Поиск состояний гонки в программах на языке Java при помощи динамического анализа

С. П. Вартанов <svartanov@ispras.ru>
М. К. Ермаков <mermakov@ispras.ru>
Факультет вычислительной математики и кибернетики,
Московский государственный университет им. М.В. Ломоносова,
119991, Россия, г. Москва, Ленинские горы, д. 1, с.52[1]

Аннотация. Язык программирования Java обладает встроенными средствами поддержки многопоточного выполнения программ. Из-за ошибок при создании подобных программ в процессе выполнения могут возникать ошибки синхронизации, одной из которых является возникновение состояния гонки между потоками, которые осуществляют доступ к разделяемому ресурсу (как минимум один поток модифицирует ресурс) и порядок работы потоков с ресурсом не фиксирован. В статье рассматривается подход к поиску состояний гонки при помощи динамического анализа. Преимущества динамического анализа по сравнению со статическим заключаются в отсутствии ложных срабатываний при определённых ограничениях, накладываемых на анализируемую программу. Для проведения динамического анализа предлагается использовать статическую инструментацию байт-кода программы, которая позволяет в ходе выполнения программы извлекать информацию о выполнении инструкций и методов, осуществляющих работу по синхронизации потоков. Построенная трасса представляет собой модель конкретного выполнения программы. На её основе с помощью отношений предшествования и механизма отслеживания блокировок определяется, возможна ли ситуация, при которой возникает состояние гонки. Для инструментации с целью сбора трассы используется инструмент динамического анализа Coffee Machine. Статическая инструментация, используемая в инструменте, позволяет проводить анализ программ на виртуальных машинах, не предоставляющих интерфейс для динамической инструментации. Анализ построенной модели и поиск потенциальных состояний гонки осуществляется при помощи инструмента ThreadSanitizer Offline. Благодаря использованию инструментации байт-кода наличие связи с исходным кодом для проведения анализа не является необходимым, однако это позволяет более точно определить причины возникновения

[1] Работа проводится при финансовой поддержке Российского фонда фундаментальных исследований, номер проекта 14-07-00609

ошибки. Реализация была проверена на ряде проектов с открытым исходным кодом и продемонстрировала свою эффективность для поиска состояний гонки.

Ключевые слова: динамический анализ, состояния гонки, ошибки синхронизации.

DOI: 10.15514/ISPRAS-2015-27(2)-3

Для цитирования: Вартанов С.П., Ермаков М.К. Поиск состояний гонки в программах на языке Java при помощи динамического анализа. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 39-52. DOI: 10.15514/ISPRAS-2015-27(2)-3.

1. Введение

В последнее время при создании программного обеспечения всё чаще используется многопоточность для увеличения производительности за счёт распараллеливания процессорных вычислений. Работа программы в нескольких потоках может способствовать увеличению её эффективности, но также приводит к росту сложности. Это в свою очередь затрудняет понимание работы программы, её разработку, поддержку и поиск в ней ошибок и уязвимостей.

Наиболее сложные для обнаружения программные ошибки при работе с потоками возникают, когда параллельно исполняемые потоки имеют доступ к одним и тем же данным или устройствам. В этом случае на программиста возлагается ответственность за обеспечение корректного взаимодействия между потоками. В противном случае в программе могут возникать ошибки, которые принято называть ошибками синхронизации. Такого рода ошибки могут быть разделены на четыре основных класса:

- взаимная блокировка (deadlock),
- состояние гонки (data race),
- нарушение атомарности операций,
- нарушение свойств детерминированности.

Для предотвращения ошибок синхронизации традиционно используется ряд механизмов, таких как: взаимоисключения, семафоры, события, критические секции и так далее. Корректное их использование может полностью устранить ошибки синхронизации, однако зачастую этому препятствуют размеры системы и сложность взаимодействия её компонентов.

Большинство современных языков программирования предоставляют широкий спектр механизмов для создания многопоточных программ. В этой статье будут рассмотрены принципы поиска состояний гонки в программах, которые представлены в виде байт-кода языка Java.

