Автоматическое тестирование линеаризуемости реализаций многопоточных структур данных

Евдокимов А.А.
Университет ИТМО
evdokimov@rain.ifmo.ru

Цителов Д.И.
ООО "Эксперт-Система"
tsitelov@acm.org

Елизаров Р.А.
ООО "Эксперт-Система"
elizarov@devexperts.com

Трифанов В.Ю.
ООО "Эксперт-Система"
trifanov@devexperts.com

***Аннотация*—В разработке алгоритма помимо теоретической корректности важна правильная реализация. Для однопоточного случая задача проверки реализации решается, например, модульным и нагрузочным тестированием. Однако, в многопоточных системах применение этих средств затруднено, поскольку выполнение программы зависит от чередования операций в потоках. Существует потребность в инструментах, которые проверяют корректность реализации многопоточного алгоритма. Аналогом понятия «корректности» для многопоточных программ служит понятие линеаризуемости — свойство алгоритма, при котором результат любого параллельного выполнения эквивалентен некоторому последовательному. В статье предлагается инструмент, проверяющий структуру данных на линеаризуемость при определенных допущениях (структура данных должна быть неблокирующейся и не зависящей от внешней среды). Инструмент работает «по определению» — ищет набор операций над структурой данных, параллельное исполнение которых не может быть объяснено последовательным. С помощью инструмента было показано, что несколько структур данных из общедоступных библиотек ведут себя некорректно на определенных наборах операций.**

Ключевые слова—линеаризуемость; Java; многопоточные алгоритмы

I. Введение

При разработке алгоритма важна не только теоретически доказанная корректность, но и правильная программная реализация. В однопоточной среде для проверки реализации применяются такие подходы к тестированию, как модульное и нагрузочное тестирование. Однако, в связи с развитием многопроцессорных систем, увеличивается число многопоточных алгоритмов. Упомянутые подходы к тестированию в целом оказываются мало эффективны в многопоточном окружении, так как программа может работать без ошибок почти всегда, кроме редких ситуаций, когда ошибка проявит себя, вследствие особой конфигурации потоков и чередования операций в них. Существует потребность в инструменте, который проверяет корректность реализации многопоточного алгоритма.

Общепринятым критерием «корректности» для многопоточных программ является понятие *линеаризуемости* [1] – свойство программы, при котором результат любого параллельного выполнения операций эквивалентен некоторому последовательному. В статье описан инструмент, который проверяет структуру данных на линеаризуемость при некоторых допущениях. Инструмент пытается найти набор операций, на котором линеаризуемость будет нарушена. Если такой набор будет найден, это значит, что структура данных нелинеаризуема. В противном случае никакого определенного ответа дать нельзя.

Для его реализации потребовалось решить задачи генерации тестовых наборов, генерации многопоточных исполнений для этих наборов и проверки полученных многопоточных исполнений.

С помощью созданного инструмента были найдены наборы операций, приводящие к некорректному поведению некоторых структур данных из общедоступных библиотек.

II. Корректность многопоточных структур данных

В качестве критерия корректности реализации многопоточной структуры данных можно использовать понятие *линеаризуемости*.

Если говорить неформально, структура данных *линеаризуема*, если можно считать, что каждая ее операция производит действие над структурой мгновенно, в один момент времени, лежащий между вызовом операции и ее возвратом. Такая операция называется *атомарной*. Место, где операция производит действие, называется *точкой линеаризации* [1].

Пример, иллюстрирующий точки линеаризации, показан на рисунке 1. Показано многопоточное обращение к структуре данных «счетчик»: по две операции incAndGet(), вызываемые из двух потоков A и B. Несмотря на то, что операции перекрываются, каждая операция возымела эффект в точке, помеченной крестом. И, соответственно, каждая операция корректно отработала, вернув свое, уникальное, значение.

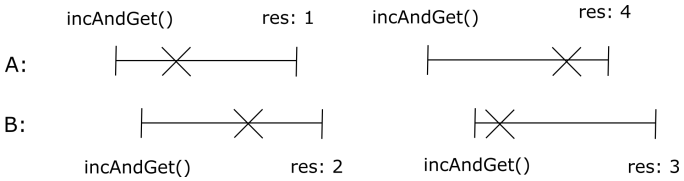

Рис. 1. Пример точек линеаризуемости

Теперь определим понятие линеаризуемости более формально, следуя классической монографии The Art of Multiprocessor Programming [1].

Исполнение многопоточной системы представлено *историей* – конечной последовательностью *событий вызова* и *возврата*, где *событие вызова* – момент вызова операции над структурой данных, а *событие возврата* – момент возврата из нее. Подыстория истории H – подпоследовательность событий истории H. Мы будем записывать событие вызова, как $\langle o\ m(args)\ A\rangle$, где o – объект, m – его метод с аргументами $args$, а A – идентификатор потока. Событие возврата будем записывать, как $\langle o\ r(ret)\ A\rangle$, где o – объект,

r – это либо Ok, либо код ошибки, ret – результирующее значение, а A – идентификатор потока.

