инженерия программного обеспечения

Трифанов Виталий Юрьевич,
Цителов Дмитрий Игоревич

УДК 004.451.2

СТАТИЧЕСКИЕ И POST-MORTEM МЕТОДЫ ОБНАРУЖЕНИЯ ГОНОК В ПАРАЛЛЕЛЬНЫХ ПРОГРАММАХ

Аннотация

Одной из самых сложных и трудновоспроизводимых ошибок в многопоточных программах являются состояния гонки (data race) – несинхронизированные обращения к одному и тому же участку памяти из различных потоков, из которых одно является записью данных. Обычно гонки слабо локализуемы, и ведут к повреждению глобальных структур данных, а их «ручное» обнаружение сильно затруднено. В этой области было проведено множество различных исследований, но автоматическое обнаружение гонок остается актуальной задачей. В данной статье рассматриваются статические и post-mortem методы обнаружения гонок, отдельное внимание уделяется вопросу поиска гонок в Java-приложениях.

Ключевые слова: многопоточность, параллельное программирование, автоматическое обнаружение ошибок, состояние гонки, обзор.

ВВЕДЕНИЕ

Используемые в настоящее время вычислительные устройства всё в большей степени являются многоядерными и многопроцессорными, что позволяет разрабатывать для них эффективные параллельные программы. Однако проектирование программы в терминах нескольких взаимодействующих потоков управления, одновременно выполняющих разные действия, является сложной задачей, что приводит к большому количеству ошибок. К самым частым и сложным в обнаружении ошибкам многопоточных программ относятся состояния гонки (data race)[1] – несинхронизированные обращения из различных потоков к одному и тому же участку памяти [34][2].

Единственным способом гарантировать отсутствие ошибок синхронизации является полное исключение доступа к механизмам возникновения таких ошибок из языка программирования. По этому пути идут некоторые современные языки – Erlang, Jocaml и другие. Однако большинство используемых в индустрии языков, таких как

[1] Термин «состояние гонки» является общепринятым переводом английского термина «data race» и используется как в отечественных научных статьях – см, например, [1] – так и на различных Интернет-ресурсах, в частности в MSDN и Wikipedia.

[2] Так, например, многопоточное программирование активно используется при разработке системного программного обеспечения [3], где особенно актуально решение проблемы обеспечения надёжности. Типичным примером являются различные задачи управления ресурсами, например, сборка мусора, которая для большинства платформ (в частности, в Java) реализована как фоновый поток.

C/C++, Java, C#, базируются на модели разделяемой памяти и не предоставляют подобной защиты.

Для избегания гонок разрабатываются дополнительные библиотеки классов и аннотаций, например Intel TBB [29] или механизмы из пакета java.util.concurrent [14], появившиеся в Java 1.5. Такие средства не дают гарантии отсутствия гонок, но помогают программисту создавать более правильный и корректный код. Другой подход – верификация программ посредством построения формальной модели программы и последующего доказательства её корректности. Существует ряд утилит [16, 25], реализующих такой подход и показавших свою состоятельность на небольших программах и модулях. Как правило, подобные утилиты используются для проверки программ, в которых цена ошибки очень высока. Например, NASA использовало утилиту JavaPathFinder для поиска ошибок в программном обеспечении космического исследовательского модуля [25]. К сожалению, задача проверки существования потенциальных состояний гонки в программе и родственная задача обнаружения тупиков являются NP-полными для конечных графов выполнения и алгоритмически неразрешимы в общем случае [33, 34]. Поэтому верифицирующие утилиты требуют колоссального количества ресурсов, экспоненциально возрастающего с увеличением размера программы, и неприменимы даже на программах среднего размера.

Таким образом, задача автоматического обнаружения гонок является признанно сложной и актуальной, и исследования здесь ведутся уже более двух десятков лет. За это время было создано множество детекторов гонок, использующих различные подходы, методы и алгоритмы, опубликовано более пятидесяти статей. В данной статье мы рассматриваем следующие подходы к обнаружению гонок – анализ программы без её запуска (статический) и обнаружение гонок уже после завершения программы на основании собранных во время её работы данных (post-mortem). Насколько нам известно, не существует обзоров детекторов гонок, покрывающих исследования последних лет. В процессе изучения современных подходов к данной области мы обнаружили, что последний[1] обзор был сделан в 2005 году [37], а с тех пор было опубликовано много работ, содержащих как новые идеи, так и качественные улучшения существующих методик и инструментов.

