Санкт-Петербургский национальный исследовательский университет
информационных технологий, механики и оптики

Кафедра компьютерных технологий

Роскошный Яков Игоревич

Применение статического анализа для оптимизации динамического поиска гонок.

Бакалаврская работа

Научный руководитель: Д. И. Цителов

Санкт-Петербург
2015

Содержание

Введение

В настоящее время все большее количество устройств становится многоядерными и многопроцессорными, и вместе с этим приходится разрабатывать эффективные параллельные программы. Но разработка программ с несколькими потоками выполнения, является сложной и влечет большое количество ошибок. Одной из таких ошибок является состояние гонки (data race, race condition) — несинхронизированные обращения из различных потоков к одному и тому же участку памяти, хотя бы одно из которых является обращением на запись. Состояние гонки в программах является частой допускаемой ошибкой и обнаружение данных ошибок является сложной и актуальной задачей.

Большинство современных языков, в том числе Java использую многопоточную модель с разделяемой памятью. Для публикации изменений, сделанных потоком, и для получения изменнений, сделанных другими потоками существуют операции синхронизации. Контракты модели памяти описывают гарантии, которые предоставляют различные операции синхронизации. Данные операции обеспечивают упорядоченность по времени между операциями. Если между операциями нет упорядочивания по времени, и хотя бы одна операция является записью, то наступает состояние гонки.

На 0.1 показан пример программы, с разделяемой памятью, в которой возникает состояние гонки.

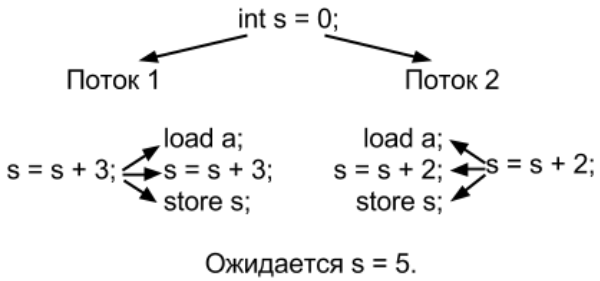

Рис. 0.1: Состояние гонки в программе

Возможно несколько вариантов исполнения данной программы, в том числе и корректный, при котором значение переменной s после исполнения двух потоков будет равно 5. Но ожидаемый результат не гарантирован. Возможный вариант исполнения программы:

1. Оба потока, загружают 0 в локальные переменные.
2. Увеличивают s параллельно.
3. Сохраняют, не важно в каком порядке значение s.

В результате в переменной s может оказаться 2 или 3. Также стоит отметить, что даже если все операции одного потока выполнятся раньше, чем первая операция другого, то корректный результат исполнения не гарантирован. Это связано с тем, что в данной программе нет никаких операций синхронизации, а следовательно нет никаких гарантий того, что изменения сделанные одним потоком будут видны другому потоку. То есть, после того как один из потоков исполнит операцию *store*, не гарантируется, что другой поток увидит изменения.

Гонки могут возникать в различных системах и наносить большой ущерб. Не синхронизировав должным образом программу, в любой финансовой системе может возникнуть проблема с балансами счетов. Простое зачисление или снятие может работать некорректно. Поймать гонку на

этапе тестирования очень сложно, поскольку обычно технические характеристики и архитектура устройств, на которых тестируется и запускается программа сильно различаются. Гонку трудно отыскать, даже когда она уже произошла. Если гонка случилась, то испорченные данные могут долго распространяться по программе, и несоответствие может обнаружиться совсем в другом месте. Повторный запуск программы на тех же данных может не привести к возникновению гонки. Таким образом, задача автоматического обнаружения гонок является сложной и актуальной.