1.1 Механизмы создания многопоточных программ на языке Java

Для работы с потоками в Java используются стандартные классы java.lang.Thread и интерфейс java.lang.Runnable. Создание нового потока происходит с помощью создания класса, реализующего интерфейс Runnable, либо с помощью переопределения метода run() у потомка класса Thread. Управление потоками происходит с помощью методов run(), join(), sleep(). Методы stop(), suspend() и resume() были объявлены небезопасными, их использование не рекомендуется. Прерывание работы потока с помощью метода stop() приводит к принудительному снятию блокировок, которыми могут быть защищены объекты, находящиеся в несогласованном состоянии. [1] Использование методов suspend() и resume() зачастую приводит к возникновениям взаимных блокировок, поскольку ни один поток не может получить доступ к ресурсу, заблокированному прерванным потоком.

1.1.1 Синхронизация в Java

Синхронизация потоков

На уровне исходного кода для синхронизации потоков в библиотеке времени выполнения (runtime library) Java присутствует ряд стандартных механизмов. Методы wait() и notify() имеются у любого объекта и используются для ожидания одним потоком действий другого. Метод join() интерфейса Runnable позволяет потоку ожидать завершения другого потока.

Также методы классов из пакета java.util.concurrent представляют широкий спектр механизмов для межпоточного взаимодействия. Пакет содержит в себе реализацию таких синхронизационных примитивов, как семафора, циклического барьера и др.

Доступ к разделяемым ресурсам

Работа с мониторами отражена также в двух инструкциях Java байт-кода: monitorenter и monitorexit, которые считывают из вершины стека ссылку на объект, который собирается захватить монитор для выполнения следующих инструкций, либо, соответственно, освободить его.

Ключевое слово synchronized может быть применено к блоку инструкций программы или метода целиком. Работа его обеспечивается при помощи создания монитора для соответствующего блока. Выполнение секции возможно только для потока, объект которого захватил монитор. Ключевое слово volatile обязывает потоки использовать единственный экземпляр переменной.

1.2 Состояние гонки

В этой статье рассматриваются методы поиска ошибок типа состояния гонки. Под состоянием гонки мы будет понимать ситуацию, при которой два или более потоков обращаются к одним и тем же данным, причём как минимум один из них производит изменение этих данных и эти обращения не синхронизованы, т. е. последовательность их выполнения зависит исключительно от скорости работы отдельных компонентов программы и от принятия системным планировщиком решений о переключении потоков. В этом случае невозможно утверждать, произойдёт ли изменение данных одним потоком к моменту, когда они будут считаны или изменены другим потоком. Данный дефект может приводить как к незначительным нарушениям функциональности программы, так и к серьёзным программным сбоям вплоть до аварийного завершения.

Особенность этого типа дефектов заключается в том, что программная ошибка, к которой приводит дефект, проявляется нерегулярно и её обнаружение и воспроизведение может быть затруднительно как в процессе тестирования из-за особенностей работы планировщика потоков, так и в процессе отладки из-за различий в поведении программы при наличии отладчика.

1.3 Автоматический поиск ошибок

Методы автоматического поиска ошибок в программном обеспечении традиционно разделяют на методы динамического и статического анализа. Также, когда используются оба подхода, говорят о гибридном анализе. Под статическим анализом обычно понимают исследование программы без запуска её на выполнение, которое основывается исключительно на её исходном коде или другом представлении. В этом случае зачастую анализируется некоторая модель, построенная на основе исходного кода программы. Основными достоинствами статического анализа являются скорость его работы (вплоть до возможности анализа «на лету» в процессе редактирования исходного кода программы) и масштабируемость (возможность анализировать либо программу целиком, либо отдельные её части — модули, файлы, функции). К недостаткам статического анализа обычно относят возможность возникновения как ошибок первого рода (false positive), так и ошибок второго рода (false negative) связанных с недостаточной полнотой модели программы в статическом анализаторе.

