DOI: 10.15514/ISPRAS–2020–32(5)–2

Обнаружение дефекта взаимной блокировки с помощью статического анализа

С.А. Поляков, ORCID: 0000-0002-8542-8035 <inly@ispras.ru>
А.Е. Бородин, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В статье описывается расширение статического анализа программ на основе резюме для поиска ошибок взаимных блокировок потоков. Анализ на основе резюме является популярным подходом для поиска ошибок в программах благодаря высокой производительности и масштабируемости. При этом реализация детекторов поиска взаимных блокировок в таком анализе является нетривиальной, так как в процессе внутрипроцедурного анализа функций отсутствует информация об удерживаемых блокировках выше по стеку вызовов. Для моделирования семантики многопоточных программ используется граф блокировок, который строится во время основного анализа. Граф блокировок является модификацией графа вызовов с добавлением информации об удерживаемых блокировках. После построения графа блокировок запускается детектор обнаружения взаимных блокировок. Как построение графа блокировок, так и алгоритм обнаружения взаимных блокировок, не требуют существенного процессорного времени. На выполненных замерах общее время анализа увеличилось на 4%. По результатам анализа 8 проектов с открытым исходном кодом на языках C/C++/Java общим размером более 14 млн. строк кода предложенный алгоритм показал высокий уровень истинных срабатываний. Описываемые алгоритмы были реализованы в инструменте Svace.

Ключевые слова: статический анализ; символьное выполнение; параллелизм; взаимные блокировки

Для цитирования: Поляков С. А., Бородин А. Е. Обнаружение дефекта взаимной блокировки с помощью статического анализа. Труды ИСП РАН, том 32, вып. 5, 2020 г., стр. 21-34. DOI: 10.15514/ISPRAS–2020–32(5)–2

Deadlock Detection using Static Analysis

S.A. Polyakov, ORCID: 0000-0002-8542-8035 <inly@ispras.ru>
A.E. Borodin, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>
Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. The paper describes an extension to summary based static program analysis to find deadlock errors. Summary based analysis is a popular approach aimed at the detection of bugs in programs due to its high performance and scalability. At the same time, the implementation of deadlock detectors in such an analysis is nontrivial, because there is no information about the locks held higher in the call stack during the process of function intraprocedural analysis. A lock graph, which is built during the main analysis, is used to model the semantics of multithreaded programs. Lock graph is a modification of call graph which contains additional information about held locks. After the lock graph is built, the deadlock detector is launched. Both the construction of the lock graph and the deadlock detection algorithm do not require significant processor time. On the performed measurements, the total analysis time increased by 4%. Based on the results of the analysis of 8 open source projects in C/C++/Java with a total size of more than 14 million lines of code, the proposed algorithm showed a high level of true positives. The described algorithms were implemented in the Svace tool.

Keywords: static analysis; symbolic execution; concurrency; deadlock freedom

For citation: Polyakov S. A., Borodin A. E. Deadlock Detection using Static Analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 5, 2020, pp. 21-34 (in Russian). DOI: 10.15514/ISPRAS-2020-32(5)-2

1. Introduction

Современные вычислительные устройства в основном являются многоядерными и многопроцессорными. Они предоставляют возможность выполнять параллельные программы, эффективность которых выше аналогичных последовательных программ. Параллельная программа представляет из себя несколько взаимодействующих потоков управления, одновременно выполняющих операции. Разработка параллельных программ является нетривиальной задачей. При проектировании параллельных программ, в отличие от однопоточных, могут быть допущены специфические ошибки проектирования (дефекты): состояния гонки и взаимные блокировки. При наличии состояний гонки работа программы зависит от порядка выполнения частей кода, что приводит к искажениям данных, последствия которых могут проявляться в непредсказуемые моменты времени. Взаимные блокировки приводят к одноименной ошибке.

Взаимная блокировка потоков – ошибка, возникающая, когда несколько потоков находятся в состоянии бесконечного ожидания ресурсов, занятых самими этими потоками. В случае двух потоков: первый поток ждет, пока второй освободит ресурс, необходимый для завершения работы первого потока, в то время как второй поток, в свою очередь, ждет освобождения другого ресурса первым.

В статье описывается подход для обнаружения взаимных блокировок в многопоточных программах с помощью статического анализа. Алгоритмы из данной статьи могут быть реализованы в любом статическом анализаторе на основе символьного выполнения с объединением состояний и резюме с обходом графа вызовов «снизу-вверх». В описываемом подходе во время фазы обхода графа вызовов собирается необходимая информация, которая будет использована после обхода всех функций. На используемые алгоритмы накладываются несколько ограничений: найти как можно больше ошибок за ограниченное время при приемлемом уровне ложных срабатываний. При этом допустимы как пропуски ошибок, так и выдача ложных срабатываний. Недопустимы любые требования по подготовке анализируемой программы, либо ограничение используемых конструкций языка. В качестве базового статического анализатора используется инструмент Svace. В данной статье описывается подход для обнаружения взаимных блокировок двух потоков. Однако предложенный подход может быть расширен на случай нескольких потоков.

В качестве анализируемого языка будем использовать императивный язык, содержащий вызовы функций, в том числе по указателю, и операции ветвления. В качестве примитива синхронизации будем использовать мьютекс, который можно заблокировать функцией *lock*, и разблокировать функцией *unlock*. Мы не рассматриваем циклы, поскольку в большинстве случаев для анализа они будут эквивалентны условным выражениям. В реальном коде в теле цикла либо не будет блокировок вообще, либо за блокировкой будет следовать разблокировка мьютекса.

