by structuring the code with making efficient use of several Fibers in one or multiple FiberScopes.

As Fibers are scheduled by Java schedulers, they do not have to be Garbage Collection (GC) roots; thus, each Fiber does not have to be bound to a new thread. Fibers are runnable or are in a state of waiting – blocked. In both cases, their reference is held by a scheduler, and in case of blocking, the reference points to the object which caused the blocking. Existing Java schedulers, including ForkJoinPool and ThreadPoolExecutor [6] serve as Thread schedulers. Additionally, third-party optimized schedulers for implementing fibers were developed to enable manipulation of stacks' calls within JVM. The default scheduler is an object of FiberForkJoinScheduler that schedules Fiber in ForkJoinPool. When scheduling fibers in a thread pool of a particular architecture or performing operations on special, event dispatching thread and not daemon thread, such as Swing, FiberExecutorScheduler might be used.

[5] Quasar, http://docs.paralleluniverse.co/quasar/

IV. Testing Environment

A. *Testing Environment Setup*

The testing environment which hosted the virtual machine on which we have performed the testing was Windows 10 64-bit OS with an x64-based processor and 16 GB of RAM. The tests were run on Ubuntu 18.04.3 64-bit Virtual Machine with a base memory of 9 GB. The testing environment relies on OpenJDK Runtime Environment, OpenJDK 64-Bit Server VM (build 11.0.6+10-post-Ubuntu-lubuntu118.04.1, mixed-mode, sharing). However, for the JDK version that supports the utilization of Fiber and FiberScope API, integrated within Project Loom build, we used the early access OpenJDK 64-Bit Server VM (15-internal build mixed mode, sharing).

B. *Test Cases*

To record and compare the differences between each runtime, we used Vegeta[6], an open-source command-line utility application written in Go programming language, which provided the detailed reports on requests and responses. Vegeta is a load testing HTTP-request based tool and library, built to consume HTTP services by constantly sending requests to the given addresses. Also, we used the implementation of the HTTP server provided within the JDK[7], bounded to a given IP address and port number and listens for the incoming TCP client connections. Dealing with requests within the implementation of this virtual server is supported by HttpHandler, which is associated with the HttpServer object. Each server could be assigned one or multiple handlers which are mapped to the server via HttpContext object.

V. Preliminary Results

After warm-up pre-runs, which were repeated 3 times and had a duration of 10 seconds, we were able to record the results and maintain the stable behavior of the program execution of tests with both fibers and threads. For the purposes of our research and to find the optimal cases for thread and fiber performances, we analyzed several configurations of test parameters.

We maintained the optimal load to test their performances and experienced the most conclusive results by declaring a request rate per time unit value of 75, which represents a number of clients trying to connect to the server in 1 second. The actual request rates slightly varied due to internal JVM's processes such as the Garbage Collection process. The internal number of CPUs of the virtual machine was 3. Additionally, we specified the constant number of 64 workers which are used in the attack to control the level of concurrency as otherwise they would be increased as needed by the JVM.

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

[7] HttpServer, https://docs.oracle.com/en/java/javase/13/docs/api/jdk.httpserver/com/sun/net/httpserver/HttpServer.html