1.3.1 Динамический анализ

Динамический анализ подразумевает запуск программы на исполнение. В ходе выполнения программы происходит анализ её состояния, проверка каких-либо условий, либо запись трассы событий. Динамический анализ отличается низкой скоростью работы из-за необходимости запуска программы

на выполнение, однако позволяет избежать ложных срабатываний в силу обнаружения ошибок непосредственно в процессе или по результатам работы программы.

Основным механизмом проведения динамического анализа является инструментация — процесс внедрения дополнительного программного кода с целью сбора информации о состоянии программы или трассе выполнения. Различают подходы статической и динамической инструментация. Статическая инструментация полностью завершается до начала выполнения программы, что позволяет избежать дополнительных накладных расходов в ходе выполнения программы. Динамическая инструментация производится непосредственно в ходе выполнения программы перед запуском целевого кода. Таким образом для её осуществления имеется вся необходимая информация.

Основной проблемой при проведении статической инструментации является необходимость на этапе до выполнения программы иметь необходимую информацию о целевых фрагментах кода и соответствующих инструментационных действиях.

2. Инструменты поиска синхронизационных ошибок

2.1 RoadRunner

Инструмент RoadRunner написан целиком на языке Java и представляет собой фреймворк для создания инструментов динамического анализа, предназначенных для поиска синхронизационных событий. Инструментация происходит на этапе загрузки классов при помощи библиотеки ASM [2]. Инструмент различает набор классов событий времени выполнения: доступ в память, блокировка, создание потока и т. д. Для проведения анализа используются так называемые теневые значения для объектов класса Thread, элементов памяти: полей классов, массивов и др. Создание инструментов на основе RoadRunner требует только описания обработчиков соответствующих событий.

Использование динамической инструментации требует дополнительных накладных расходов на этапе загрузки классов в виртуальной машине.

2.2 Sherlock

Инструмент Sherlock [3] представляет собой инструмент динамического анализа для поиска взаимных блокировок. Одним из основных понятий, используемых в инструменте является понятие расписания, которое определяется как последовательность событий. Для предварительного поиска потенциальных мест, в которых возможно возникновение взаимных блокировок, используется инструмент GoodLock.

Sherlock реализует механизм итеративного динамического анализа. На первой итерации программа выполняется на начальных входных данных. На каждой следующей итерации происходит выполнение инструментированной версии программы и записывается трасса, в которую входят синхронизационные события, а также информация об ограничениях, накладываемых на путь выполнения. Полученное в итоге расписание преобразуется с целью достижения ситуации взаимной блокировки потоков. Для воспроизведения полученного расписания строятся наборы входных данных для программы при помощи SMT-решателя на основе построенных ограничений пути. Это становится возможным за счёт контролирования инструментом системного планировщика потоков.

2.3 ThreadSanitizer

Проект ThreadSanitizer посвящён поиску состояний гонки в программах. Он представляет собой набор инструментов динамического анализа, которые производят построение модели конкретного пути выполнения программы и на основе этой модели осуществляют поиск состояний гонки.

В проекте используются два основных подхода к поиску состояний гонки:

- установление отношений предшествования между синхронизационными событиями,
- отслеживание множества блокировок для каждого события.

Основной инструмент проекта осуществляет поиск состояний гонки в программах, представленных в бинарном коде. В первых версиях инструмента этот анализ происходил на основе фреймворков динамической инструментации Valgrind [7] и PIN [8]. Более поздняя версия основывается на статической инструментации, что позволяет снизить накладные расходы.

В проект также входят два экспериментальных инструмента. Один из них — ThreadSanitizer Offline — представляет из тебя утилиту, которая на основе трассы событий делает вывод о наличии или отсутствии в ней состояния гонки. Второй — Java ThreadSanitizer — экспериментальный динамический анализатор, который использует интерфейс утилиты ThreadSanitizer Offline для поиска состояний гонки в программах на языке Java.