Событие возврата *соответствует* событию вызова, если они принадлежат одному объекту и потоку. В случае, если метод был вызван несколько раз из одного потока, образуется несколько событий возврата и все они соответствуют методу вызова. Введем понятие *вызова метода*. *Вызовом метода* будем называть пару из события вызова и следующего соответствующего ему события возврата. Событие вызова называется *незавершенным*, если дальше в H нет ни одного соответствующего ему события возврата.

Расширение истории H – история, полученная из H добавлением *событий возврата* к некоторым (нулю или более) незавершенным событиям вызова. Будем обозначать подисторию H, состоящую из всех соответствующих друг другу событий вызова и возврата, как $complete(H)$.

История H *последовательна*, если она состоит из чередующихся событий вызова и соответствующих им событий возврата и начинается с события вызова.

Подыстория потока, или $H|A$ – подпоследовательность всех событий истории H для потока A. Две истории H и H' *эквивалентны*, если для любого потока A $H|A$ совпадает с $H'|A$. История H *правильно сформирована*, если для каждого потока A $H|A$ – последовательна.

Последовательная история H *корректна*, если подыстория для каждого объекта корректна, то есть соответствует спецификации поведения объекта.

Вызов метода m_0 *предшествует* вызову m_1 в истории H, если событие возврата для m_0 произошло до события вызова m_1.

Итак, история H *линеаризуема*, если есть такое ее расширение H' и такая корректная последовательная история S, что:

- $complete(H')$ эквивалентна S;
- Если вызов метода m_0 предшествовал вызову m_1 в H, то это должно соблюдаться и в S.

Будем называть S *линеаризацией* H.

Таким образом, было введено формальное определение линеаризуемости. И если найдется способ проверять его программным путем, то появится возможность автоматически проверять структуру данных на линеаризуемость.

III. Похожие разработки

Есть несколько других работ, в которых описаны подобные инструменты.

В одной из работ описан инструмент COLT [2]. Этот инструмент проверяет атомарность композиции заведомо атомарных операций структуры данных, пользуясь определением линеаризуемости. Для этого в произвольных местах проверяемой композиции вставляются другие операции над структурой данных, как будто выполняемые параллельно. Это возможно вследствие атомарности операций. Далее полученные наборы операций исполняются и полученные результаты проверяются. Однако такой подход не подходит для тестирования непосредственно структуры данных, но в работе есть полезные замечания о взаимовлиянии операций, которые учтены в разработанном инструменте.

В другой работе описан инструмент Line-Up [3], который проверяет линеаризуемость структуры данных. В нем используется иной подход генерации тестовых наборов: там не учитываются взаимовлияние операций и согласованность аргументов. Также там используется иной подход к генерации многопоточных исполнений. Для генерации используется программа проверки моделей (model checker) CHESS [4], что вносит свою специфику, так как в ней используется управление чередованием операций за счет вмешательства в механизм диспетчеризации исполнения нативного кода и такой метод плохо подходит для обнаружения ошибок, которые могут возникнуть при JIT-оптимизации и связанных с моделью памяти аппаратного уровня. В нашем подходе используется генерация многопоточных исполнений конечного, скомпилированного и оптимизированного кода.

Также существует инструмент нагрузочного тестирования jcstress [5]. Он позволяет тестировать многопоточное исполнение кода, но не в атоматическом режиме. Требуется непосредственно реализовать функции, которые будут исполняться параллельно, и все ожидаемые результаты. В нашем же подходе различные многопоточные сценарии и возможные корректные результаты генерируются в автоматическом режиме. В некотором роде созданный инструмент можно считать развитием jcstress.

IV. Метод проверки корректности

Целью исследования является разработка и реализация метода, позволяющего проверять линеаризуемость структуры данных в автоматическом режиме «по определению», и при этом работающего достаточно эффективно и требующего минимального количества настроек со стороны программиста.

Формальное доказательство линеаризуемости является сложной задачей. Наш метод будет стремиться найти нарушения линеаризуемости, а именно искать наборы операций, на которых многопоточная структура данных ведет себя некорректно.

В качестве первого шага на пути к решению сузим решаемую задачу и не будем рассматривать структуры данных, которые при вызове какой-то операции могут заблокироваться, ожидая выполнения какого-то условия. Это значит, что любая операция должна завершаться вне зависимости от текущего состояния структуры. Если же операция не завершается, значит внутри произошел dead-lock, зацикливание или проблема иного рода и это сразу же сигнализирует об ошибке. После этого допущения во всех опредениях не требуется рассматривать операции, которые являются *незавершенными*. Таким образом, рассмотрение блокирующихся структур данных выходит за рамки нашего исследования.