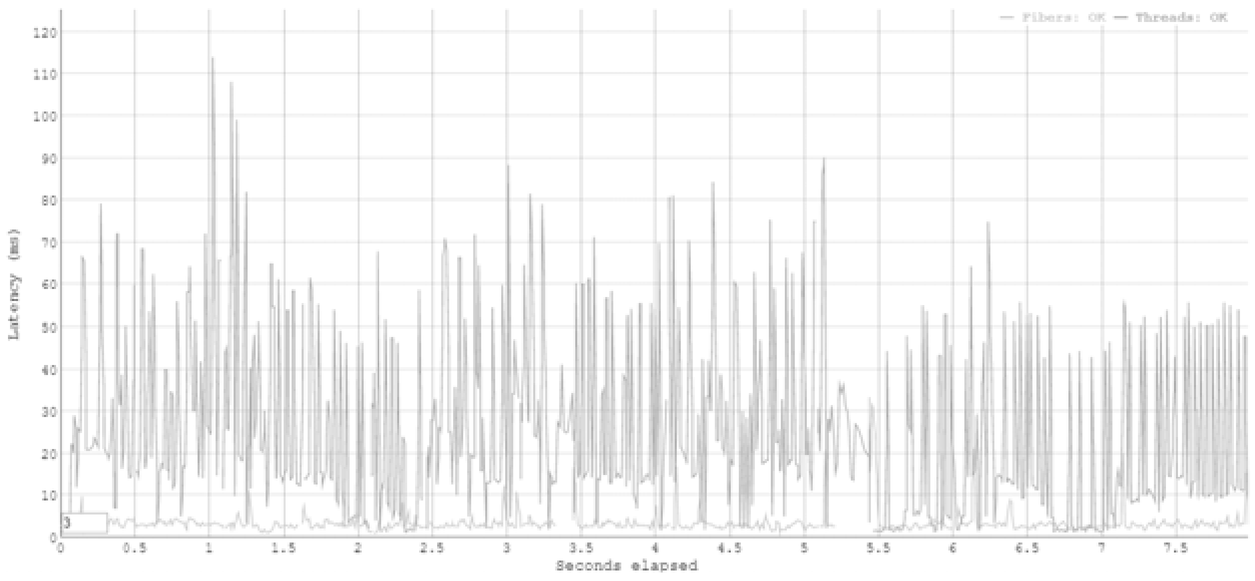


Figure 1. Variations of latencies in attacks of 8 seconds duration with scheduled fibers and threads (1 millisecond sleep)

Both threads and fibers were assigned one Runnable object, which represented a task that they would perform. In our first test case, the body of Runnable set a Thread to sleep for 1 millisecond. Similarly, the wait method was invoked on Fiber which correlates with the sleep method invoked on Thread object. In the second case, Thread and Fiber waited for 10 milliseconds. The sleep period could represent a task with constant duration. With this kind of test that performs a simple task, we were able to investigate the speed of scheduling and execution by the JVM and underlying OS.

In the figures, the vertical axis represents latency in milliseconds, and the horizontal axis represents the duration of each request, which was 8 seconds for every run. Based on the conducted results, the total execution of the task with fibers had on average 6.15 times lower latency than threads in heavier-load tests (with passing the greater number as a parameter to either sleep or the wait method) and around 1.02 times faster on tests where the duration of waiting was 10 times shorter. On average, fiber latency is approximately 16.26% of thread latency, which makes them more efficient in heavier-load cases. In both cases, fibers' amounts of latencies are more constant than threads and do not have significant variations in latency.

In the tests with shorter sleep periods, fiber's startup period is significantly longer, and more time is needed for them to reach stable execution. However, in a heavier-load test case, the startup difference between threads and fibers at the beginning of the runtimes is significantly lower. According to these results, we might conclude that fibers are able to handle an extensive workload more efficiently in comparison to lightweight scheduled tasks.