В инструменте Java ThreadSanitizer применяется динамическая инструментация при помощи библиотеки ASM [2]. Это не позволяет применять инструмент для поиска ошибок на программах, которые запускаются на виртуальных машинах с отсутствующей опцией javaagent. Динамическая инструментация также приводит к дополнительным накладным расходам в ходе выполнения программы, которых можно избежать с применением инструментации на начальном этапе анализа, поэтому от использования инструмента Java ThreadSanitizer было решено отказаться.

3. Используемые подходы

3.1 Отношение предшествования

Отношение предшествования определяется как отношение строгого частичного порядка на множестве событий на одном из путей выполнения программы. Т. е. такое отношение, что

$$\forall x : x \prec x,$$
$$\forall x, y : x \prec y \wedge y \prec x \Rightarrow x = y,$$
$$\forall x, y, z : x \prec y \wedge y \prec z \Rightarrow x \prec z.$$

Согласно определению, событие A предшествует событию B, если оно предшествует ему в рамках одного потока, либо событие A — событие отправки сообщения, а событие B — приёма того же сообщения, либо существует событие C, такое что A предшествует C, а C предшествует B (свойство транзитивности). Другими словами, событие A предшествует событию B, если событие A может повлиять на событие B. [4]

3.2 Отслеживание блокировок

Модель отношений предшествования учитывает синхронизацию событий при помощи отправки и получения сообщений. Для синхронизации с использованием блокировок используется механизм, который строит для каждого объекта в ходе выполнения программы множество блокировок, которые захвачены объектом в данный момент.

Совместное использование двух описанных механизмов позволяет инструменту ThreadSanitizer производить эффективный поиск состояний гонки. В терминах ThreadSanitizer состоянием гонки считается ситуация, при которой существуют два или более событий доступа к одним данным, которые происходят в разных потоках, по крайней мере одно из которых — операция записи и которые не связаны отношением предшествования.

4 Реализация поиска состояний гонки

Поиск состояний гонки осуществляется при помощи совмещения работы двух инструментов: Coffee Machine и ThreadSanitizer Offline.

4.1 Инструментация

Модульная структура инструмента Coffee Machine позволяет реализовать на его основе компоненты для печати трассы с синхронизационными событиями в ходе выполнения программы.

В качестве параметров инструменту Coffee Machine передаётся путь к директории, в которой располагаются класс-файлы либо JAR-файлы, в которых содержится целевой байт-код, а также список методов, которые

необходимо проинструментировать.

В зависимости от того, известен ли пользователю исчерпывающий список методов, которые будут вызваны в ходе выполнения программы, передаваемый список методов воспринимается иснтрументом либо как окончательный, либо как начальный набор единиц инструментации. В первом случае процесс инструментации заканчивается после прохода по полученному списку. Во втором случае запускается итеративный процесс поиска необходимых методов, на каждой итерации которого выбранный из списка метод инструментируется, удаляется из списка, и в список добавляются все методы, которые не были проинструментированы и вызовы которых присутствуют в текущем методе. Во втором случае, в зависимости от особенностей анализируемой программы, может быть проинструментировано значительное количество методов, которые никогда не будут вызваны в силу их недостижимости при заданных входных данных программы, однако следует заметить, что это влияет лишь на размер результирующих класс-файлов и на время работы инструментатора на начальном этапе. В силу того, что методы в языке Java могут быть вызваны динамически, статическое построение полного списка может быть невозможно. В связи с этим, если в ходе выполнения программы, вызывается метод, который не был ранее проинструментирован, выполнение может быть прервано для дополнительной инструментации.