СОСТОЯНИЯ ГОНКИ

Семантика многопоточных программ во многом определяется архитектурно-зависимой моделью памяти (memory model) – набором правил, по которым потоки могут взаимодействовать друг с другом посредством общей памяти. К сожалению, правила взаимодействия, заданные архитектурой, и простые контракты синхронизации по типу volatile-переменных в языке C не обеспечивают достаточной детерминированности поведения программ, поскольку не накладывают достаточных ограничений на возможные оптимизации, применяемые компилятором при генерации кода, такие как переупорядочивание кода, кэширование значений, упреждающее чтение и т. п. В результате формальная семантика существенного числа традиционно применяемых языков программирования не обладает свойством последовательной консистентности (sequential consistency)[2] в условиях многопоточного исполнения. Это означает, что в общем случае путь выполнения программы не может быть описан как чередование операций в различных потоках, поскольку «видимость» изменений, производимых в общей памяти, остаётся недетерминированной [8]. Осознав эту проблему, разработчики современных языков программирования стали вносить в спецификацию языка собственную архитектурно-независимую модель исполнения, которая позволяла бы описывать поведение многопоточных программ достаточно де-

[1] Среди более поздних работ, касающихся обнаружения гонок, нам удалось обнаружить лишь две, посвященные проблематике в целом, – в [27] авторы сравнивали утилиты для обнаружения гонок в программах, использующих OpenMP, в [6] кратко описывается текущее состояние предметной области.

[2] Перевод взят из монографии [2].

терминировано. Одним из первых общепризнанных языков с такой формальной моделью стал язык Java (начиная с версии 1.5) [24].

Согласно модели памяти Java, каждому потоку при его запуске выделяется отдельный фрагмент памяти, доступный только ему, – так называемая собственная память потока. Изначально в памяти потока хранятся значения разделяемых переменных, скопированные из общей памяти программы. Когда поток обращается к разделяемой переменной на чтение, на самом деле он обращается к своей копии переменной, находящейся в его памяти. При изменении значения разделяемой переменной поток также изменяет значение своей копии переменной. Для публикации изменений, сделанных потоком, а также для получения изменений, сделанных другими потоками, существуют операции синхронизации – загрузка изменений из памяти потока в общую память программы и, наоборот, загрузка чужих изменений из общей памяти программы в память потока. Без осуществления должной синхронизации возможны проблемные ситуации, когда потоки «не видят» изменений, сделанных друг другом.

Состояние гонки наступает, когда два или более потока в параллельной программе одновременно обращаются к одной структуре данных в общей памяти, между этими обращениями нет принудительного упорядочивания по времени, и хотя бы одно из них – обращение на запись [34].

Гонки могут быть очень опасными. Например, пусть два разных потока пытаются одновременно увеличить сумму банковского счета на x и y рублей соответственно. При отсутствии должной синхронизации вместо итогового увеличения на $x + y$ рублей может произойти увеличение лишь на x или на y (см. рис. 1). Если же речь идет о медицинских системах, то гонки могут привести к ещё более серьёзным последствиям. Например, от передозировок, допущенных аппаратом лучевой терапии Therac-25 по причине гонок, скончалось, как минимум, два человека [31].

Конкретный порядок выполнения потоков, приводящий к состоянию гонки, специфичен для каждого запуска программы и зависит от порядка выполнения операций в потоках, определяемого планировщиком задач операционной системы. Поэтому состояния гонки трудновоспроизводимы даже при последовательных запусках одной и той же программы с одинаковыми входными данными. Более того, как правило, состояния гонки слабо локализованы во времени и, являясь причиной повреждения глобальных структур данных, не приводят к немедленным заметным ошибочным действиям программы – она будет продолжать выполняться, что впоследствии может привести к трудно объяснимым сбоям.