В главе 1 будут более подробно разобраны подходы к автоматическому поиску гонок в программах. Пока, отметим, что существует два принципиально разных подхода к обнаружению гонок : статический и динамический. Статический подход анализирует программу без ее запуска. Динамический подход работает вместе с программой и анализирует конкретный вариант работы программы. Задача нахождения гонок статическим анализом является NP-трудной. Поэтому при статическом подходе снижается точность и полнота результата. Главной проблемой динамического подхода является наносимый ущерб производительности и потребления памяти анализируемой программы. Целью данной работы является оптимизация динамического детектора путем проведения предварительного статического анализа.

Статический анализ не наносит ущерб исполнению программы. С другой стороны, во время статического анализа можно выделить поля, при обращении к которым не может возникнуть состояние гонки(далее корректно-синхронизированные поля).

Динамический анализ использует достаточно большие структуры данных для полей и операций синхронизации. Информация о том, что часть полей можно не анализировать позволит уменьшить время анализа и объем потребляемой памяти, и следовательно уменьшить ущерб наносимый динамическим детектором.

В рамках данной работы будет разработан алгоритм для выделения корректно-синхронизированных полей и реализована соответствующая программа. Программа будет интегрирована с одним из существующих динамических детекторов.

В первой главе проведен обзор возможностей статического анализа. Также рассмотрены подходы к динамическому обнаружению гонок и возможности для их улучшения. Во второй главе приведен алгоритм поиска корректно-синхронизированных полей. В третьей главе рассмотрена программная реализация алгоритма и представлены полученные результаты.

Глава 1. Обзор

В первом разделе главы рассмотрены различные подходы к поиску гонок, оценены их возможности, преимущества и недостатки. Во втором модель памяти Java, динамический детектор для Java программ и возможности для его улучшения статическим анализом. Автоматические подходы к обнаружению гонок и сам динамический детектор более подробно описаны в [1]. В третьем разделе рассмотрены основные возможности и подходы статического анализа для решения исходной задачи.

1.1. Методы автоматического обнаружения гонок

1.1.1. Статический подход

Статический анализ требует только исходный код или скомпилированные файлы. Для проведения анализа не требуется запуск программы. Но задача обнаружения гонок статическим анализом является NP-трудной. Поэтому статически выявить гонки за приемлимое время невозможно. Существующие утилиты используют различные эвристики, уменьшают глубину анализа, что приводит к существенно неточным и неполным результатам, а также допускают ложные срабатывания. Подводя итог, отметим основные преимущества и недостатки статического подхода. К преимуществам относится то, что в отличии от динамического подхода, теоретически возможен анализ всех участков программы, а также то, что статический подход не требует запуска программы и не наносит ущерб ее выполнению. Главным и существенным недостатком является пропуск большого числа гонок, а также ложные срабатывания.

1.1.2. Динамический подход

Динамические детекторы выполняются одновременно с программой и отслеживают синхронизационные события и обращения к разделяемым переменным. Среди динамических детекторов выделяют 2 вида :

- on-the-fly - получают информацию и анализируют её во время выполнения программы.
- post-mortem - сохраняют информацию во время выполнения программы, а анализируют её уже после завершения работы программы.

Динамический анализ является неполным, так как анализируется только конкретный путь исполнения программы. Однако теоретически он гарантирует тончость, то есть отсутствие ложных срабатываний.

Для динамического анализа используются два принципиально различных алгоритма : *happens-before* и *lockset*, которые описаны в [1,2,3].

1.2. Динамический детектор гонок для java программ

1.2.1. Модель памяти java

Для языка программирования java существует спецификация его модели памяти(Java memory model), которая входит в стандарт языка. Данная спецификация содержит архитектурно-независимые гарантии исполнения многопоточных программ. Для загрузки изменений из памяти потока в общую память программы, а также для загрузки чужих изменений из общей памяти программы в память потока существуют операции синхронизации. В JMM описано отношение *happens-before*. Если операция A *happens-before* B, то при выполнении операции B видны изменения, выполненые A. Если операции A и B происходят из разных потоков

и обращаются к одному участку памяти, то они не образуют гонку тогда и только тогда, когда A $happens\text{-}before$ B либо B $happens\text{-}before$ A.

