

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ
ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение
высшего образования

УРАЛЬСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
имени первого Президента России Б.Н. Ельцина

ИНСТИТУТ ЕСТЕСТВЕННЫХ НАУК И МАТЕМАТИКИ

Кафедра алгебры и фундаментальной информатики

**ОПТИМИЗАЦИИ МЕХАНИЗМА FIBERS В
JVM ЯЗЫКАХ ПРОГРАММИРОВАНИЯ**

Направление подготовки 02.03.01 «Математика и
компьютерные науки»

Заведующий кафедрой:
д. ф. м. н., проф. М.В. Волков

Выпускная квалификационная
работа бакалавра

**Башкарова Ивана
Андреевича**

Нормоконтролер:
д. п. н., проф. А.Г. Гейн

Научный руководитель:
А.Х. Хакимов

Научный соруководитель: д. ф.
м. н. В.А. Баранский

Екатеринбург
2023

РЕФЕРАТ

Башкаров И.А. Оптимизации механизма fibers в JVM языках программирования, Выпускная квалификационная работа: УрФУ, 2023, стр. 34, рис. 1, табл. 9 ; библ.: 10 назв.

Ключевые слова: файберы, Project Loom, JVM, виртуальные потоки, асинхронность, Cats Effect 3.

В данной работе рассмотрены теоретические аспекты файберов в JVM языках программирования, их преимущества и недостатки в сравнении с альтернативным подходом к описанию асинхронных операций, таким как Project Loom и виртуальные потоки. Проведено сравнение механизма файберов с виртуальными потоками JVM. Также исследована возможность комбинирования двух механизмов и рассмотрены примеры программ, в которых комбинирование данных подходов позволяет добиться наилучшего результата.

МЕСТО ВЫПОЛНЕНИЯ РАБОТЫ

Выпускная квалификационная работа была выполнена на кафедре алгебры и фундаментальной информатики института естественных наук и математики уральского федерального университета имени первого президента России Б.Н. Ельцина.

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ И СОКРАЩЕНИЙ	3
ВВЕДЕНИЕ	5
ОСНОВНАЯ ЧАСТЬ	6
1 Постановка задачи	6
2 Анализ предметной области	6
2.1 Продолжения. Программирование в стиле передачи про- должений.	6
2.2 Неблокирующие операции	8
2.3 Монада IO	9
2.4 Файберы в Scala	11
2.5 IO-нотация	12
2.6 Виртуальные потоки в Java	18
3 Практическая часть	20
3.1 Техническое окружение, используемое в практической части	20
3.2 Реализации файберов	20
3.3 Методика замеров производительности	21
3.4 Задача прохождения сообщения по цепочке акторов . .	21
3.5 Задача обеспечение многопоточного доступа к изменя- ющимся данным	25
3.6 Задача осуществления асинхронных сетевых взаимо- действий	30
ЗАКЛЮЧЕНИЕ	32
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРА- ТУРЫ	33

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Мьютекс - структура, обеспечивающая взаимное исключение доступа к общим ресурсам или фрагментам кода. [1]

Сборщик мусора — это низкоприоритетный процесс, который запускается периодически и освобождает память, использованную объектами, которые больше не нужны.

Асинхронность - способ организации выполнения задач, при котором код может продолжать работу, не ожидая завершения определенной операции или функции.

JVM - это виртуальная машина, которая выполняет Java-байткод. Является ключевой компонентой платформы Java и позволяет запускать Java-приложения на различных операционных системах.

Файберы (fibers) - это легковесные потоки выполнения, представляющие собой механизм, позволяющий создавать сопрограммы или подпотоки, которые могут выполняться независимо друг от друга, с возможностью приостановки, возобновления и передачи управления между собой.

Интерпретация языка - это процесс выполнения программы, написанной на определенном языке, с помощью интерпретатора. Интерпретатор анализирует код программы, выполняя соответствующие действия на каждом шаге.

Аллокация в JVM - относится к процессу выделения памяти для объектов и массивов во время выполнения программы на Java или других языках, компилируемых в байт-код JVM. Аллокация памяти в JVM происходит в куче (heap), которая является областью памяти, управляемой JVM.

Project Loom - проект, разрабатываемый в рамках открытого проекта OpenJDK, который нацелен на улучшение масштабируемости и производительности приложений на Java, особенно в области конкурентности и параллелизма. Он предоставляет новые средства разработки для работы с потоками и асинхронными операциями. [2]

Пул потоков - механизм управления и переиспользования потоков в приложении. Вместо создания и завершения отдельных потоков для каждой задачи, пул потоков предварительно создает определенное количество пото-

ков и управляет ими для выполнения поступающих задач.

Рантайм файберов - подпрограмма, занимающаяся интерпретацией Ю-нотации и исполнением программы, описанной с его помощью.

Сокет - название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одной ЭВМ, так и на различных ЭВМ, связанных между собой сетью. Сокет — абстрактный объект, представляющий конечную точку соединения.
[3]

ВВЕДЕНИЕ

В современных языках программирования одним из основных требований является поддержка неблокирующих операций. Задачи, предполагающие ожидание некоторого внешнего вычисления, находящегося за пределами процесса инициатора, могут быть оформлены различным образом. Блокирующее ожидание подразумевает, что процесс инициатор будет задействовать поток операционной системы для того, чтобы ожидать внешнего события. Неблокирующее же ожидание предполагает, что поток, в котором был инициирован запрос к внешней системе, будет освобожден до тех пор, пока внешнее вычисление не будет завершено. И лишь после завершения внешнего вычисления работа будет продолжена. Это позволяет создавать меньше потоков, поскольку открывается возможность переиспользования потоков, которые, в случае с блокирующими операциями, могли быть заблокированы.

Вопрос далее сводится к двум основным моментам - как неблокирующие операции реализованы внутри, как они представлены в коде. В данной работе особенное внимание будет уделено внутреннему устройству неблокирующих операций в JVM языках программирования, рассмотрен механизм файберов, а также нововедение в JVM последних версий - виртуальные потоки.

Одним из интересных механизмов, открывающим новые возможности оптимизаций файберов, являются виртуальные потоки представленные в рамках Project Loom. Их отличительная особенность состоит в том, что, в отличие от классических файберов, используемых в языке Scala, они реализованы на уровне виртуальной машины Java, что позволяет им, за счет использования более простых структур и других оптимизаций, быть оптимальнее по потребляемой памяти и скорости работы. [4]

В данной работе будет представлен подход, позволяющий комбинировать файберы, используемые в языке Scala, и виртуальные потоки, представленные в Project Loom. Будут проведены замеры производительности, показывающие, что этот подход позволяет превзойти классические файберы в некоторых задачах.

