Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем

Кафедра системного программирования

Озерцов Александр Сергеевич

Верификация потокобезопасности структур данных с помощью управляемых точек синхронизации

Выпускная квалификационная работа

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

Научный консультант:
руководитель направления dxLab Цителов Д. И.

Рецензент:
асистент СПбГПУ Беляев М. А.

Санкт-Петербург
2017

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems

Software Engineering

Ozertsov Alexander

Verification thread-safe data structures by controllable synchronisation points

Graduation Thesis

Scientific supervisor:
Professor Andrey Terehov

Scientific consultant:
Head of dxLab Dmitriy Citelov

Reviewer:
teaching assistant SPbSTU Mikhail Belyaev

Saint-Petersburg
2017

Оглавление

Введение **4**

1. Постановка задачи **6**

2. Обзор **7**

2.1. Классификация методов проверки линеаризуемости . . 7

2.2. jcstress 8

2.3. jPredictor 9

2.4. Penelope 9

2.5. Chess 9

2.6. STORM 10

2.7. Текущая реализация 11

3. Формализация операций виртуальной машины Java **12**

4. Архитектура системы управления ходом исполнения программы **14**

5. Доработка существующего алгоритма **16**

6. Разработка алгоритма, предоставляющего гарантии над пространством переключений **17**

6.1. Неформальное описание работы алгоритма 19

6.2. Формальное описание работы алгоритма 22

7. Сравнение **24**

8. Заключение **28**

Список литературы **29**

Введение

Разработка промышленных систем требует создания высокопроизводительных алгоритмов. Одним из способов увеличения производительности является многопоточность. Многопоточное программирование становится более популярным с распространением многоядерных процессоров. Однако, многопоточное программирование ставит перед программистом вопрос построения и тестирования потокобезопасных компонент, чьи методы корректны в многопоточной среде без использования неуместных барьеров и иных методов синхронизации.

Потокобезопасность компонент не является четко формализованным термином. Более корректные условия корректности типов данных включают отсутствие гонок [2], сериализуемость [8], линеаризуемость [4]. Последнее время общепринятым стандартом потокобезопасности является наличие линеаризуемости [3]. Говоря неформально, структура данных является линеаризуемой, если результат любого многопоточного алгоритма над этой структурой данных эквивалентен результату некоторого последовательного алгоритма. Иными словами многопоточное поведения можно рассматривать как набор последовательных операций.

Наиболее известными примерами линеаризуемых компонент являются типы данных, такие как очереди или множества, которые предоставлены множеством различных библиотек, таких как Java Concurrency [7] и PLINQ [9]. Классические подходы к тестированию [13] 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. Это порождает необходимость инструмента, который проверяет корректность реализации многопоточной структуры данных.

Объектно-ориентированная парадигма программирования подразумевает взаимодействие структуры данных с внешним миром при помощи некоторого конечного набора методов. Таким образом, любой алго-

ритм над структурой данных можно рассматривать как последовательность методов этой структуры и набор входных данных.

Задача доказательства линеаризуемости является довольно трудоемкой и зачастую проводится вручную. Однако, на практике оказывается довольно удобным альтернативный подход путем приведения контрпримеров [14], то есть таких алгоритмов, которые могут работать некорректно. Соответственно, при достаточно большом количестве рассмотренных алгоритмов можно дать вероятностную оценку линеаризуемости всей структуры данных в целом.

Одним из инструментов, предлагающих автоматизировать проверку линеаризуемости, является lin-check [18] — библиотека, генерирующая большой набор тестовых алгоритмов над структурой данных и проверяющая их корректность путем многократного исполнения в многопоточной среде и проверки результатов на соответствие некоторому последовательному исполнению. Однако, данная библиотека не учитывает внутреннюю структуру функций тестируемых типов данных и не имеет планировщика выполнения для управления ходом исполнения тестового алгоритма, что порождает множество ненужных запусков для корректного (с точки зрения линеаризуемости) алгоритма, что в свою очередь отрицательно влияет на производительность всей системы в целом.

1. Постановка задачи

Целью данной работы является разработка и внедрение системы управления порядком исполнения многопоточных алгоритмов над структурой данных в проекте lin-check.

В рамках данной работы сформулированы следующие задачи:

- выполнить обзор методологий тестирования многопоточных программ;
- разработать архитектуру для создания стратегий управления ходом исполнения программы;
- формализовать операции виртуальной машины Java, влияющие на поведение программы в многопоточной среде;
- доработать существующий алгоритм управления порядком исполнения с учетом формализованных операций виртуальной машины Java;
- разработать алгоритм управления порядком исполнения многопоточных программ;
- сравнить полученные алгоритмы с нынешней реализацией.

2. Обзор

Проверка структур данных основана на подборе нелинеаризуемых исполнений, поэтому основной задачей является определение потокобезопасности сгенерированного алгоритма.

На данный момент есть массив инструментов, которые успешно обнаруживают ошибки в многопоточных программах используя широкий диапазон эвристических техник, которые были разработаны чтобы облегчить проблему быстрого роста чередований, который присущ в анализе параллельных программ. Данная глава посвящена обзору основных инструментов.

