Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем

Системное программирование

Булгаков Андрей Вадимович

Верификация многопоточных алгоритмов методом управляемого исполнения легковесных потоков

Выпускная квалификационная работа

Научный руководитель:
д. ф.-м. н., профессор Терехов А. Н.

Научный консультант:
руководитель направления dxLab Цителов Д. И.

Рецензент:
старший преподаватель СПбГПУ Ахин М. Х.

Санкт-Петербург
2017

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems

Software Engineering

Bulgakov Andrey

Multithreaded algorithms verification by controllable execution of lightweight threads

Graduation Thesis

Scientific supervisor:
Professor Andrey Terehov

Scientific consultant:
Head of dxLab Dmitriy Citelov

Reviewer:
Assistant professor SPbSTU Marat Akhin

Saint-Petersburg
2017

Оглавление

1. Введение

С ростом количества многопоточных алгоритмов растёт потребность в их тестировании и доказательстве корректности работы. Одним из базовых критериев корректности многопоточных алгоритмов является линеаризуемость [2] — свойство программы, при котором результат любого параллельного выполнения операций эквивалентен некоторому последовательному. Для проверки линеаризуемости существует несколько подходов [4]. Одним из них является поиск последовательности операций, при многопоточном исполнении которой результат выполнения не совпадает ни с одним результатом из множества последовательных выполнений. Если такая последовательность существует, то алгоритм нелинеаризуем. В противном случае, утверждать обратного нельзя.

Для поиска такого множества операций существует инструмент lin-check [11]. Он генерирует множества результатов последовательных исполнений. Затем составляет наборы операций и запускает их на нескольких потоках. Процесс тестирования является псевдослучайным, так как между операциями вставляются различные задержки. Это вызвано отсутствием необходимого для тестирования механизма синхронизации в проверяемых структурах. Данный фактор заставляет осуществлять многократный запуск одних и тех же тестовых данных и надеяться, что переключение потоков произойдёт в отличных от предыдущего запуска точках. Этого можно избежать в случае контролируемого переключения между потоками изнутри структуры.

Ход работы инструмента lin-check состоит из множества независимых друг от друга итераций. Данное утверждение позволяет выдвинуть гипотезу об увеличении скорости проверки структуры, если запускать каждую итерацию в своём потоке. Но такое улучшение вызовет значительное падение производительности при большом значении параметра запущенных одновременно исполнений. Это связано с особенностями модели работы стандартных JVM потоков.

На текущий момент виртуальная машина Java стандартно использует native модель для расспараллеливания потоков. Native-потоки — это

потоки, за управление которыми отвечает операционная система. Создание большого их количества нагружает систему. Кроме того, стандартные средства Java не позволяют эффективно управлять их переключением [7, 8]. В целях решения этой проблемы были созданы легковесные потоки (Green Threads). Они эмулируют многопоточное исполнение, являясь по сути либо корутинами либо континуациями [9]. Континуации — это функции, чьё выполение может быть приостановлено и возобновлено. Корутины — механизм управления ходом выполнения программы, позволяющий приостанавливать выполняемую задачу, сохранять её состояние и выключать из потока. Проблема управления потоками заключается в том, что тестируемые структуры не обладают нужными точками переключений потоков. Устанавливать такие точки вручную внутри структур является сложной задачей, так как требует глубокого понимания логики работы многопоточного алгоритма.

2. Постановка задачи

Целью данной работы является разработка и реализация системы управления порядком выполнения потоков в инструменте lin-check.

Для её достижения были поставлены следующие задачи.

- Проанализировать существующие решения в области легковесных потоков.
- Разработать требования к реализуемой системе.
- Разработать новую архитектуру для внедрения точек синхронизации и управления точками из легковесных потоков.
- Выполнить реализацию системы.
- Провести апробацию.

3. Обзор библиотек легковесных потоков

Данная глава посвящена обзору библиотек на языке Java, реализующих легковесные потоки.

3.1. Kilim

Библиотека легковесных потоков Kilim [7, 8] работает на основе модели акторов. Актор — объект позволяющий принимать и отправлять сообщения взаимодействия при помощи "почтовых ящиков". Почтовые ящики не предоставляют процессам доступа к общей памяти, тем самым изолируя их друг от друга. Акторы не подразумевают синхронного выполнения и, таким образом, предотвращают возможность взаимных блокировок.

Принцип работы Kilim строится на основе CPS преобразовании байт-кода. Все методы, которые могут быть остановлены, помечаются специальной сигнатурой. Скомпилированный код передается в процесс трансформации классов "weaver". Он распознаёт методы вызывающие блокировки по сигнатуре и добавляет внутрь него 3 области с байт-кодом: prelude, pre-call, post-call. Преобразованный метод также принимает один дополнительный аргумент типа Fiber. Этот объект нужен для накопления информации о стеке метода. Таким образом, weaver "оборачивает" вызовы блокировок в два метода Fiber.down и Fiber.up, которые управляют сохранением, раскруткой стека и возвратом состояния. Все измененные методы обрабатываются планировщиком, который управляет пулом потоков и переключением контекста.

Важной особенностью библиотеки Kilim являются объекты, хранящие состояния стека методов. Каждый метод имеет свой набор данных: примитивы, ссылочные типы. Хранение их в универсальной структуре дорого с точки зрения преобразования типов и поиска. Поэтому Kilim генерирует для каждого "Pausable" метода свой тип State, который содержит весь набор переменных стека. Это позволяет быстро восстанавливать состояние, что значительно увеличивает скорость работы библиотеки.

Основным преимуществом библиотеки является её малый размер и лёгкость создания акторов. Достаточно наследоваться от класса Task и помечать блокирующиеся методы соответствующей сигнатурой.

Недостатком является отсутствие процесса трансформации классов во время выполнения программы, механизмов прямого блокирования потоков, и поддержки данной библиотеки.