После этого определение линеаризуемости принимает следующий вид:

История H *линеаризуема*, если существует такая корректная последовательная история S, что:

- H эквивалентна S;
- Если вызов метода m_0 предшествовал вызову m_1 в H, то это должно соблюдаться и в S.

В качестве второго шага вводится следующее предположение: для проверки структуры данных на линеаризуемость нужно перебрать достаточно большое количество многопоточных историй, проанализировать результаты выполнения и попытаться их объяснить последовательной перестановкой. Если каждая многопоточная история будет объяснена, то можно с достаточно хорошей вероятностью утверждать, что структура данных линеаризуема.

Стоит отметить, что такой подход не позволяет непосредственно найти ошибку в реализации, но он может сиг-

нализировать о ее наличии, предоставляя тест, на котором поведение структуры данных в многопоточной среде будет отличаться от ожидаемого.

Введенные предположения позволяют предложить следующий метод.

1) Описание метода: Выбирается небольшое количество потоков, в которых будет тестироваться структура данных. Далее создается тестовый набор операций. Он представляет собой небольшой фиксированный набор операций для каждого потока, которые будут вызваны над объектом. Для этого набора генерируются все возможные последовательные исполнения для объекта и фиксируются результаты выполнения операций каждого такого исполнения. Последовательное исполнение состоит из набора операций, являющихся перестановкой исходного набора операций. Перестановка, суженная на конкретный поток, должна давать в точности те операции и в том же порядке, что предназначены для этого потока. Набор последовательных исполнений конечен, так как рассматриваются только перестановки исходного набора операций.

Мы получаем эталонные исполнения этих операций над объектом и, если в многопоточном режиме получается какое-то новое исполнение, это однозначно сигнализирует об ошибке.

Причем желательно, чтобы этот набор операций был не абсолютно случайным. Некоторые операции над структурой данных принимают аргументы. Можно ввести понятие классов эквивалентности входных параметров. Входные параметры из одного класса эквивалентности вызывают одно и тоже внутреннее поведение структуры.

Например, скорее всего неважно, какое значение добавлять в очередь. Поведение структуры никак не изменится от этого значения. Таким образом, все входные параметры для операции добавления в очередь образуют один класс эквивалентности.

Но если значение – ключ для хеш-таблицы, то операции, вызванные с разными ключами в разных потоках параллельно могут никак не пересечься по логике структуры и будут выполнены изолированно друг от друга. Поэтому, каждое значения ключа для хеш-таблицы образует свой класс эквивалентности (если не учитывать коллизии). Если ключи будут принадлежать одному классу эквивалентности, то есть будут одинаковыми, то могут возникать различные эффекты, которые и могут помочь выявить ошибку. Например вставка в хеш-таблицу разных значений с одинаковым ключом.

Также стоит обратить внимание на операции, никак не изменяющие структуру данных, а лишь обращающиеся к данным и определенным образом их обрабатывающие. Подобные операции, выполненные параллельно, скорее всего будут работать ровно также как и в однопоточном режиме. Поэтому при генерации тестовых наборов стоит это учесть и не ставить в одновременное параллельное выполнение такие операции.

Следующим шагом после генерации последовательных исполнений является генерация многопоточных исполнений набора операций. Для каждого многопоточного исполнения фиксируется результат выполнения каждой операции. Далее этот набор результатов нужно проверить. Для любого многопоточного исполнения должно существовать последовательное исполнение, приводящее к тому же результату. Поэтому перебираются многопоточные исполнения, и ищется соответствие результатов среди последовательных. Если найдено последовательное исполнение, совпадающее с многопоточным, значит многопоточное исполнение корректно, и мы переходим к следующему. В противном случае обнаружилось многопоточное исполнение, которое нелинеаризуемо, а значит, структура данных не удовлетворяет этому свойству. Дополнительно получен тестовый набор операций, на котором произошла ошибка и его, далее, можно использовать для анализа и поиска ошибки в реализации.

Метод можно представить в виде блок-схемы, изображенной на рисунке 2.