ОСНОВНАЯ ЧАСТЬ

1 Постановка задачи

На текущий момент фиберы, особый вид легковесных потоков, распространенных в Scala, являются самым эффективным из известных способом интерпретации и исполнения функционального языка, описывающего ход выполнения программы в языке Scala. Фиберы и особый функциональный язык являются важнейшей составляющей современного программирования на Scala, однако обладают некоторыми недостатками. Один из таких недостатков состоит в том, что механизм фиберов не полагается в полной мере на возможности виртуальной машины, на которой впоследствии будет исполняться. Среди таких возможностей особенно можно выделить виртуальные потоки, которые способны решать некоторые задачи, возложенные сегодня на фиберы, более эффективно, за счёт непосредственной поддержки на уровне JVM.

Необходимо найти и рассмотреть задачи, в которых фиберы, адаптированные под виртуальные потоки из Project Loom, использующие преимущества, которые он способен дать, показывают результат лучший, нежели классические фиберы, не полагающиеся на виртуальные потоки. Основными показателями, которые позволят сравнивать те или иные способы решения данной задачи на предмет эффективности, будут являться время выполнения программы и объем аллокаций, создаваемых программой.

2 Анализ предметной области

2.1 Продолжения. Программирование в стиле передачи продолжений.

Дана некоторая задача, которая имеет возвращаемый результат, обозначаемая как **Task[Int]**. Продолжением этой задачи будет называться некоторая функция, зависящая от результата выполнения данной задачи, возвращающая новую задачу.

Таким образом, можно определить новую задачу, которая будет получаться из комбинирования некоторой исходной задачи и ее продолжения.

Пример неблокирующей операции:

Исходная задача: Вычисление $\sin(\frac{\pi}{6})$

Продолжение: Возведение в квадрат.

Комбинируя исходную задачу и продолжение, можно получить задачу, которая сначала находит синус $\frac{\pi}{6}$, а затем возводит полученный результат в квадрат.

Программирование в стиле передачи продолжений заключается в том, что все задачи либо достаточно просты, либо сложны и представляют из себя комбинации некоторой исходной задачи (которая тоже может являться комбинированной), и цепочки продолжений. Такой подход более принято называть continuation-passing style, либо же просто CPS. Далее в работе будет использоваться именно эта аббревиатура. Пример последовательных вычислений, описанных в стиле продолжений, приведен на рисунке 1.

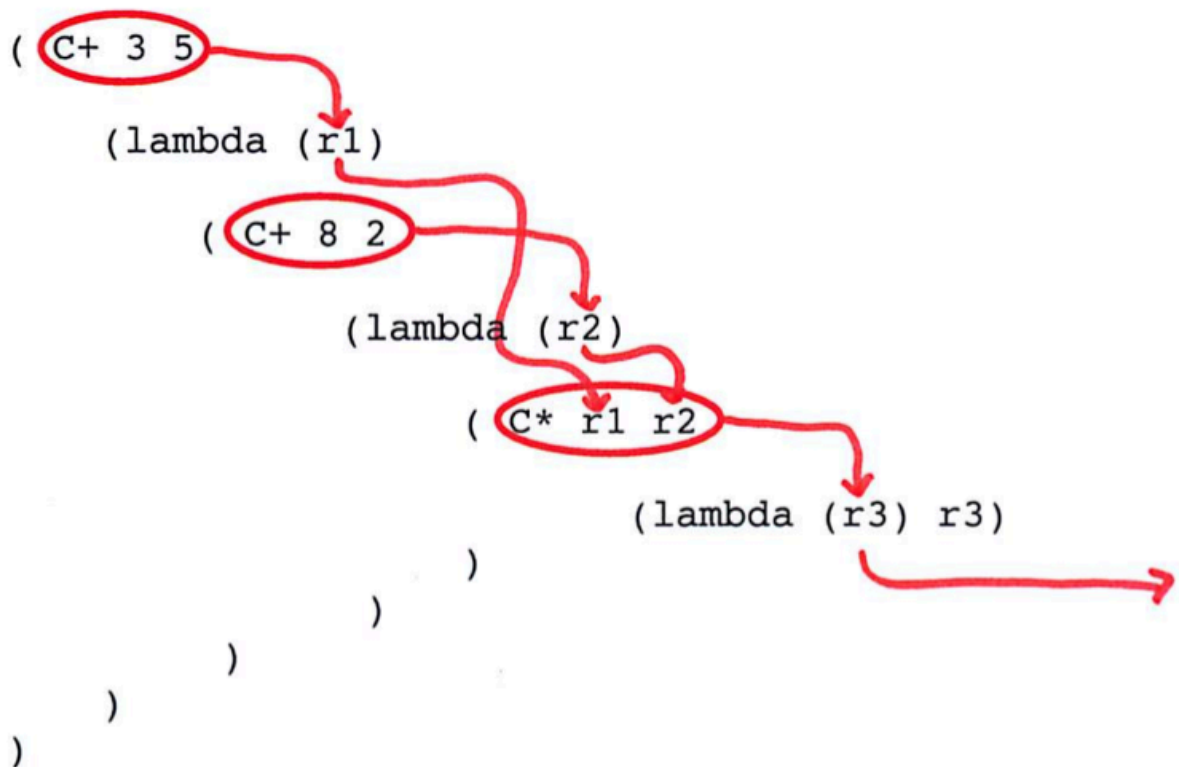


Рисунок 1: Пример программирования в стиле продолжений

1. Callbacks

Одна из самых первых реализаций CPS. Подход состоит в том, что все задачи, к которым хотелось бы назначить продолжение, принимают

некоторый параметр `callback`, представляющий из себя продолжение. Как только задача выполнится, она вызовет этот параметр `callback`, передаст ему результат своего выполнения, запустит дальнейшее выполнение цепочки задач.

2. Монады

Реализация CPS, зародившаяся в функциональных языках программирования, в частности в Haskell. Здесь задача представлена некоторым контейнером `IO[A]`, где `A` - тип результата выполнения задачи. Также у задачи есть функция `flatMap`, иногда также называемая `bind` или просто `>>=`. Эта функция позволяет прикрепить к задаче какое-то продолжение и возвращает новую задачу - комбинацию исходной и ее продолжения.

Классический CPS предполагает, что продолжение задачи явно указывается в коде как продолжение. Но в некоторых языках CPS появляется только после этапа компиляции, а в самом исходном коде отсутствует. Один из интересных примеров такого подхода - язык Kotlin и его корутины. В исходном коде все вызовы выглядят как обычные последовательные вычисления, однако если одно из этих вычислений помечено ключевым словом `suspend`, компилятор переводит функцию в вид CPS.