ПОДХОДЫ К ОБНАРУЖЕНИЮ ГОНОК

Подходы к обнаружению гонок традиционно принято классифицировать по тому, в какое время относительно этапа выполнения программы осуществляется анализ. Выделяют следующие типы детекторов гонок [37]:

– статические (static, ahead-of-time) – не требуют запуска программы, анализируют исходный код программы или скомпилированные файлы, не запуская их [9, 17, 21, 30, 32];

– post-mortem – собирают информацию во время выполнения программы, но анализируют её уже после завершения работы [11, 38];

– динамические (dynamic, on-the-fly) – выполняются одновременно с программой, отслеживая синхронизационные события и обращения к разделяемым переменным [10, 12, 15, 20, 35, 39].

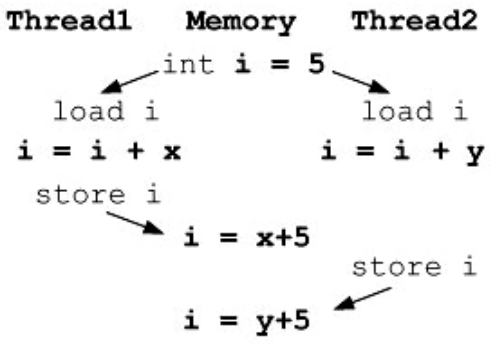

Рис. 1. Пример гонки в Java-программе. Оба потока увеличивают значение переменной i на x и y, соответственно, но в итоге оно увеличивается не на $(x + y)$, а только на y

В этой работе мы остановимся на кратком обзоре статических и post-mortem походов с акцентом на обнаружение гонок в Java-приложениях. Утилиты, которые будут рассмотрены в данной работе, представлены в табл. 1.

СТАТИЧЕСКИЕ ПОДХОДЫ К ОБНАРУЖЕНИЮ ГОНОК

Статические детекторы анализируют исходный код программы и строят её модель, основывающуюся, как правило, на графе потока выполнения (control flow graph) и графе использования объектов (object use graph) [36]. С помощью этой модели анализируются синхронизационные события и формируется множество обращений к данным, которые могут быть вовлечены в гонки. Эти средства обладают существенно меньшей точностью, чем динамические детекторы, поскольку они не располагают информацией о ходе выполнения программы. Однако, они имеют значительные преимущества при анализе больших объемов кода. Во-первых, статические детекторы не воздействуют на программу во время её выполнения и не замедляют ход её работы. Во-вторых, они анализируют весь код программы, в том числе и тот, который выполняется очень редко, и могут обнаружить те гонки, которые сложно обнаружить с помощью динамических утилит.

Многие статические методики основываются на расширении существующей системы типов языка[1] . В работе [9] представлена новая статическая система типов для языка Java, позволяющая предотвратить как гонки, так и взаимные блокировки (deadlocks). Это исследование основывается на идее, что корректно типизированные программы гарантированно не будут содержать подобных дефектов. Предложенная в работе система типов позволяет программисту описать используемые в программе правила синхронизации. Для избегания взаимных блокировок предоставляется возможность ввести несколько классов эквивалентности на множестве блокировок и ввести отношение частичного порядка на множестве этих классов. Допускаются динамические изменения этого отношения, если они не приводят к образованию циклов. Вводятся дополнительные типы для привязки объекта к потоку (это означает, что поток владеет объектом, то есть объект локален для него). Это позволяет статически вычислить принадлежность объектов к тем или иным потокам. Авторами был реализован JVM-совместимый прототип, который транслиру-

Табл. 1. Статические и post-mortem детекторы гонок

Статические детекторы		
Название утилиты	**Язык программирования**	**Год появления**
RaceFree Java type system	Java	2001
ESC/Java	Java	2001
RacerX	C++	2003
Chord	C++, Java	2006
Rccjava	Java	2000, 2006
Post-mortem детекторы		
Déjà vu	Java	1998
RecPlay	Java	1999

[1] Несмотря на ряд преимуществ, этот подход не нашел широкого применения в практическом программировании, главным образом, благодаря большому количеству ручной работы (вставка типовых аннотаций в исходный код программы), которую нужно проделать программисту. В настоящее время ведутся различные исследования, направленные на преодоление этого недостатка – например, в работе [4] предлагается перспективный подход, в котором программист должен выполнить аннотирование лишь для небольшой части программы, а препроцессор добавит в код динамические проверки корректности для тех случаев, когда статической проверки типов недостаточно.