Для сбора необходимой информации используется абстракция «Граф блокировок», содержащая информацию о последовательности заблокированных мьютексов и вызовов функций. Упоминание и краткое описание графа блокировок приведено в статье [1], описывающей особенности анализа Java программ. Поиск ошибок там осуществляется только для стандартного средства управления потоками – конструкции *synchronize*. Использование конструкции *synchronize* аналогично использованию «сбалансированных» вызовов блокировки и разблокировки мьютексов, что упрощает анализ. Под «сбалансированностью» подразумевается следующее правило: цепочка заблокированных мьютексов может быть разблокирована только в порядке, обратном порядку их блокировки.

Данная статья содержит описание модифицированного алгоритма для анализа программ на языках C, C++ и Java, в том числе для «несбалансированных» вызовов блокировки и разблокировки мьютексов.

2. Базовый анализ

Предлагаемый подход к поиску ошибок взаимоблокировок является расширением статического анализа на основе резюме. В качестве базового анализатора в данной работе был использован анализатор Svace. Используемые алгоритмы могут быть реализованы в любом другом похожем анализе. В данном разделе опишем какими свойствами должен обладать базовый анализ, чтобы его можно было расширить.

Анализ на основе резюме является достаточно популярным, так как имеет ряд существенных преимуществ. При таком анализе функции анализируемой программы обходятся «снизу-вверх» по графу вызовов. После анализа функции создаётся ее резюме, описывающее поведение функции, которое дальнейшем используется при анализе вызова функции. При отсутствии циклов в графе вызовов каждая функция посещается только один раз. Многие существующие реализации игнорируют циклы в графе вызовов. Анализы, учитывающие циклы в графе вызовов, анализируют некоторые функции более одного раза, но как правило количество проходов не велико. Анализ на основе резюме имеет высокую произоводительность и масштабируемость. Скорость анализа достигается за счёт того, что каждая функция анализируется только один раз. Масштабируемость получается за счёт ограничения размера резюме. Кроме того, анализ на основе резюме легко распараллеливается, так как информация передается только от вызываемых функций к вызывающим, и поэтому значительное количество функций могут анализироваться независимо. Дополнительным преимуществом анализа на основе резюме является возможность анализировать библиотеки, а не только целые программы. Из"=за множества преимуществ анализа на основе резюме, его довольно часто используют в инструментах статического анализа. Например, следующие инструменты используют анализ на основе резюме: PREfix [2], Archer [3], Saturn [4], Calysto [5], SharpChecker [6], Coverity [7].

Базовый анализ функции должен обладать потоковой чувствительностью, т.е. учитывать порядок инструкций. Кроме этого, он должен отслеживать значения переменных. Мы будем рассматривать символьное выполнение с объединением абстрактных состояний в точках объединения путей. В базовом анализе потребуется реализация некоторых вспомогательных анализов.

3. Граф блокировок

3.1 Расширение анализа на основе резюме

Во время анализа функции мьютексы, которые были заблокированы выше по графу вызовов, не известны, поэтому анализ на основе резюме плохо подходит для поиска взаимных блокировок. Для поиска таких ошибок предлагается расширить базовый анализ на основе резюме с помощью использования графа блокировок. В предложенном анализе поиск ошибок осуществлялся в две фазы.

- Фаза обхода функций программы по графу вызовов «снизу-вверх». На этой фазе происходит построение графа блокировок. Во время обхода функции доступны резюме вызываемых функций, содержащие информацию о блокировке и разблокировке мьютексов.
- Анализ графа блокировок для обнаружения ошибок в программе. На данной фазе информация, содержащаяся в резюме уже не доступна, что сделано из соображений производительности, чтобы не требовалось хранить резюме функций на протяжении всего анализа.

Таким образом, алгоритм поиска взаимных блокировок представляет собой надстройку над анализом на основе резюме. Вторая фаза анализа графа блокировок не занимает значительного времени и не требует много памяти. Построение графа блокировок на первой фазе незначительно увеличивает время первой фазы, и это увеличение пропорционально количеству инструкций в программе. Время анализа первой фазы зависит от реализации в немодифицируемом анализаторе.

Первая фаза может занимать существенное время в зависимости от алгоритмов, реализованных в оригинальном анализаторе.

3.2 Определение графа блокировок

Граф блокировок приближенно описывает параллельную программу и используется для обнаружения взаимных блокировок. Будем считать, что в программе присутствует взаимная блокировка двух потоков, если существует такой путь выполнения, на котором последовательно блокируются мьютексы *l*1 и *l*2, а также существует путь выполнения, на котором последовательно блокируются мьютексы *m*2 и *m*1, причем *m*1 является алиасом *l*1, а *m*2 является алиасом *l*2.

Рассмотрим пример из листинга 1. Внутри функции *foo* происходят последовательные блокировки мьютексов, а затем – разблокировки мьютексов в обратном порядке. При вызовах данной функции из *test*1 и *test*2 в качестве формальных параметров передаются глобальные мьютексы *g*1 и *g*2. В данном примере возможна взаимная блокировка потоков, если функции *test*1и *test*2 выполняются параллельно. Смоделируем ошибочное поведение программы. В одном потоке выполняется функция *test*1, глобальные переменные *g*1 и *g*2 передаются в функцию *foo* в качестве первого и второго параметров соответственно, *g*1 блокируется вызовом *pthread_mutex_lock* на 6 строке, предпринимается попытка заблокировать *g*2 на 7 строке. Параллельно в другом потоке выполняется функция *test*2, глобальные переменные *g*2 и *g*1 передаются в функцию *foo* в качестве первого и второго параметров соответственно, *g*2 блокируется на 6 строке, предпринимается попытка заблокировать *g*1 на 7 строке.

```
1:  #include <pthread.h>
2:
3:  pthread_mutex_t *g1, *g2;
4:
5:  void foo(pthread_mutex_t *p1, pthread_mutex_t *p2) {
6:      pthread_mutex_lock(p1);
7:      pthread_mutex_lock(p2);
8:      pthread_mutex_unlock(p2);
9:      pthread_mutex_unlock(p1);
10: }
11:
12: void test1() {
13:     foo(g1, g2);
14: }
15:
16: void test2() {
17:     foo(g2, g1);
18: }
19: // alias.c
```