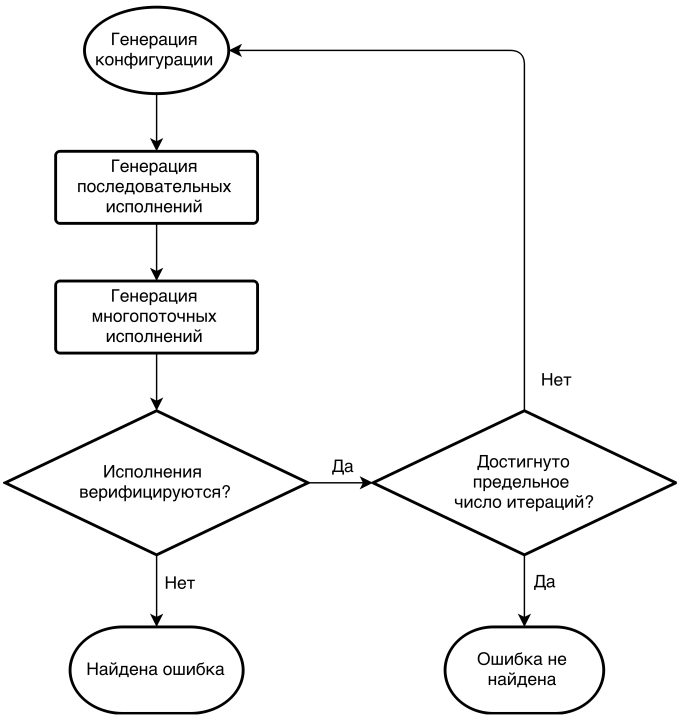

Рис. 2. Блок-схема алгоритма

2) Границы применимости метода: Структура данных не должна зависеть от внешних факторов – например, внутри нее не должно быть работы с сетью. При повторных выполнениях одной и той же последовательности операций должны получаться одни и те же результаты. В противном случае нет возможности создать эталонные последовательные исполнения и, таким образом, проверить многопоточные.

Как было сказано, данный метод находит не непосредственно ошибки, а набор операций, на которых многопоточная структура данных ведет себя некорректно. Некорректное поведение может быть вызвано такими ошибками, как незащищенный доступ к разделяемой памяти, неправильная синхронизация исполнения или логические ошибки в частях алгоритма, призванных обеспечить корректность алгоритма при многопоточном исполнении – иными словами, любые ошибки, которые приводят к иному поведению в многопоточной среде по сравнению с однопоточной.

В случае нахождения набора операций, на котором структура данных ведет себя некорректно, можно утверждать, что структура данных нелинеаризуема. В противном случае никакого определенного ответа дать нельзя. Будем

считать, что ожидается корректность структуры данных. То есть нулевая гипотеза – структура данных линеаризуема. Тогда инструмент может давать следующие варианты ответов: «линеаризуема и это неправильно», «линеаризуема и это правильно» и «нелинеаризуема и это правильно». Таким образом, ошибки первого рода отсутствуют (ложных срабатываний быть не может), второго рода – присутствуют (могут не найтись наборы операций для некорректных структур).

Итого, границы применимости метода можно определить так: метод точен, но не полон и его можно применять на неблокирующихся структурах данных, не зависящих от внешней среды и заявляющих себя, как безопасные при исполнении в многопоточной среде (то есть удовлетворяющие свойству линеаризуемости).

V. Особенности реализации метода

A. *Генерация тестовых наборов*

Требуется каким-то образом предоставить входные данные анализатору. Просто передать ссылку на соответствующий класс недостаточно – непонятно, какие методы вызывать, с какими аргументами их вызывать и с какими параметрами тестировать данный класс.

Для решения этой проблемы был реализован механизм аннотаций. Идея была взята из библиотеки jcstress [5]. Чтобы протестировать какую-то структуру данных, требуется создать класс и разметить его специальным образом с помощью аннотаций.

```
1  @CTest(iter = 300, actorsPerThread = {"1:3", "1:3"})
2  @CTest(iter = 300, actorsPerThread = {"1:3", "1:3", "1:3"})
3  public class QueueTest {
4      public Queue<Integer> q;
5
6      @Reset
7      public void reload() {
8          q = new GenericMPMCQueue(16);
9      }
10
11
12     @Actor(args = {"1:10"})
13     public void offer(Result res, Object[] args) throws Exception {
14         Integer value = (Integer) args[0];
15         res.setValue(q.offer(value));
16     }
17
18     @Actor(args = {})
19     public void poll(Result res, Object[] args) throws Exception {
20         res.setValue(q.poll());
21     }
22
23     @Immutable
24     @Actor(args = {})
25     public void peek(Result res, Object[] args) throws Exception {
26         res.setValue(q.peek());
27     }
28 }
```

Листинг 1. Пример класса для тестирования очереди

Рассмотрим пример такого класса (листинг 1). Для настройки анализатора применяется аннотация @CTest, которая имеет следующие параметры: iter – количество сгенерированных тестовых наборов, actorsPerThread задает количество потоков и диапазон количества сгенерированных операций для каждого потока. Значение для actorsPerThread – набор строк, количество строк соответствует количеству потоков, а отдельная строка задает открытый справа диапазон количества операций для соответствующего потока.