2.2 Неблокирующие операции

Продолжение задачи не обязательно должно выполниться тотчас на том же потоке. CPS предполагает более высокую гибкость в управлении тем, как задачи будут выполняться друг за другом. Например, такой подход не накладывает никаких ограничений на привязанность задач к конкретному потоку. Исходная задача может быть выполнена на одном потоке, а ее продолжение на другом. Что самое важное - CPS предоставляет достаточную гибкость для того, чтобы с его помощью описывать неблокирующие операции.

Пример:

Исходная задача: прочитать данные из файла

Продолжение: вывести данные в консоль

Комбинация: поскольку чтение данных из файла представляет из себя операцию, время выполнения которой зависит не только от действий, связан-

ных с потоком инициатором, а по большей части упирается во время работы жесткого диска, после отправки запроса на чтение данных из файла, поток может быть освобожден от текущей задачи. Как только данные будут считаны с жесткого диска и доставлены в оперативную память, будет запущено продолжение - вывод данных в консоль.

Таким образом, если использовать в приложении ограниченный пул потоков, один из потоков будет занят, выполняя данную задачу лишь небольшое количество времени. Намного дольше будет производиться чтение с жесткого диска, но в этот момент поток будет свободен от текущей задачи.

2.3 Монада IO

Под монадой в Scala принято понимать некоторый контекст $F[_]$ для другого типа A , обладающий следующими свойствами:

1. В этот контекст можно что-то обернуть, то есть определено отображение $A \Rightarrow F[A]$ для любого типа A
2. Определено отображение $(F[A], (A \Rightarrow F[B])) \Rightarrow F[B]$, также называемое связыванием вычислений.

На языке Scala такой контекст представим следующим интерфейсом:

```
sealed trait Monad[F[_]] {  
  def pure[A](t: A): F[A]  
  def flatMap[A, B](el: F[A])(f: A => F[B]): F[B]  
}
```

Далее считается, если для типа высшего порядка $F[_]$ определен экземпляр такого интерфейса, тип $F[_]$ является монадой.

В качестве примера, можно определить экземпляр интерфейса **Monad**[F[_]] для списка. Тогда список будет считаться монадой.

```
val listMonad = new Monad[List] {  
  override def pure[A](t: A): List[A] = List(t)
```

```

override def flatMap[A, B](el: List[A])(
    f: A => List[B]
): List[B] =
    el.map(f).foldLeft(List.empty[B]) {
        case (a, b) => a ::: b
    }
}

```

Далее можно использовать этот экземпляр **Monad[List]** для двух необходимых операций:

- 1) Положить элемент в контекст списка. Согласно реализации будет создан список, состоящий из одного элемента
- 2) Отображение листа с использованием функции, которая каждый элемент списка отображает в новый список, определено таким образом, что все элементы списка отображаются сначала в новые списки, а затем эти списки конкатенируются.

Особенно важным понятием для языка Scala является монада IO, которую нельзя не упомянуть, прежде чем приступить к рассмотрению механизма фиберов. Монада IO представляет из себя контекст вычислений. Контекст можно обозначать следующим образом: **IO[]**. Можно рассматривать экземпляр типа **IO[A]** как задачу, которая может быть выполнена. В процессе выполнения будет вычислен результат, обладающий типом A. То есть, в общем случае существует отображение **IO[A] => A**, называемое запуском вычисления. Особенность написания программ с использованием монады IO состоит в том, что запуск вычисления происходит лишь единожды, в одном участке программы, называемым концом мира. Верно будет утверждать, что программа представляет из себя некоторое вычисление, имеющее тип **IO[ExitCode]**, а далее в конце мира, в классе запуска программы, происходит запуск этого вычисления.

Монады IO представляют особый интерес в контексте чистоты функций. Есть два важных для функционального программирования понятия: чистота функций и ссылочная прозрачность. Под чистотой функции понима-

ется то, что она не имеет никаких побочных эффектов. То есть, функцию можно рассматривать как математическое отображение из одного множества в другое. Аналогами множеств в функциональных языках программирования являются типы, а их элементы называются обитателями. Таким образом, чистая функция ставит каждому обитателю входящего типа в соответствие обитателя результирующего типа, причем такое отображение строго детерминировано, не зависит от состояния программы, не изменяет состояние программы. Другое важное понятие это ссылочная прозрачность. Под ссылочной прозрачностью понимается свойство программы, предполагающее, что замена любого выражения на произвольное эквивалентное по возвращаемому результату выражение, не способна оказать никакого значимого для хода выполнения влияния на программу.

Сторонним эффектом называется некоторая деятельность, выполняющаяся функцией, которая:

1. Влияет на возвращаемый результат функции, при этом зависит не только от значений аргументов.
2. Изменяет состояние программы или системы, которое влияет на ход выполнения программы и область видимости которого расположена за пределами текущей функции.
3. Взаимодействует с операционной системой напрямую

Примеры стороннего эффекта: функция, выполняющая поход в сеть; функция, печатающая в консоль или файл; функция, вычисляющая случайное значение.

2.4 Файберы в Scala

Файбер является частным случаем легковесного потока. Каждому файберу в соответствие ставится определенная задача. Файбер выполняет эту задачу, используя потоки уровнем ниже, например потоки операционной системы. Полезное свойство файберов состоит в том, что они в любой момент способны поставить на паузу своё выполнение, не занимая реальных потоков, то есть отдать ресурсы другим файберам. Все состояние файбера хранится в

JVM куче и он может быть в любой момент возобновлен. Это используется, например, для выполнения асинхронных операций.

Файберы представляют из себя цепочку последовательных вычислений. Они, в отличие от потоков JVM, могут быть остановлены извне без прерывания реального потока, и прекратят свое выполнение на одном из звеньев цепочки.

Монада IO, в сущности, лишь способ описания программы. Он не накладывает сильных ограничений на то, как именно программа будет выполняться. Выполнение монады IO определяет отображение $IO[A] \Rightarrow A$. Полученная в результате комбинирования различных элементов программы задача $IO[ExitCode]$, представляющая из себя программу целиком, в сущности является лишь описанием программы с использованием IO-нотации. Вообще говоря, любая монада IO представима в виде синтаксического дерева, полученного путем комбинирования различных структур, представленных в IO-нотации. Выполнение программы представляет из себя интерпретацию такого синтаксического дерева. Именно такая интерпретация и определяет отображение $IO[A] \Rightarrow A$. Вычисление эффекта не является чистым отображением и именно поэтому его принято использовать лишь в конце мира, чтобы весь остальной код представлял из себя чистые функции.

Файберы в Scala - один из способов интерпретации IO эффектов. Он предъявляет к IO-нотации несколько основных требований:

1. Создание задачи поверх некоторого выражения.
2. Операция связывания задач, где каждая последующая зависит от результатов выполнения предыдущих задач.
3. Описание параллельных вычислений.
4. Описание асинхронных вычислений.
5. Описание отмены задачи.