1.2.2. Динамический детектор jDRD

jDRD — динамический детектор для java программ. Он основан на $happens\text{-}before$ алгоритме.

Рассмотрим подробнее $happens\text{-}before$ алгоритм, чтобы выявить места для ускорения jDRD.

Для каждого потока t будут храниться векторные часы $t.vc$. Также часы будут храниться для всех разделяемых переменных $v.vc$ и синхронизационных объектов $l.vc$. Векторные часы являются массивом целых чисел, каждая компонента которого является целым числом, отвечающим за компоненту часов соответсвующего потока. Векторные часы имеют длину равную общему количеству потоков. Каждый поток хранит свою локальную копию векторных часов, синхронизируясь с копиями часов других потоков во время синхронизационных операций. Сравнение часов происходит при обращениях к разделяемых переменных.
Изначально:

- $\forall i:\ t_i.vc[i]\ :=\ 1$
- $\forall i,\ j\ \neq i:\ t_i.vc[j] := 0$
- $\forall v,\ j:\ v.vc[j]\ :=\ 0$
- $\forall l,\ j:\ l.vc[j]\ :=\ 0$

При захвате потоком t синхранизационного объекта l:

- $\forall j:\ t.vc[j] = max(t.vc[j]\ , l.vc[j])$

При освобождении потоком t_i синхранизационного объекта l:

- $t_i.vc[i]{+}{+}$
- $\forall j:\ l.vc[j] = max(l.vc[j]\ , t_i.vc[j])$

При обращении потока t к разделяемой переменной v:

- Если $\exists j : \ v.vc[j] \ > \ t.vc[j]$, то значит найдена гонка.
- $v.vc = t.vc$

В случае java программ данными разделяемыми переменными являются поля.

1.2.3. Возможности статического анализа для оптимизации jDRD

Как видно из описания работы jDRD, каждое поле является потенциальным местом возникновения гонки. Для каждого поля в jDRD приходится хранить векторные часы. Статическим анализом можно выяснить, что некоторые поля — - то есть, при обращении к ним невозможно состояние гонки. Это позволит уменьшить потребление памяти и времени детектора.

1.3. Статический анализ для поиска корректно-синхронизированных полей

1.3.1. Существующие утилиты для статического анализа

В настоящее время существует достаточное количество утилит, которые производят статический анализ. Немногие из них, такие как $FindBugs$ и $ThreadSafe$ ориентированы на анализ конкурентного доступа к данным. Но все эти утилиты ориентированы на поиск ошибок в программах и для решения исходной задачи не подходят. Существуют утилиты, которые статическим анализом позволяют получать различные представления исходного кода для дальнейшего анализа. Такие утилиты не могут решить исходную задачу, но могут быть использованы, как вспомогательные в данной работе. Таким образом, задача поиска корректно-

сихронизированных полей статическим анализом актуальна, и в открытом доступе нет утилиты, позволяющей решать данную задачу.

1.3.2. Возможные подходы к анализу

Основным объектом исследования при статическом анализе программ является граф потока исполнения выполнения.

Граф потока исполнения (англ. control flow graph, CFG) — все возможные пути исполнения части программы, представленые в виде графа. Вершинами данного графа являются последовательности операций, не содержащие в себе ни операций передачи управления, ни точек, на которые управление передается из других частей программы. Ребра показывают возможные переходы между операциями. Построение данных графов зависит от представления программы, которое используется. Для java программ изначально доступно представление в виде байт-кода. Если доступен исходный код, то можно проводить анализ самой java программы. Данные представления неудобны для последующего анализа. Байт-код имеет большое количество инструкций. Java код имеет большое количество синтаксических конструкций, все из которых трудно проанализировать. Поэтому, часто оказываются удобными для анализа промежуточные представления.

Промежуточное представление (англ. Intermediate representation, IR) — язык абстрактной машины, упрощающий проведение анализа. Для java программ большинство промежуточных представлений строится на основе байт-кода.