Листинг 1. Пример взаимной блокировки
Listing 1. Deadlock example

Граф блокировок – ориентированный граф, содержащий информацию о вызовах функций и захвате мьютексов. Вершина в графе блокировок сопоставляется событию в выполнении программы одного из следующих типов: блокировка мьютекса; передача мьютекса или

содержащего его объекта в функцию, где в дальнейшем мьютекс будет заблокирован; вызов функции; начало выполнения функции. Все вершины аннотируются строками в исходном коде. Заметим, что в графе блокировок явно не содержится информация об освобождения мьютексов.

Вершина типа **VLock** описывает блокировку мьютекса, она содержит информацию о мьютексе. Вершина **VLock** означает, что мьютекс мог быть заблокирован. Вершина типа **VCall** описывает вызов функции. Вершина типа **VStart** описывает начало выполнения функции. Вершины последних двух типов содержат информацию о функции, с помощью которой ее можно идентифицировать. Вершина типа **VLockAlias** ⊂ **VLock** описывает передачу мьютекса в функцию. Вершина такого типа описывает захват блокировки внутри вызова функции, а также является «алиасом» вершин типа **VLockAlias** или **VLock**. Вершина типа **VLockAlias** содержит информацию о мьютексе и его алиасе.

На изображениях графов блокировок, приведенных в данной работе, вершины типа **VLock** изображены в виде октагона, вершины типа **VAliasLock** – октагона с двойным контуром, вершины типа **VCall** – прямоугольника, а вершины типа **VStart** – пентагона.

Ребра в графе блокировок могут быть следующих видов.

- **VLock**[$l1$] → **VLock**[$l2$] Ребро между вершиной, описывающей захват мьютекса $l1$, и вершиной, описывающей захват мьютекса $l2$. Наличие такого ребра означает, что $l1$ является последним захваченным мьютексом, предшествующей захвату $l2$. Данный факт будем обозначать $l1 \to l2$.
- **VLock**[l] → **VCall**[p] Ребро между вершиной, описывающей захват блокировки l, и вершиной, описывающей вызов функции p. Наличие такого ребра означает, что l является последней захваченной блокировкой, предшествующей вызову p. Данный факт будем обозначать $l \to p$.
- **VCall**[p] → **VStart**[p'] Ребро между вершиной p, описывающей вызов функции, и вершиной p', описывающей начало выполнения вызываемой функции. Фактически это ребро от вызова функции к её определению. Такое ребро существует только тогда, когда p и p' идентифицируют одну и ту же функцию.

  Отсутствие исходящих ребёр из **VCall**[p] означает отсутствие информации о вызванной функции. Несколько исходящих ребёр означают несколько альтернатив для разрешения вызова процедуры. Последнее возможно при вызове функции по указателю, когда возможен вызов нескольких функций. Таким образом вершина вызова функций имеет либо пустое множество исходящих ребёр, либо содержит рёбра на исчерпывающее множество возможных определений вызванных функций. Данный факт будем обозначать $p \to p'$.
- **VStart**[p] → **VLock**[l] Ребро между вершиной, описывающей начало выполнения функции p, и вершиной, описывающей блокировку мьютекса l. Наличие такого ребра означает, что в ГПУ существует путь выполнения, на котором все инструкции выполняются при заблокированном мьютексе l. Данный факт будем обозначать $p \to l$.
- **VStart**[$p1$] → **VCall**[$p2$] Ребро между вершиной, описывающей начало выполнения функции $p1$, и вершиной, описывающей вызов функции $p2$. Наличие такого ребра означает, что $p1$ вызывает $p2$, не удерживая никакую блокировку. Данный факт будем обозначать $p1 \to p2$.
- **VCall**[p] → **VLockAlias**[l] Ребро между вершиной, описывающей вызов функции p, и вершиной, описывающей блокировку мьютекса l внутри функции p. Данный факт будем обозначать $p \to l$.
- **VLockAlias**[l'] → **VLock**[l] Ребро между вершинами, описывающими блокировку мьютекса l, алиасом которого является l'. Наличие такого ребра означает, что мьютекс l' (или объект, его содержащий) был передан в качестве фактического параметра в некоторую функцию p, внутри которой мьютекс l был заблокирован, причем l (или

объект, его содержащий) является формальным параметром функции p, и l' является алиасом l. Данный факт будем обозначать $l' \to l$.
- **VLockAlias**[$l1$] → **VLockAlias**[$l2$] Ребро между вершинами, описывающими мьютексы $l1$ и $l2$, которые являются алисами. Наличие такого ребра означает, что мьютекс $l1$ (или объект, его содержащий) был передан в качестве фактического параметра в некоторую функцию $p1$, внутри которой мьютекс $l2$ был передан в другую функцию $p2$, причем $l2$ (или объект, его содержащий) является формальным параметром функции $p1$ и $l1$ является алиасом $l2$. Внутри функции $p2$ мьютекс l', являющийся алиасом $l2$, может как блокироваться, так и передаваться в качестве параметра дальше, но в конечном итоге окажется заблокированным. Данный факт будем обозначать $l1 \to l2$.