4.1.1 Инструментация отдельных инструкций

Инструментация представляет собой внедрение дополнительной функциональности в байт-код таким образом, чтобы это в как можно меньшей степени влияло на изначальную функциональность анализируемой программы. О полном отсутствии влияния говорить не приходится, поскольку любые дополнительные действия в любом случае занимают процессорное время, что приводит к замедлению работы программы, часто неравномерному. Этот эффект является одной из основных проблем динамического анализа в целом и может решаться различными способами, в зависимости от задачи, которая встаёт перед анализом. Так, для избавления от неравномерности замедления, в программу могут быть принудительно добавлены задержки, либо же для нивелирования воздействия инструментации на системные механизмы они могут быть откалиброваны с учётом особенностей инструментации на уровне интерпретатора языка или операционной системы. К таким механизмам можно отнести системные часы, планировщик выполнения потоков, сборщик мусора и т. д.

Для выбранного подхода поиска ситуаций состояния гонки эта проблема актуальна в меньше степени из-за особенностей механизма проверки, построенной в ходе выполнения модели работы программы. В силу этих особенностей, результирующая модель не зависит от работы системного планировщика выполнения потоков и поэтому влиянием временных задержек можно пренебречь.

Инструмент Coffee Machine посредством использования библиотеки BCEL [5] предоставляет интерфейс для внедрения в байт-код для каждого типа инструкции при определённых условиях двух дополнительных наборов инструкций: предварительного и заключительного. Каждый из двух наборов при этом может быть пустым. Если оба набора пусты, это означает, что для данного типа инструкций инструментационного кода не создаётся и этот тип не представляет интереса с точки зрения анализа программы.

В рамках архитектуры инструмента Coffee Machine для определённого типа инструментации создаётся собственный класс. Класс для печати синхронизационных событий имеет имя Concurrency и содержит в себе всю необходимую функциональность. Непосредственно в предварительный и заключительный наборы инструкций, которые будут добавлены до и после целевой инструкции соответственно, входят инструкции дублирования элементов стека, загрузки константных значений и инструкция вызова инструментационной функции класса Concurrency.

Например, для инструкции байт-кода monitorenter, существуют оба набора инструкций. Предварительный набор состоит из одной инструкции дублирования вершины стека (DUP) для передачи в инструментационный код ссылки на объект, а в заключительный набор входят инструкции добавления в стек текущего значения симулятора счётчика команд (PUSH) и инструкция вызова метода monitorEnter(Object, int) класса Concurrency (инструкция INVOKESTATIC). Таким образом, вместо одной инструкции monitorenter, в байт-коде инструментированной программы будут четыре инструкции:

```
DUP
MONITORENTER
PUSH <program counter>
INVOKESTATIC
   ru.ispras.coffeemachine.target.Concurrency.monitorEnter
   (Ljava/lang/Object;I)V
```

Листинг 1. Список инструкций байт-кода, на которые заменяется инструкция MONITORENTER

Хотя бы один из инструментационных наборов не пуст для следующих инструкций:

загрузки значения в стек,сохранения значения в локальную переменную,вызова функции и выход из неё,обращения к полям объектов,выброса исключенияmonitorenter, monitorexit.

Оба блока инструментационного кода будут сгенерированы для инструкций вызова функции и инструкции monitorenter.

Методы класса Concurrency, вызовы которых присутствуют в инструментационном коде, производят действия по обработке переданных им параметров, обновлению соответствующих объектов и печати строки

синхронизационного события. Информация о синхронизационных событиях записывается в стандартный поток вывода с определённым префиксом, что позволяет отличить её от вывода программы. В этой строке содержится тип события, номер текущего потока, значение симулятора счётчика команд и дополнительные параметры события, если такие есть. Формат данных соответствует формату входных файлов инструмента ThreadSanitizer Offline, который производит построение модели выполнения программы.

```
...
UNLOCK 0 8be 2a675b7986 0
SBLOCK_ENTER 0 8c 0 0
WRITE 0 8c 2f9ee1ac00000003 1
SBLOCK_ENTER 0 8c 0 0
RTN_CALL 0 0 0 0
SBLOCK_ENTER 0 88 0 0
RTN_EXIT 0 88 0 0
SBLOCK_ENTER 0 8c 0 0
READ 0 8c 2f9ee1ac00000000 1
THR_START 1 0 0 0
THR_FIRST_INSN 1 8c 0 0
SBLOCK_ENTER 0 8c 0 0
...
```

Листинг 2. Трасса синхронизационных событий

В приведённом примере присутствуют события обращения к объекту на чтение (READ), запись (WRITE), вызов функции (RTN_CALL) и выход из неё (RTN_EXIT), снятие блокировки (UNLOCK), начало нового потока (THR_START и THR_FIRST_INSN).