В данной работе будут искаться поля, обращения к которым всегда защищены блокировкой. В качестве блокировок будут рассмотрены стандартные операции захвата и освобождения монитора, а также блокировки пакета java.util.concurrent.

Не рассмотренным в данной работе останется другой возможный подход к выделению корректно-синхронизированных полей. Статическим

анализом можно пытаться доказать, что переменная не покидает контекст одного потока. То есть, если можно статически доказать, что все обращения к полю происходят только в рамках одного потока, то данное поле корректно-синхронизировано.

Глава 2. Описание алгоритма

В данной главе будет рассмотрен алгоритм для поиска корректно-синхронизированных полей. Поле является корректно-синхронизированным, когда существует блокировка, такая, что любая операция с полем проводится с этой блокировкой. Алгоритм будет искать для полей, такие блокировки. Алгоритм можно разделить на 3 части.

1. Получение промежуточного представления и графа потока исполнения.
2. Получение множества возможных блокировок.
3. Обход графа потока исполнения и выделение корректно-синхронизированных полей.

В первой главе будет рассмотренно промежуточное представление, на основе которого будет строиться граф потока исполнения. Во второй показано как выделить множество переменных, которые могут являться блокировкой. Эта часть алгоритма необходима, так как при статическом анализе не каждая локальная переменная и не каждое поле может являться потенциальной блокировкой. В третей части будет рассмотрена основная часть алгоритма — обход графа потока исполнения, отслеживание и обработка и операций сихронизаций и обращений к полям, выделение корректно-синхронизированных полей.

2.1. Описание промежуточного представления и CFG для него

Как уже говорилось ранее, существуют утилиты для получения различных промежуточных представлений и их графов потоков исполнения. Для данной работы было выбрано промежуточное представление приве-

денное в SSA форму. SSA форма технически упрощает получение множества блокировок. В промежуточном представлении используемом в данной работе, вся работа со стеком заменена на локальные переменные. Как и в байт-коде будет две примитивные операции синхронизации: $monitorEnter$ и $monitorExit$. Эти операции отвечают за взятие и освобождение монитора объекта.

Каждая переменная имеет единственное место присваивания, так как представление удовлетворяет SSA форме. Для присваивания переменной может использоваться ϕ-функция. Если локальная переменная может принимать несколько значений, то данная переменная присваивается ϕ-функции из всех возможных ее значений. Подробнее о SSA форме и ϕ-функциях можно прочитать в [2]. Присваивания в данном представлении удоветворяют следующей грамматике($local$ — локальная переменная, $field$ — поле класса, $constant$ – константа).

imm	$local$ $constant$
expr	imm_1 $binop$ imm_2 (type) imm imm_1 instanceof type invokeExpr new refType $newarray$ (type) [imm] neg imm
assignStmt	$local = \phi(imm_1, imm_2, ...)$ $local = imm \mid field \mid local.field \mid expr$ $field = imm$ $local.field = imm$

Таблица 2.1: Грамматика присваиваний используемого IR

Также в данном языке, выделены операции инициализации. Они используются для того чтобы получить ссылку на $this$, на возникший $exception$ или на аргумент функции.

Далее будет приведен простейший пример метода, полученное промежуточное представление и его CFG.

```
public void a() {
    int x = 5;
    int y = 0;
        while (x > 5) {
            y += x - 1;
        }
        System.out.println(y);
}

public void a()     {
        r0 := @this;
        b0 = 5;
        i1 = 0;
     label1:
        i1_1 = Phi(i1 #0, i1_2 #1);
        if b0 <= 5 goto label2;
        $i2 = b0 - 1;
        i1_2 = i1_1 + $i2;
        goto label1;
     label2:
        $r1 = java.lang.System.out;
        $r1.println(i1_1);
        return;
}
```