2.5 IO-нотация

Под IO-нотацией понимается множество различных выражений, описывающих программу, а также специфику её выполнения. Каждое такое вы-

ражение представляет из себя монаду IO и может быть получено как специальными операциями, оборачивающими какие-либо вычисления, так и путем комбинирования задач через операцию связывания. IO-нотация является краеугольным камнем функциональных языков, поскольку он позволяет явно описывать различные вычислительные процессы, такие как асинхронность, параллельность, предоставляет более удобный интерфейс работы с ошибками, контекстами, ресурсами и многим другим. Зачастую IO-нотации сопряжены также с фибер языками. Под фибер языком следует понимать некоторый язык, имеющий достаточную выразительность для того, чтобы его можно было интерпретировать с помощью фиберов. Например, в языке Kotlin фибер язык не представлен IO-нотацией, но при этом удовлетворяет всем достаточным условиям для того, чтобы его можно было выполнять на фиберах, называемых в Kotlin корутинами.

В Kotlin фибер язык представляет из себя окраску функций ключевым словом `suspend`, которое говорит о том, что вычисление, потенциально, будет выполняться на отдельном фибере, на легковесном потоке, а не на том потоке, который вызывает данную функцию. Это добавляет некоторые ограничения на использование данной функции, поскольку для ее вызова необходимо чтобы, либо функция вызывалась из другой `suspend` функции, что эквивалентно комбинированию задач с использованием IO-нотации, либо вызов функции должен производиться явно из специального **CoroutineScope**, который исполнит эту функцию не на вызывающем потоке, а на некотором корутине, то есть на фибере.

IO-нотация важна не только тем, что она является фибер языком и её можно интерпретировать на фиберах, но и в том, что такой она делает все вычисления прозрачными. Такими становятся асинхронные, параллельные операции, поскольку имеются явные методы и конструкции нотации, которые указывают на то, что задачи будут выполняться соответствующим образом. IO-нотация делает программу более ясной, позволяет обеспечить более высокую надежность за счет того, что многие детали выполнения отражены на уровне типов.

При этом сама IO-нотаций накладывает существенную нагрузку на работу приложения, поскольку его интерпретация требует большого количе-

ство аллокаций и дополнительных обработок.

Пример IO-нотации:

Создание задачи поверх некоторого выражения зачастую определяется следующей конструкцией:

```
val task: IO[Int] = IO.delay(  
    1008 * 4000  
)
```

У контекста есть некоторый метод, который принимает выражение, при этом не вычисляя его сразу, создает задачу, имеющую результирующий тип как у этого выражения.

Комбинирование задач:

```
val task: IO[Int] = IO.delay(  
    1008 * 4000  
)  
  
def andMultiple2x(i: Int): IO[Int] = IO(i * 2)  
  
val task2 = task.flatMap(res => andMultiple2x(res))
```

Для комбинирования как правило используется метод `flatMap`.

Параллельное выполнение задач:

```
(task, task).parMapN {  
    case (res1, res2) => res1 + res2  
}
```

Допустимо наличие методов, которые принимают список задач и возвращают новую задачу, которая вычисляет каждую из исходных задач параллельно.

Асинхронное выполнение задач:

```
val io: IO[Int] = IO.async(
  cb => IO(
    future.map(result => cb(Right(result)))
  ).as(None)
)
```

Обычно вводится функция, которая предоставляет callback. Его необходимо вызвать, как только произойдет событие и нужно будет возобновить дальнейшее выполнение.

Отмена вычисления:

```
val fiber: IO[Fiber[IO, Throwable, Int]] = io.start
fiber.flatMap(_ .cancel)
```

Как правило предоставляется метод start, который позволяет параллельно запустить задачу и возвращает интерфейс управления файбером, называющийся Fiber. У этого объекта есть метод cancel, позволяющий отменить выполнение файбера.

IO-нотация представлена следующими составляющими:

```
sealed abstract class Rei[+A](val tag: Int)

case class Pure[A](f: A) extends Rei[A](0)
case class Delay[A](f: () => A) extends Rei[A](2)
case class Async[A](
  cb: (A => Unit) => Rei[Unit]
) extends Rei[A](5)
case class FlatMap[E, A](
  u: Rei[E],
  fu: E => Rei[A]
) extends Rei[A](7)
```

Интерпретация IO-нотации происходит на некотором фибере. В следующем примере реализации фиберов на языке Scala **Rei[A]** - представляет из себя задачу, монаду IO, имеющую возвращаемый тип - *A*. Приведенный фибер представляет из себя класс, создаваемый поверх некоторой задачи, которую необходимо интерпретировать. Далее, фибер можно запустить, вызвав метод *runAsync*. Он начнет интерпретировать IO-дерево, также вызывая различные сторонние эффекты.

```
class Eva00Fiber[A](rei: Rei[A], cb: A => Unit)
  extends Fiber[A] {
  def run(): Unit = {
    ???
  }
  override def join: Rei[A] = Join(this)
}
```

Метод *runAsync* определяет логику интерпретации монады IO. Выполнение этого метода происходит с использованием внутренней рекурсивной функции, называемой *runLoop*. Данная функция имеет 2 аргумента: указатель на текущую разбираемую задачу и стек продолжений, которые необходимо выполнить, как только текущая задача будет выполнена.

```
def run(): Unit = {
  @tailrec
  def runLoop(
    current: Rei[Any],
    conts: List[Any => Rei[Any]]
  ): Unit = {
    current match {
      case Pure(f) => ???
      case Delay(f) => ???
      case Async(cb) => ???
      case FlatMap(u, fu) => ???
    }
  }
}
```

```

    }
  }
}

```

В зависимости от того, на какой тип задачи указывает ссылка `current`, принимаются различные действия.