Аннотация @Reset помечает метод, который будет использован для инициализации структуры (требуется переинициализировать структуру после каждого исполнения операций на ней).

Для того чтобы указать методы, требующие тестирования, и аргументы, для них нужно создать специальные методы и пометить их аннотацией @Actor. В качестве параметра args задается диапазон сгенерированных значений для каждого аргумента метода. Внутри созданного метода требуется вызвать нужный метод структуры данных с предоставленными значениями и сообщить результат исполнения операции в служебную структуру Result с помощью соответствующих процедур. Если вызываемый метод над структурой данных завершается исключением (это может быть частью контракта структуры данных – например, извлечение из пустой очереди), то это будет корректно обработано и записано соответствующим образом в структуру Result.

Достаточно сложно оценить качество тестового набора. Но о некоторых наборах можно сказать, что они вряд ли помогут выявлению ошибок. К таким наборам относятся те, которые никак не изменяют структуру данных, или те, в которых результаты всех операций неотличимы друг от друга (ничего не возвращают).

Существует опциональная возможность пометить метод как неизменяющий структуру данных (если есть такое априорное знание) для улучшения качества сгенерированного набора с помощью аннотации @Immutable. В таком случае в тестовом наборе среди параллельных операций обязательно найдется операция, изменяющая структуру данных.

Представленный класс определяет следующую конфигурацию: два теста, в каждом генерируется по 300 тестовых наборов. В первом тесте 2 потока, во втором 3. Для каждого потока генерируется от 1 до 2 (включительно) операций. Возможные операции: offer (с аргументом от 1 до 10), poll и peek.

Пример тестового набора представлен в листинге 2.

```
1  A: q.peek(); q.peek();
2  B: q.offer(3); q.offer(4);
```

Листинг 2. Пример тестового набора для очереди

B. *Генерация многопоточных исполнений*

После того, как тестовый набор операций был создан, необходимо сгенерировать многопоточные исполнения для него.

Для этого созданный набор исполняется большое количество раз в многопоточном режиме и результаты исполнения сохраняются.

1) Качество генератора: Важным условием, влияющим на работоспособность создаваемого анализатора, является качество генератора многопоточных исполнений. Если при запусках будет преобладать какой-то конкретный порядок исполнения команд, то значимых результатов не получится. Нужно стремиться к тому, чтобы операции над

структурой данных исполнялись максимально разнообразно, покрывая все возможные состояния структуры.

Также требуется, чтобы параллельные операции исполнялись действительно параллельно, иначе многопоточное исполнение вырождается в последовательное, и нарушений линеаризуемости найдено не будет. Пример такого вырождения – когда вследствие особенностей диспетчеризации потоков операции для первого потока исполняются раньше, чем запускается второй поток, и генерируется один и тот же вариант исполнения.

Качество генератора оценивается по скорости работы, так как более быстрый генератор в единицу времени сгенерирует большее количество многопоточных исполнений, что должно привести к лучшему покрытию всевозможных исполнений. Также оценивается распределение возможных исходов – если последовательный анализ показал какое-то количество определенных исходов, а генератор покрывает только некоторые из них, либо доминирует какой-то конкретный исход, скорее всего мы не получим достоверного покрытия вероятных исходов, что ухудшит способность генератора к нахождению ошибок реализации.

Для того, чтобы наборы операций стартовали одновременно и исполнялись без лишних задержек, были проделаны следующие шаги.

Для общего снижения издержек на создание и инициализацию различных служебных структур данных создается пул потоков, в котором и исполняются наборы операций. При моделировании многопоточных исполнений никаких служебных структур не создается – они создаются заранее и передаются в многопоточную модель.

Для того, чтобы наборы операций начинали исполняться одновременно, потоки, исполняющие их, синхронизируются с помощью барьера. При достижении барьера поток блокируется. После того, как все потоки достигнут барьера, барьер снимается, и все потоки одновременно продолжают исполнение.

2) Вызов методов: Следующий вопрос, требующий решения – каким образом вызывать методы у структуры данных. Java предоставляет специальную библиотеку, Reflection API [6], позволяющую получать методы у класса и вызывать их. Однако на практике она показала себя не очень хорошо при использовании в нашем инструменте. Во-первых, вызов методов с ее помощью происходит с большими накладными расходами по времени, а с учетом того, что нам требуется делать большое количество вызовов методов у структуры, это становится критичным. Во-вторых, ее использование вносит различные, неконтролируемые задержки между вызовами операций (стек служебных вызовов, приводящий в итоге к вызову нужной нам операции), что снижает качество генератора и снижает возможности по контролируемому исполнению кода. В-третьих, библиотека может вносить неявную синхронизацию при вызове методов, что негативно влияет на качество генерации многопоточных исполнений. Поэтому было решено отказаться от Reflection API в пользу генерации Java байткода.