3.2. Quasar

Библиотека Quasar [6] предоставляет высокопроизводительные легковесные потоки, механизм каналов как в языке Go, и возможность использования акторов. Она основана на более неподдерживаемой библиотеке Матиасса Мана [5]. Принцип работы построен на использовании корутин и планировщика. Коруритины занимаются сохранением и возврата стека, а планировщик управляет запуском, включением и выключением корутин.

В Quasar корутины представлены классом Fiber, являющимся оболочкой для Runnable. Он реализует в полной мере функциональность стандартных потоков с дополнительной возможностью блокировок. Это позволяет напрямую управлять переключением контекста на нужный поток. Как и в Kilim, все приостанавливаемые методы необходимо помечать соответствующей сигнатурой. В данном случае библиотека предоставляет два варианта маркировки методов: через механизм исключений ”throws SuspendExecution” и через аннотацию ”@Suspendable”. Первый вариант не является удобным так как данное исключение пробрасывается для управления состояниями корутины. Это вынуждает программиста заботиться о контроле исключения в вызовах. Кроме того, в случае реализации сторонних интерфейсов, не всегда есть возможность доопределить сигнатуру метода, что несёт дополнительные трудности и неизбежные ошибки реализации.

Байт-код помеченных методов преобразуется, но в отличие от Kilim это происходит в ходе выполнения программы, через подключенного Java агента или с помощью загрузчика классов. Процесс преобразова-

ния происходит в несколько фаз и требует много времени, что является недостатком библиотеки, если необходимо подгружать много классов непрерывно.

Преимуществом библиотеки является возможность управления одновременно корутинами и стандартными потоками при помощи класса Strand, который делегирует вызов команд на команды корутин, если выполнение происходит в них или на команды настоящих потоков, когда программа не находится в активном Fiber. Также библиотека предоставляет механизм сериализации корутин. Это позволяет возвращать поток в нужное состояние.

3.3. Jephyr

Проект Jephyr [3] — это библиотека, предоставляющая реализацию легковесных потоков и плагин для посткопиляции байт-кода. Инструмент предназначен для обработки кода и замены одной реализации потоков на другую. Он позволяет осуществлять быстрый перенос программы на легковесные потоки, не изменяя исходного кода, чем не обладает ни один из рассмотренных выше аналогов. При обычном использовании код пишется на стандартных потоках с использованием метода блокировки LockSupport.park и разблокировки LockSupport.unpark. Подключенный плагин преобразует компилируемый код, внедряя управление стеком метода и механизм планирования исполнения и заменяя все вхождения Thread на готовую легковесную реализацию потоков.

Недостатком библиотеки является отсутствие встроенных инструментов динамического преобразования байт-кода и поддержки данной библиотеки.

4. Требования к системе

Перед начало разработки были сформулированы критерии, которым должна соответствовать разрабатываемая программа. Они позволили выделить основные фазы разработки. Был сформулирован следующий набор необходимых элементов системы.

- Функциональность внедрения точек синхронизации в структуры - необходима для предоставления доступа переключения между потоками внутри структуры. Должна включать в себя модули загрузки классов и инструментирования байт-кода.

- Функциональность запуска на легковесных потоках - необходима для создания и управления пулом легковесных потоков. Должна включать в себя пул потоков, планировщик и исполнитель поступивших заданий.

- Функциональность параллельного запуска итераций - необходима для ускорения поиска нелинеаризумых исполнений за счёт выполнения итераций одновременно на всех доступных ядрах системы.

Для доступа к новым функциям необходимо разработать ряд пользовательских API. В него входят:

- API для управления потоками изнутри,

- API пользователя для доступа к новым функциям.

Кроме того, реализованные элементы должны обладать гибкостью и устойчивостью к дальнейшим улучшениям. Вся разработка должна вестись на языке программирования Java, и включать документирование методов и дополнительные комментарии к коду на английском языке.

5. Архитектура системы

Разработанная система имеет следующую структуру, представленную на диаграмме компонент 1.

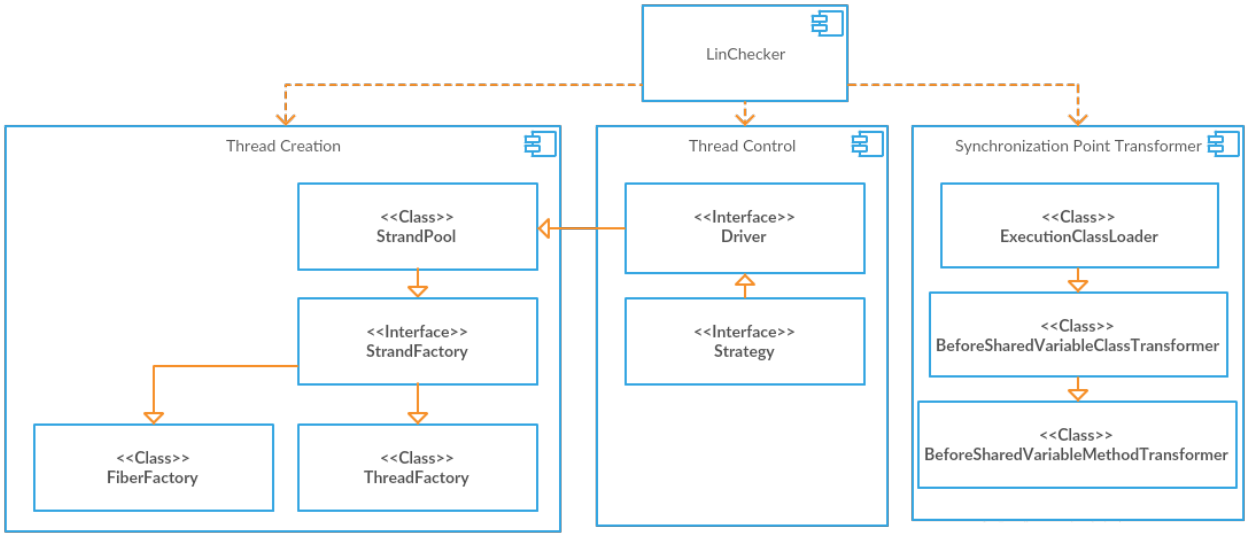

Рис. 1: Диаграмма компонент

Точкой входа начала проверки структуры является метод check класса LinChecker. Прежде чем начать тестирование, необходимо выбрать один из сценариев использования с помощью конфигурирования параметров аннотаций в тестовом классе. В дополнении к обычному стресс-тестированию, с внедрением системы управления потоками, появляется следующий набор методов проверки структуры:

1. Стресс-тестирование;
2. Тестирования на стандартных потоках с помощью стратегии;
3. Тестирования на легковесных потоках с помощью стратегии;
4. Параллельное выполнение итераций тестирования на стандартных потоках с помощью стратегии;
5. Параллельное выполнение итераций тестирования на легковесных потоках с помощью стратегии.

Во всех сценариях, кроме стресс-тестирования, тестовый класс передаётся в загрузчик классов. Он является загрузчиком верхнего уровня и осуществляет инструментирование, объявление в JVM, кэширование байт-кода и объявленных классов. Поступив в загрузчик класс отправляется в преобразователь байт-кода, который изменяет все разрешенные для этого методы, вставляя внутрь точки синхронизации. На этом заканчивается этап инструментирования байт-кода, результатом которого является объявление в JVM инстрементированного класса.

Для создания, запуска и управления потоками используется пул потоков. На основе сгенерированных исполнений [11] он создаёт необходимое количество задач с помощью фабрики. Настройка фабрики происходит путем конфигурирования её по типу используемых потоков: "Thread" или "Fiber". Кроме того, это оптимизирует скорость работы, так как нет необходимости выбора, поток какого вида создать. После выбора типа пула изменить его нельзя и данный факт предоставляет безопасность того, что не будет создано потоков разных типов. Независимость от типа позволяет пулу получать статус ошибок и состояния как легковесных так и стандартных потоков, делегируя вызов нужной реализации.

Управление переключением и остановкой потоков осуществляется внутри исполнений, путем вызова точек синхронизаций. Вызов точки происходит с помощью доступа к выбранной стратегии, содержащейся в статическом хранилище. Стратегия — планировщик, который контролирует ход исполнения и принимает решение к продолжению или переключению потока посредством содержащегося в ней драйвера. Интерфейс драйвера имеет набор специфицированных методов, которые изолируют объекты потоков в пуле посредством идентификаторов и делегируют команды прерываний.

6. Реализация

Внедрение точек синхронизации подразумевает наличие вызовов внутри структуры, которые обращаются к стратегии-планировщику. Это возможно двумя способами:

- Ручная вставка вызовов методов планировщика или намеренного переключения потоков в исходный код;
- Автоматическая вставка вызовов методов на всех операциях чтения-записи общих переменных.

Точки синхронизации представляют из себя инструкции вызова методов стратегии, вставленные непосредственно в байт-коде перед определенными командами и локацию точки. Локация точки — уникальный идентификатор, состоящий из имени класса, имени метода, описания метода и порядкового номера инструкции байт-кода в методе. Все локации помещаются в единое хранилище точек, которое реализовано по принципу шаблона "Одиночка" и является потокобезопасным.

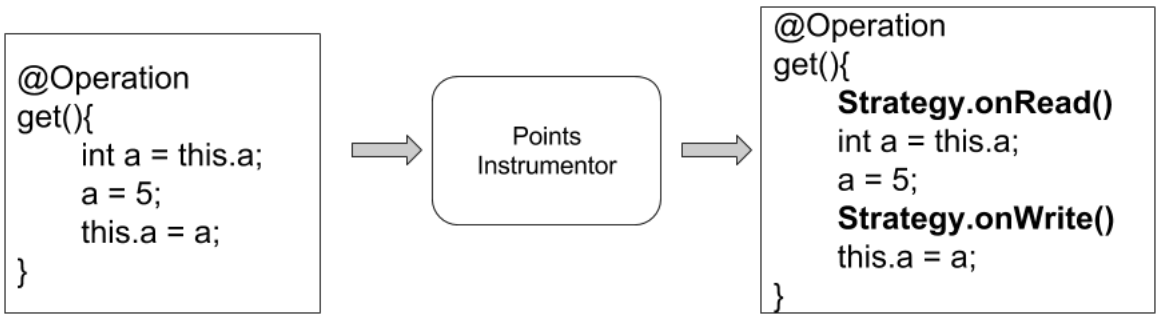

Рис. 2: Вставка точки синхронихации в исходном коде

Ручная вставка синхронизаций требует наличия исходного кода и понимания критических точек в тестируемом алгоритме. Наличие первого пункта может быть невозможным, а второго будет требовать досконального понимания алгоритма, что неудобно, если необходимо быстро проверить алгоритм на корректность. Поэтому в контексте данной работы рассматривает автоматическая вставка точек синхронизации путём трансформации байт-кода. Обычно в таких случаях используют один или несколько следующих методов инструментирование кода.

- Посткопиляционный процесс преобразования байт-кода. Это AOT процесс, который работает после завершения компиляции программы, но до её запуска. Он обрабатывает набор ".class" файлов и является хорошим выбором, если в программе не будет генерироваться новых классов.

- Агент подключаемый к JVM. Это сопрограмма, запускаемая перед основной программой и предоставляющая доступ к механизму управления байт-кодом. Позволяет обрабатывать любые классы, включая стандартный пакет "java.". Минусом является сложность поддержки агента и необходимость явного подключения при использовании библиотеки.

- Загрузчик классов. Является обычным объектом JVM и требует переопределения определённых методов. Кроме того, все требуемые классы в цепочке загрузки будут также проходить через него, что даёт возможность целенаправленно менять байт-код только необходимых классов. Трудностью является невозможность менять стандартные пакеты языка Java без переименования классов.

