

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

Содержание	2
Введение	4
Глава 1. Обзор	8
1.1 Методы автоматического обнаружения гонок	8
1.1.1 Статический подход	8
1.1.2 Динамический подход	9
1.2 Динамический детектор гонок для java программ	9
1.2.1 Модель памяти java	9
1.2.2 Динамический детектор jDRD	10
1.2.3 Возможности статического анализа для оптимизации jDRD	11
1.3 Статический анализ для поиска корректно- синхронизированных полей	11
1.3.1 Существующие утилиты для статического анализа .	11
1.3.2 Возможные подходы к анализу	12
Глава 2. Описание алгоритма	14
2.1 Описание промежуточного представления и CFG для него .	14
2.2 Получение множества возможных блокировок	17
2.2.1 Нахождение переменных, которые могут ссылаться на разные поля	18
2.2.2 Нахождение переменных, ссылающихся на не final поля	19
2.3 Обход графа потока исполнения	19
2.3.1 Обработка операций синхронизации	21
2.3.2 Обработка операции обращения к полям	21

Глава 3. Практическая реализация алгоритма	22
3.1 Особенности реализации	22
3.2 Тестирование	22
3.3 Интеграция с jDRD	22
Заключение	23
Источники	24

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

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.

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 отличие 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 точность, 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 есть операции синхронизации. В JMM описано отношение *happens-before*. Если операция *A happens-before B*, то при выполнении операции *B* видны изменения, выполненные *A*. Если операции *A* и *B* происходят из разных потоков и об-

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

1.2.2. Динамический детектор jDRD

jDRD — динамический детектор для java программ. Он основан на *happens-before* алгоритме.

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. 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 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. 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-

синхронизированных 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, 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. 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 рассмотрено 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 третьей части будет рассмотрена основная часть алгоритма — обход графа потока исполнения, отслеживание и обработка операций синхронизаций 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 удовлетворяют следующей грамматике.

local	Локальная переменная
field	Поле
imm	<i>local</i> <i>constant</i>
expr	<i>imm₁ binop imm₂</i> <i>(type) imm</i> <i>imm₁ instanceof type</i> <i>invokeExpr</i> <i>new refType</i> <i>newarray (type) [imm]</i> <i>neg imm</i>
assignStmt	<i>local = ϕ(imm₁, imm₂, ...)</i> <i>local = imm field local.field expr</i> <i>field = imm</i> <i>local.field = imm</i>

Таблица 2.1: Грамматика присваиваний используемого IR

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

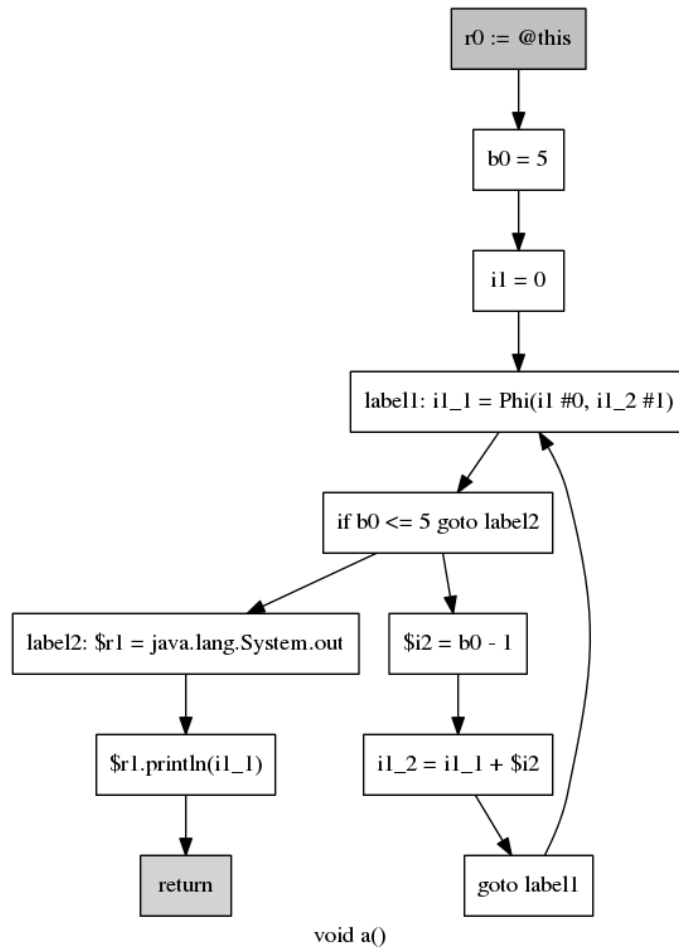


Рис. 2.1: CFG метода

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, которое может измениться, то операции взятия блокировки по таким пере-

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

```
Class A {  
    final Object lock1 = new Object();  
    final Object lock2 = new Object();  
    void a() {  
        Object lock;  
        if (System.currentTimeMillis() % 10000 == 0) {  
            lock = lock1;  
        } else {  
            lock = lock2;  
        }  
        synchronized(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 полученное представление является SSA, то каждая переменная имеет единственное место инициализации. Если переменная инициализируется как ϕ -функция, то значит эта переменная может иметь не единственное значение. 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 единственное 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 синхронизации 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 $v.locks \not\subseteq curLocks$, 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 обращении 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  $\subseteq$  currentLocks then
    break
  else
    v.locks  $\leftarrow$  v.locks  $\cap$  currentLocks
    currentLocks  $\leftarrow$  v.locks  $\cap$  currentLocks
  op  $\leftarrow$  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  $\leftarrow$  op.getField()
    field.locks  $\leftarrow$  field.locks  $\cap$  currentLocks
  for all CFGVertex c : v.childs do
    VISIT(c, currentLocks)
end function

```

Далее будет более подробно рассмотрена 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.concurrent` 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*.

Глава 3. Практическая реализация алгоритма

3.1. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.2. ТЕСТИРОВАНИЕ

3.3. ИНТЕГРАЦИЯ С JDRD

Заключение В ходе данной работы был получен алгоритм для поиска корректно-синхронизированных полей. Также была разработана программа реализующая данный алгоритм. Программа была запущена на различных библиотеках и показала следующие результаты : . Текст разный [1].

Источники

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