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, используют многопоточную модель с разделяемой памятью. Для публикации изменений, сделанных потоком, и для получения изменений, 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 рис. 1 показан пример программы с разделяемой памятью, в которой возникает состояние гонки.

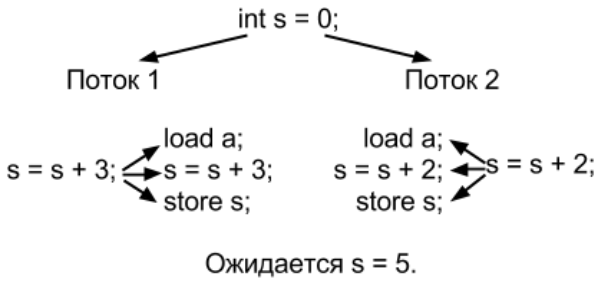

Рис. 1 Состояние гонки в программе.

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 значение 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. 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. При отсутствии правильной синхронизации балансов счетов простое зачисление или снятие средств 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 более подробно разобраны подходы к автоматическому поиску гонок в программах. 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 снижение производительности анализируемой программы. Целью данной работы является оптимизация динамического поиска гонок путем проведения предварительного статического анализа.

Статический анализ не использует вычислительные ресурсы во время выполнения программы. С другой стороны, статическим анализом можно выделить поля, при обращении к которым не может возникнуть состояние гонки (далее корректно-синхронизированные поля).

Динамические детекторы используют 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 детектором памяти.

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

ческих детекторов.

В первой главе проведен обзор возможностей статического анализа. Также рассмотрены динамические подходы к 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. Возможности статического анализа для оптимизации поиска гонок

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 оптимизации 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

Для статического анализа требуется только исходный код или скомпилированные файлы. Проведение анализа не требует запуска программы. Но задача обнаружения гонок статическим анализом является NP-трудной [2, 3]. Поэтому статически выявить гонки за приемлемое время невозможно. Существующие реализации используют различные эвристики, уменьшают глубину анализа, что приводит к существенно неточным и неполным результатам. Одной из наиболее эффективных существующих реализаций является Chord [4, 5]. В данной работе проводится анализ пар обращений к объектам. В результате выделяются пары обращений, при которых возможно возникновение гонки. Также существуют инструменты, которые проверяют, что программа удовлетворяет некоторой модели, на основании чего делается вывод, что она свободна от гонок. Примером такой модели является расширенная система типов, рассмотренная в работе [6]. При

данном подходе приходится проводить много ручной работы — расставлять типовые аннотации. С другими существующими подходами и реализациями можно ознакомиться в [7, 8].

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 использует ресурсы во время выполнения программы. 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 данным. Среди динамических детекторов выделяют 2 вида:

- on-the-fly — получают информацию и анализируют её во время выполнения программы [9];
- post-mortem — сохраняют информацию во время выполнения программы, а анализируют её уже после завершения работы программы [10].

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

Для динамического анализа используются два принципиально различных алгоритма: *happens-before* [1, 11] и *lockset* [1, 12, 13]. Также существуют гибридные алгоритмы [1, 9], которые совмещают преимущества *happens-before* и *lockset* алгоритмов.

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, JMM), которая входит в стандарт языка. Данная спецификация содержит архитектурно-независимые гарантии исполнения многопоточных программ. Для гарантированной загрузки изменений из памяти потока в общую память программы, а также для загрузки чужих изменений из общей памяти программы в память потока предусмотрены операции синхронизации. В JMM описано отношение *happens-before*. Если операция A *happens-before* B, то в момент начала выполнения операции B видны изменения, выполненные A. Если операции A и B выполняются из разных потоков и обращаются к одному участку памяти, то они не образуют гонку тогда и только тогда, когда A *happens-before* B либо B *happens-before* A [14].

1.2.2. Динамический детектор jDRD

jDRD — динамический детектор гонок для Java-программ. Он основан на *happens-before* алгоритме [1].