Если ссылка указывает на структуру **Pure**, есть два варианта. Если текущая задача выполнена и у нее нет продолжений, то фибер считается завершенным и может вызвать функцию **cb**, сообщив о своем успешном завершении. Если же у текущей задачи есть продолжения, можно взять продолжение с вершины стека, применить к результату, лежащему в **Pure**, получить новую задачу, которая будет назначена текущей. Так, стек уменьшится:

```

case Pure(f) => conts match {
  case head :: tail => runLoop(head(f), tail)
  case Nil => cb(f.asInstanceOf[A])
}

```

Если же ссылка указывает на структуру **FlatMap**, то его необходимо интерпретировать как дополнительное продолжение, примененное к внутренней задаче:

```

case flatMap: FlatMap[Any, Any] =>
  runLoop(flatMap.u, flatMap.fu :: conts)

```

Если ссылка указывает на структуру **Async**, то необходимо запустить выполнение асинхронной задачи, а затем отдать управление потоком. Как только асинхронная задача выполнится, фибер продолжит выполняться на другом освободившемся потоке:

```

case Async(cb) => {
  val continue: Any => Unit =

```

```

    res => executor.execute(() => runLoop(Pure(res), conts))
  runLoop(cb(continue), Nil)
}

```

Если ссылку указывает на структуру **Delay**, которая означает отложенное вычисление, поэтому, в отличии от **Pure**, он принимает не экземпляр типа *A*, а функцию, которая вычислит такой экземпляр. Может интерпретироваться вычислением функции и конвертацией в **Pure**:

```
case Delay(f) => runLoop(Pure(f()), conts)
```

2.6 Виртуальные потоки в Java

Виртуальные потоки в Java, разработанные в рамках проекта Loom - нововведение, добавляющее поддержку легковесных потоков на уровне виртуальной машины Java. Эти потоки могут быть использованы в самом языке Java, но в рамках данной работы интерес к виртуальным потокам Java обусловлен тем, что они могут дать языку Scala и фиберам, в частности таким библиотекам как Cats Effect 3 и ZIO. Scala может использовать любой код, доступный Java, поскольку оба этих языка выполняются на одной и той же виртуальной машине. Это открывает возможность использовать виртуальные потоки в Scala.

Виртуальные потоки доступны по умолчанию в Java 21, но на текущий момент Java 21 находится в раннем доступе. Также они доступны в Java 20, но только при использовании специальных параметров запуска JVM, таких как **—enable-preview —add-modules jdk.incubator.concurrent**

Для виртуальных потоков Java был представлен следующий интерфейс взаимодействия:

Executor - класс, обладающий единственным методом, позволяющим выполнять произвольные функции особым образом, например, на пуле потоков. В Loom доступен Executor, который позволяет выполнять задачи на виртуальных потоках. Он может быть создан следующим образом:

```
val executor =  
Executors.newVirtualThreadPerTaskExecutor()
```

Каждая задача, выполненная с помощью такого `Executor`, попадает на новый виртуальный поток, который использует ресурсы потоков операционной системы. Это очень похоже на файберы, но происходит уровнем ниже, в JVM.

Особенно интересно устроена асинхронность. Когда задача выполняется на виртуальном потоке, все значимые блокирующие операции, такие как чтение из сокета, работа с блокирующими коллекциями, блокировка потока на заданное время, и некоторые другие системные вызовы, начинают работать в специальном режиме, который не приводит к блокировке потока операционной системе, а блокирует лишь виртуальный поток, освобождая реальный.

Основное преимущество виртуальных потоков JVM перед файберами в Scala состоит в том, что они выполняются на уровне JVM, а это позволяет использовать некоторые оптимизации. Например, сократить количество аллокаций, сократить потребление памяти. Это приводит к основной гипотезе - виртуальные потоки доступные в Loom должны быть эффективнее файберов.

Одним из недостатков файберов в Scala является то, что само описание IO-нотации требует от программы большого количества аллокаций, поскольку каждая монада IO является оберткой над вычислениями, которую необходимо аллоцировать. Этого не происходит в случае с виртуальными потоками, поскольку в написании Java кода с ними отсутствуют специфичные конструкции языка, которые могли бы привести к дополнительным аллокациям.

Несмотря на то, что виртуальные потоки Java в определенной степени несовместимы с файберами в языке Scala, поскольку интерфейс взаимодействия с ними, все же, предполагает процедурный стиль программирования и не заточен под функциональный, все равно остается возможность извлечь некоторую пользу из этого механизма и оптимизировать работу файберов, ускорив работу приложения. Самое важное из того, что может дать Project Loom файберам - это примитивное представление продолжений внутри самого JVM. Таким образом, некоторые асинхронные вызовы можно представ-

лять не с помощью продолжений на уровне языка Scala, а используя продолжения, доступные на уровне JVM. Этого возможно добиться, если возложить всю ответственность за асинхронность на виртуальные потоки Java, а на файберы возложить непосредственно интерпретацию IO-нотации, который в случае с использованием виртуальных потоков не будет вызывать дополнительных аллокаций, связанных с выполнением асинхронных операций, созданием продолжений на уровне языка Scala.

3 Практическая часть

3.1 Техническое окружение, используемое в практической части

- Процессор: AMD Ryzen 7 5800X3D 8-Core Processor
- ОЗУ: 32GB 3200 MT/s
- OS: Linux archlinux 6.3.4-arch1-1 x86_64 GNU/Linux
- Java: OpenJDK 21

3.2 Реализации файберов

В большей части замеров производительности в качестве файберов была выбрана простейшая реализация названная **Rei**. В некоторых сценариях она может уступать таким библиотекам как **Cats Effect 3** и **ZIO**, однако отлично подходит для исследования, поскольку её рантайм является упрощенным рантаймом из **Cats Effect 3**, и все выводы относительно файберов **Rei** будут верны также относительно **Cats Effect 3** - опять же, в силу схожести их рантаймов.

Также, в некоторых из замеров программа выполняется с использованием файберов, адаптированных под Project Loom. Адаптация состоит в том, что вся асинхронность в них перенесена на уровень виртуальных потоков за счет использования соответствующих блокирующих структур. Данная реализация файберов носит название ФСПВП (Файберы с поддержкой виртуальных потоков). Файберы, работающие на виртуальных потоках, но при этом

не оптимизированные под них, носят название ФВП (Файберы на виртуальных потоках).

3.3 Методика замеров производительности

Во всех замерах производительности будут сравниваться различные способы решения одной и той же задачи различными способами. Как правило, будут сравниваться такие реализации как: файберы, виртуальные потоки, файберы комбинированные с виртуальными потоками. Поскольку любое измерение производительности зависит не только от сложности выполняемой непосредственно программой работы, а также от работы сборщика мусора, общей нагруженности процессора и компьютера, все замеры будут предполагать 1000 повторений выполнения программы, что будет задавать выборку, достаточную для выявления тенденций.

Далее, под интенсивностью аллокаций будет пониматься средний объем памяти в мегабайтах, аллоцированный в ходе одной итерации выполнения программы. Под скоростью выполнения будет пониматься длительность выполнения одной итерации в миллисекундах. Интенсивность аллокаций важна по той причине, что она влияет на работу сборщика мусора. При высокой интенсивности аллокаций сборщик мусора начинает работать активнее, нагружая процессор, что приводит к замедлению выполнения всех программ, исполняемых на процессоре.

3.4 Задача прохождения сообщения по цепочке акторов

Актор - элемент программы, определенным образом обрабатывающий сообщения, приходящие в его почтовый ящик извне.