ет корректно составленные программы в байт-код и может быть выполнен обычной JVM. Полученная реализация поддерживает все существовавшие на момент выхода работы (2001 год) механизмы синхронизации в языке Java. Предложенная система типов получилась достаточно эффективной, но при использовании требует большого количества аннотаций, которые программистам нужно «вручную» вставлять в исходный код. Кроме того, для поддержки такой системы типов требуется специальный компилятор или промежуточный транслятор.

В работе [23] развивается идея о том, что отсутствие гонок не является ни необходимым, ни достаточным условием отсутствия ошибок, возникающих по причине нежелательных взаимодействий потоков. Авторы рассматривают более сильное условие атомарности блоков (atomicity). Это означает, что последовательное выполнение инструкций атомарного блока кода одним потоком не может быть прервано другим потоком. Авторы представляют систему типов для задания и верификации свойств атомарности. Эта система типов позволяет помечать блоки кода и функции ключевым словом atomic и гарантирует, что при любом чередовании потоков во время произвольного запуска программы существует эквивалентный путь выполнения, в котором инструкции в каждом атомарном блоке выполняются одним потоком без прерывания. Такое свойство позволяет программистам исследовать поведение программ на более высоком уровне абстракции, где каждый атомарный блок выполняется «в один шаг», что значительно упрощает как неформальные, так и формальные рассуждения о корректности программы. Доказательство корректности системы типов базируется на теореме о сокращении (reduction theorem) Коэна-Лампорта [13]. Преимущества системы типов проиллюстрированы на стандартном библиотечном Java-классе java.util.Vector. Достоинство предлагаемого подхода состоит в том, что трактовка атомарного блока кода как одной потокобезопасной инструкции приводит к существенному упрощению рассуждений о взаимодействиях потоков. Однако соответствующая система типов должна быть использована вместе с другими утилитами по обнаружению гонок и не гарантирует их отсутствие сама по себе. Кроме того, подход требует использования специального языка программирования или модификации существующего для реализации этой системы типов.

Ещё один подход к статическому обнаружению гонок в Java-программах предложен в работе [21]. В рамках этого подхода проверяется выполнение правил блокировки путём отслеживания переменных блокировки, которыми защищаются разделяемые поля и проверки того, что нужная переменная блокировки захвачена при обращении к полю (аналог динамического алгоритма lockset). Для выполнения этих проверок авторами реализовано расширение системы типов языка Java, которое покрывает многие популярные приемы (patterns) синхронизации, такие как классы с внутренней синхронизацией, классы, требующие синхронизации на стороне клиента и локальные классы потока. Предоставляются механизмы для отключения системы типов в тех местах, где она накладывает чрезмерно строгие ограничения, а также в ситуациях, когда гонки безопасны (benign races). Реализованный авторами статический детектор RccJava был проверен более чем на 40 тысячах строках кода. Несколько гонок было обнаружено в стандартных Java-библиотеках. Утилита является более эффективной для обнаружения гонок, чем утилиты, верифицирующие порядок событий, но требует «ручного» аннотирования когда – примерно 20 аннотаций на 1000 строк кода.