6.1. Преобразование байт-кода

Загрузчик классов

Из перечисленных выше методов используется способ инструментирования через загрузчик классов. Точкой входа в процесс преобразования байт-кода является метод loadClass. Переопределение данного метода нарушает стандартный способ загрузки классов, так как обычно принято, при проектировании загрузчиков, работать только с методом findClass. Данное решение связано с необходимостью запрета делегирования загрузки нижележащим загрузчикам. Если руководствоваться стандартным способом, то загрузчик сначала будет пытаться искать класс в родительских загрузчиках, а уже потом попробует определить его текущим. В этом случае можно потерять часть точек синхронизации, получить ошибки в ходе выполнения и итоговое преобразование

не обеспечит должного покрытия переключений.

Игнорирование пакетов

После поступления имени класса в ExecutionClassLoader происходит проверка на необходимость инструментирования. Стандартные правила Java запрещают преобразование классов из пакетов ”java.” и ”sun.”, поэтому данные пакеты игнорируются. Кроме того, необходимо запретить трансформацию классов из пакета “lincheck.”, так как нет необходимости вставлять точки синхронизации в собственный исходный код, который никак не относится к тестируемой структуре и будет лишь замедлять работу библиотеки.

Хранение байт-кода

Следующим этапом алгоритма является поиск байт-кода в статическом кэше. В связи с эквивалентностью преобразований нет смысла проводить их каждый раз в новом загрузчике. Поэтому происходит сохранение классов в общем кэше. Такое решение даёт определённый выигрыш в скорости, но не позволяет использовать разные типы потоков при работе программы. Весь кэш хранится в потокобезопасной структуре данных для возможности использования загрузчиков в параллельных потоках и экономии количества проводимых действий по инструментированию.

Вставка точек синхронизации

Для преобразования байт-кода используется библиотека ASM [1]. Данный выбор обуславливается наличием зависимости к этой библиотеке и простотой использования. Сначала считывается байт-код класса и отправляется в объект-обработчик классов BeforeSharedVariableClassVisitor. Он запоминает имя класса (для дальнейшей генерации локаций) и делегирует обработку методов объекту-обработчику методов. Важной особенностью является пропуск следующих типов сигнатур методов:

- Методы имеющие модификатор доступа ”native”;

- Методы конструкторы;
- Методы инициализаторов класса.

Методы с сигнатурой ”native” выходят за рамки обработки байт-кода, поэтому пропускаются. Игнорирование инициализаторов и конструкторов связано одновременно с запретом на приостановку таких методов в библиотеке легковесных потоков Quasar и возможными ошибками. Например: прерывание инициализаторов классов может привести к взаимной блокировке, если другой поток попытается получить доступ к приостановленному недоинициализированному классу. Таким образом, в связи с необходимостью внесения правок в библиотеку легковесных потоков, на текущий момент, инструментирование таких методов не осуществляется, и добавляется ряд ограничений к тестируемым структурам.

Преобразование метода начинается в момент появления одной из формализованных инструкций доступа к общей памяти [10]. Такими операциями являются:

- xALOAD (где ”x” - тип переменной);
- xASTORE (где ”x” - тип переменной);
- GETSTATIC;
- PUTSTATIC;
- GETFIELD;
- PUTFIELD.

Как только в байт-коде встречаются такая операция, происходит вставка команды вызова объекта стратегии из хранилища стратегий. Затем происходит генерация локации точки на основе собранной информации о классе и сигнатуре метода, путем запроса к хранилищу локаций, и вызывается один из методов стратегии, в зависимости от типа операции: чтение, запись.

Листинг 1 содержит пример байт-кода выполняющий чтение из поля ”a”. Таким образом, демонстрируется вставка точки синхронизации перед инструкцией GETFIELD.

Listing 1: Вставка точки синхронизации в байт-коде

```
INVOKESTATIC StrategyHolder.getStrategy ()LStrategy;
BIPUSH location
INVOKEINTERFACE Strategy.onSharedVariableRead (I)V
GETFIELD somepackage/LockFreeDeque.a : LSomeObject;
```

Маркировка прерываемых методов

Вторым шагом преобразования является маркировка прерываемых методов. Библиотека Quasar нуждается либо в добавлении исключения к сигнатруе прерываемого метода, либо пометки в виде аннотации ”@Suspendable” к методу. Кроме того, должен быть помечен весь стек вызовов прерываемого метода. Маркировать только методы, в которых есть вызов точек синхронизации, неправильно, так как есть возможность, что вызываемые им методы будут прерываемыми и тогда загрузчик библиотеки Quasar некорректно осуществит инструментирование кода. Поэтому все методы маркируются с помощью аннотации ”@Suspendable”. Такая разметка вызывает падение производительности при инструментировании, но не в ходе исполнения, так как библиотека потоков поддерживает оптимизацию, проверяющую возможный стек метода и, в зависимости от результатов, вставляет методы сохранения и возврата стека. Таким образом, все по настоящему непрерываемые методы не будут преобразованы.

Решение добавлять контролируемые исключения к сигнатурам методов вместо аннотаций заставило бы заботиться об их правильном пробрасывании. Дополнительной проблемой были бы ”try-catch” блоки внутри тестируемого алгоритма, которые могут перехватить данное исключение, что вызовет трудноопределяемые ошибки.

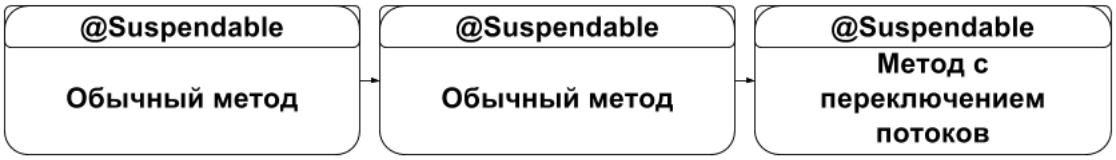

Рис. 3: Маркировка прерываемых методов

Завершение преобразования

После завершения процесса инструментирования, байт-код загружается в JVM и в случае успеха кэшируется вместе с полученным классом.

