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

Кафедра компьютерных технологий

Роскошный Яков Игоревич

Применение статического анализа для оптимизации динамического поиска гонок.

Бакалаврская работа

Научный руководитель: Д. И. Цителов

Санкт-Петербург
2015

Содержание

Введение

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

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

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

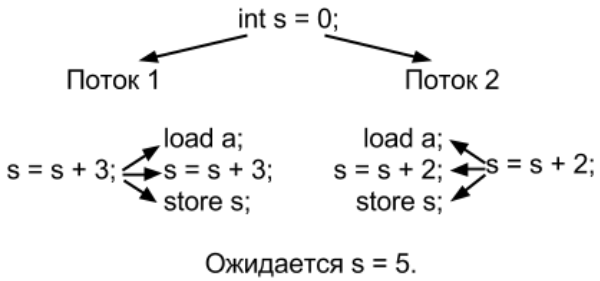

Рис. 0.1: Состояние гонки в программе

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

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

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

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

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

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

Статический анализ не наносит ущерб исполнеию программы. С другой стороны, во время статического анализа можно выделить поля, при обращении к которым не может возникнуть состояние гонки.

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.

Глава 1. Обзор

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

1.1. Методы автоматического обнаружения гонок

1.1.1. Статический подход

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

1.1.2. Динамический подход

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

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

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

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

1.2. Динамический детектор гонок для Java программ

1.2.1. Модель памяти Java

Для языка программирования Java существует спецификация его модели памяти(Java memory model), которая входит в стандарт языка. Данная спецификация содержит архитектурно-независимые гарантии исполнения многопоточных программ. Для загрузки изменений из памяти потока в общую память программы, а также для загрузки чужих изменений из общей памяти программы в память потока есть операции синхронизации.

1.2.2. Динамический детектор jDRD

Описание jDRD

1.2.3. Возможности статического анализа для оптимизации jDRD

Как видно из описания работы jDRD, каждое поле является потенциальным местом возникновения гонки. Следовательно для каждого поля в jDRD приходится хранить векторные часы. Статическим анализом можно выяснить, что некоторые поля корректо-синхронизированы - то есть, при обращении к ним невозможно состояние гонки. Это позволит уменьшить потребление памяти и времени детектора. Также статическим анализом можно выделить корректно-синхронизированные последовательности операций - последовательность операций, которая гарантировано исполняется в одном потоке.

1.3. Статический анализ для поиска корректно-синхронизированного кода

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-сихронизированного кода 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 выполнения (control flow graph). На его основе может быть построен граф использования объектов(object use graph).

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(intermediate representation).

Глава 2. Алгоритм поиска корректно-синхронизированных полей

2.1. Получение графа потока управления

2.2. Описание алгоритма

2.2.1. Построение множества возможных блокировок

2.2.2. Обход CFG

2.3. Реализация алгоритма

2.3.1. Тестирование

2.4. Практическое применение

Заключение

Текст разный [1].

Источники

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