Листинг 2.1: Описанное промежуточном представлении

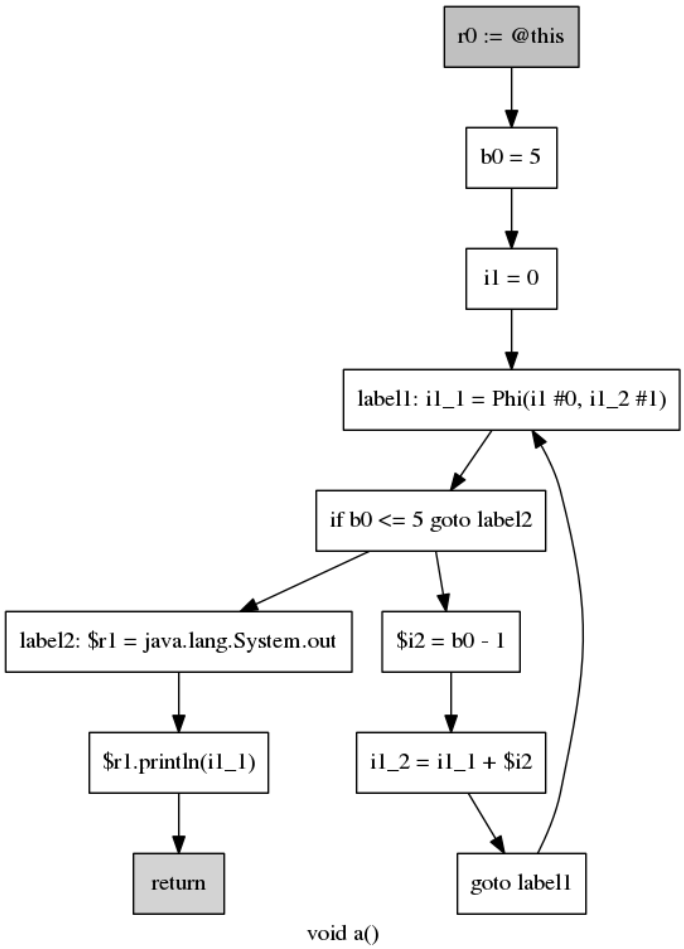

Рис. 2.1: CFG метода

2.2. Получение множества возможных блокировок

Алгоритм должен быть точным, то есть сообщать только о тех полях, которые действительно корректно-синхронизированы. В полученном в предыдущем разделе представлении операция взятия блокировки осуществляется с локальной переменной. При обходе графа, который будет рассмотрен в следующей главе нужно поддерживать текущее множество взятых блокировок. Если нельзя статически доказать, что локальная переменная всегда ссылается на одно поле, или переменная ссылается на поле, которе может изменится, то операции взятия блокировки по таким пере-

менным не должна добавлять информацию относительно взятых блокировок. В листинге 2.2 демонстрируется соответсвующий пример.

```
Class A {
    final Object lock1 = new Object();
    final Object lock2 = new Object();
    void a() {
        Object lock;
        if (System.currentTimeMilis() % 10000 == 0) {
            lock = lock1;
        } else {
            lock = lock2;
        }
        syncronized(lock) {
            // some code
        }
    }
}
```

Листинг 2.2: Блокировка с несколькими возможными значениями

В данном примере переменная $lock$ может ссылаться на поле $lock1$ и $lock2$. И куда будет ссылаться $lock$ статическим анализом выяснить нельзя. Таким образом, операция синхронизации не должна изменять множество взятых блокировок.

2.2.1. Нахождение переменных, которые могут ссылаться на разные поля

В данном разделе будет показано, как найти все локальные переменные, которые могут иметь не единственное значение. Так как полученое представление является SSA, то каждая переменная имеет единственное место инициализации. Если переменная инициализируется как ϕ-функция, то значит эта переменная может иметь не единственное значеие. Также если локальная переменная $l1$ присваивается другой локальной переменной $l2$, или полю локальной переменной $l2.field$, и $l2$ может иметь не единственное значение, то $l1$ может иметь не единственно значение и не может

являться блокировкой.