Рассмотрим подробнее *happens-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 соответствующего потока. В общем случае векторные часы имеют длину, равную общему количеству потоков. Каждый поток хранит свою локальную копию векторных часов, синхронизируясь с копиями ча-

сов других потоков во время выполнения синхронизационных операций. Сравнение часов происходит при обращениях к разделяемым переменным. 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 синхронизационного объекта l:

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

При освобождении потоком t_i синхронизационного 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. Применяя статический анализ можно выяснить, что некоторые поля корректно-синхронизированы — то есть, при обращении к ним невозможно состояние гонки. Для таких полей можно не анализировать обращения и не хранить векторные часы. Это позволит ускорить динамический детектор и уменьшить его потребление памяти.

1.3. Статический анализ для поиска корректно-синхронизированных полей

1.3.1. Существующие возможности статического анализа

В настоящее время статический анализ рассмотрен довольно широко. В разделе 1.1.1 были рассмотрены статические детекторы гонок. Существует достаточное количество программ, которые проводят статический анализ для различных целей. Немногие из них, такие как $FindBugs$ [15] и $ThreadSafe$ [16], проводят анализ конкурентного доступа к данным. Но все эти реализации ориентированы на поиск ошибок в программах и для решения исходной задачи не подходят.

В работе [17] приведен способ для нахождения объектов, которые не покидают контекст потока или метода. Результаты данной работы используются для выделения объектов на стеке вместо кучи, а также для ликвидации ненужных операций синхронизации. Если объект используется только в методе, в котором создается, то этот объект можно выделить на стеке. Если объект используется только в одном потоке, то при обращении к этому объекту можно избавиться от операций синхронизации. Целью работы являлась оптимизация компилятора, однако метод анализа, рассмотренный в данной работе представляет интерес для решения исходной задачи.

Существуют библиотеки, которые статическим анализом позволяют получать различные представления исходного кода для дальнейшего анализа. Такие программы не решают исходную задачу, но могут быть использованы, как вспомогательные в данной работе. Таким образом, задача поиска корректно-синхронизированных полей статическим анализом актуальна.

1.3.2. Возможные подходы к анализу

Объектами исследования при статическом анализе программ среди прочего являются представления исследуемой программы в виде графов (граф потока исполнения, граф использования объектов, *point-to* граф [17]). В данной работе проводится анализ графа потока исполнения.

Граф потока исполнения (англ. control flow graph, CFG) — это все возможные пути исполнения части программы, представленные 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 [18]. 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, которые 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 выделены поля, обращения к которым всегда защищены блокировкой. В качестве блокировок рассмотрены стандартные операции захвата и освобождения монитора, а также блокировки пакета java.util.concurrent.

Глава 2. Описание алгоритма

В данной главе рассмотрен алгоритм для поиска корректно-синхронизированных полей. Поле является корректно-синхронизированным, когда существует такая блокировка, что любая операция с полем проводится с этой блокировкой. Алгоритм ищет 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 первом разделе рассмотрено промежуточное представление, на основе которого построен 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 третьей части рассмотрена основная часть алгоритма — обход графа потока исполнения, отслеживание и обработка операций синхронизации и обращений к полям, выделение корректно-синхронизированных полей.

2.1. Описание промежуточного представления и CFG для него

В настоящее время существует достаточное количество готовых библиотек для получения различных промежуточных представлений и графов потока исполнения. Для данной работы было выбрано промежуточное представление приведенное в SSA-форму [19]. 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 имеется 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. Промежуточное представление и CFG получены с помощью библиотеки $Soot$ [20]. Присваивания в данном представлении удовлетворяют 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, его промежуточное представление и CFG.