Симулятор счётчика команд — простая целочисленная статическая переменная класса-инструментатора, её значение известно на этапе инструментирования кода и в инструментационном коде она представляет собой константное значение. Это значение будет добавлено в строковое представление инструментационного события и будет использовано для связывания этого события с инструкцией байт-кода, которая привела к возникновению этого события. Инструкция байт-кода в свою очередь при наличии соответствующей информации в класс-файле может быть связана со строкой исходного кода программы, что позволит определить место возникновения событий, которые приводят к состоянию гонки. Номер потока вычисляется динамически, в ходе исполнения инструментационного кода.

4.1.2 Инструментация инструкций вызова метода

Особая работа производится с инструкциями вызова метода. Инструментация этой инструкции не может быть универсальной и производится в основном для покрытия функциональности методов из пакета java.util.concurrent. Также

инструментация производится для вызовов базовых механизмов взаимодействия потоков из пакета java.lang и для вызовов набора методов пакета org.jtsan, которые могут быть встроены пользователем в программу для управления ходом работы анализа.

До операции вызова в байт-код внедряются операции дублирования вершины стека — параметров вызываемого метода. Если инструментационные действия необходимо производить и до, и после вызова, вершина стека дублируется дважды. Затем происходит вызов предварительного инструментационного кода, если он не пуст. После выполнения возвращение из целевого метода происходит вызов заключительного инструментационного кода, опять же, если он не пуст. Метод, реализующий заключительный набор инструментационных инструкций, обязан возвращать значение метода, которое он получает на вход вместе с продублированными параметрами, если метод не объявлен с ключевым словом void.

Из пакета java.util.concurrent инструментируются методы классов

- locks.AbstractQueuedSynchronizer,
- locks.Condition,
- locks.ReentrantLock,
- locks.ReentrantReadWriteLock,
- locks.ReentrantReadWriteLock_ReadLock,
- locks.ReentrantReadWriteLock_WriteLock,
- locks.LockSupport,
- CountDownLatch,
- CyclicBarrier и
- Semaphore.

Также частично осуществляется поддержка системных коллекций.

4.2 Поиск ошибок

Сгенерированная описанным путём трасса подаётся на вход инструменту ThreadSanitizer Offline. В случае обнаружения состояния гонки, инструмент выдаёт сообщение, в котором указывается тип ошибки: сколько потоков участвует в гонке, какие из потоков обращаются к данным посредством чтения, а какие посредством записи. Также добавленные в трассу ссылки на исходный код программы, если он доступен, указывают программисту на место в коде, где произошла ошибка.

5. Результаты

Работа инструмента была проверена на ряде проектов с открытым исходным кодом. Для проверки базовой функциональности инструмента был расширен набор стандартных тестов инструмента Java ThreadSanitizer. Инструмент

продемонстрировал эффективность поиска дефектов, как на проектах с искусственно внесёнными ошибками, так и на проектах без модификаций. Основные исследования проводились на стандартных приложениях платформы Android, которые в силу использования графического пользовательского интерфейса активно используют механизмы многопоточности.

В результате были обнаружены ситуации состояния гонки, одна из которых приводит к аварийному завершению приложения. Для воспроизведения обнаруженных дефектов на сегодняшний момент автоматические средства не реализованы, и эта задача решается вручную путём внедрения в программу временных задержек. Наличие в класс-файлах информации о связи байт-кода с исходным кодом программы не является обязательным для работы механизма поиска дефектов, однако это существенно упрощает для разработчика поиск причин возникновения дефекта и его воспроизведения.