2.1. Классификация методов проверки линеаризуемости

С классической точки зрения техники тестирования часто сочетаются с понятием покрытия (код, условия, входные данные и т. д.), что дает технике гарантию. Проверка линеаризуемости выделяет два основных пространства, влияющие на поведение программы в многопоточной среде: пространство чередований и пространство входных данных. То есть, можно утверждать о линеаризуемости тестового алгоритма после покрытия обоих пространств. Соответственно, техники тестирования параллельных программ можно разбить на 3 категории.

1. Без каких-либо гарантий покрытия. Подходы, основанные на философии использования эвристик для достижения прерываний, которые наиболее вероятно содержат ошибки и тестирование данных прерываний так много насколько это возможно. Данные подходы очень успешны в нахождении ошибок, однако не предоставляют никаких гарантий покрытия прерываний и, следовательно, доказательств линеаризуемости.

2. Предоставляющие гарантии покрытия пространства чередований. Данные методы используют идеи ограничения контекста [10], огра-

ничений глубины [6] для приоритизации поиска в пространстве прерываний. Эти приоритизации позволяют нам дать количественную оценку как много пространства чередований протестировано, свойство, которое техника эвристического поиска категории 1 не предоставляет.

3. Гарантии покрытия над пространствами программных вводов и чередований. Техники, основанные на ограничении контекста (или иных видов приоритизации поиска), перевода параллельных программ в последовательную программу, которая имеет аналогичное поведение (до определенного момента), а затем статичного анализа программы на интересующее свойство, если сравнение провалилось, то пространства вводов и прерываний (до определенной границы) найдено. Однако, эти техники только исследуют подмножество поведений программы и следовательно не могут быть нацелены на максимальное покрытие.

2.2. jcstress

The Java Concurrency Stress Tests (jcstress) [17] представляет собой библиотеку нагрузочного тестирования многопоточных программ на языке Java. Данная библиотека тестирует некоторый многопоточный алгоритм, который описывается в методах соответствующих номеру потока. В основе данной библиотеки лежит принцип сопоставления результата работы потока заранее заданным, результатам. Таким образом, запуская один и тот же нелинеаризуемый алгоритм большое количество раз можно найти новый результат, который не ожидается в работе данного алгоритма, однако, никаких гарантий работы данная библиотека не предоставляет, то есть заведомо неверный алгоритм может отработать в соответствии с ожиданиями программиста в следствии удачного стечения обстоятельств.

2.3. jPredictor

jPredictor [16] — инструмент, основанный на принципе нарезанной трассы. Данный инструмент исполняет многопоточную программу и записывает трассу её исполнения, которая содержит следующие данные (с учетом потока):

- начало и конец метода;
- true-переходы условных операторов;
- доступ к переменным.

После этого данный инструмент анализирует трассу на предмет возможных ошибок и нарушений линеаризации. Данный инструмент также не дает никаких гарантий потокобезопасности программы по причине неполноты трассы.

2.4. Penelope

Данная библиотека [11] так же как и jPredictor выполняет программу на входных данных, записывая необходимую информацию. Затем она анализирует трассу программы и составляет расписание, учитывая шаблоны, которые могут привести к нарушению линеаризуемости. После этого в исходной программе расставляются глобальные точки синхронизации в соответствии с таблицей переключений для приведения исполнения к желаемому. Далее следует процесс повторного исполнения, который выявляет наличие ошибки или доказывает её отсутствие. Очевидно, что данная библиотека также относится к библиотекам 1 категории, так как одно единственное выполнение не может выявить все возможные расписания.

2.5. Chess

Лаборатория Microsoft Research представила инструмент проверки моделей Chess [5] для проверки многопоточных программ. Все значи-

мые операции Chess заменяет на соответствующие обертки. Это необходимо для работы планировщика. Сам планировщик работает в 3 этапа:

1. Воспроизведение. Планировщик имеет файл расписания. На 1 прогоне он пуст. Данный файл содержит частичный график, который был сгенерирован на предыдущем запуске программы.

2. Запись. В каждой точке (операциях, влияющих на линеаризуемость) отмечается какие потоки могут работать параллельно. На операциях переключения потоков выбираем поток, основываясь на приоритетах.

3. Поиск. Данный этап генерирует однопроцессорное расписание в каждой точке.

Таким образом, Chess проверяет устойчивость каждой точки, которая может нарушить линеаризуемость. Такой подход позволяет быть уверенным в покрытии пространства прерываний. Однако, Chess реализован для платформы .Net Framework, а также не учитывает паттерны SMT.

2.6. STORM

Данная библиотека [15] предназначена для исследования многопоточных программ, написанных на языке программирования C. Она основана на принципах ограничения контекста. Принцип работы можно описать тремя основными этапами.

1. Устранение сложностей C таких как динамическое выделение памяти, арифметика указателей и т. д. путем компиляции в язык BoogiePL.

2. Построение последовательной программы на языке BoogiePL, которая фиксирует все поведения параллельной программы BoogiePL (и, следовательно, исходной программы на языке C), вплоть до ограничений контекста.

3. Генерация условий проверки из последовательной программы BoogiePL и проверка с помощью универсального решателя разрешимости логических формул (SMT solver).

Данный подход не приемлем так как требуется наличие стороннего языка программирования, в который будет транслироваться язык Java. Более того, в представленном подходе мы должны угадывать значения общих переменных в начале каждого контекста. Однако неправильные догадки могут привести к достижению недопустимых состояний программы.