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

Листинг 2.1: Пример Java-программы.

```
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 2.2: Описанное промежуточное представление.

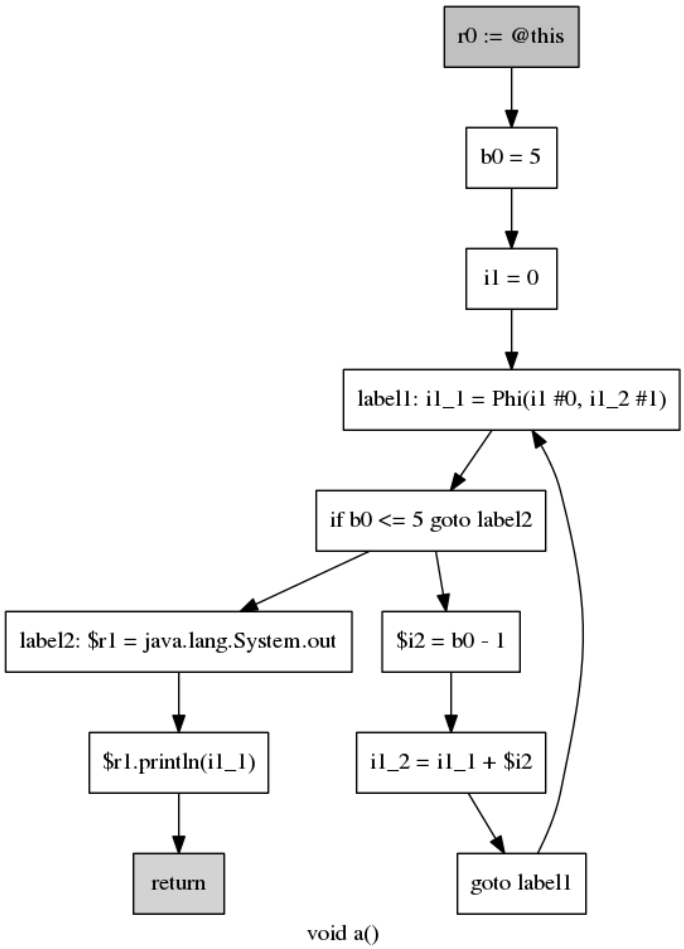

Рис. 2.1: Схема CFG метода.

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, которое может измениться, то операции взятия блокировки по таким

переменным не должна добавлять информацию относительно взятых блокировок. В листинге 2.3 демонстрируется соответствующий 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 2.3: Блокировка с несколькими возможными значениями.

В данном примере переменная $lock$ может ссылаться на поле $lock1$ и $lock2$. Статическим анализом нельзя выяснить значение переменной $lock$. 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 полученное представление является SSA, то каждая переменная имеет единственное место инициализации. Если переменная инициализируется как ϕ-функция, то эта переменная может иметь не единственное значение. Также если локальная переменная $l1$ принимает значение другой локальной переменной $l2$ или поля локальной переменной $l2.field$, и $l2$ может иметь не единственное значение, то $l1$ тоже может иметь не единственное значение и не может

являться блокировкой.

Рассмотрим алгоритм. Сначала выделим множество переменных, которые инициализируются 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 2.4: Блокировка по полю другого класса.

В листинге 2.4 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 непосредственно 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. Описан обход, обработка операций синхронизации и обращений к полям.

Обход является рекурсивным, напоминает обход в глубину, но с некоторыми отличиями. При обходе поддерживается множество текущих взятых блокировок $curLocks$. Также для каждой вершины CFG хранится 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 $curLocks$ нужно при операциях взятия и освобождения блокировки. Для каждого поля сохраним множество блокировок, с которыми обращались к данному полю $f.locks$. При обращении к полю f нужно пересечь $f.locks$ 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 более подробно рассмотрена обработка операций синхронизации и обращений 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 добавляем блокировку в $currentLocks$; а если операция освобождения, то удаляем блокировку из $currentLocks$. Помимо стандартных операций $monitorEnter$ и $monitorExit$ в данной работе рассмотрены блокировки пакета java.util.concurrent и их парные операции $lock()$ и $unlock()$.

2.3.2. Обработка обращений 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. Соответственно, 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 2.5: Класс без внутренней синхронизации.

На листинге 2.5 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-синхронизировано.

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 метод, 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 нужны, 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 защищенным блокировкой l. Оценим количество таких обходов. 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 обходов 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 программа 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 *глобальный*. *Локальный* режим подразумевает, что анализируемый код может использоваться сторонними приложениями, не входящими в область анализа. Таким образом, если поле или метод корректно-синхронизировано в рамках анализируемой программы, но оно доступно для изменения, то в локальном режиме оно не считается корректно-синхронизированным. Такой режим нужен для анализа различных библиотек. *Глобальный* режим подразумевает, что анализируемый код используется только из области анализа. Если поле корректно-синхронизировано в рамках анализируемой программы, то даже если оно доступно для изменения, глобальный режим отметит его как корректно-синхронизированное. Данный режим предназначен для тестирования законченных приложений. Множество переменных, выделенное в глобальном режиме включает в себя множество, выделенное при работе в локальном режиме.