6. Заключение

В статье была рассмотрена реализация механизма поиска состояний гонки в программах на языке Java, либо программах, которые могут быть конвертированы в Java байт-код. Механизм представляет собой динамический анализ трассы выполнения программы с её предварительной статической инструментацией, — набора событий, которые представляют интерес с точки зрения взаимодействия между потоками. На основе полученной трассы сторонним инструментом ThreadSanitizer Offline производится построение модели выполнения программы и поиск состояний гонки. Проведённые исследования показали эффективность работы инструмента для поиска состояний гонки в многопоточных программах.

Список литературы

[1]. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, 21(7), 1978. pp. 558–565. doi: 10.1145/359545.359563

[2]. Java Thread Primitive Deprecation [HTML] (http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html)

[3]. K. Serebryany and T. Iskhodzhanov. ThreadSanitizer – data race detection in practice. Proceedings of the Workshop on Binary Instrumentation and Applications, 2009. pp. 62-71. doi: 10.1145/1791194.1791203

[4]. Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel)

[5]. Bruneton E. ASM 4.0. A Java bytecode engineering library, 2011 [PDF] (http://download.forge.objectweb.org/asm/asm4-guide.pdf)

[6]. M. Eslamimehr, J. Palsberg. Sherlock: scalable deadlock detection for concurrent programs // Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014. P. 353–365

[7]. Nethercote N., Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation // PLDI. 2007.

[8]. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation // PLDI. 2005.

Detecting Race Conditions in Java Programs Using Dynamic Analysis

S. Vartanov <svartanov@ispras.ru>
M. Ermakov <mermakov@ispras.ru>
Department of Computational Mathematics and Cybernethics, Moscow State University, 1/52, Leninskie Gory, Moscow, Russia, 119991

Abstract. Multithreading support is one of Java programming language built-in features. Synchronization defects are a serious issue despite the extensive thread processing API provided by standard classes. One of the most common synchronization defects is a race condition. It arises when two different threads read and modify shared data and data access order is not fixed. This paper focuses on an approach to automatically detect race conditions using dynamic analysis. The main advantage of dynamic analysis is the lack of false positives if program under analysis fits a number of requirements. Our approach for dynamic analysis uses static bytecode instrumentation to extract execution trace at run-time which allows to track usage for thread synchronization methods and functions. Generated trace is a concrete execution model which is processed to identify happen-before relations and lock sets for possible data race detection. We use Coffee Machine tool for static instrumentation. Static instrumentation approach is applicable even to virtual machines which do not provide support for dynamic instrumentation. We use ThreadSanitizer Offline to process generated trace and detect race conditions. While debug information in bytecode is unnecessary it provides more precise error messages. Described tool chain has been tested on a set of open source projects.

Keywords: dynamic analysis, race condition, synchronization defects.

DOI: 10.15514/ISPRAS-2015-27(2)-3

For citation: Vartanov S.P., Ermakov M.K. Detecting Race Conditions in Java Programs Using Dynamic Analysis. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 39-52 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-3.

References

[1]. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, 21(7), 1978. pp. 558–565. doi: 10.1145/359545.359563

[2]. Java Thread Primitive Deprecation [HTML] (http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html)

[3]. K. Serebryany and T. Iskhodzhanov. ThreadSanitizer – data race detection in practice. Proceedings of the Workshop on Binary Instrumentation and Applications, 2009. pp. 62-71. doi: 10.1145/1791194.1791203

[4]. Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel)

[5]. Bruneton E. ASM 4.0. A Java bytecode engineering library, 2011 [PDF] (http://download.forge.objectweb.org/asm/asm4-guide.pdf)

[6]. M. Eslamimehr, J. Palsberg. Sherlock: scalable deadlock detection for concurrent programs // Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014. P. 353–365

[7]. Nethercote N., Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation // PLDI. 2007.

[8]. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation // PLDI. 2005.