2.7. Текущая реализация

Нынешний принцип работы lin-check подобен jcstress. Различие заключается в том, что lin-check перед запуском многопоточного алгоритма, генерирует все возможные линеаризуемые результаты исполнения, что снимает с программиста необходимость искать все возможные результаты. Также, при каждом запуске тестового алгоритма между вызываемыми методами вставляются задержки различной величины, что делает каждой новый запуск потенциально уникальным. Ниже представлена визуализация процесса запуска тестового алгоритма.

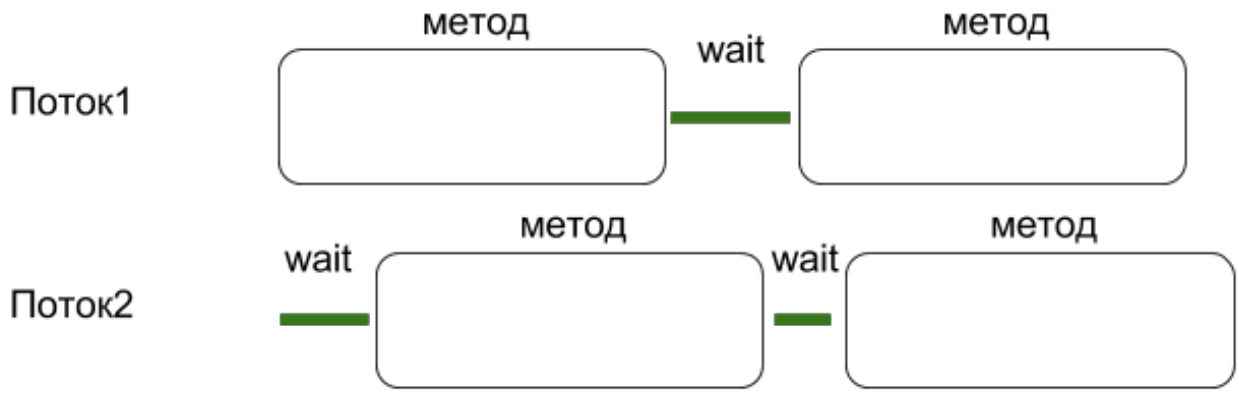

Рис. 1: Тестовый алгоритм, модифицированный lin-check

Такой подход не дает никаких гарантий покрытия, надеясь лишь на то, что ошибка проявит себя в процессе исполнения.

3. Формализация операций виртуальной машины Java

Классический подход к исследованию линеаризуемости подразумевает чтение и запись в переменные, как основные операции, которые могут изменить поведение алгоритма в многопоточной среде. Однако, такой подход не учитывает внутреннюю организацию языка программирования, с помощью которого исполняется алгоритм. Так как Lin-Checker предназначен для языка программирования Java, следовательно, нужно выделить операции виртуальной машины Java, которые могут привести к нарушению линеаризуемости.

Виртуальная машина Java являет собой низкоуровневую реализацию стековой машины с собственным набором инструкций [12]. При создании объекта, для хранения его полей создается пул постоянных констант. Любое обращение к полям структуры происходит через него. При вызове метода, JVM гарантирует его изолированность. Для выполнения операций, каждый вызванный метод имеет стек локальных переменных и стек операндов, которые вычисляются при компиляции. Таким образом, имеем 3 различные структуры, для которых определены операции чтения и записи.

Очевидно, что обращения в пул постоянных констант следует рассматривать как операции, которые влияют на линеаризуемость, так как через данный объект происходит чтение и запись в общие поля структуры данных.

Так как стек операндов предназначен для выполнения промежуточных вычислений JVM, то можно отказаться от рассмотрения данного элемента, так как с ним оперирует только JVM.

Несмотря на то, что в стек локальных переменных могут записываться значения полей, тем не менее, изменение этих значений будет изменено только в стеке локальных переменных. А при записи в стек локальных переменных ссылок на сторонние структуры данных, изменение полей структуры данных возможно только через пул постоянных констант сторонней структуры данных.

Таким образом, подозрительные операции происходят только при обращении к пулу постоянных констант, которые можно разделить на 3 группы:

1. операции доступа к полям класса;
2. операции доступа к полям объекта;
3. операции доступа к элементам массива.

4. Архитектура системы управления ходом исполнения программы

В связи с тем, что существует множество реализаций управления исполнением многопоточных программ и планируется дальнейшее расширение спектра методов, реализованных в lin-check, было решено использовать паттерн стратегия. Для этого необходимо заранее предусмотреть все методы, которые могут потребоваться для создания новых способов управления исполнением.

Вызываемые методы условно были разделены на следующие группы.

1. Управляющие ходом. Методы, которые вызываются непосредственно из многопоточной программы.

2. Проводящие подготовку к запуску. Методы, которые вызываются на этапе подготовки к запуску многопоточной программы или по окончании работы многопоточной программы.