Задача определена следующим образом - дана цепочка из 100000 акторов, каждому из которых в соответствие поставлена очередь, представляющая из себя почтовый ящик, а также легковесный поток.

Каждый актор настроен таким образом, что, как только в его почтовый ящик поступает новое сообщение, он отправляет его следующему актору. Выстраивается цепочка, где изначально сообщение получает первый актор, а далее они передают это сообщение по цепочке, пока оно не достигнет последне-

го актора. Основные показатели, которые будут измеряться - время доставки сообщения от первого актора к последнему, а также объем аллоцированной памяти в ходе прохождения сообщения по цепочке. Почтовая очередь устроена таким образом, что, как только в ней оказывается сообщение, актор пробуждается, занимая некоторый свободный поток, а затем обрабатывает сообщения. Пока сообщений в очереди нет, актор не занимает никаких потоков.

Замер производительности 1.1. Файберы и виртуальные потоки:

В качестве очереди для актора в случае с **Project Loom** используется блокирующая очередь из языка Java с максимальным размером 16. В качестве очереди для программы с использованием файберов используется высокопроизводительная асинхронная очередь, использующая в своей основе механизм файберов, позволяющий им отдавать управление потоком операционной системы до тех пор, пока не произойдет некоторое событие (в данном случае событием будет являться пополнение очереди).

Цель: сравнить скорость прохождения сообщения по цепочке, а также интенсивность аллокаций для файберов и для Project Loom.

Результаты измерений представлены в таблице 3.4.1.

Таблица 3.4.1 - Замеры производительности

Рантайм	Среднее время на итерацию	Интенсивность аллокаций
Project Loom	90 мс	1.43 мб/ит
Файберы	110 мс	20.4 мб/ит

Анализ измерений: виртуальные потоки действительно показывают более высокую производительность, нежели файберы, при наличии большого числа взаимодействий с неблокирующей очередью. На файберах данная задача в среднем выполняется на 22% дольше, нежели на виртуальных потоках JVM.

Огромное различие заметно в интенсивности аллокаций. Файберы за одну итерацию аллоцируют почти в 20 раз больше памяти, нежели Loom, что говорит об значительно более эффективном использовании памяти в виртуальных потоках.

Замер производительности 1.2. Файберы на виртуальных потоках:

в текущем замере производительности описанная ранее задача будет выполняться на файберах, но в качестве потоков, поверх которых выполняются файберы, будут использоваться не потоки операционной системы, а виртуальные потоки.

Цель: сравнить скорость прохождения сообщения по цепочке, а также интенсивность аллокаций для файберов, для Project Loom, а также для запуска файберов на виртуальных потоках.

Результаты измерений представлены в таблице 3.4.2.

Таблица 3.4.2 - Замеры производительности

Рантайм	Среднее время на итерацию	Интенсивность аллокаций
Project Loom	90 мс	1.43 мб/ит
Файберы	110 мс	20.4 мб/ит
ФВП	160 мс	33 мб/ит

Анализ измерений: несмотря на то, что выполнение данной задачи на Project Loom создает лишь незначительные аллокации в размере 1.43 мб/ит, запуск файберов поверх Project Loom увеличивает интенсивность аллокаций более чем в полтора раза. При этом по производительности такой рантайм уступает как файберам на потоках операционной системе, так и виртуальным потокам.

Замер производительности 1.3. Файберы на виртуальных потоках с использованием блокирующей очереди: в третьем замере производительности будет рассмотрен запуск файберов на виртуальных потоках, но вместо асинхронной очереди и перебрасывания файбера на другой поток после того как выполнится операция чтения из очереди, виртуальный поток, привязанный к файберу, будет ставиться на паузу. Использоваться для этого будет механизм из Project Loom. Вместо асинхронной очереди и *Async* задачи будет использоваться **ArrayBlockingQueue**, который при использовании на виртуальных потоках не блокирует реальный поток операционной системы, а лишь виртуальный поток.

Цель: сравнить скорость прохождения сообщения по цепочке, а также интенсивность аллокаций для Project Loom, фиберов, а также для их комбинации с использованием виртуальных потоков и блокирующей очереди.

Результаты измерений представлены на таблице 3.4.3.

Таблица 3.4.3 - Замеры производительности

Рантайм	Среднее время на итерацию	Интенсивность аллокаций
Project Loom	90 мс	1.43 мб/ит
Файберы	110 мс	20.4 мб/ит
ФВП	160 мс	33 мб/ит
ФСВП	150 мс	10 мб/ит

Анализ измерений: запуск фиберов поверх виртуальных потоков, если использовать не стандартную асинхронную очередь, а блокирующую очередь **ArrayBlockingQueue**, незначительно увеличивает скорость работы программы по сравнению с запуском фиберов поверх виртуальных потоков с использованием асинхронной очереди, но при этом все равно уступает в производительности использованию Project Loom и фиберам на потоках операционной системы.

Однако, несмотря на отставание в производительности по сравнению с классическим применением фиберов, использование виртуальных потоков и блокирующей очереди позволяет значительно сократить количество аллокаций, уменьшив нагрузку на сборщик мусора и оперативную память. Такой эффект наблюдается за счет того, что асинхронные операции, которые являются крайне тяжелыми для рантайма фиберов, перекладываются на Project Loom. Вместо оформления асинхронности, сложной логики, вызова продолжения фибера на пуле потоков, происходит обычная постановка виртуального потока на паузу при чтении из блокирующей очереди, а затем возобновление. Всю ответственность за асинхронность берет на себя JVM.

3.5 Задача обеспечение многопоточного доступа к изменяющимся данным

Перед различными реализациями теперь будет стоять задача параллельного инкрементирования некоторой переменной. При этом, что важно, синхронизация потоков будет выполняться с помощью мьютекса. В следующих сценариях будут рассмотрены различные вариации выполнения этой задачи, будет производиться сравнение таких реализаций как: Project Loom, Cats Effect 3, ФСПВП. Основным элементом, который будет исследоваться в процессе последующих тестов, является мьютекс, который будет останавливать различные потоки от входа в критическую секцию мутирования переменной. Каждый поток инкрементирует число на 1000 единиц, при этом всего создается 100 задач, которые должны быть выполнены конкурентно. Итого, после выполнения полного цикла инкрементации число должно быть равно 100000.

Проверка необходимости примитивов синхронизации для обеспечения корректного многопоточного доступа к данным: в следующем за мере демонстрируется тот факт, что без использования соответствующих примитивов синхронизации программа работает некорректно и вместо ожидаемого значения, которое должно оказаться в переменной после того, как она будет многократно инкрементирована различными потоками, там находится другое число. Следующий код, используя пул из 12 потоков, периодически запускает задачу инкрементации числа.