Идея состоит в генерации кода с минимальным добавлением служебных операций, который легко бы поддавался оптимизации JIT-компилятором. Для реализации этого подхода используется библиотека ASM [7]. Был создан базовый класс, в котором определен нужный метод. Далее генерируется по одному наследнику для каждого потока от этого класса и у них переопределяется этот метод так, чтобы он содержал вызовы нужных операций. После этого при генерации многопоточных исполнений достаточно вызывать метод у нужного наследника.

3) Кеш-линии: На выполнение многопоточного кода может повлиять так называемая проблема False Sharing [8]. Процессор оперирует памятью не побайтово, а так называемыми кэш-линиями, которые представляют собой блоки фиксированного размера (часто используется размер 64 байта, но могут быть и другие размеры). И если два разных объекта разделяют общую кэш-линию, то изменение одного объекта может повлечь за собой загрузку данных из памяти для второго объекта заново, хотя данные и не менялись, что может повлиять на исполнение многопоточного кода. Один из подходов для решения этой проблемы – добавление фиктивных полей в структуру, чтобы ее размер был кратен размеру кэш-линии и, таким образом, она не разделяла какую-то кэш-линию с другим объектом. Для облегчения выравнивания в Java 8 была введена аннотация @Contended и проаннотированный таким образом класс выравнивается автоматически. Аннотация была применена к структуре Result, в которую и происходит запись данных из разных потоков. Выравнивание объектов Result никак не влияет на структуру памяти исследуемого объекта и все связанные с этим особенности его реализации остаются неизменными.

4) Управление выполнением: После того как наборы операций исполняются эффективно, можно попробовать внести контролируемые изменения в порядок исполнения. Для этого нужно наблюдать за получающимися результатами и каким-то образом влиять на исполнение для обеспечения максимального покрытия. Это может быть сделано добавлением небольших, динамически изменяющихся задержек перед каждой операцией. Однако на данный момент в инструменте используется упрощенный подход. Фаза генерации многопоточных исполнений была разбита на две части. В первой фазе все операции исполняются без задержек. Во второй фазе перед каждой операцией вносится случайная задержка из небольшого диапазона. Такой подход привел к немного лучшему распределению результатов многопоточных исполнений (многопоточным исполнениям соответствуют однопоточные и хороший генератор будет производить различные многопоточные исполнения, приводящие к разным результатам, для плохого же генератора будут доминировать одни и те же многопоточные исполнения) и повысил процент тестовых наборов, засчитанных как приводящие к ошибке.

Нужно отметить, что на первых итерациях многопоточного запуска может наблюдаться поведение, необычное для последующих запусков. Это так называемая фаза прогрева, когда большую роль играют особенности диспетчеризации потоков и JIT-оптимизация. Однако это не имеет негативного влияния на разработанный метод, поскольку он не имеет ложных срабатываний. Потребовалось учесть фазу прогрева так, чтобы итерации этой фазы не составляли большую часть общего количества итераций.

Количество многопоточных запусков ограничено фиксированным числом итераций и является разумным компромиссом между временем работы инструмента и качеством нахождения ошибок. Для этого было взято достаточно большое количество запусков, при котором ошибки находились. Далее было определено реальное число запусков, которое обеспечивало нахождение ошибок (к примеру, делалось

100000 запусков, но ошибки находились уже на первых 50000 итераций).

C. *Проверка многопоточных исполнений*

Для проверки многопоточного исполнения требуется найти последовательное исполнение, ему соответствующее. Соответственно, необходим способ построения последовательных исполнений.

Для построения последовательных исполнений требуется сгенерировать все перестановки исходного набора операций такие, что перестановка, суженная на конкретный поток, дает в точности те операции и в том же порядке, что для него предназначены. Можно сказать, что планировщик исполнения случайным образом выбирает поток, из которого он будет выполнять следующую операцию.

Пример последовательных перестановок для примера на листинге 2 представлен в листинге 3.

```
[q.peek(); q.peek(); q.offer(3); q.offer(4); ]
[q.peek(); q.offer(3); q.peek(); q.offer(4); ]
[q.peek(); q.offer(3); q.offer(4); q.peek(); ]
[q.offer(3); q.peek(); q.peek(); q.offer(4); ]
[q.offer(3); q.peek(); q.offer(4); q.peek(); ]
[q.offer(3); q.offer(4); q.peek(); q.peek(); ]
```

Листинг 3. Пример последовательных перестановок

После того, как все такие перестановки были сгенерированы, для каждой перестановки последовательно выполняются ее операции над структурой данных и сохраняется результат.