Алгоритм будет следующим. Сначала выделим множество переменных, которые инициализуются ϕ-функциями. Затем построим замыкание данного множества относительно операций присваивания. Полученное множество, очевидно будет искомым.

2.2.2. Нахождение переменных, ссылающихся на не final поля

```
public class A {
    final B b = new B();
    public void a() {
        synchronized (b.lock) {
            /*some code*/
        }
    }
}
```

Листинг 2.3: Блокировка по полю другого класса

В листинге 2.3 показан пример, когда берется блокировка по полю другого класса. Чтобы являться блокировкой, поле b класса A должно иметь модификатор $final$ и поле $lock$ класса B также должно быть $final$. В общем случае каждое поле в пути блокировки должно иметь модификатор $final$. Отметим, что данные рассуждения относятся к не *static* полям. В случае со *static* полями достаточно проверить, что поле имеет модификатор $final$.

Алгоритм будет следующим. Сначала выделим множество переменных, которые ссылаются на не $final$ поля. Затем построим замыкание данного множества относительно операций присваивания. Полученное множество, очевидно будет искомым.

2.3. Обход графа потока исполнения

В данной главе будет рассмотрен алгоритм обхода CFG каждого метода. Описан сам обход, обработка операций сихронизации и обращения к полям.

Обход будет рекурсивным, напоминающий обход в глубину, но с некоторыми отличиями. При обходе будем поддерживать множество текущих взятых блокировок $curLocks$. Также для каждой вершины CFG будем сохранять множество блокировок, с которыми обход уже посещал данную вершину $v.locks$. При входе в вершину v нужно сравнить $curLocks$ и $v.locks$. Если $v.locks \subseteq curLocks$, то можно не продолжать обход вершины v. Если $v.locks \nsubseteq curLocks$, то в $v.locks$ и $curLocks$ запишем $v.locks \cap curLocks$ и продолжим обход. Записывать нужно пересечение, так как если существует ветка обхода, в которой вершина v посещена без блокировки l, то нельзя гарантировать, что операция вершины v защищена блокировкой l.

Изменять $curLocks$ будем при операциях взятия и освобождения блокировки. Для каждого поля будем хранить, множество блокировок, с которыми обращались к данному полю $f.locks$. При обращениии к полю f, будем также пересекать $f.locks$ и $currentLocks$. Далее приведен псевдокод описанного алгоритма.

Алгоритм 1 Алгоритм обхода CFG метода

```
function VISIT(CFGVertex v, Set<Lock> currentLocks)
    if v.locks ⊆ currentLocks then
        break
    else
        v.locks ← v.locks ∩ currentLocks
        currentLocks ← v.locks ∩ currentLocks
    op ← v.getOperation()
    if op.isMonitorEnterOperation() then
        currentLocks.add(v.getOperations.getLock())
    if op.isMonitorExitOperation() then
        currentLocks.remove(v.getOperations.getLock())
    if op.isFieldAssignmentOperation() then
        field ← op.getField()
        field.locks ← field.locks ∩ currentLocks
    for all CFGVertex c : v.childs do
        VISIT(c, currentLocks)
end function
```

Далее будет более подробно рассмотренна обработка операций синхронизации и обращения к полям.

2.3.1. Обработка операций синхронизации

Если при обходе встретилась операция синхронизации, то нужно изменить $currentLocks$. Но сначала нужно проверить, что переменная, над которой осуществляется операция синхронизации может являться блокировкой. Данная проверка описана в предыдущем разделе. Далее, если текущая операция — операция взятия блокировки, то добавить блокировку в $currentLocks$, если операция освобождения, то удалить блокировку из $currentLocks$. Помимо стандартных операций $monitorEnter$ и $monitorExit$ в данной работе рассмотрены блокировки пакета java.util.concurent и их парные операции $lock()$ и $unlock()$.