Заметим, что только ребра типа **VCall**[p] → **VLockAlias**[l] могут быть добавлены в граф блокировок во время обхода графа вызовов программы. Ребра типа **VLockAlias**[l'] → **VLock**[l] и типа **VLockAlias**[$l1$] → **VLockAlias**[$l2$] создаются после обхода графа вызовов перед анализом графа блокировок. Для создания таких ребер используется информация, сохраненная в вершинах **VLockAlias**, а сами ребра создаются с целью оптимизации анализа графа блокировок.

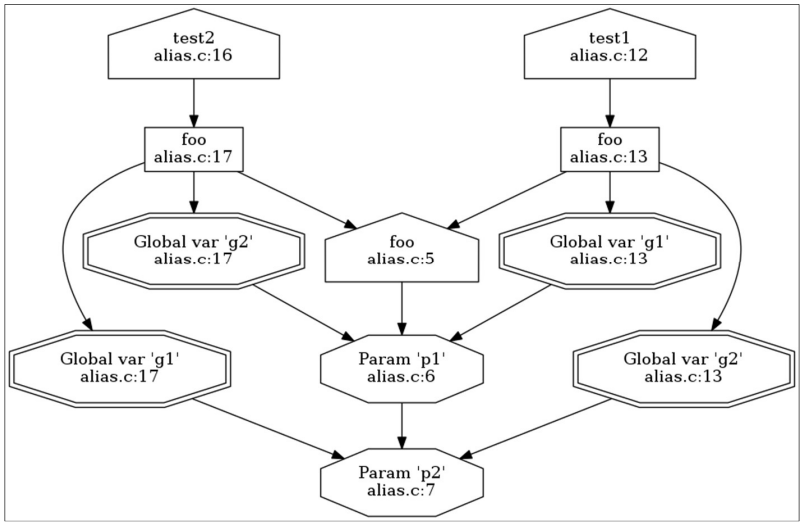

Рис. 1. Граф блокировок, соответствующий листингу 1
Fig. 1. Lock graph corresponding to listing 1

Граф блокировок, который соответствует листингу 1, изображен на рис. 1. В результате анализа графа блокировок будет выдано предупреждение о возможной взаимной блокировке на основании следующих фактов.

- Внутри функции *foo* мьютексы блокируются в следующем порядке: $p1 \to p2$.
- В случае вызова функции *foo* из *test*1 мьютекс $g1$ соответсвует $p1$, а мьютекс $g2$ соответствует $p2$. Получаем пару заблокированных мьютексов в порядке: $g1 \to g2$.
- В случае вызова функции *foo* из *test*2 мьютекс $g2$ соответсвует $p1$, а мьютекс $g1$ соответствует $p2$. Получаем пару заблокированных мьютексов в порядке: $g2 \to g1$.
- В программе существует путь выполнения, на котором последовательно блокируются мьютексы $g1$ и $g2$, а также существует путь, на котором последовательно блокируются $g2$ и $g1$.

3.3 Вспомогательные анализы

Для построения графа блокировок необходимо несколько вспомогательных анализов. Анализы устанавливают некоторые свойства программы. Для описания свойств используются атрибуты. Атрибуты могут ассоциироваться с ребром в ГПУ, либо с идентификатором значения.[1]

Атрибут типа **LockStatus**, с одной стороны, описывает состояние мьютекса в некоторой точке программы, с другой стороны – эффект от вызова некоторой функции, то есть каким образом вызов функции влияет на состояние мьютекса. Возможные состояния описываются множеством значений данного атрибута.

- *default* – над мьютексом не выполнялись операции блокировки/разблокировки. Вызов функции не влияет на состояние мьютекса.
- *lock* – мьютекс заблокирован. Вызов функции блокирует мьютекс.
- *may_lock* – мьютекс возможно заблокирован, то есть мьютекс блокируется хотя бы на одном пути выполнения, но не на всех возможных путях. Вызов функции возможно блокирует мьютекс.
- *unlock* – мьютекс разблокирован, то есть над мьютексом была выполнена операция разблокировки. Вызов функции разблокирует мьютекс.
- *may_unlock* – мьютекс возможно разблокирован. Вызов функции возможно разблокирует мьютекс.
- *unlock_then_lock* – мьютекс был разблокирован, а затем заблокирован. Вызов функции разблокирует, а затем заблокирует мьютекс.
- *unlock_then_may_lock* – мьютекс был разблокирован, а затем возможно заблокирован. Вызов функции разблокирует, а затем возможно заблокирует мьютекс.
- *may_unlock_then_lock* – мьютекс возможно был разблокирован, а затем заблокирован. Вызов фукнции возможно разблокирует, а затем заблокирует мьютекс.
- *may_unlock_then_may_lock* – мьютекс возможно был разблокирован, а затем возможно заблокирован. Вызов фукнции возможно разблокирует, а затем возможно заблокирует мьютекс.

Значения данного атрибута ассоциируются с идентификаторами значений.