Методы первой группы должны обеспечить возможность полного контроля ходом исполнения многопоточной программы. Для этого были внедрены методы, вызываемые в начале исполнения каждого потока, в конце выполнения и непосредственно перед доступом к разделяемой переменной, внутри которых стратегия будет решать, каким образом продолжать выполнение хода программы. При этом, перед каждым доступом к общей переменной вызываются соответствующие чтению и записи методы. Среди основных методов проверки были выделены следующие:

- onStartThread;
- onSharedVariableRead;
- onSharedVariableWrite;
- onEndThread.

Методы второй группы обеспечивают возможность провести некоторый анализ исходного кода сгенерированной программы, составление и пересчет внутренних метрик. Также, необходимо учесть возможность параллельной проверки тестовых алгоритмов. В результате было решено для каждого тестового алгоритма создавать создавать собственный объект стратегии. Ввиду того, что анализ тестового алгоритма проводится один раз перед исполнением программы, было решено проводить проводить его непосредственно в конструкторе стратегии, что гарантирует установку верных стартовых параметров стратегии. Также был учтен возможный перерасчет параметров в связи с новой информацией, полученной в ходе исполнения тестового алгоритма. В результате были созданы следующие методы:

- onStartInvocation;
- onEndInvocation;
- isNeedStop;

Данная архитектура позволяет выбирать стратегию проверки линеаризуемости программы из набора реализованных, а также разрабатывать и использовать свои собственные.

5. Доработка существующего алгоритма

Разработанная архитектура позволила заметно улучшить предыдущую реализацию управления ходом исполнения программы. А именно позволила исключить вероятность вставки следующих друг за другом переключений. Теперь переключения между потоками следуют непосредственно перед обращением к общим переменным. Такая модификация позволила несколько увеличить производительность библиотеки lin-check, однако, тем не менее не дает никаких гарантий линеаризуемости сгенерированной программы. То есть, сгенерированная программа может верно работать большое количество вызовов, однако на практике возможен вариант исполнения приводящий к нарушению принципа линеаризуемости.

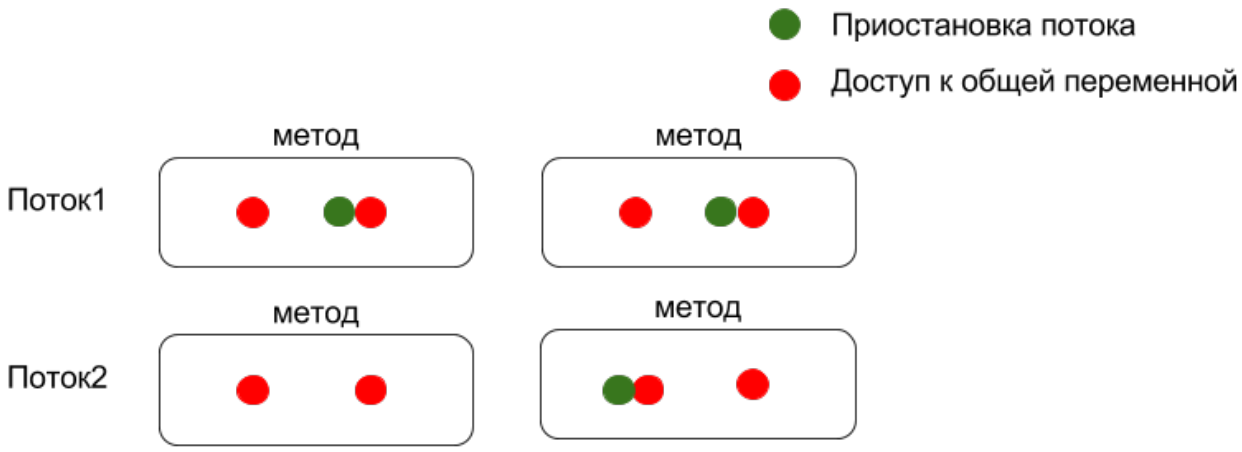

Рис. 2: Пример тестовой программы, полученной в результате доработанного алгоритма.

6. Разработка алгоритма, предоставляющего гарантии над пространством переключений

Теория проверки многопоточных программ определяет планировщик исполнения многопоточной программы как основной инструмент проверки линеаризуемости. В статье [1] проведен обзор основных видов планировщиков исполнения с доказательством их корректности. Также данная статья приводит формальное описание языка, на котором может быть написана многопоточная программа. Грамматика и семантика языка Java полностью соответствуют приведенному в статье формальному языку, что позволяет реализовать планировщик, соответствующий принципам построения планировщиков, описанных в данной статье.

Для реализации был выбран планировщик типа Round-robin. Одна из реализаций данного планировщика представлена в инструменте проверки линеаризуемости CHESS [5].

Данная глава посвящена описанию работы реализованного планировщика исполнения.

Перед описанием принципа работы алгоритма введем некоторые определения.

- Расписание – некоторая последовательность исполнения многопоточной программы.
- Очередь исполнения – перестановка чисел от 0 до k, определяющая в каком порядке будут выполняться потоки.
- Прерываемые потоки – пара чисел, обозначающая какие потоки будут прерываться между собой.
- Параметры расписания – пара из очереди исполнения и прерываемых потоков. Уникальным образом определяет расписание.

- Точка – кортеж <loc, tid, type>, где loc – локация точки, tid – номер потока, type – тип доступа к общей переменной (чтение или запись).

- Локация точки – уникальный номер точки в исследуемой структуре данных.