Загрузчик без преобразований

Важной особенностью реализации является наличие загрузчика классов, который никак не изменяет классы. Он необходим при генерации возможных результатов, так как линеаризуемые исполнения не должны содержать точек переключений и обрабатываться загрузчиком библиотеки Quasar. Использование такого загрузчика значительно увеличивает скорость работы за счёт экономии времени преобразования байт-кода.

6.2. Создание и управление потоками

Создание потоков

Одной из подзадач добавления функциональности легковесных потоков является сохранение возможности запуска теста на стандартных потоках. Поэтому используется единый пул, зависящий лишь от выбранного типа. Он управляет объектами класса Strand. Strand — единая абстракция для легковесных и стандартных потоков, делегирующая команды в зависимости от выбранной реализации. Тип пула конфигурируется при создании и больше не изменяется, для того чтобы избежать возможности исполнения потоков разных типов.

Внутри пула для создания нужного потока используется абстрактная фабрика. В момент создания пула инициализируется фабрика нуж-

ного типа. Отличие в наследниках фабрики состоит в том, что легковесные потоки уже являются реализацией интерфейса Future, а для стандартных потоков приходится оборачивать исполнение в объект FutureTask. Кроме того, легковесные потоки нуждаются в явном задании своего внутреннего планировщика, чтобы предотвратить переключение по таймауту.

Точкой входа в пул является метод invokeAll. Он принимает коллекцию сгенерированных исполнений, затем для каждого исполнения создаёт свой поток и кладёт его в ядро пула. После того, как все исполнения обработаны, все объекты в ядре запускаются и на выход передаётся коллекция вычисляющихся результатов.

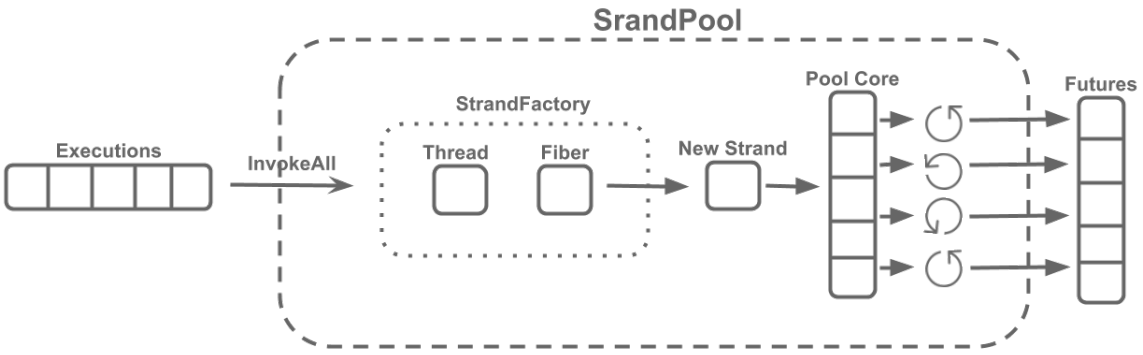

Рис. 4: Организация пула

Управление потоками

Для того чтобы инкапсулировать функциональность стратегий от управления потоками, разработан интерфейс драйвера потоков. Его основными задачами являются:

- Блокировка потока;
- Ожидание потока;

- Переключение потока;

Переключение осуществляется в два шага: сначала разблокируется целевой поток, а затем блокируется текущий. Это действие является критическим, так как в определённый момент будет разблокировано два потока. В случае работы стандартных потоков возможна смена контекста, до момента блокировки текущего потока, что может привести к взаимной блокировке. Кроме того, согласно спецификации Java, возможен случайный выход потоков из блокировки. Для предотвращение возобновления используется live-lock по параметру текущего потока. Таким образом достигается атомарность переключения.

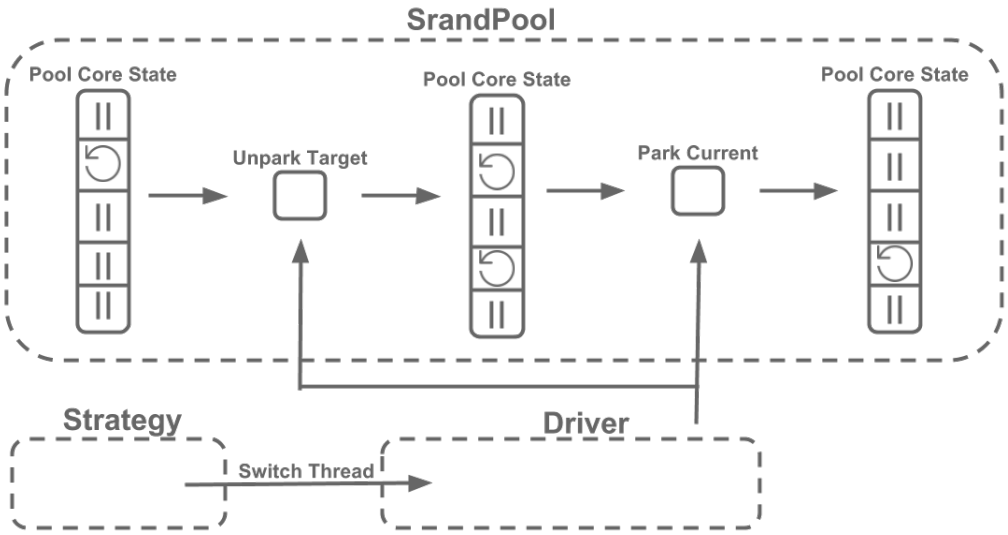

Рис. 5: Переключение потока

Важной особенностью разработки стратегии является необходимость смены значения параметра, отвечающего за номер обрабатываемого ей потока, перед тем как вызвать команду драйвера.