Введем функцию *val*, которая для каждой переменной программы возвращает ассоциированный с ней идентификатор значения.

```
1:  #include <pthread.h>
2:
3:  pthread_mutex_t *g;
4:
5:  void safe();
6:  void unsafe();
7:
8:  void may_lock_f(int lock) {
9:      if (lock) {
10:         pthread_mutex_lock(g);
11:     }
12: }
13:
14: void may_unlock_f(int unlock) {
15:     if (unlock) {
16:         pthread_mutex_unlock(g);
17:     }
18: }
```

[1] Под идентификатором значения будем понимать символьную переменную в терминах символьного выполнения.


```
19:
20: void test1(int safe) {
21:     may_lock_f(safe);
22:     unsafe();
23:     may_unloc_f(safe);
24: }
25:
26: void do_unsafe(void (*do_smth_ptr)()) {
27:     pthread_mutex_unlock(g);
28:     (*do_smth_ptr)();
29:     pthread_mutex_lock(g);
30: }
31:
32: void test2(int all_safe) {
33:     pthread_mutex_lock(g);
34:     safe();
35:     if (all_safe) {
36:         unsafe();
37:     } else {
38:         do_unsafe(&unsafe);
39:     }
40:     pthread_mutex_unlock(g);
41: }
```

Листинг 2. Пример распространения атрибута ***LockStatus***
Listing 2. Example of ***LockStatus*** *attribute flow*

Рассмотрим распространение данного атрибута на примере листинга 2. Функция *may_lock_f* блокирует мьютекс g в зависимости от условия, заданного параметром функции. Внутри данной функции значение g не меняется, и *val*(g) в каждой вершине ГПУ будет возвращать один и тот же идентификатор значения. Обозначим его v_g^{ml}. В ГПУ образуется ветвление из-за условного перехода на строке 9. В первом базовом блоке не будет ни одной инструкции, и значение атрибута, ассоциированное с v_g^{ml}, не будет изменено, то есть останется равным *default*. Однако во втором блоке мьютекс g будет заблокирован после вызова функции *pthread_mutex_lock* на строке10, и значение атрибута станет *lock*. После слияния путей в ГПУ на строке 11, согласно описанной функции *joinval*[**LockStatus**], получим значение *may_lock*. Данное значение, ассоциированное с v_g^{ml}, попадет в резюме функции *may_lock_f*. Аналогично, в резюме функции *may_unlock_f* попадет значение *may_unlock*, ассоциированное с v_g^{mu}. Функция *test*1 вызывает функцию *unsafe*, блокируя мьютекс g при условии, что *safe* != 0. Данный факт реализован посредством вызова *may_lock_f* на строке 21. Внутри функции *test*1 значение g не меняется, обозначим идентификатор значения *val*(g) = v_g^{t1}. До вызова *may_lock_f* значение атрибута, ассоциированное с v_g^{t1}, равно *default*. После вызова данной функции значение изменяется на *may_lock*, поскольку в момент вызова v_g^{t1} соответствует v_g^{ml}, а с v_g^{ml} в резюме ассоциировано значение *may_lock*. Получаем значение *default* после вызова функции *may_unlock_f* на строке 23, поскольку в момент вызова v_g^{t1} соответствует v_g^{mu}, а с v_g^{mu}в резюме ассоциировано значение *may_unlock*.

Функция *do_unsafe* вызывает функцию по указателю, переданному в качестве параметра. Она спроектирована таким образом, что вначале разблокирует мьютекс g, затем вызовет требуемую функцию, а затем заново заблокирует g. В резюме *do_unsafe* попадет значение *unlock_then_lock*, ассоциированное с идентификатором v_g^{do}. Внутри функции *test*2 значение g не меняется, обозначим идентификатор значения *val*(g) = v_g^{t2}. После вызова функции *pthread_mutex_lock* на строке 33 значение атрибута, ассоциированное с v_g^{t2}, равно *lock*. Данное значение распространится в каждый базовый блок, полученный в результате условного перехода на строке 35. В первом базовом блоке состояние мьютекса не меняется, следовательно, не поменяется и значение атрибута. Во втором базовом блоке вызывается

do_unsafe. Учитывая резюме функции (с v_g^{do}, который соответствует v_g^{t2}, ассоциировано значение *unlock_then_lock*) и значение *lock* атрибута до вызова, получим новое значение *lock*, ассоциированное с v_g^{t2}. После вызова функции *pthread_mutex_unlock* на строке 40, значение атрибута вновь станет равным *default*.

Атрибут типа **LocksOnPath** содержит информацию о том, какие мьютексы заблокированы на ребре, входящем в конкретную вершину ГПУ. Значением атрибута данного типа является множество всех возможных последовательностей блокировок. Формально записывается $\{LockChain_1, LockChain_2, \ldots, LockChain_n\}$,

$LockChain \equiv [lock_1 \rightarrow lock_2 \rightarrow \ldots \rightarrow lock_m]$,

$lock \equiv (lId, vId, trace)$, где

$vId \in V$ – множество идентификаторов значений,

$lId \in L$ – множество идентификаторов мьютексов, *trace* – последовательность точек в программе;

Значением по умолчанию для атрибута **LocksOnPath** является множество из одного элемента – пустой последовательности блокировок {[]}, которая означает отсутствие заблокированных мьютексов на ребре ГПУ. Значения данного атрибута ассоциируются с ребрами в ГПУ.

Реализация функции *joinval*[**LocksOnPath**] (функции определения значения атрибута при слиянии путей в ГПУ) тривиальна – поскольку значением атрибута является множество цепочек блокировок, то определим данную функцию следующим образом:

$Joinval\ (LocksOnPath_l, LocksOnPath_r) = LocksOnPath_l \cup LocksOnPath_r$.