Впоследствии авторы RccJava дополнительно оптимизировали утилиту для лучшей применимости к большим объемам кода, добавив автоматическое выведение предположительных аннотаций и пользовательский интерфейс, упрощающий анализ выдаваемых утилитой предупреждений [19]. Для упрощения практического анализа больших программ авторы разработали механизм для создания аннотаций Houdini/rcc, базирующийся на библиотеке Houdini [22], который автоматически вставляет аннотации в анализируемые программы. Дополнительно был использован ряд методик для уменьше-

ния количества ложных срабатываний, вызванных автоматическим аннотированием. Улучшенная версия RccJava [5] предоставляет информацию о потенциальных гонках в удобном виде через простой интерфейс. Дополнительно она группирует гонки по причине их происхождения, чтобы программист мог просматривать связанные гонки в совокупности. Однако утилита всё ещё ограничена рассмотрением только тех механизмов синхронизации, которые основаны на захвате блокировок. Для поддержки иных механизмов синхронизации нужно создавать дополнительные аннотации «вручную», что очень трудоёмко.

Другой подход к статическому обнаружению гонок, основанный на автоматическом доказательстве теорем, используется в утилите ESC/Java [30]. Новые версии утилиты [18] помимо отслеживания незащищённого доступа к переменным обладают возможностью проверки более сильных условий. К сожалению, этот подход также требует значительного количества «ручного» аннотирования для передачи информации детектору и уменьшения количества ложных срабатываний.

В работе [17] представлена статическая утилита RacerX для C-программ, основанная на подходе lockset. RacerX фактически не требует аннотирования, в отличие от ранее рассмотренных утилит. Единственным требованием является необходимость указать пары методов, которые используются для захвата и освобождения блокировок, а также (опционально) тип блокировки: циклическая блокировка (spin lock), приостановление потока (blocking), использование повторных попыток в случае неудачи (reentrance). Как правило, полученная таблица таких пар содержит не более 30 записей, что существенно снижает требуемое количество «ручной» работы в случае больших систем. На первом этапе RacerX обходит все файлы с исходным кодом и строит, используя такую таблицу, граф потока управления программы (control flow graph). Этот граф содержит информацию о вызовах функций и обращениях к разделяемой памяти. Далее запускается детектор гонок, который обходит построенный граф. Он выполняет проверку на наличие гонки для каждой инструкции в графе, обновляя при необходимости множества захваченных блокировок. Поскольку обход всего графа сложен, авторы применяют различные методики редукции, которые минимизируют нагрузку, но ведут к возможной потере части гонок. На последнем этапе обнаруженные гонки приоритезируются по важности и потенциальной опасности. Авторы приложили значительные усилия к уменьшению количества ложных срабатываний и повышения достоверности обнаруживаемых гонок. На анализ 1.8 миллионов строк кода RacerX тратит 2–14 минут. Утилита была апробирована на исходном коде FreeBSD, Linux (обнаружено суммарно 16 ошибок) и ряде крупных коммерческих приложений.

Аналогичная методика многофазного потокозависимого анализа для поиска гонок в Java-программах используется в утилите Chord [32]. Авторы разработали сложный многоэтапный алгоритм, который составляет множество всех пар обращений к объектам и последовательно удаляет пары, которые гарантированно не могут быть вовлечены в гонку. Подобный подход является типичным для современных статических детекторов. Схема работы Chord представлена на рис. 2.

На первом этапе отбрасываются все пары обращений, которые недостижимы, т.е. не могут возникнуть одновременно ни на каком пути выполнения программы. На втором этапе отбрасываются пары обращений к разным участкам памяти, на третьем этапе – пары, которые обращаются к локальным данным потока, а не к разделяемым данным. На последнем этапе остаются только те пары объектов, обращение к которым осуществляется без захвата общей блокировки. Авторы провели тщательную апробацию подхода как на тестах производительности, так и на больших приложениях. Полученные результаты показали высокую точность алгоритма и сравнительно небольшое время работы. Так, анализ всего исходного кода реляционной СУБД Apache Derby [7] (650000 строк кода), написанной полностью на Java, занял 26 минут и выявил 1018 пар обращений к памяти, образующих гонки. Это выявило 319 ошибок, среди которых не