В листинге 3.1 поле *balance* корректно-синхронизировано, 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 файлы Java-программ. На вход ей можно подать либо *jar*-файл с программой, либо указать путь до папки с *class*-файлами. Промежуточное представление и граф потока исполнения получены на основе байт-кода с помощью библиотеки *Soot* [21]. Промежуточное представление, используемое в данной работе для анализа, называется *Shimple* [20].

3.2. Тестирование

Было проведено тестирование разработанной 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 оценки результата работы программы.

jDRD получает список корректно-синхронизированных полей через конфигурационный файл, генерируемый разработанной программой. В 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. 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.

jtt — система менеджмента времени, использующая в качестве сервера JIRA.

Результаты приведены в таблице.

Приложение	Общее количество полей	Корректно-синхронизированных полей	Процент
dxFeed	5730	493	8
MARS	2753	418	15
Tomcat	7600	1826	24
jtt	1684	127	8

Таблица 3.1: Полученные результаты.

Полученные данные верны с теоретической точки зрения. С практической точки зрения важны найденные не final поля.

Приложение	Общее количество полей	Корректно-синхронизированных полей	Процент
dxFeed	1597	60	4
jtt	416	15	4
Tomcat	2031	42	2
MARS	590	10	2

Таблица 3.2: Полученные результаты, без учета final полей.

Практические результаты имеют перспективы развития. Большинство найденных не final полей представляют нетривиальные структуры данных. Согласно работе [17], можно доказать, что некоторые объекты не покидают одного потока. Тогда, для динамического детектора можно будет отключить любые вызовы методов данного объекта.

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 synchronized методов.

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

Листинг 3.3: Использование блокировки для синхронизации.

Заключение

Гонки в программах являются распространённым видом ошибок, доставляющих много проблем. Поэтому обнаружение гонок является актуальной задачей. Работы в области динамического обнаружения в настоящее время ведутся активно. Главной проблемой динамического подхода является снижение производительности анализируемой программы. В данной работе был рассмотрен вопрос оптимизации динамических детекторов, путем проведения предварительного статического анализа.

В рамках работы был разработан алгоритм для поиска корректно-синхронизированных полей. Данный алгоритм позволяет обнаруживать поля, при обращении к которым не может возникнуть гонки. Данный анализ позволяет сократить время выполнения и объем потребления памяти динамического детектора.

Полученный алгоритм позволяет отслеживать различные операции синхронизации, используемые для защиты обращений к разделяемым переменным. Алгоритм основан на обходе графов потока исполнения методов.

Также была разработана программа, реализующая разработанный алгоритм. Корректность работы программы была протестирована большим набором тестов. Получены результаты запуска программы на типичных приложениях, которые подтвердили предположения о возможностях данного подхода.

Одной из целей данной работы являлась оптимизация динамического поиска гонок. Полученная программа предоставляет интерфейс для интеграции с динамическими детекторами. В рамках данной работы она была интегрирована с существующим динамическим детектором гонок для Java-программ. Статистика, полученная динамическим детектором, подтвердила прирост его производительности.