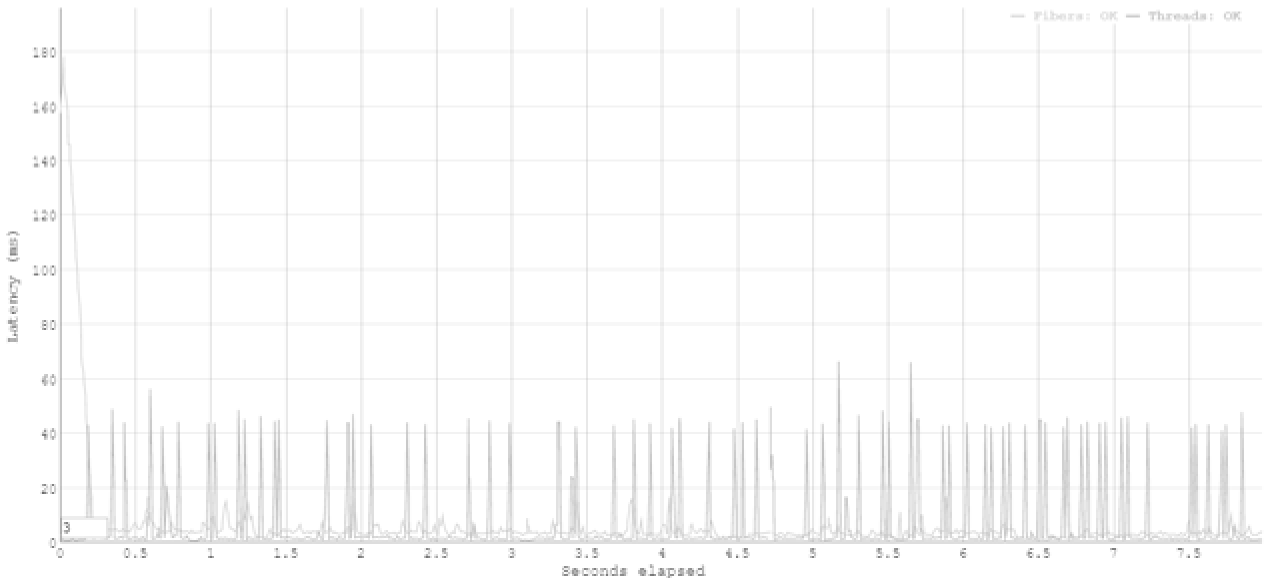
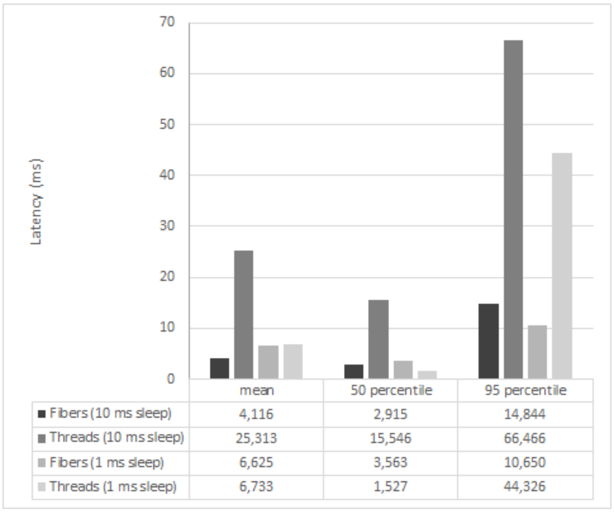


Figure 2. Variations of latencies in attacks of 8 seconds duration with scheduled fibers and threads (10 milliseconds sleep)



	mean	50 percentile	95 percentile
■ Fibers (10 ms sleep)	4,116	2,915	14,844
■ Threads (10 ms sleep)	25,313	15,546	66,466
■ Fibers (1 ms sleep)	6,625	3,563	10,650
■ Threads (1 ms sleep)	6,733	1,527	44,326

Figure 3. Amount of latency for fibers and threads (with 1 and 10 milliseconds sleep) - mean value, 50th percentile, and 95th percentile

When compared to different amounts of time taken for the waiting for responses in all test cases, the arithmetic mean of the latencies of all requests in an attack was the longest in case of threads execution with a sleep of 10 milliseconds. The average latency rises up to 20 ms, which takes the longest period of time to receive responses. In comparison to fibers, scheduled to wait for an equal time period of 10ms, fibers performed remarkably better with an average latency of 4.116 ms.

The only test case in which threads outperformed fibers is visible in the case with regard to the maximum duration of waiting for the responses in one attack, where latency rises up to 170ms in cases of fibers with a wait time of 1 millisecond. However, on the 50th percentile of all requests' latencies, the amount was the highest in the thread-case scenario in which each thread sleeps for 10 milliseconds. A similar report is visible on the 95th percentile, where threads scheduled to wait for 1 and 10 milliseconds had a notably higher amount of latencies.