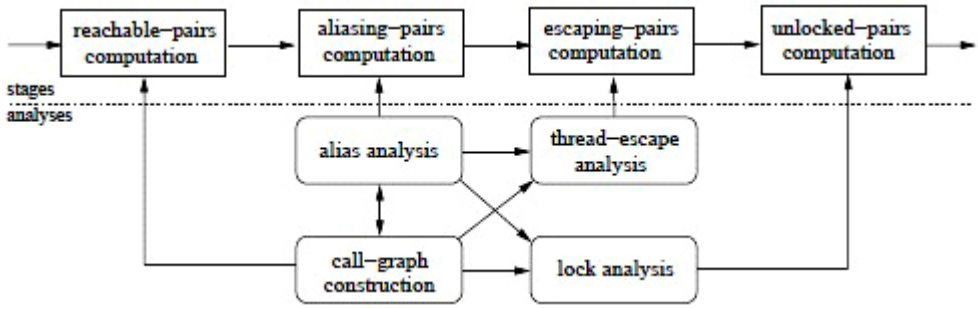

Рис. 2. Схема работы статического детектора Chord (рисунок взят из работы [32])

было ни одного ложного срабатывания. Также нулевой процент ложных срабатываний при большом количестве реально обнаруженных гонок был выявлен при анализе ещё трех популярных библиотеках группы Apache. На анализ 120–160 тысяч строк кода у Chord уходило от полутора до пяти минут. Таким образом, по-видимому, Chord является наиболее эффективным из рассмотренных продуктов для статического анализа Java-программ – при сравнительно небольшом времени, затрачиваемом на анализ, его отличает очень высокая точность и большое количество обнаруженных ошибок – на порядок больше, чем у предшественников. Кроме того, исходный код утилиты находится в открытом доступе [26].

ОБНАРУЖЕНИЕ ГОНОК ПОСЛЕ ЗАВЕРШЕНИЯ РАБОТЫ ПРОГРАММЫ

Данный подход (post-mortem approach) предназначен для поиска гонок путём динамического сбора информации о ходе выполнения программы с последующим ее анализом уже после завершения работы приложения [28]. Подходы в рамках этой парадигмы минимизируют накладные расходы во время выполнения программы за счёт выполнения анализа трассы программы после окончания работы приложения, во время работы программы информация о событиях только записывается. При этом, в отличие от статических детекторов, поиск осуществляется только по тем путям выполнения программы, которые были достигнуты во время ее запуска. Основной недостаток post-mortem подхода заключается в том, что он не позволяют обнаружить ошибку непосредственно сразу же после её возникновения и предотвратить дальнейшее повреждение данных. Кроме того, без дополнительных улучшений такой метод не подходит для больших систем, время выполнения велико или не ограничено.

Одним из самых перспективных направлений парадигмы post-mortem является replay-подход. В рамках этого подхода во время выполнения программы собирается минимальное количество информации, необходимое для ее перезапуска (с некоторой степенью достоверности), а затем программа перезапускается и проводится её полный анализ. Одна из таких утилит RecPlay описана в работе [38]. При повторном запуске программы утилита доходит до первой гонки и выдаёт предупреждение пользователю. После устранения этой гонки она запускается повторно до следующей гонки, и так продолжается до тех пор, пока все гонки не будут устранены. Для обнаружения гонок при первом запуске программы RecPlay записывает только информацию о синхронизационных событиях с искусственными логическими временными метками, что похоже на векторные часы, ипользующиеся в алгоритме happens-before. Во время непосредственного поиска гонок (т.е. во время повторного запуска программы) RecPlay анализирует обращения к памяти, с помощью векторных часов находит конфликтующие обращения и, наконец, находит команды, которые привели к гонке. Эта утилита совершенно независима от компилятора и языка программирования, но работает только на ОС Solaris. Накладные расходы программы составляют примерно 90%.