2.3.2. Обработка операции обращения к полям

При обращении к полю может возникнуть состояние гонки. Но, если существует блокировка l, такая что любая операция чтения и записи с полем v производится со взятой l, то поле v корректно-синхронизировано. Таким образом, для каждого поля f надо поддерживать множество блокировок $f.locks$, с которыми гарантировано обращались к данному полю. А при обращении к полю f сужать $f.locks$ до пересечения $f.locks$ и $currentLocks$.

2.3.3. Обработка методов, защищенных блокировкой

Пока в работе рассматривался обход CFG каждого метода независимо. Но существуют методы, вызовы которых всегда защищены определенной блокировкой. И соответсвенно, любая операция в данном методе защищена этой блокировкой. Анализ методов может добавить информации относительно текущих блокировок, что приведет к увеличению найденных корректно-синхронизированных полей. Предварительно выделив для метода множество блокировок, с которым он гарантировано вызывается, можно улучшить анализ.

```
public class A {
    private Integer x;
    public Integer getX() {
        return x;
    }

    public void setX(Integer x) {
        this.x = x;
    }
}
```

Листинг 2.4: Класс без внутренней синхронизации

На листинге 2.4 показан пример класса, в котором отсутствует синхронизация. Однако, если все вызовы $getX()$ и $setX()$ защищены одной блокировкой, то все обращения к полю x защищены этой же блокировкой. Таким образом, получим, что поле x корректно-синхронизирована.

Рассмотрим алгоритм поиска блокировок, которыми защищен метод. Метод m *защищен блокировкой* l, если любая операция вызова метода m защищена блокировкой l. Для поиска блокировок, которыми защищен метода можно использовать алгоритм аналогичный алгоритму для поиска корректно-синхронизированных полей. Для каждого метода m надо поддерживать множество блокировок $m.locks$, с которыми гарантировано вызывался данный метод. При операции вызова метода m записывать в $m.locks$ пересечение $currentLocks$ и $m.locks$.

После одного обхода для каждого метода m будет сформирован $m.locks$. Далее можно повторить обход графа с появившимися новыми блокировками. Второй и последующие обходы будут полезны, так как после каждого обхода множество блокировок, которые защищают метод может увеличиться. Если после очередного обхода в множество блокировок, защищающих метод m, добавлена блокировка l, то все операции метода m защищены l. Это означает, что при следующем обходе, любой метод k, вызываемый из m может стать защищеным блокировкой l. Когда останавливать данные обходы? Если после очередного обхода для каждого метода m

не изменилось $m.locks$, то можно завершать алгоритм. Теоретически может понадобиться $countMethods * countLocks$ обходов. На практике нескольких (трех или четырех) таких обхода достаточно, так как операции синхронизации редко используются для того, чтобы синхронизировать операции через пять вызовов метода.

Глава 3. Практическая реализация алгоритма

В рамках данной работы была написана утилита для поиска корректно-синхронизированных полей, использующая алгоритм из предыдущего раздела. Программа написана на языке программирования java и интегрирована с jDRD.

3.1. Особенности реализации

Данная программа имеет два режима работы: *локальный* и *локальный. Локальный* режим подразумевает, что анализируемый код может использоваться строронними приложениями, не входящими в область анализа. Таким образом, если поле корректно-синхронизированно в рамках анализируемой программы, но оно доступно для изменения, то в локальном режиме оно не считается корректно-сихронизированным. Такой режим нужен для анализа различных библиотек. *Глобальный* режим подразумевает, что анализируемый код, никем используется. Если поле корректно-синхронизированно в рамках анализируемой программы, то даже если оно доступно для изменения, глобальный режим отметит отметит его как корректно-синхронизированное. Данный режим необходим для тестирования законченых приложений. Множество переменных, выделенных в глобальном режиме включает в себя множество, выделенное при работе в локальном режиме.