6.3. Параллельный запуск легковесных потоков

Добавление функциональности легковесных потоков избавило операционную систему от необходимости переключения их контекста и освободило системные ресур. Появилась возможность запускать каждую итерацию в своём потоке, а создание итерации откладывать с помощью ленивой очереди.

Для того чтобы не перегружать систему большим количеством потоков, число одновременно запущенных итераций считается равным числу свободных ядер. Также необходим контроль за ходом каждой итерации. Если нелинеаризуемые результаты будут найдены, работа всех итераций должна быть прекращена и программа завершится с ошибкой. Для этого используется общий слушатель, который завершает пул итераций и разблокирует главный поток.

6.4. Ограничения

Текущая реализация обладает некоторыми ограничениями, связанными с использованием библиотеки Quasar.

1. Нет возможности проверять алгоритмы и структуры содержащие вызовы методов внутри конструкторов классов.

2. Нельзя тестировать алгоритмы и структуры принадлежащие пакету ”java.”.

3. Нельзя проверять алгоритмы и структуры унаследованные от любого из классов пакета ”java.” .

Существуют ряд подходов, позволяющий избавиться от этих ограничений. Они описаны в дальнейших исследованиях.

7. Оценка производительности

После окончания основной части работы необходимо было провести сравнение следующих сценариев работы:

- Запуск на легковесных потоках;
- Запуск на стандартных потоках;
- Запуск на легковесных потоках и параллельными итерациями;
- Запуск на стандартных потоках и параллельными итерациями.

Для тестирования использовался ПК обладающий соответствующими характеристиками:

- Процессор Intel Core i5, 4 ядра;
- 8 GB RAM ;
- Java Version 1.8.0_77 ;
- Операционная система Linux 4.10.13-1-ARCH x86_64 .

Тестирование проводилось на корректной структуре данных, с использованием планировщика EnumerationStrategy. Без использования стратегии легковесные потоки будут исполняться последовательно, без переключений. Каждый тест запускался трижды, после чего бралось среднее значение. Важной особенностью является отсутствие предварительного прогрева JIT компилятора для имитации реального использования библиотеки.

Было выделено два варианта сравнений, образующих плоскость роста времени от параметров:

- Зависимость времени работы программы от количества итераций (в глубину);
- Зависимость времени работы от количества исполнений в каждой итерации (в ширину).

Как видно из графика 6 время работы линейно от числа итераций. Параллельное выполнение итераций всегда быстрей однопоточного. Работа на легковесных потоках выполняется быстрей чем на стандартных.

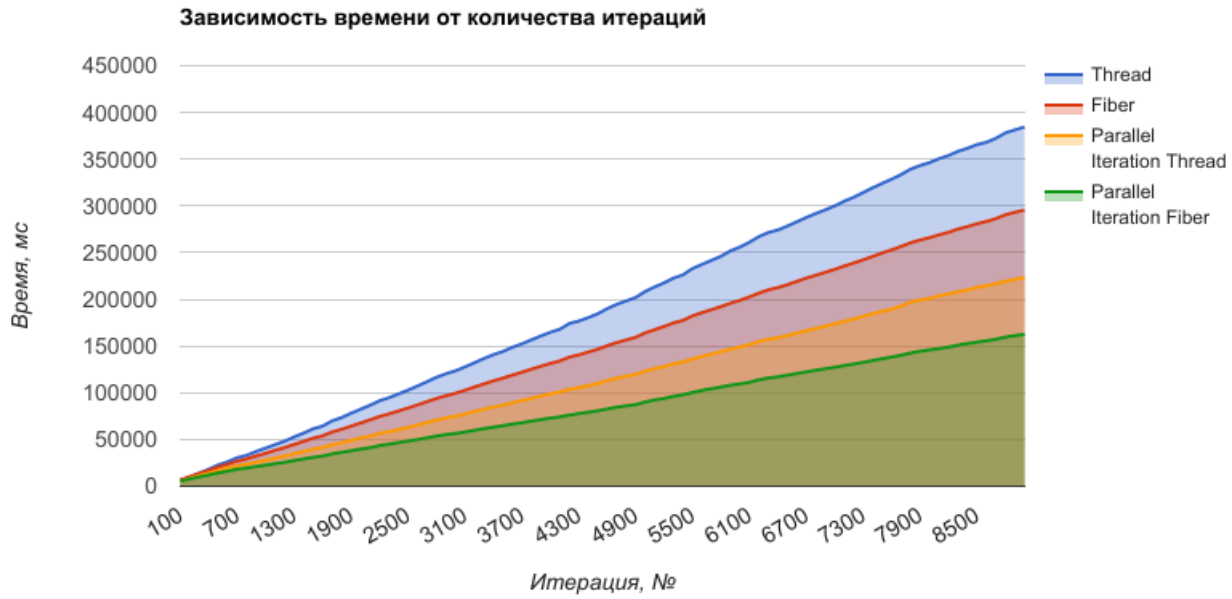

Рис. 6: График зависимости времени работы от количества итераций

Из графика 7 видна похожая на экспоненциальную скорость роста времени от числа одновременных исполнений в каждой итерации. Время работы для стандартных потоков на 5 исполнениях и 4 одновременных итерациях отличается на пять минут от исполнения для легковесных потоков.

Замеры времени для каждой сотни из 9000 итераций показывают, что скорость работы легковесных потоков в среднем на 31% быстрей стандартных. Разница в уровне столбцов говорит о том, что для одной сотни итераций генерировались более ”тяжёлые” исполнения, чем для другой.

Таким образом, можно сделать вывод, что внедрение легковесных потоков в инструмент верификации многопоточных алгоритмов lin-check значительно уменьшило время тестирования, по сравнению с использованием стандартной модели потоков.