- История – упорядоченный набор пройденных точек.

- Окно – набор инструкций между двумя обращениями к разделяемой переменной, на которых может быть вызвана остановка потока.

- Размер окна – количество инструкций между двумя обращениями к разделяемой переменной. Соответственно, чем больше размер окна, тем выше вероятность остановки или переключения данного потока.

- Паттерн SMT – набор обращений к разделяемой переменной, которые могут вызвать нарушение линеаризуемости.

- Устойчивость окна – сохранение линеаризуемости при возникновении порядка обращения, соответствующего одному из паттернов SMT.

- Глубина устойчивости окна – количество точек, удовлетворяющее паттернам SMT, которое может выдержать окно без нарушения линеаризуемости.

6.1. Неформальное описание работы алгоритма

Данный алгоритм проверяет устойчивость каждого окна. Он основан на выполнении в каждый момент времени одного потока, что создает необходимость создания генератора очередности потоков. Это достигается за счет перебора всех возможных перестановок потоков. Также, для каждой перестановки требуется выбрать прерываемые потоки учитывая порядок следования данных потоков. Данные операции создают уникальные параметры расписания.

Таким образом, для 3 потоков будут сгенерированы следующие параметры расписания:

Очередь исполнения	Прерываемые потоки
0,1,2	0,1
0,1,2	1,2
0,2,1	0,2
0,2,1	2,1
...	
3,2,1	3,2
3,2,1	2,1

Проверка линеаризуемости алгоритма основана на приближении к однопоточному выполнению, делая переключение потоков только в исследуемом окне. Для этого стратегии сообщается порядок очередности потоков и номера потоков, которые будут прерываться (необязательно данные потоки будут первыми в очереди потоков). Во время работы, стратегия останавливает ненужные потоки, оставляя рабочим только тот поток, который должен исполняться в данный момент. Проходя каждую точку, стратегия записывает её в историю. Это необходимо для идентификации окон, так как стратегии ничего не известно о состоянии исследуемой структуры данных. То есть, окно идентифицируется по следующим параметрам:

1. история прохождения точек, приведшая к открыванию окна;
2. точка, которая закрывает окно;

После того, как мы попадаем в новое окно, необходимо переключить поток и найти новую точку, после чего переключить исполнение обратно. Важным дополнением, которое не учитывает планировщик round-robin, позволяющим уменьшить количество необходимых вызовов стал учет паттернов SMT. То есть, после переключения потока ищем новую точку, которая может влиять на линеаризуемость тестового алгоритма.

После того, как было выполнено 2 переключения, все остальные потоки выполняются без переключений. Ниже представлен рисунок иллюстрирующий ход исполнения потоков согласно описанной стратегии.

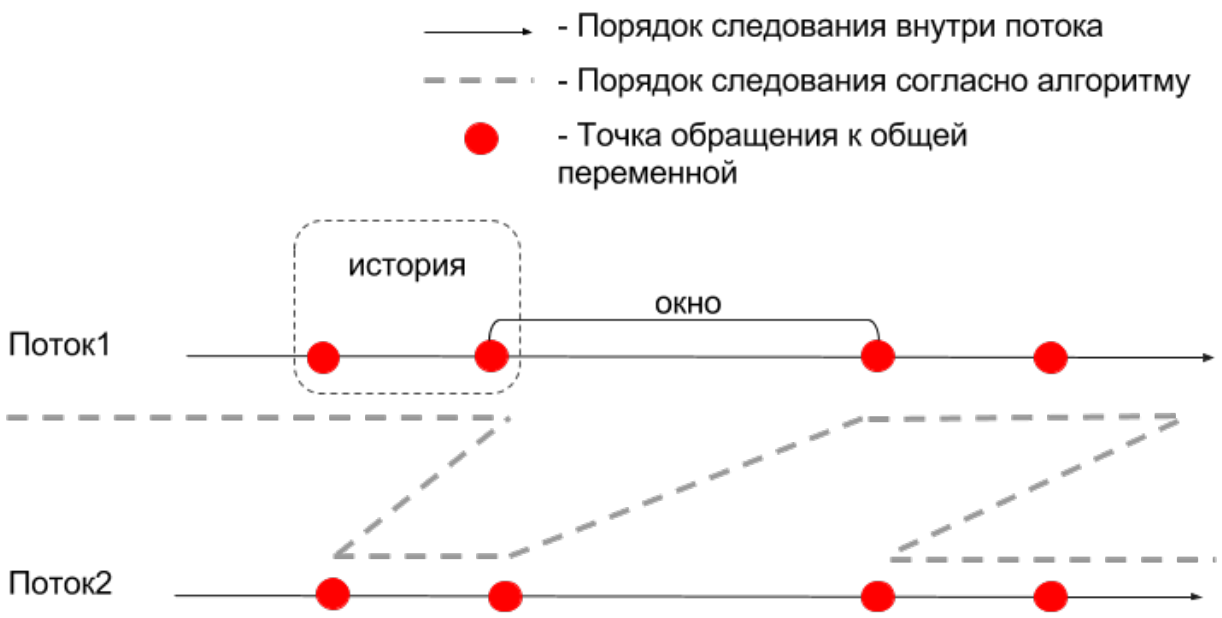

Рис. 3: Пример проверки устойчивости окна

Также важным является критерий приоритизации поиска. В классическом варианте при нахождении нового окна исследуется максимальная глубина устойчивости окна, то есть, сперва проверяется глубина устойчивости 1, на следующий запуск проверяется устойчивость этого же окна на глубину 2 и так далее, пока окно, либо не нарушит линеаризуемость, либо не закончится второй прерываемый поток и можно будет перейти к рассмотрению другого окна. Такая расстановка приоритетов проверки устойчивости окна гарантирует проверку всех окон,