Таким образом, главными практческими результатами являются разработання программа и полученная оптимизация динамического детектора. Гланым теоретическим результатом является предложенный алгоритм для поиска корректно-синхронизированных полей.

Источники

[1] Трифанов В.Ю. Динамическое обнаружение состояний гонки в многопоточных Java-программах. 2013.

[2] Netzer R. H. B., Miller B. P. What Are Race Conditions?: Some Issues and Formalizations // ACM Lett. Program. Lang. Syst. New York, NY, USA, 1992. Vol. 1, no. 1. P. 74–88. URL: http://doi.acm.org/10.1145/130616.130623.

[3] Netzer R. H. B. Race Condition Detection for Debugging Shared-memory Parallel Programs. Ph.D. thesis. Madison, WI, USA: University of Wisconsin at Madison, 1991. UMI Order No. GAX91-34338.

[4] Naik M., Aiken A., Whaley J. Effective Static Race Detection for Java // SIGPLAN Not. New York, NY, USA, 2006. Vol. 41, no. 6. P. 308–319. URL: http://doi.acm.org/10.1145/1133255.1134018.

[5] Chord: A Program Analysis Platform for Java. URL: http://pag.gatech.edu/chord.

[6] Boyapati C., Rinard M. A Parameterized Type System for Race-free Java Programs // SIGPLAN Not. New York, NY, USA, 2001. Vol. 36, no. 11. P. 56–69. URL: http://doi.acm.org/10.1145/504311.504287.

[7] Engler Dawson, Ashcraft Ken. RacerX: Effective, Static Detection of Race Conditions and Deadlocks // SIGOPS Oper. Syst. Rev. New York, NY, USA, 2003. Т. 37, № 5. С. 237–252. URL: http://doi.acm.org/10.1145/1165389.945468.

[8] Flanagan C., Freund S. N. Type-based Race Detection for Java // SIGPLAN Not. New York, NY, USA, 2000. Vol. 35, no. 5. P. 219–232. URL: http://doi.acm.org/10.1145/358438.349328.

[9] Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs / J.-D. Choi, K. Lee, A. Loginov et al. // SIGPLAN Not. New York, NY, USA, 2002. Vol. 37, no. 5. P. 258–269. URL: http://doi.acm.org/10.1145/543552.512560.

[10] Трифанов В.Ю., Цителов Д.И. Статические и post-mortem средства обнаружения гонок в параллельных программах. 2011.

[11] Lamport L. Time, Clocks, and the Ordering of Events in a Distributed System // Commun. ACM. New York, NY, USA, 1978. Vol. 21, no. 7. P. 558–565. URL: http://doi.acm.org/10.1145/359545.359563.

[12] Eraser: A Dynamic Data Race Detector for Multi-threaded Programs / S. Savage, M. Burrows, G. Nelson et al. // SIGOPS Oper. Syst. Rev. New York, NY, USA, 1997. Vol. 31, no. 5. P. 27–37. URL: http://doi.acm.org/10.1145/269005.266641.

[13] Elmas Tayfun, Qadeer Shaz, Tasiran Serdar. Goldilocks: A Race and Transaction-aware Java Runtime // SIGPLAN Not. New York, NY, USA, 2007. Т. 42, № 6. С. 245–255. URL: http://doi.acm.org/10.1145/1273442.1250762.

[14] Java Language Specification, Third Edition. Threads and Locks. URL: http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html.

[15] FindBugs. URL: http://www.ibm.com/developerworks/java/library/j-findbug1/.

[16] ThreadSafe. URL: http://www.contemplateltd.com/threadsafe.

[17] Whaley J., Rinard M. Compositional Pointer and Escape Analysis for Java Programs // SIGPLAN Not. New York, NY, USA, 1999. Vol. 34, no. 10. P. 187–206. URL: http://doi.acm.org/10.1145/320385.320400.

[18] Allen Frances E. Control Flow Analysis // SIGPLAN Not. New York, NY, USA, 1970. Т. 5, № 7. С. 1–19. URL: http://doi.acm.org/10.1145/390013.808479.

[19] Bilardi G., Pingali K. Algorithms for Computing the Static Single Assignment Form // J. ACM. New York, NY, USA, 2003. Vol. 50, no. 3. P. 375–425. URL: http://doi.acm.org/10.1145/765568.765573.

[20] Soot - a Java Bytecode Optimization Framework / R. Vallée-Rai, P. Co, E. Gagnon et al. 2000. URL: http://dl.acm.org/citation.cfm?id=781995.782008.

[21] Soot. A framework for analyzing and transforming Java and Android Applications. URL: http://sable.github.io/soot/.