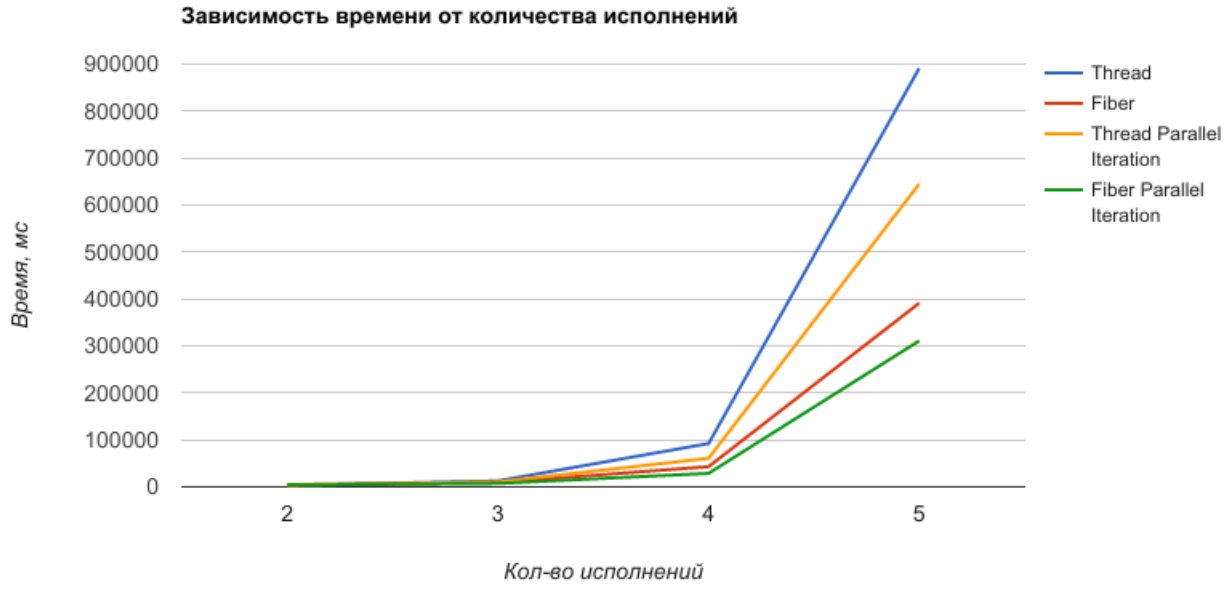

Рис. 7: График зависимости времени работы от количества исполнений

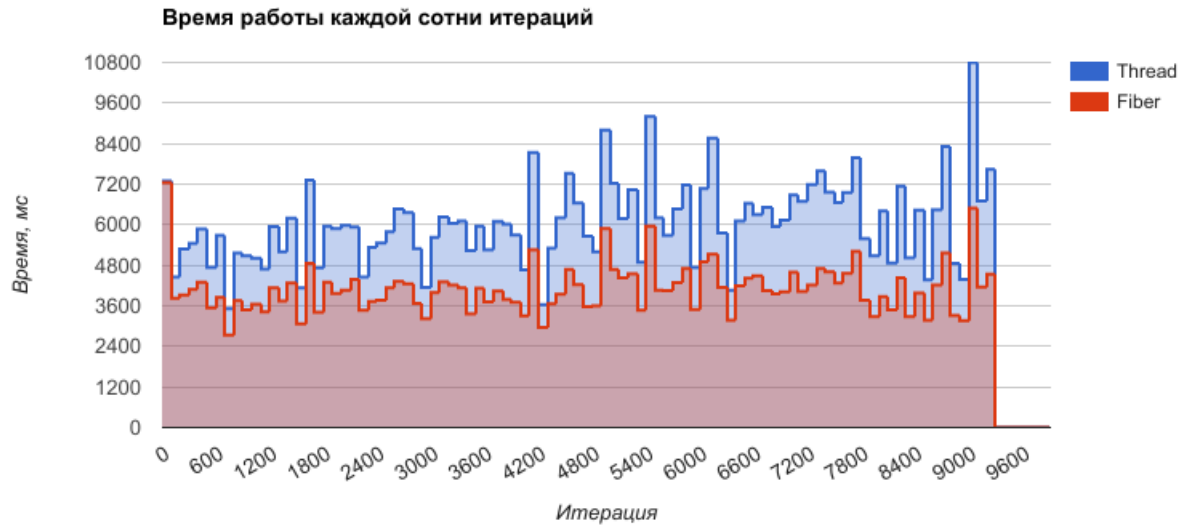

Рис. 8: График времени работы каждой сотни итераций

8. Заключение

В рамках данной работы были достигнуты следующие результаты.

- Проведён обзор библиотек Quasar и Kilim легковесных потоков для языка Java.
- Разработаны функциональные требования выделяющие фазы работы.
- Спроектирована новая архитектура инструмента lin-check обладающая гибкостью для дальнейших усовершенствований.
- Выполнена реализация:
  - Средства внедрения точек синхронизации путем трансформации кода на лету;
  - Функциональности тестирования алгоритмов на легковесных потоках;
  - Системы управления легковесными потоками.
- Проведена апробация для существующих структур данных и проведены замеры скорости работ.

Список литературы

[1] Bruneton Eric. ASM 4.0. A Java bytecode engineering library. — 2011. — URL: http://download.forge.objectweb.org/asm/asm4-guide.pdf.

[2] Herlihy M., Shavit N. The Art of Multiprocessor Programming, 1st ed. — Morgan Kaufmann, 2008.

[3] Jephyr. — URL: https://github.com/yngui/jephyr.

[4] Lowe Gavin. Testing for linearizability // Concurrency and Computation: Practice and Experience. — 2017. — Vol. 29.

[5] Mann Matthias. Continuations library. — URL: http://www.matthiasmann.de/content/view/24/26/.

[6] Quasar, Parallel Universe. — URL: docs.paralleluniverse.co/quasar/.

[7] Srinivasan Sriram. A Thread of One's Own.

[8] Srinivasan Sriram, Mycroft Alan. Kilim: Isolation-Typed Actors for Java // ECOOP. — 2008.