однако зачастую неустойчивое окно, либо неустойчиво совсем (то есть глубина его устойчивости 0), либо имеет небольшую глубину устойчивости. Этот факт наводит на мысли об изменении приоритизации поиска неустойчивого окна. В моей реализации алгоритм исследует устойчивость окон по очереди. Сперва исследуется устойчивость всех окон на глубину 1, если ошибок не выявлено, то проверяется устойчивость всех окон на глубину 2 и т. д., пока, либо все окна не будут исследованы, либо не будут потрачены все попытки вызовов.

Предложенная реализация алгоритма построения расписаний для проверки линеаризуемости характеризуется гарантией над пространством прерываний и учитывает статистические особенности нарушения линеаризуемости.

6.2. Формальное описание работы алгоритма

Работу алгоритма можно разбить на 2 части.

- Внешнее управление работой – обеспечивает своевременную смену параметров расписания и прекращение работы алгоритма.
- Внутреннее управление работой – обеспечивает выполнение расписания и своевременное переключение потоков.

Представим, что у нас имеется циклический список параметров parametersL для каждого параметра есть множество непроверенных окон (nonCheckedWind Так как перед началом работы у нас нет информации об окнах, данное множество пусто. Каждому параметру расписания сопоставляется последнее окно, которое было обнаружено (lastWindow). Если не переключиться на нем, то переключений не будет совсем, то есть будет потрачен лишний запуск. Также, для каждого запуска имеется счетчик количества переключений (intereavingsCounter) и флаг (needChangeParameters), оповещающий внешнюю часть о необходимости смены параметров запуска. Наконец, каждому окну сопоставляется множество историй, которыми была проверена глубина устойчивости окна (passedPaths).

Тогда, формально алгоритм внешнего управления работой можно представить в следующем виде:

Algorithm 1 внешнее управление работой

```
1: while parametersList ≠ ∅ do
2:     if needChangeParameters then
3:         prepareForStart
4:     runAlgorithm
5:     if needChangeParameters && nonCheckedWindows ≠ ∅ then
6:         removeParameterFromList
```

Где prepareForStart подразумевает подготовку к запуску (сброс счетчика переключений и т.д.), runAlgorithm подразумевает под собой запуск программы в многопоточном режиме, а removeParameterFromList — удаление данного параметра из списка параметров запуска.

В момент исполнения программы планировщик осуществляет внутренее управление ходом исполнения путем обращения к методам, которые можно описать следующим образом:

Algorithm 2 логика работы при обращении к общей переменной

```
1: // не было прерываний
2: if interleavingsCounter = 0 then
3:     // окно новое или не исследовано
4:     if isNew(window) || window ∈ nonChecked then
5:         // добавляем окно в список неисследованных
6:         // и переключаем поток
7:         nonCheckedWindows ←put history
8:         switchThread
9:     // если уже переключались
10:    // и точка удовлетворяет SMT паттернам
11: else if isSMT(point) then
12:    switchThread
13: // В противном случае пропускаем
```

Algorithm 3 логика работы при завершении потока

```
1: if interleavingsCounter == 0 then
2:     // Если не было прерываний и поток должен был быть прерван
3:     if currentThread ∈ interleavingThreads then
4:         // Сообщаем наружу, что нужно сменить параметры
5:         needChangeParameters ← true
6:     setNextThread
7: if interleavingsCounter == 1
   && currentThread == interleavingThreads[1] then
8:     // Включаем прерванный поток
9:     setPreviousThread
10: if interleavingsCounter == 2 then
11:    // Включаем следующий поток в расписании
12:    setNextThread
```

7. Сравнение

После окончания основной части были проведены сравнения производительности различных стратегий. Для проведения данного исследования использовался ПК со следующими характеристиками:

- процессор Intel Core i5-3230M с 4 процессорными ядрами;
- объем оперативной памяти 8 GB;
- операционная система Ubuntu 16.04;
- архитектура процессора x64;
- виртуальная машина Java версии 1.8.0_101-b13.

Представленные ниже замеры производительности производились в 3 потоках, в каждом потоке не более 5 вызываемых методов, для каждого тестового алгоритма было задано ограничение в 5000 вызовов. Также была гарантирована детерминированность генерируемых тестовых алгоритмов.

Основным достоинством планировщика Enumeration является уменьшение количества вызовов, для чего была выбрана корректная структура данных. Так как доработка старого алгоритма не меняет своё поведение на корректных структурах данных, было решено сравнивать новый алгоритм (Enumeration) и доработанный (Stress). Для сравнения был выбран некоторый набор структур данных и составлен график, показывающий среднее количество вызовов, необходимое для проверки тестового алгоритма.