Данный атрибут является межпроцедурным, то есть его значения хранятся в резюме и в случае вызова функции, значение атрибута меняется и в вызывающей функции. Причем для создания нового значения используется как значение данного атрибута в резюме, так и значения атрибута **LockStatus** из резюме, которые соответствуют идентификаторам значений, сохраненных в тройке *lock*.

```
1:  #include <pthread.h>
2:
3:  pthread_mutex_t *g1, *g2;
4:
5:  void unlock_then_lock() {
6:      pthread_mutex_unlock(g1);
7:      pthread_mutex_lock(g1);
8:  }
9:
10: void test(int cond) {
11:     // p1
12:     pthread_mutex_lock(g1);
13:     // p2
14:     pthread_mutex_lock(g2);
15:     // p3
16:     unlock_then_lock();
17:     // p4
18:     if (cond) {
19:         pthread_mutex_unlock(g1);
20:         pthread_mutex_unlock(g2);
21:         // p5
22:     }
23:     // p6
24: }
```

Листинг 3. Пример распространения атрибута ***LocksOnPath***
Listing 3. Example of ***LocksOnPath*** *attribute flow*

Рассмотрим распространение данного атрибута на примере листинга 3. В точке *p*1 на ребре ГПУ, входящем в вершину инструкции вызова функции *pthread_mutex_lock* на строке 12, атрибут **LocksOnPath** имеет значение по умолчанию {[]}. В точке *p*2 атрибут принимает {[*lockId*(*val*(*g*1))]} после вызова *pthread_mutex_lock* на строке 12. На строке 14 функция *pthread_mutex_lock* вызывается при уже заблокированном мьютексе~$g1$, таким образом, в точке~$p3$ атрибут принимает значение {[*lockId*(*val*(*g*1)) → *lockId*(*val*(*g*2))]}. Заметим, что функция *unlock_then_lock*, вызываемая на строке 16, сначала разблокирует, а затем опять заблокирует мьютекс *g*1. Таким образом, значение атрибута изменится на {[*lockId*(*val*(*g*2)) → *lockId*(*val*(*g*1))]} в точке *p*4. Далее в ГПУ образуется ветвление из-за условного перехода на строке 18. В первом базовом блоке не будет ни одной инструкции, и значение атрибута не будет изменено. Однако во втором блоке оба мьютекса *g*1 и *g*2 будут разблокированы, и значение атрибута в точке *p*5 станет равным значению по умолчанию. В точке *p*6 после слияния путей в ГПУ, согласно описанной функции *joinval*[**LocksOnPath**], получим значение {[], [*lockId*(*val*(*g*2)) → *lockId*(*val*(*g*1))]}.

Атрибут типа **PathToLock** содержит информацию, необходимую для идентификации мьютекса, являющегося алиасом ранее заблокированного мьютекса.

3.4 Построение графа блокировок

Во время анализа графа вызовов для каждой функции запускается анализ графа потока управления (ГПУ). Для каждой функции во время анализа ГПУ строится локальный граф блокировок, который добавляется в глобальный граф блокировок по завершении анализа текущей функции. Локальные графы добаляются в глобальный путем создания ребер типа **VCall**[*p*] → **VStart**[*p'*].

Опишем построение локального графа блокировок.

Перед началом обхода графа потока управления анализируемой функции *f* в граф добавляется вершина **VStart**[*f*].

Встретив инструкцию вызова функции *p* добавляем в граф вершину **VCall**[*p*]. Пути из вершины **VStart**[*f*]. в нее строятся с использованием значения атрибута **LocksOnPath** на ребре ГПУ, входящем в вершину, соответствующую инструкции вызова функции *p*. Соединим добавленную вершину с вершиной **VStart**[*f*] в случае если атрибут **LocksOnPath** имеет значение по умолчанию. В ином случае ищем путь из вершины **VStart**[*f*], проходящий через вершины типа **VLock**, описывающие мьютексы, которые соответствуют одной из цепочек мьютексов, содержащейся в атрибуте. Из последней вершины, лежащей на пути, проводим ребро к добавленной ранее вершине **VCall**[*p*]. Повторяем данную процедуру для каждой цепочки мьютексов, содержащейся в значении атрибута **LocksOnPath**. Если в резюме вызываемой функции *p* атрибут *PathToLock* имеет значение, не равное значению по умолчанию, то в граф блокировок добавляется вершина типа **VLockAlias**. Проводится ребро из ранее созданной вершины **VCall**[*p*] в вершину **VLockAlias**.

Встретив инструкцию вызова функции, в резюме которой значение атрибута **LocksOnPath** не равно значению по умолчанию, для каждой цепочки мьютексов для каждого мьютекса в цепочке создаем вершину типа **VLock**. Способ ее добавления в граф блокировок аналогичен способу добавления вершины типа **VCall**.

3.5 Анализ графа блокировок

Для обнаружения дефекта взаимной блокировки потоков предлагается следующий алгоритм.

1. Обходим граф блокировок в глубину. В каждый момент времени отслеживается множество захваченных блокировок на пути к текущей вершине. Также отслеживается множество вызванных функций на пути к текущей вершине.
2. Встретив вершину захвата блокировки **VLock**, запускаем процедуру поиска её ближайших потомков такого же типа, а также процедуру поиска ее алиасов **VLockAlias**.

Данные процедуры также отслеживают множество захваченных блокировок и множество вызванных функций.

3. При добавлении новой пары (*l*1, *l*2) проверяется, была ли инвертированная пара (*l*2, *l*1) ранее добавлена в коллекцию.
   - Если такой пары (*l*2, *l*1) не было, то пара (*l*1, *l*2) добавляется в коллекцию.
   - Если такая пара была ранее добавлена в коллекцию, то, пропуская её добавление в коллекцию, выполняются несколько дополнительных проверок, и, если проверки будут пройдены, будет выдано сообщение о дефекте.