VI. Практические результаты

A. *Тестирование на синтетических примерах*

Для первичного тестирования инструмента было реализовано несколько структур данных и в них были допущены синтетические ошибки.

Рассмотрим структуру данных **счетчик** и ее ошибочную реализацию и покажем, что было найдено анализатором. Также это поможет проиллюстрировать его работу. Примером использования **счетчика** является подсчет количества появлений каких-либо событий в многопоточной среде.

В листинге 4 представлена корректная реализация.

```
public class Counter {
    private volatile int c = 0;

    public int get() {
        return c;
    }

    public synchronized int getAndIncrement() {
        return c++;
    }
}
```

Листинг 4. Структура данных "счетчик"

Ключевое условие, обеспечивающее корректность реализации – защищенный с помощью ключевого слова synchronized метод incrementAndGet. Это значит, что внутри метода единовременно может находиться только один поток.

В случае, если по каким-то причинам ключевое слово будет потеряно, возникает классическая проблема – *состояние гонки*. Это происходит, потому что операция c++ не атомарна. При совершении этой операции сначала значение переменной c читается, потом увеличивается, потом записывается обратно. Поэтому, если два потока считали одно и тоже значение, потом увеличили его и записали обратно, счетчик увеличится на значение 1. А ожидалось, что на 2.

Или, в худшем случае, один поток считал значение, а другой в это время увеличил счетчик несколько раз и записал результат. Первый же поток увеличивает свое значение и записывает его обратно, что приводит к откату счетчика.

Анализатор находит данную ошибку. В листинге 5 отображена найденная конфигурация и недопустимое поведение для многопоточного счетчика. В фигурных скобках показан результат выполнения операции. Обе операции incrementAndGet вернули значение 1, что не является допустимым поведением.

```
[0_incrementAndGet() {1}]
[1_incrementAndGet() {1}]
```

Листинг 5. Пример для ошибочного счетчика

Также были реализованы многопоточная очередь и примитив банковского приложения (функциональность, которую должна обеспечивать структура – получение количества средств на счете клиента, установка значения средств на счете клиента, и перевод средств между разными клиентами) и в них были допущены ошибки синхронизации при доступе к разделяемым переменным. Для всех реализаций инструмент представил пример, на котором линеаризуемость нарушается.

В качестве примеров были выбраны несложные структуры данных, корректность работы которых обеспечивалась с помощью блокировок, так как эти ошибки в таких структурах достаточно наглядны, и это те примеры, на которых анализатор точно должен находить ошибки. Однако потенциально, спектр обнаруживаемых ошибок не ограничивается потерями блокировок, ведь анализируется не корректность синхронизации, а общее поведение структуры данных, которое в многопоточном режиме должно также быть корректным и соответствовать спецификации поведения структуры данных. Исходя из сути метода, могут быть обнаружены только те ошибки, которые приводят к неправильному поведению именно в многопоточной среде.

B. *Тестирование на доступных библиотеках*

Было проведено тестирование некоторых доступных библиотек созданным инструментом.

Сначала была протестирована стандартная библиотека java.util.concurrent [9] и тестирование не выявило ошибок. Далее была протестирована библиотека Google Guava [10] и ошибок также не нашлось.

Эти библиотеки используются в большом количестве проектов и скорее всего все возможные ошибки уже были выявлены и исправлены.

Далее были протестированы экспериментальные библиотеки, содержащие реализации многопоточных очередей и хэш-таблиц - jctools [11], zchannel [12] и hish_scale_lib [13]. Тестирование показало, что некоторые структуры оттуда реализованы некорректно и, действительно, возможны

Таблица I. Статистика для найденных ошибок

Структура	Тестовый набор	Многопоточное исполн.	Время (мс)
Counter	1 – 1 (1)	2 – 1695 (530)	4 – 110 (33)
Queue	1 – 28 (7)	6 – 997 (307,6)	5 – 38786 (8635)
Accounts	1 – 39 (14,9)	19 – 5547 (1199,4)	3 – 50606 (18697)
NonBlockingSetInt	1 – 7 (3,2)	1846 – 54522 (29736,4)	52 – 10414 (4291)
NonBlockingHashSet	1 – 33 (8,5)	1 – 58749 (24661,7)	5 – 66975 (16111)
MpmcArrayQueue(2)	2 – 25 (11,1)	335 – 67590 (34806,5)	2352 – 45628 (19974)
MPMCQueue(2)	1 – 18 (10,6)	9034 – 96977 (52290)	1814 – 33958 (19714)
MPMCQueue(16)	2 – 20 (5,8)	454 – 23292 (14409,8)	1877 – 33858 (8651)
LockFreeQueue	6 – 193 (76,375)	2588 – 66147 (32718)	8825 – 342199 (135202)

такие наборы операций, что структура ведет себя неправильно.