В листинге 3.1 поле *balance*, корректно-синхронизирован и будет выделено при работе программы и в локальном и в глобальном режиме. Если убрать модификатор *private* у поля *balance* и предположить, что в рамках программы снаружи к полю *balance* не обращаются, то оно будет корректно-синхронизировано только с точки зрения глобального режима.

```
public class Account {
    private Integer balance = 0;
    synchronized void incrementBalance(Integer value){
        balance += value;
    }

    synchronized Integer getBalance() {
        return balance;
    }
}
```

Листинг 3.1: Пример корректно-синхронизированного поля для локального режима

Программа на вход принимает скомпилированные фалйы java программ. На вход ей можно подать либо *jar*-файл с программой либо указать путь до папки с *class*-файлами. Промежутончое представление и граф потока исполнения получено на основе байт-кода с помощью библиотеки *Soot*. Промежуточное представление используемое в данной работе для анализа называется *Shimple*.

3.2. Тестирование

Было проведено тестирование данной программы на различных тестах. Создан набор тестов, покрывающий большинство конструкций java программ. В качестве тестов были использованы программы использующие различные операции синхронизации, обработку ошибок(exception), статические и нестатические блокировки и поля, внутренние классы и т.д.

Также были проведены запуски на реальных библиотеках и приложениях с последующей проверкой результатов.

3.3. Сбор статистики и интеграция с jDRD

В программу статического анализа был добавлен модуль сбора статистики, который подсчитывает различные метрики работы алгоритма. Этот модуль необходим для оценивания результата работы программы.

jDRD получает список корректно-синхронизированных полей через конфигурационный файл, генерируемый разработанной программой. На стороне jDRD был также включен сбор статистики, который отслеживает количество обращений к корректно-синхронизированным полям, выделенным статическим анализом.

3.4. Полученные результаты

Программа была запущена на различных приложениях, которые активно используют конкурентный доступ к данным . Результаты приведены в таблице.

Приложение	Общее количество полей	Корректно-синхронизированных полей	Процент
dxFeed	5730	493	8,6
MARS	4437	340	7,6
Tomcat	7600	390	5,1

Таблица 3.1: Полученные результаты

Далее представлено краткое описание тестируемых библиотек.
MARS(Monitoring and reporting system) — Система мониторинга реального времени. Используется для отображения различных данных приложения.
dxFeed — система, отвечающая за быструю доставку больших данных(котировок).
Tomcat — контейнер сервлетов. Позволяет запускать веб-приложения.

По результатам видно, что для данных библиотек в среднем статическим анализом можно обнаружить около

Далее приведены найденные часто используемые паттерны, применяемые для защиты блокировкой операций блокировкой.

```
public class A {
    private Object o;
    public synchronized void a() {
        //operations with o
    }
    public synchronized void b() {
        //operations with o
    }
}
```

Листинг 3.2: Использование syncronized методов

```
public class A {
    private static Object o;
    private final static Object LOCK = new Object();
    public void a() {
        synchronized (LOCK) {
            //operations with o
        }
    }
    public synchronized void b() {
        synchronized (LOCK) {
            //operations with o
        }
    }
}
```

Листинг 3.3: Использование блокировки для синхронизации

```
public class A {
    private static Object o;
    private final Lock lock = new ReentrantLock();
    public void a() {
        try {
            lock.lock();
            //operations with o
        } finally {
            lock.unlock();
        }

    }
}
```

Листинг 3.4: Использование блокировки для синхронизации

Заключение

В данной работе был получен алгоритм для поиска корректно-синхронизированных полей. Также была разработана программа, реализующая данный алгоритм. Корректность работы программы была протестирована большим набором тестов.

Одной из целей данной работы являлась оптимизация динамического поиска гонок. Программа была и интегрирована с существующим детектором jDRD.

Источники

[1] В.Ю Трифанов. Динамическое обнаружение состояний гонки в многопоточных Java-программах // Мир. 2010. С. 20–30.