Ещё одна replay-утилита для Java под названием Déjà vu представлена в работе [11] и основана на концепции логического расписания потоков. Такое расписание является последовательностью временных интервалов, где каждый интервал соответствует событиям, происходящим последовательно в некотором потоке, а границы интервала – точкам, в которых происходит переключение потоков. Утилита сохраняет все критические события, т.е. все синхронизационные события и обращения к разделяемым переменным внутри логических интервалов. Для формирования этих интервалов для каждого потока используются глобальные и локальные часы. Все синхронизационные события отслеживаются посредством обновления глобальных часов и сохранения их значений с помощью локальных часов. Когда поток приостанавливается, его локальные часы также останавливаются, а глобальные продолжают идти. Таким образом, эти часы независимы от планировщика операционной системы или языка программирования. Наличие списка таких интервалов для каждого потока в программе достаточно для её детерминированного перезапуска — потоки будут вести себя (с точки зрения обнаружения гонок и порядка доступа к ресурсам) точно также, как при первом запуске программы. В начале перезапуска каждый поток получает список своих логических интервалов. Когда достигается критическое событие, поток увеличивает глобальные часы и ждет, пока они дойдут до начала его следующего интервала. Для реализации механизма записи данных и последующего повторения программы авторы модифицировали JVM. На достаточно больших программах их утилита показала накладные расходы не более 80–90%. К сожалению, утилита обеспечивает детерминизм только на уровне чередования потоков и не поддерживает ввод-вывод данных, события интерфейса и системные вызовы. Эта проблема является основной для replay-утилит: с одной стороны, точный перезапуск требует сохранения как можно большего числа недетерминированных событий; с другой стороны, сохранение всех видов таких событий – очень сложная, фактически, невыполнимая задача, к тому же приводящая к большим накладным расходам.

ЗАКЛЮЧЕНИЕ

В данной работе мы рассмотрели статические и post-mortem подходы к обнаружению гонок в многопоточных программах, уделив особое внимание программам, написанным на языке Java.

В табл. 2 представлены все утилиты, которые были упомянуты в данной статье. В таблице приведен сравнительный анализ рассмотренных утилит по производительности. Эта информация представлена следующим образом:

– для статических утилит указывалось, как правило, время работы утилиты в зависимости от количества строк кода исходной программы (KLOC/MLOC — тысячи/миллионы строк) или средняя скорость его обработки (KLOC/сек.);

– для post-mortem утилит приводилось замедление работы стандартных тестов производительности в процентах на этапе сбора информации во время первичного пуска программы.

Статический подход к обнаружению гонок основывается на анализе исходного кода программы и выделении множества обращений к памяти, потенциально вовлеченных в гонки. Утилиты, основанные на этом подходе, не ограничены временем работы исходного приложения, поскольку не требуют его запуска. Однако полный анализ на наличие гонок является алгоритмически неразрешимой задачей [33, 34], поэтому для получения достоверного результата необходима дополнительная информация о наиболее вероятных путях исполнения программы. Статическим детекторам эта информация недоступна, или требуется дополнительное трудоемкое аннотирование исходных текстов. Статические детекторы также допускают большое количество ложных срабатываний. После ряда исследований авторам утилиты RacerX [17] удалось частично решить первую проблему (трудоемкость «ручного» аннотирования), а авторам Chord [32] – уменьшить количество ложных срабатываний фак-

тически до нуля без потери в производительности. Однако большинство статических детекторов основывается на алгоритме lockset (поскольку они не могут отслеживать реальные действия потоков), что приводит к неполному обнаружению гонок и неполному покрытию механизмов синхронизации языков программирования. Эти проблемы делают актуальным post-mortem подход.

В рамках post-mortem подхода анализируются реальные трассы работы программы, собранные в процессе ее выполнения. Ключевым недостатком такого подхода является невозможность сигнализации об ошибке и принятия соответствующих действий непосредственно после её обнаружения, поэтому такой подход может быть применен лишь на этапе тестирования и не может быть использован для обнаружения гонок на работающих системах. Среди post-mortem утилит наибольшее распространение получил replay-подход, в котором информация, собранная во время работы программы, используется для повторного запуска программы с тем же чередованием потоков, во время которого и осуществляется анализ. К со-

Табл. 2. Сравнительный анализ статических и post-modern детекторов обнаружения гонок