The figures represent the average results of the results gathered after 3 iterations of each test case. The fibers had constant request rates with a quantity of 75.12, but with the increase of sleep duration by 10 times, in the case of threads, the rate increased by 1%. When performed lighter-load tests, with a shorter wait period, fibers on average performed slightly better than threads with average latency that is 0.11 milliseconds lower. However, heavier-load tests present results with a difference in the average latency of 21.2 milliseconds.

VI. Discussion

In [14], Cuadrado and Aracil presented scheduling techniques for model-to-model transformations (M2M) with which they had performed reverse engineering based on continuations. Whenever a particular rule requires a target element that is not yet created, it saves its state in a continuation which might be resumed later, when available for use. During the explained event, other nondependent parts of transformation are being executed concurrently. Based on conducted performance testing, their model transformation based on continuations has shown to be efficiently scalable and brought useful methodologies to deal with language interoperability.

Stadler's, Würthinger's, and Wimmer's implementation of fibers in JRuby [8], the second most used implementation of Ruby programming language, uses the underlying system of coroutines that handles a great amount of workload during the applications' runtimes. The API for coroutines was introduced by modifying the Java HotSpot^{TM} VM. This system uses two strategies that enable the implementation of coroutines within JVM: each coroutine is allocated a separate stack and the stacks are copied to support larger amounts of coroutines to be allocated for the execution of the necessary tasks. Minimal changes in JVM were required to achieve an efficient performance of this novel system, including the addition of stack management code and new code in the Garbage Collection system. Different benchmarks in Ruby environments were used to evaluate the outcome of the coroutine support. The results prove that JRuby's implementation with coroutines is 7 times faster than JRuby without them. Therefore, the Java API for coroutines has shown to be beneficial to this programming language implemented on top of the JVM.

A benchmark that performs recursive method calls in which each function captures and resumes a continuation was performed to measure the performance of the algorithm that lazily stores the contents of continuations in [13]. This approach is implemented for the interpreter and the compiler for Java HotSpot^{TM} VM, but could be applicable in other Java VMs and might operate in runtimes of other programming languages. The results show that runtimes increase linearly with the addition of local variables handled within each continuation. Also, the duration of resuming a continuation is lower than capturing because there is no need for memory allocation. The algorithm for the execution of storing contents in continuation and resuming a continuation uses an optimized approach by restoring only stack frames that are not bound to the current thread's state. In comparison to continuation libraries that do not modify JVM, the performances of VM-based implementation of continuations are shown to be more efficient.

The coroutine support in the Abstract Behavioral Specification (ABS) system performs resuming of execution by the underlying scheduler, which provides the avoidance of errors [9]. With implementing such a feature, the explicit command to invoke the resuming of an operation is not allowed and the program quality is improved. Instead of creating a thread for every asynchronous function call, such calls are stored within Callable objects that represent tasks. The architecture consists of a thread-pool in which at most one thread is assigned to an actor (a lightweight representation of a thread), a queue assigned to each actor and each thread that holds an actor executes the main task that iteratively runs tasks from the assigned queue. The instances of ABSTask objects are created by calling send or spawn method and are stored in a queue of an actor which was generated to hold a group of tasks. When a particular task terminates, its return value is saved in the field of ABSFuture object which keeps track of completed tasks. ABSFuture implements a mechanism that notifies the actors about the tasks' completions. A test-case example that creates a stack of synchronous, recursive calls was used to measure the impact of coroutine implementation on the overall performance. Storing the tasks in heap memory instead of using native threads for each function call resulted in a decrease of memory overhead.