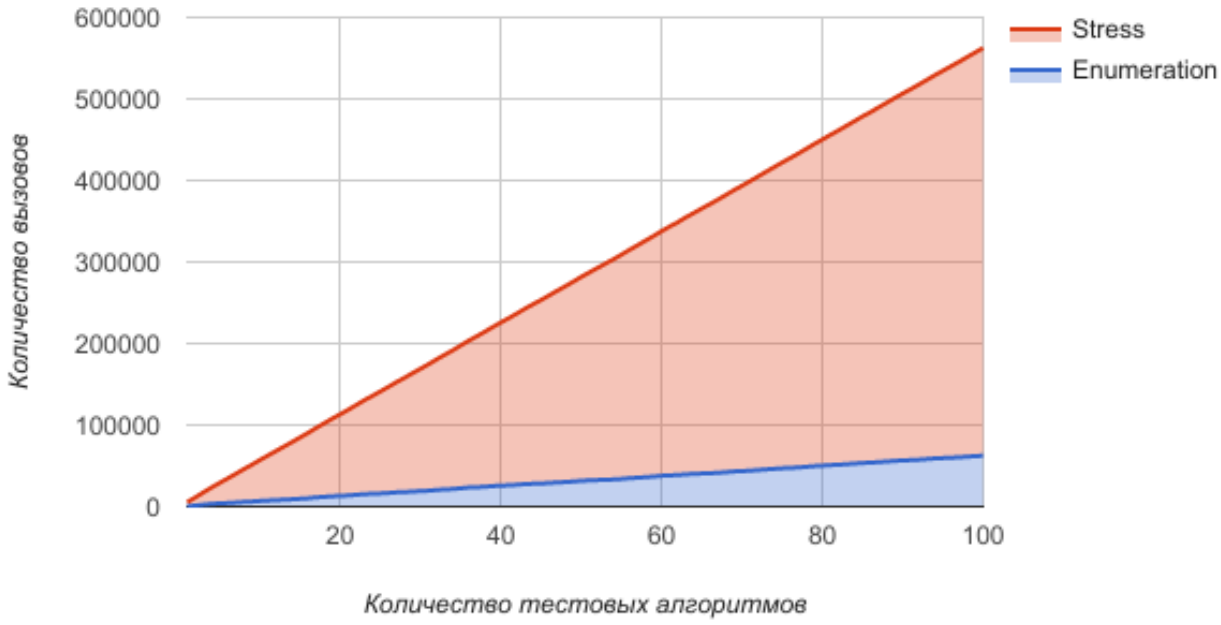

Рис. 4: График количества вызовов, необходимых для проверки линеаризуемости структуры.

Данный график иллюстрирует значительное уменьшение количества вызовов, необходимое для проверки тестового алгоритма, что потенциально уменьшает время, необходимое для проверки корректности структуры данных.

Разработанный алгоритм подразумевает один поток в рабочем состоянии в каждый момент времени. Можно считать, что многопоточный алгоритм становится однопоточным (в некоторой степени), что увеличивает время его работы. Более того, введение планировщика добавляет некоторые накладный расходы. Всё это влечет необходимость измерения времени, необходимого для проверки тестового алгоритма. Для сравнения времени исполнения был выбран набор структур данных и составлен график, показывающий среднее время, требуемое для проверку определенного количества тестовых алгоритмов. В данном сравнении были исследованы предыдущий (Old), доработанный (Upgraded)

и разработанный (Enumeration) методы проверки тестового алгоритма.

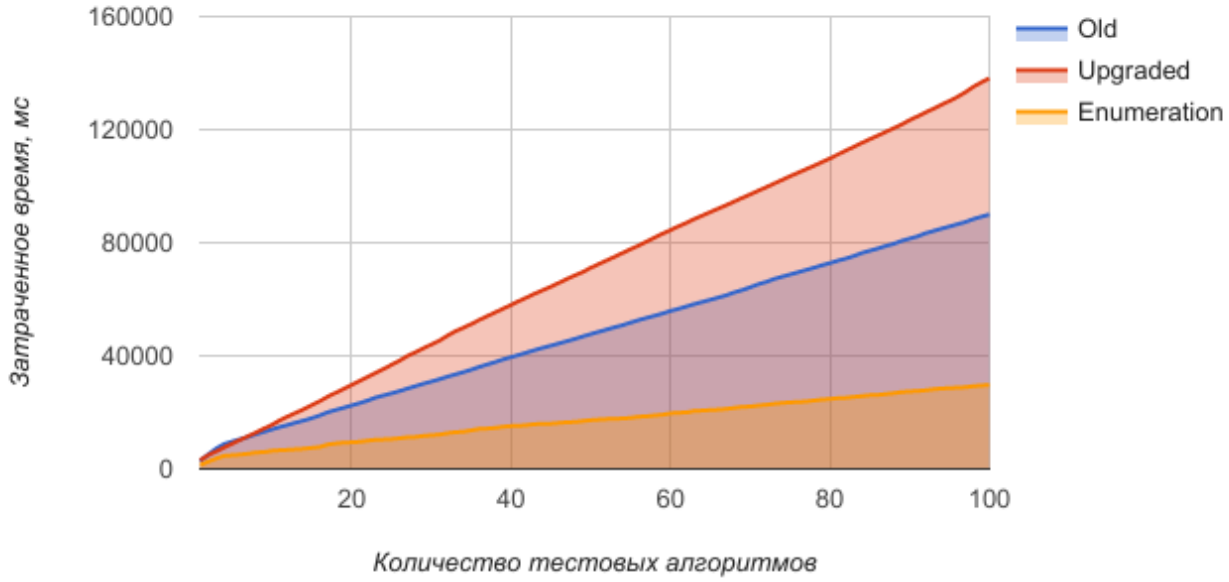

Рис. 5: График затраченного времени, необходимого для проверки линеаризуемости структуры.