В качестве примера дополнительной проверки для подавления ложных срабатываний рассмотрим ситуацию, когда в программе используется так называемый guarding/gate lock – такая блокировка специального мьютекса, что несколько потоков не могут одновременно достигнуть области возможной взаимной блокировки потоков. Для предотвращения выдачи ложного предупреждения необходимо выполнить следующую проверку. Для каждой вершины, являющейся первой в своей паре, собрать множество доминирующих её вершин, имеющих тип **VLock**, либо **VAliasLock**. Если пересечение полученных множеств не пусто, необходимо отменить выдачу предупреждения.

4. Результаты

В табл. 1 приведены проекты, на которых был протестирован предложенный в работе алгоритм поиска взаимных блокировок. Также в таблице приведены данные о размерах проектов и указано время анализа каждого отдельного проекта.

Анализ проектов проводился на машине со следующими характеристиками: 4 восьмиядерных процессора Intel Xeon E5-2680, 128 Gb RAM. Работа модулей, предназначенных для поиска дефектов многопоточности, занимала не более 4% времени всего анализа.

Табл. 1. Оценка производительности
Table 1. Performance evaluation

Проект	Строк кода, тыс.	Функций, тыс.	Время анализа, мин.
Android 5.2 (Java)	4364	489	40
Android 5.2 (C/C++)	8561	1147	164
jenkins	121	26	5.36
tomcat	299	28	11.41
cassandra	297	49	9.22
flink	363	36	4.38
phoenix	234	32	10.55
storm	188	21	2.41

В табл. 2 приведены результаты работы алгоритма поиска взаимных блокировок. Указаны только те из проанализированных проектов, в исходном коде которых были найдены дефекты данного типа.

Алгоритм показал низкий уровень ложных срабатываний. Однако в некоторых случаях оказалось невозможно определить является ли выданное предупреждение истинным или ложным, ввиду сложной логики работы некоторых частей проектов и запутанности их исходного кода.

Табл. 2. Результаты работы алгоритма поиска взаимных блокировок
Table 2. Deadlock algorithm results

Проект	Истинные	Ложные	Сложно определить
Android 5.2 (Java)	34	11	21

Android 5.2 (C/C++)	72	47	63
jenkins	4	0	2
tomcat	2	1	2
cassandra	2	0	1
flink	2	9	9

Также был проанализирован исходный код самого анализатора Svace на наличие состояний взаимной блокировки потоков. Однако таковые не были найдены.

5. Похожие работы

Подходы к обнаружению ошибок синхронизации потоков в параллельных программах можно разделить на две группы: статические и динимаческие методы.

Инструменты, использующие динамический подход [8-11], анализируют только те пути выполнения программы, которые она проходит на тестовых запусках. Также агрессивная инструментация кода, которая сопровождает динамический анализ, исключает использование данного подхода на низкоуровневом коде: операционные системы, драйверы устройств – то есть в тех случаях, когда ошибки многопоточности наиболее опасны. Динамический анализ может влиять на взаимодействие потоков, из-за чего дефект может быть пропущен в ходе анализа. Преимущество же динамического подхода заключается в высокой точности анализа, так как в момент выполнения программы детектор имеет точные сведения о значениях переменных и существующих алиасах. Однако точность достигается путем больших накладных расходов – скорость работы программы, которую инспектирует динамический детектор, снижается в десятки раз, а потребление памяти растет.

Инструменты, использующие статический подход, не вмешиваются в работу программы и не замедляют скорость ее выполнения. Кроме того, они анализируют все пути выполнения программы, даже те, которые выполняются крайне редко. Главным недостатком статического подхода является невысокая точность анализа.

В работе [12] описывается система типов для языка программирования Java. Она разработана с целью предотвратить и взаимные блокировки, и состояния гонки. С помощью определенной в данное работе системы типов программисту необходимо описать используемые в программе механизмы синхронизации потоков, то есть аннотировать программу. Отсутствие в программе взаимных блокировок обеспечивается с помощью введения классов эквивалентности на множестве блокировок и определения частичного порядка на множестве введенных классов. Несмотря на то, что такая система типов достаточно эффективна, данный подход не используется в промышленном программировании по двум причинам. Во-первых, масштабное «ручное» аннотирование кода, необходимое в данном случае, требует значительных затрат человеческих ресурсов. Во-вторых, такая система типов требует либо специальный компилятор, либо инструмент, который будет транслировать корректно типизированную программу в Java байт-код.

В анализаторе Chord [13, 14] используется потоко-чувствительный анализ для обнаружения взаимных блокировок и состояний гонки в программах на языке Java. Алгоритм обнаружения взаимных блокировок использует комбинацию нескольких статических анализов, каждый из которых аппроксимирует различные необходимые условия существования дефектов взаимной блокировки в исходном коде программ. Авторы выделяют шесть задач, которые эффективно могут быть решены методами статического анализа. Первая задача – задача достижимости: может ли поток выполнения достигнуть точки захвата объекта блокировки B после того, как он уже завладел объектом блокировки A. Вторая задача – анализ указателей, определяющий, какие указатели и переменные в программе ссылаются на один и тот же участок памяти. Третья задача – анализ сохраненных значений: доступна ли блокировка, захваченная одним потоком выполнения, из других потоков. Четвертая задача – задача

определения следующего факта: могут ли два потока выполнения одновременно достичь точек, где первый поток попытается завладеть блокировкой, удерживаемой вторым потоком, а второй поток попытается завладеть блокировкой, удерживаемой первым потоком. Пятая и шестая задачи сформулированы с целью уменьшения числа ложных предупреждений, которые могут появиться по двум причинам. Если поток выполнения пытается завладеть блокировкой, которой он уже завладел ранее, то взаимная блокировка произойти не может, поскольку спецификация языка Java не запрещает этого. Также взаимная блокировка невозможна, если два потока владеют общей блокировкой (guarding/gate lock). Каждая из шести задач является в общем случае неразрешимой. Поэтому любое решение отдельной задачи является либо неточным, либо неполным. Предложенный в статье алгоритм консервативно аппроксимирует первые четыре задачи. Однако для решения последних двух задач используется неконсервативные алгоритмы, поэтому некоторые случаи взаимных блокировок могут быть пропущены.