As explained before, Fiber and FiberScope API is still experimental and under construction. However, after research about available solutions with similar approaches to structured concurrency in the form of concrete lightweight thread implementations, we believe the further development of such features might show promising results in future updated versions of JDK. As most of our tests' results showed a better performance of fibers in terms of low latency and higher throughput, we support the claim that suggests handling requests concurrently without reliance on the memory provided by the operating system. We believe that the cost of external resources might decrease by the replacement of the first version of threads with the integration of some ideas of these novel approaches into future Java releases.

VII. Conclusion

Due to the insufficient amount of support that Java receives from kernel native threads, this novel approach to achieving concurrency with the concrete implementation of JVM's internal threads in the form of fibers brings an appropriate solution for improving the performance. Based on conducted reports and our preliminary results, the system which incorporates these lightweight threads promises future progress in handling heavy workloads that might appear during the execution of multi-threaded server applications. Furthermore, Project Loom's features enable the code scalability and Fibers might allow Java programs to manage concurrency under the heavy-load in a more efficient way. Although this fiber-based mechanism is still declared as experimental, we believe that further development could be promising and definitely worth exploring. With this novel approach, we might be able to increase the level of optimization and reduce costs when providing services to a large number of users.

References

[1] N. J. Smith, "Notes on structured concurrency, or: Go statement considered harmful," April 25, 2018. https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-considered-harmful/

[2] E. W. Dijkstra, "Notes on Structured Programming," 2nd ed., T. H.-Report 70-WSK-03, April 1970. https://www.cs.utexas.edu/~EWD/ewd02xx/EWD249.PDF

[3] C. Tismer, "Continuations and Stackless Python," in Proc. of the 8th International Python Conference, vol. 1, 2000.

[4] Z. Li and E. Kraemer, "Programming with Concurrency: Threads, Actors, and Coroutines," in Proc. of the 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum (IPDPSW), Boston, USA, May 2013, pp. 1304-1311.

[5] R. Malhotra, "Java Language and Ecosystem Recap," in Rapid Java Persistence and Microservices: Persistence Made Easy Using Java, EE8, JPA and Spring, Apress, Berkeley, 2019.

[6] R. Pressler, "Project Loom: Fibers and Continuations for the Java Virtual Machine," August 23, 2018. https://cr.openjdk.java.net/~rpressler/loom/Loom-Proposal.html

[7] M. Sústrik, "Structured Concurrency," February 7, 2016. http://250bpm.com/blog:71

[8] L. Stadler, T. Würthinger, and C. Wimmer, "Efficient Coroutines for the Java Platform," in Proc. of the 8th International Conference on the Principles and Practice of Programming in Java, pp. 20-28, 2010.

[9] V. Serbanescu, F. de Boer, and M. M. Jaghoori "Actors with Coroutine Support in Java," in Formal Aspects of Component Software (FACS 2018), K. Bae and P. Ölveczky, Eds. Lecture Notes in Computer Science, vol. 11222, pp. 237-255, Springer, Cham, October 05, 2018.

[10] P. Pufek, H. Grgić, B Mihaljević, "Analysis of Garbage Collection Algorithms and Memory Management in Java," in Proc. of the 42nd Internationa. Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, 2019, pp. 1677-1682.

[11] H. Grgić, B. Mihaljević, and A. Radovan, "Comparison of Garbage Collectors in Java Programming Language," Proc. of the 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, 2018, pp. 1539-1544.

[12] P. Sestoft, "Programming Language Concepts," 2nd ed., Undergraduate topics in computer science, Springer, Berlin, 2017.

[13] L. Stadler, C. Wimmer, T. Würthinger, H. Mössenböck, J. Rose, "Lazy Continuations for Java Virtual Machines," in Proc. of the 7th International Conference on Principles and Practice of Programming in Java (PPPJ '09). Association for Computing Machinery, New York, USA, 2009, pp. 143-152.

[14] J. S. Cuadrado and J. P. Aracil, "Scheduling model-to-model transformations with continuations," in Software: Practice and Experience, vol. 44, issue 11, November 2014, pp. 1351-1378.