Результаты, представленные на графике демонстрируют, что введение планировщика отрицательно сказалось на стресс-тестировании с точки зрения производительности, а также превосходство разработанного алгоритма управления исполнением.

Другим преимуществом планировщика Enumeration является его детерминированность (иными словами, если сгенерированный тестовый алгоритм содержит ошибку, то она будет найдена), однако стресс-тестирование не является детерминированным. В следствии чего, было решено исследовать линеаризуемость структур данных из библиотек с открытым исходным кодом, а также заведомо нелинеаризуемые структуры данных. Так как доработанный алгоритм стресс-тестирования учитывает внутреннее строение тестируемых структур данных, ожидается уменьшение числа вызовов, необходимое для нахождения ошибки.

Структура данных	Метод тестирования	Количество тестовых алгоритмов	Количество вызовов
List	Old Upgraded Enumeration	260 260 260	1 295 243 1 295 017 118 852
Queue	Old Upgraded Enumeration	356 356 217	1 777 046 1 775 588 116 027
Deque	Old Upgraded Enumeration	3085 1322 1322	15 421 064 6 610 320 853 456
WrapperQueue	Old Upgraded Enumeration	17 17 17	80 026 80 011 1 648

Таблица 1: Сравнение методов проверки линеаризуемости для некорректных структур данных.

В процессе сравнения было подтверждено, что планировщик Enumeration находит нелиниаризуемость тестового алгоритма, что не гарантирует стресс-тестирование. Также в процессе исследования были найдены ошибки в структурах данных библиотеки с открытым исходным кодом amino_cbbs [1].

[1]Исходный код доступен на $https://github.com/figoxu/aminocbbs$

8. Заключение

В ходе работы были достигнуты следующие результаты:

- проведен обзор методологий проверки линеаризуемости многопоточных программ;
- разработана архитектура для создания стратегий управления ходом исполнения программы;
- формализованы операции виртуальной машины Java, влияющие на поведение алгоритма в многопоточной среде ;
- доработан существующий алгоритм управления порядком исполнения с учетом формализованых операций виртуальной машины Java;
- разработан алгоритм управления порядком исполнения многопоточных программ на языке Java;
- сравнение полученных алгоритмов с текущей реализацией выявило увеличение производительности разработанного алгоритма.

Список литературы

[1] Emmi Michael, Qadeer Shaz, Rakamaric Zvonimir. Delay-bounded scheduling // POPL. — 2011.

[2] Flanagan C. Freund S. Efficient and precise dynamic race detection // Programming Languages: Design and Implementation / ACM. — 2009. — P. 121–133.

[3] Fraser Keir, Harris Timothy L. Concurrent programming without locks // ACM Trans. Comput. Syst. — 2007. — Vol. 25. — P. 5.

[4] Herlihy Maurice, Wing Jeannette M. Linearizability: A Correctness Condition for Concurrent Objects // ACM Trans. Program. Lang. Syst. — 1990. — Vol. 12. — P. 463–492.

[5] Musuvathi Madanlal, Qadeer Shaz, Ball Thomas. CHESS: A Systematic Testing Tool for Concurrent Software. — 2007.

[6] P. Godefroid. Model Checking for Programming Languages using Verisoft // POPL. — 1997.

[7] Package java.util.concurrent. — URL: `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html`.

[8] Papadimitriou Christos H. The serializability of concurrent database updates // J. ACM. — 1979. — Vol. 26. — P. 631–653.

[9] Parralel LINQ (PLNQ). — URL: `https://msdn.microsoft.com/ru-ru/library/dd460688(v=vs.110).aspx`.

[10] Qadeer S. Rehof J. Context-Bounded Model Checking of Concurrent Software // TACAS. — 2005.

[11] Sorrentino F. Farzan F. Madhusudan P. PENELOPE: Weaving Threads to Expose Atomicity Violations // FSE. — 2010. — P. 37–46.

[12] The Structure of the Java Virtual Machine, Oracle. — URL: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.11.1.

[13] Unit testing. — URL: https://en.wikipedia.org/wiki/Unit_testing.

[14] Vechev M. Yahav E. Yorsh G. Experience with model checking linearizability // SPIN. — 2009. — P. 261–278.

[15] Z. Rakamaric. STORM: static unit checking of concurrent programs // 2010 ACM/IEEE 32nd International Conference on Software Engineering. — 2010. — Vol. 2. — P. 519–520.

[16] jPredictor. — URL: http://fsl.cs.illinois.edu/index.php/JPredictor.

[17] jcstress. — URL: http://openjdk.java.net/projects/code-tools/jcstress/.

[18] Автоматическое тестирование линеаризуемости реализаций многопоточных структур данных.