В работах [15, 16] используется метод проверки модели. Идея проверки модели концептуально очень проста: исследовать все возможные пути выполнения для всех возможных значений переменных, чтобы определить, могут ли возникать те или иные виды нежелательного поведения. Конечно, сформулированная таким образом задача является алгоритмически неразрешимой. По этой причине создаются такие абстракции, как поток управления или значения данных для конкретной программы, и именно эта модель параллельной программы изучается.

6. Заключение

В данной работе описан граф блокировок для многопоточных программ, написанных на языках C/C++/Java. Во время анализа, в графе блокировок накапливается информация, необходимая для поиска дефектов, связанных с некорректной синхронизацией потоков. Данный граф представляет из себя модификацию графа вызовов – в него добавляются специальные вершины, содержащие информацию об инструкциях блокировки, полученную во время обхода графа потока управления.

Построение такого графа требует незначительное количество ресурсов – памяти и процессорного времени.

Основные результаты работы.

- Предложен граф блокировок для многопоточных программ, расширяющий граф вызовов программы вершинами с информацией об инструкциях блокировки.
- Предложен алгоритм, выявляющий взаимные блокировки потоков.
- Разработанные алгоритмы реализованы в статическом анализаторе Svace, демонстрируют высокий уровень истинных срабатываний (60-90%) и занимают не более 4% времени работы всего инструмента.

Главное преимущество предложенного в работе подхода заключается в том, что описанные алгоритмы возможно реализовать в любом анализаторе, использующем анализ «снизу-вверх» на основе резюме. Такой подход является популярным из-за высокой скорости и масштабируемости анализа. Однако статические анализаторы, нацеленные на обнаружение взаимных блокировок потоков, используют обход графа вызовов сверху-вниз, что упрощает построение и анализ структур, аналогичных графу блокировок. Использование обхода сверху-вниз негативно сказывается на скорости и масштабируемости анализа, а также возможностях обнаружения других типов дефектов, не связанных с неправильной синхронизацией потоков.

Другими преимуществами предложенного подхода являются отсутствие необходимости ручного аннотирования кода, возможность анализировать как полные программы, так и отдельные библиотеки.

Список литературы/References

[1] А.П. Меркулов, С.А. Поляков, А.А. Белеванцев. Анализ программ на языке Java в инструменте Svace. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 57-74 / A.P. Merkulov, S.A. Polyakov, A.A. Belevantsev. Supporting Java programming in the Svace static analyzer. Trudy ISP RAN/Proc.ISP RAS, vol.29, issue 3, 2017, pp. 57-74 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-5.

[2] W.R. Bush, J D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. Software: Practice and Experience, vol. 30, no. 7, 2000, pp. 775-802.

[3] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In Proc. of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, 2003, pp. 327-336.

[4] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In Proc. of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2007, pp. 435-445.

[5] D. Babic and A.J. Hu. Calysto: scalable and precise extended static checking. In Proc. of the 30th international conference on Software engineering, 2008, pp. 211-220.

[6] В.К. Кошелев, И.А. Дудина, В.Н. Игнатьев, А.И. Борзилов. Чувствительный к путям поиск дефектов в программах на языке C# на примере разыменования нулевого указателя. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 59-86 / V. Koshelev, I. Dudina, V. Ignatyev, A. Borzilov. Path-Sensitive Bug Detection Analysis of C# Program Illustrated by Null Pointer Dereference. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015, pp.59-86 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-5.

[7] McPeak, C.-H. Gros, and M. K. Ramanathan. Scalable and incremental software bug detection. In Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering, 2013, pp. 554-564.

[8] R. Agarwal and S. D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In Proc. of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging, 2006, pp. 51-60.

[9] K. Havelund. Using runtime analysis to guide model checking of java programs. Lecture Notes in Computer Science, vol. 1885, 2000, pp. 245-264.

[10] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems (TOCS), vol. 15, no. 4, 1997, pp. 391-411.

[11] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. ACM SIGPLAN Notices, vol. 42, no. 6, 2007, pp. 245-255.

[12] C. Boyapati and M. Rinard. A parameterized type system for race-free java programs. In Proc. of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2001, pp. 56-69.

[13] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In Proc. of the IEEE 31st International Conference on Software Engineering, pp. 386-396.

[14] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In Proc. of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2006, pp. 308-319, 2006.

[15] S. D. Stoller. Model-checking multi-threaded distributed java programs. Lecture Notes in Computer Science, vol. 1885, 2000, pp. 224-244.

[16] T.A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In Proc. of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation, 2004, pp. 1-13.

Информация об авторах / Information about authors

Сергей Андреевич ПОЛЯКОВ – сотрудник. Сфера научных интересов: статический анализ, параллелизм, JVM языки.

Sergey Andreevich POLYAKOV – employee. Research interests: static analysis, concurrency, JVM languages.

Алексей Евгеньевич БОРОДИН – кандидат физико-математических наук, научный сотрудник. Сфера научных интересов: статический анализ исходного кода программ для поиска ошибок.

Alexey Evgenevich BORODIN – PhD, researcher. Research interests: static analysis for finding errors in source code.