А именно, были найдены ошибки в

- NonBlockingSetInt, NonBlockingHashMap, NonBlockingHashSet из high_scale_lib.
- MpmcArrayQueue из jctools
- GenericMPMCQueue(2), GenericMPMCQueue(16) из zchannel
- lockfreequeue, найденная на GitHub [14]

Инструмент показал, что некоторые структуры данных из доступных библиотек, предоставляющих многопоточные реализации структур данных, реализованы некорректно, и в некоторых случаях их поведение может отличаться от ожидаемого.

Может быть, эти ошибки и нечасто встречаются в реальной жизни, но сам факт того, что они могут встретиться, важен. И структуры, которые заявлены как безопасные к использованию в многопоточной среде, по существу, таковыми не являются.

В таблице 1 показана статистика параметров работы инструмента для найденных ошибок. В ней отражен номер тестового набора, приводящего к ошибке, количество многопоточных запусков, потребовавшихся для выявления некорректного поведения, а также общее время поиска. В таблице указаны диапазоны полученных значений и усредненное значение для 10 запусков.

VII. Заключение

В работе было предложено решение, позволяющее проверять линеаризуемость структуры данных в автоматическом режиме. Был создан инструмент, реализующий это решение. Он не способен доказать выполнение линеаризуемости, однако способен предъявить тест, на котором удалось обнаружить нарушение линеаризуемости. Далее этот тест может быть использован для анализа реализации и нахождения ошибки.

Инструмент состоит из трех частей – генератора тестовых наборов для структуры, генератора последовательных исполнений этих тестовых наборов и генератора многопоточных исполнений тестовых наборов.

Созданный инструмент находит наборы операций, вызывающие неправильное поведение реализаций с допущенными ошибками, а также с помощью него было показано, что некоторые структуры данных из доступных библиотек, таких как jctools, zchannel и high_scale_lib, на некоторых найденных тестовых наборах ведут себя некорректно.

Ключевыми моментами, влияющими на качество анализатора, являются генератор тестовых наборов и генератор многопоточных исполнений.

В качестве возможного направления развития инструмента можно выделить развитие генератора наборов, используя априорные сведения о взаимовлиянии операций и исключая из анализа наборы, которые не могут привести к некорректному многопоточному исполнению.

Другим возможным развитием работы может являться улучшенный генератор многопоточных исполнений, который использует взамен множественного запуска тестовых наборов контролируемое исполнение этих наборов и обеспечивает более эффективное покрытие возможных путей исполнения. Применение данного подхода приведет к своим компромиссам в виде выбора подмножества всех возможных вариантов исполнения и искажения низкоуровневых эффектов реализации. Потребуется отдельное сравнительное исследование эффективности и скорее всего имеет смысл поддерживать оба варианта. Также в случае использования такого подхода будет получен не только тест, приводящий к ошибке, но и непосредственно образец выполнения кода, что поможет в локализации проблемы.

Также возможна интеграция инструмента с анализатором DRD (Data Race Detector) [15], что расширит диапазон обнаруживаемых ошибок.

На данном этапе реализованный инструмент показал свою практическую применимость и может быть использован при разработке многопоточных структур данных.

Список литературы

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

[2] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav, “Testing atomicity of composed concurrent operations” *ACM SIGPLAN Notices - OOPSLA '11*, vol. 46, no. 10, pp. 51--64, Oct. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2048073

[3] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan, “Line-up: a complete and automatic linearizability checker” *ACM SIGPLAN Notices - PLDI '10*, vol. 45, no. 6, pp. 330--340, Jun. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1806634

[4] “Chess”. [Online]. Available: http://research.microsoft.com/en-us/projects/chess/

[5] “jcstress”. [Online]. Available: http://openjdk.java.net/projects/code-tools/jcstress/

[6] “Reflection API”. [Online]. Available: https://docs.oracle.com/javase/tutorial/reflect/

[7] “ASM library”. [Online]. Available: http://asm.ow2.org/

[8] “False Sharing”. [Online]. Available: http://robsjava.blogspot.ru/2014/03/what-is-false-sharing.html

[9] “java.util.concurrent”. [Online]. Available:

http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html
[10] “Google Guava”. [Online]. Available: https://code.google.com/p/guava-libraries/
[11] “jctools”. [Online]. Available: https://github.com/JCTools/JCTools
[12] “zchannel”. [Online]. Available: http://landz.github.io/
[13] “High Scale Lib”. [Online]. Available: https://github.com/stephenc/high-scale-lib
[14] “lockfreequeue”. [Online]. Available: https://github.com/yaitskov/lock-free-queue
[15] “Data Race Detector”. [Online]. Available: https://code.devexperts.com/display/DRD