Цель: убедиться, что наличие примитивов синхронизации необходимо для корректной работы программы при многопоточном доступе к переменной.

Результаты измерений представлены на таблице 3.5.1.

Таблица 3.5.1 - Замеры производительности

Номер итерации	Значение числа	Ожидаемое значение
1	100000	100000
2	99000	100000
3	99993	100000
4	98000	100000
5	99000	100000

Анализ измерений: по выборке в 5 полных итераций можно сделать вывод, что иногда число действительно равно 100000 и потоки не мешают друг другу, но случается, что потоки конфликтуют, и число не равно ожидаемому значению.

Замер производительности 2.1. Cats Effect 3, Project Loom и ФСПВП.

В текущем замере будет добавлен мьютекс, который не позволит нескольким потокам одновременно заходить в критическую секцию. Будет сравниваться производительность таких решений как: **Project Loom**, **Cats Effect 3** и **ФСПВП** с использованием виртуальных потоков. При чем для **ФСПВП** будут использоваться соответствующие блокирующие структуры, которые поддерживаются виртуальными потоками и позволяют блокировать только лишь виртуальный поток, но не блокировать поток операционной системы.

Цель: сравнить производительность приведенных решений, а также интенсивность аллокаций при использовании каждого из них.

Результаты измерений представлены на таблице 3.5.2.

Таблица 3.5.2 - Замеры производительности

Рантайм	Среднее время на итерацию	Интенсивность аллокаций
Cats Effect 3	147 мс	20 мб/ит
Project Loom	2 мс	0.24 мб/ит
ФСПВП	25 мс	5 мб/ит

Анализ измерений: выполнение данной задачи с использованием Project Loom происходит существенно быстрее, нежели с использованием файбе-

ров библиотеки **Cats Effect 3**, при этом потребляя меньшее количество памяти. Несмотря на это, фиберы, запускаемые поверх виртуальных потоков, **ФСПВП**, с использованием примитива синхронизации **ReentrantLock** позволяют добиться более высокой производительности нежели фиберы из **Cats Effect 3** с использованием асинхронного мьютекса. При этом потребление памяти становится значительно меньше.

Замер производительности 2.2. Cats Effect 3, Project Loom и ФСПВП с добавлением асинхронного ожидания. В текущем замере производительности различные рантаймы будут решать эту же задачу, но с одним исключением. Исключение состоит в том, что, если раньше после инкрементации легковесный поток продолжал занимать поток операционной системы, то теперь легковесный поток будет отдавать управление потоком, используя асинхронное ожидание в течении одной микросекунды.

То есть, вместо такого цикла:

```
def incrementLoop(id: Int, remain: Int): Rei[Unit] =
  if(remain <= 0) Rei.unit else {
    (for {
      _ <- Rei.delay(lock.lock())
      _ <- Rei.delay(variable += 1)
      _ <- Rei.delay(lock.unlock())
    } yield ()).flatMap(
      _ => incrementLoop(id, remain - 1)
    )
  }
```

Теперь будет использован следующий следующий:

```
def incrementLoop(id: Int, remain: Int): Rei[Unit] =
  if(remain <= 0) Rei.unit else {
    (for {
      _ <- Rei.delay(lock.lock())
```

```

    _ <- Rei.delay(variable += 1)
    _ <- Rei.sleep(1.micro)
    _ <- Rei.delay(lock.unlock())
  } yield ().flatMap(
    _ => incrementLoop(id, remain - 1)
  )
}

```

После инкрементации переменной, но прежде чем будет отпущен мьютекс, легковесный поток отдает управление потоком операционной системы. Это приводит к тому, что легковесные потоки захватывают мьютекс, но не отпускают, отдавая управление другим легковесным потокам, которые с большей вероятностью будут входить в состояние блокировки.

Цель: сравнить производительность **Cats Effect 3**, **Project Loom** и **ФСПВП**, а также интенсивность аллокаций при использовании каждого из них, при условии выполнения асинхронного ожидания в течении одной микросекунды.

Результаты измерений представлены на таблице 3.5.3.

Таблица 3.5.3 - Замеры производительности

Рантайм	Среднее время на итерацию	Интенсивность аллокаций
Cats Effect 3	500 мс	55 мб/ит
Project Loom	300 мс	25 мб/ит
ФСПВП	120 мс	20 мб/ит

Анализ измерений: при добавлении асинхронного ожидания виртуальные потоки значительно замедляются, время выполнения полной итерации программы увеличивается в 150 раз. При этом в Cats Effect время увеличивается лишь в 2.5 раза. Это говорит о дополнительной нагрузке, создаваемой асинхронным ожиданием. Причем, в случае с **Project Loom** нагрузка проявляется намного сильнее.

Однако, в такой ситуации **ФСПВП** опережает две эти реализации по скорости выполнения и аллоцирует меньше памяти, что говорит о том, что он

решает эту задачу оптимальнее других аналогов. В данном примере комбинирование файберов и виртуальных потоков JVM дает эффективность недостижимую использованием лишь одной из двух составляющих.

Замер производительности 2.3. Cats Effect 3, Project Loom и ФСПВП с добавлением асинхронного ожидания. Данный замер производительности аналогичен предыдущему, но отличие будет состоять в том, что вместо асинхронного мьютекса будет использоваться *AtomicLong* - специальная структура из языка Java, позволяющая гарантировать атомарность различных операций над числом. В данном примере понадобится лишь одна операция - инкрементация. Теперь вместо захвата мьютекса перед инкрементацией обычной числовой переменной, будет вызываться метод инкрементации атомарного числа, который обеспечит транзакционность текущей программы и делает код потокобезопасным. Основное отличие от предыдущих замеров состоит в том, что такой код будет работать должен быстрее за счет того, что пропадает лишняя нагрузка, создаваемая асинхронным мьютексом, ведь все операции над *AtomicLong* реализованы через специальную процессорную команду известную как *compareAndSet*.

Цель: сравнить производительность **Cats Effect 3, Project Loom** и **ФСПВП**, а также интенсивность аллокаций при использовании каждого из них, при условии выполнения асинхронного ожидания в течении одной микросекунды, а корректный доступ к переменной гарантируется с помощью структуры **AtomicLong**

Результаты измерений представлены на таблице 3.5.4

Таблица 3.5.4 - Замеры производительности

Рантайм	Среднее время на итерацию	Интенсивность аллокаций
Cats Effect 3	8 мс	20 мб/ит
Project Loom	75 мс	25 мб/ит
ФСПВП	13 мс	25 мб/ит

Анализ измерений: в данной конфигурации наилучшие результаты показывает рантайм **Cats Effect 3**. При этом худшие по производительности

результаты показывает **Project Loom**, что подтверждает гипотезу о неустойчивости этого рантайма к операциям смены контекста. Суммируя несколько последних замеров, можно сделать более общий вывод: файберы в языке Scala более эффективны в смене контекста, но при этом блокирующий мьютекс из **Project Loom** более эффективен. Именно за счет этого комбинирование двух этих рантаймов, используемое в **ФСВП**, дает столь хорошие результаты по производительности.