[9] de Moura Ana Lúcia, Ierusalimschy Roberto. Revisiting coroutines // ACM Trans. Program. Lang. Syst. — 2009. — Vol. 31. — P. 6:1–6:31.

[10] А.С. Озерцов. Оптимизация покрытия графа управления при тестировании многопоточных алгоритмов путём внедрения управляемых точек синхронизации. — 2017.

[11] Евдокимов А.А. Цителов Д.И. Елизаров Р.А. Трифанов В.Ю. Автоматическое тестирование линеаризуемости реализаций многопоточных структур данных. — 2015. — URL: http://tmpaconf.org/images/pdf/2015//evdokimov-tsytelov-elizarov-trifanov-automat-testing.pdf.

9. Приложение

В данном приложении представлены таблицы оценки производительности.

Кол-во потоков	Thread	Fiber	Thread Parallel Iteration	Fiber Parallel Iteration
2	3219	4983	3459	3863
3	12811	9380	10805	7340
4	92218	43030	60854	28742
5	891240	391251	645041	310959

Таблица 1: Таблица зависимости времени работы от числа потоков

Итерация	Thread	Fiber	Parallel Iteration Thread	Parallel Iteration Fiber	Thread 100 Iteration Time	Fiber 100 Iteration Time
100	6363	6943	5668	5532	7294	7245
200	9922	10225	8347	7773	4449	3825
300	14091	13625	11672	10006	5285	3918
400	18210	17007	14050	12019	5449	4083
500	22856	20650	16949	14281	5877	4295
600	26291	23552	19484	16108	4739	3544
700	30553	26860	21849	18223	5687	3854
800	33143	29306	23477	19491	3522	2736
900	37212	32407	25770	21019	5161	3758
1000	40851	35295	27984	22553	5076	3486
1100	44624	38485	30154	24196	5006	3652
1200	48170	41361	32231	25603	4691	3417
1300	52799	44769	34979	27486	5946	4133
1400	56824	47972	37037	29180	5200	3745
1500	61470	51454	39703	30886	6196	4273
1600	64378	53984	41439	32294	4136	3067
1700	69952	58045	44484	34620	7317	4857
1800	73460	60937	46568	36260	4722	3414
1900	78086	64329	49161	37913	5956	4297
2000	82404	67643	51619	39705	5884	3968
2100	86641	71048	53566	41206	5980	4067
2200	91467	74617	56504	43480	5936	4382
2300	94751	77433	58430	45004	4452	3468
2400	98767	80551	60629	46641	5328	3736
2500	102789	83688	63131	48275	5464	3765
2600	107229	87051	65479	49965	5791	4133
2700	111858	90513	68231	51707	6466	4322
2800	116563	93889	70848	53722	6363	4254
2900	120367	96918	73216	55214	5297	3675
3000	123463	99462	74998	56410	4143	3221
3100	127635	102665	77326	57986	5621	3991
3200	132093	106200	79905	59843	6224	4314
3300	136691	109429	82485	61557	6043	4221
3400	140754	112767	84753	63128	6110	4139
3500	144132	115654	86643	64735	5238	3369
3600	148709	118946	89351	66448	5960	4119
3700	152344	121991	91651	67954	5263	3721
3800	156628	125261	94114	69636	6098	4036
3900	160806	128361	96470	71282	6019	3788
4000	164719	131385	98835	72839	5693	3705
4100	168006	134065	100795	74171	4662	3307
4200	174107	138117	103827	76067	8136	5256
4300	176762	140546	105486	77663	3632	2966
4400	180438	143560	107702	79169	5304	3663
4500	184543	146622	110062	80724	6211	3939
4600	189969	150307	112981	82654	7524	4681
4700	194627	153750	115631	84592	6647	4230
4800	198273	156520	117765	85972	5658	3571
4900	201975	159372	119761	87301	5201	3606
5000	208453	164065	123185	90140	8798	5888
5100	213247	167548	125888	92223	7230	4677
5200	217650	170965	128219	93845	6177	4431
5300	222740	174818	131070	96232	7039	4550
5400	226247	177560	133001	97931	4891	3471
5500	233050	182365	136268	100286	9203	5961
5600	237431	185809	139064	102596	6207	4060
5700	241764	189001	141393	104349	5688	4053
5800	246189	192269	143986	106136	6463	4288
5900	251942	195884	146835	108080	7175	4699
6000	255998	198917	148924	109430	4736	3493
6100	261156	202666	151517	111256	7066	4904
6200	267249	206672	154993	113913	8564	5132
6300	271551	210228	157163	115804	5758	4144
6400	274233	212597	159044	116998	4066	3169
6500	278454	215709	161279	118828	6114	4196
6600	283008	219249	163943	120595	6627	4412
6700	287678	222763	166399	122361	6299	4486
6800	291914	225824	169013	124114	6511	4055
6900	295809	228953	171367	125711	5951	3957
7000	300076	232189	173839	127320	6129	4018
7100	305117	235659	176513	129057	6883	4595
7200	309239	238910	178921	130754	6694	4023
7300	314431	242334	181779	132564	7184	4215
7400	319433	245996	184694	134503	7590	4698
7500	324200	249668	187227	136429	6968	4607
7600	328686	253059	189656	138245	6652	4280
7700	333694	256657	192874	140231	6955	4565
7800	339440	260615	197220	142605	7980	5211
7900	343275	263561	199243	144392	5590	3762
8000	346424	266018	201041	145947	5082	3284
8100	350632	269177	203923	147347	6413	3878
8200	354028	271760	205909	148861	4863	3484
8300	358484	275317	208251	151005	7140	4427
8400	361809	277896	210046	152697	5018	3278
8500	365682	281005	212885	154203	6426	3983
8600	368190	283411	214608	155709	4368	3170
8700	372503	286488	216742	157338	6441	4216
8800	378225	290621	219400	159584	8314	5160
8900	381428	293235	221414	161122	4858	3321
9000	384543	295598	223362	162546	4379	3157

Таблица 2: Таблица зависимости времени работы от числа итераций