3.6 Задача осуществления асинхронных сетевых взаимодействий

В следующей задаче будет использоваться асинхронное чтение из сокетов, которое предоставляется в рамках Project Loom. При чтении из блокирующего сокета, если использовать Project Loom, не будет происходить блокировки потока операционной системы, на котором выполняется чтение. Вместо этого заблокируется лишь виртуальный поток. Асинхронным аналогом такого подхода являются классы и механизмы, предоставляемые пакетом **java.nio**.

Замер производительности 3.1. java.nio и Project Loom. В текущем замере будет сравниваться производительность сокет сервера, написанного с помощью **java.nio** с производительностью аналогичного сервера, написанного с использованием блокирующих сокетов на виртуальных потоках Java.

Цель: сравнить производительность и интенсивность аллокаций в случае использования **java.nio** сокетов и в случае с использованием виртуальных потоков Java и сокетов из **java.net**.

Результаты измерений представлены на таблице 3.6.1.

Таблица 3.6.1 - Замеры производительности

Рантайм	Среднее время на итерацию	Интенсивность аллокаций
java.nio	120 мс	12 мб/ит
Loom + java.net	70 мс	12 мб/ит

Анализ измерений: Project Loom в сочетании с блокирующими сокетами из пакета **java.net** способен дать большую производительность, нежели

использование пакета **java.nio** в обстоятельствах высокой нагрузки.

Замер производительности 3.2. Cats Effect 3, Project Loom и ФСПВП с добавлением асинхронного ожидания. Теперь, исходя из гипотезы, что сокеты в Project Loom по скорости работы превосходят сокеты, используемые в **java.nio**, необходимо провести сравнение этих двух подходов, но в случае с использованием фибров. Для этого необходимо описать весь жизненный цикл сервера как некоторую IO задачу. Функция, которая будет являться обработчиком входящих сообщений, также будет иметь возвращаемый тип IO. Для того, чтобы использование фибров в данном замере было оправданным, к вычислению запроса будет добавлена асинхронная операция, представляющая из себя ожидание в течении одной микросекунды.

Результаты измерений представлены на таблице 3.6.2.

Таблица 3.6.2 - Замеры производительности

Рантайм	Среднее время на итерацию	Интенсивность аллокаций
CE3 + java.nio	200 мс	47 мб/ит
ФСПВП	95 мс	22 мб/ит

Анализ измерений: использование фибров из библиотеки Cats Effect 3 дает менее высокую производительность, нежели оптимизированные для Project Loom фибров поверх блокирующих сокетов. При этом также сокращается интенсивность аллокаций. Таким образом, **ФСПВП** показывает себя эффективнее **Cats effect 3 with java.nio**, используя преимущество в скорости обработки запросов блокирующих сокетов поверх виртуальных потоков над асинхронным аналогов.

ЗАКЛЮЧЕНИЕ

В большей части рассмотренных примеров задачи, полагающиеся на асинхронность, намного эффективнее выполнялись на виртуальных потоках из Project Loom. Так происходит по той причине, что виртуальные потоки реализованы на уровне JVM, где допустимо использовать различные оптимизации, недоступные при этом на уровне языка Scala при написании механизма фиберов.

Однако, в ходе работы было установлено, что:

1. Использование блокирующих структур поверх виртуальных потоков на фиберах эффективнее асинхронных структур, реализованных на языке Scala, по потреблению памяти, что позволяет сократить нагрузку на сборщик мусора и освободить процессор от лишней работы.
2. Существуют задачи, которые более эффективно решаются комбинацией фиберов и виртуальных потоков, нежели чем-то из перечисленного по отдельности.
3. Сетевые взаимодействия с использованием виртуальных потоков реализованы эффективнее, нежели их аналоги представленные в пакете асинхронного ввода-вывода. Поэтому использование виртуальных потоков при запуске сервера на фиберах оправдано для оптимизации сетевых взаимодействий через сокеты.
4. Блокирующий мьютекс, использующийся на виртуальных потоках, в задаче обеспечения многопоточного доступа к данным работает значительно эффективнее асинхронного аналога, написанного на языке Scala.

Таким образом, была исследована возможность использования виртуальных потоков для оптимизации фиберов в Scala. Был представлен подход, позволяющий увеличить эффективность выполнения некоторых задач. Полученные результаты представляют большой интерес и могут быть использованы в дальнейшем для оптимизации различных программ, написанных на языке Scala с использованием фиберов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

- [1] ReentrantLock [электронный ресурс] // Oracle. URL: <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html> (дата обращения 26.05.2023)
- [2] Project Loom: Fibers and Continuations for the Java Virtual Machine [электронный ресурс] // OpenJDK. URL: <https://cr.openjdk.org/~rpressler/loom/Loom-Proposal.html> (дата обращения 26.05.2023)
- [3] Socket [электронный ресурс] // Oracle. URL: <https://docs.oracle.com/javase/8/docs/api/java/net/Socket.html> (дата обращения 26.05.2023)
- [4] How will Loom's fibers change the Cats Effect and ZIO parts of the ecosystem [электронный ресурс] // Reddit. URL: https://www.reddit.com/r/scala/comments/sa927v/how_will_looms_fibers_change_the_cats_effect_and/ (дата обращения 26.05.2023)
- [5] Impact of Loom on “functional effects” [электронный ресурс] // ScalaLang. URL: <https://contributors.scala-lang.org/t/impact-of-loom-on-functional-effects/5821> (дата обращения 26.05.2023)
- [6] Fabio Labella — How do Fibers Work? A Peek Under the Hood // SystemFW. URL: https://www.youtube.com/watch?v=x5_MmZVLiSM (дата обращения 26.05.2023)
- [7] Cats Effect 3 (программное обеспечение) [электронный ресурс] // Github. URL: <https://github.com/typelevel/cats-effect> (дата обращения 26.05.2023)
- [8] Rise of Loom [электронный ресурс] // Ziverge. URL: <https://ziverge.com/blog/zio-london-october/> (дата обращения 26.05.2023)

- [9] JVM Garbage Collection [электронный ресурс] // Baeldung. URL: <https://www.baeldung.com/jvm-garbage-collectors> (дата обращения 26.05.2023)
- [10] Asynchronous [электронный ресурс] // Techtarget. URL: <https://www.techtarget.com/searchnetworking/definition/asynchronous> (дата обращения 26.05.2023)