Статические детекторы			
Название утилиты	**Краткое описание**	**Производительность**	**Год появления**
RaceFree Java type system	Статическая система типов для языка Java, позволяющая предотвратить как гонки, так и взаимные блокировки.	–	2001
ESC/Java	Утилита использует автоматическое доказательство теорем, требует значительного количества аннотаций, неприменима для интерактивного использования вследствие неразрешимости задачи доказательства теорем в общем случае.	Обрабатывает 98% функций приложения размером 41 KLOC при ограничении «не больше минуты на функцию» (667 MHz Alpha CPU).	2001
RacerX	Утилита позволяет единожды указать пары синхронизационных операций, чтобы впоследствии избежать «ручного» аннотирования. Применяет множество различных оптимизаций ценой потери некоторых гонок.	14 мин. на анализ 1.8 MLOC (подробностей нет)	2003
Chord	Многоэтапный детектор с высокой производительностью и точностью. Есть open-source реализация для Java.	1.5-5 мин. на 120-140 KLOC; 28 мин на 650 KLOC (2.4 GHz CPU, 4 GB RAM)	2006
Rccjava	Расширение системы типов Java, поддерживает аннотации для повышения производительности и точности, графический интерфейс.	В среднем обрабатывает порядка 2 KLOC в минуту (667 MHz Alpha CPU)	2000, 2006
Post-mortem детекторы			
Déjà vu	Позволяет перезапустить программу с тем же чередованием потоков за счёт построения логических временных интервалов работы потоков.	80-90%	1998
RecPlay	Собирает информацию о синхронизационных событиях во время работы программы, после этого позволяет перезапустить программу с таким же порядком чередования потоков.	порядка 90%	1999

жалению, детерминизм перезапуска программы ограничивается лишь взаимодействием потоков и не поддерживает ввод-вывод данных, события интерфейса и системные вызовы.

Указанные выше недостатки статического и post-mortem подходов привели к развитию динамических методов обнаружения гонок.

Литература

1. *Кудрин М., Прокопенко А., Тормасов А.* Метод нахождения состояний гонки в потоках, работающих на разделяемой памяти // Труды МФТИ. Том 1, № 4, 2009. С. 182–201.
2. *Одинцов И.* Профессиональное программирование. Системный подход. БХВ-Петербург, 2006.
3. *Салищев С.И., Ушаков Д.С.* Использование языков и сред управляемого исполнения для системного программирования // Системное программирование. Вып. 4: Сб. статей / Под ред. А.Н. Терехова, Д.Ю. Булычева. 2009. С. 197–216.
4. *Сергей И.Д.* Реализация гибридных типов владения в Java посредством атрибутных грамматик // Системное программирование. Вып. 6: Сб. статей / Под ред. А.Н. Терехова, Д.Ю. Булычева. 2011. С. 49–79.
5. *Abadi M., Flanagan C., Freund S.* Types for safe locking: Static race detection for Java. In ACM Transactions on Programming Languages and Systems, Vol. 28, Issue 2, March 2006. P. 207–255.
6. *Adve S.* Data races are evil with no exceptions: technical perspective. In Communications of the ACM, Vol. 53, Issue 11, November 2010. P. 84–84.
7. Apache Derby Project, http://db.apache.org/derby/ (дата обращения: 15.12.2011).
8. *Boehm H.* Threads cannot be implemented as a library. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, Vol. 40, Issue 6, June 2005. P. 261–268.
9. *Boyapati C., Rinard M.* A parameterised type system for race-free java programs. In ACM Conference on Object-Oriented Programming Systems Languages and Applications, 2001. P. 56–69.
10. *Choi J., Lee K., Loginov A., O'Callahan R., Sarkar V., Sridharan M. Efficient* and precise datarace detection for multithreaded object-oriented programs. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, 2002. P. 258–269.
11. *Choi J., Srinivasan H.* Deterministic replay of Java multithreaded applications. SPDT '98 Proceedings of the SIGMETRICS symposium on Parallel and distributed tools, 1998. P. 48–59.
12. *Christiaens M., Brosschere K.* TRaDe: A topological approach to on-the-fly race detection in Java programs. In Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology Symposium, Vol. 1, 2001. P. 105–116.
13. *Cohen E., Lamport L.* Reduction in TLA. In International Conference on Concurrency Theory, 1998. P. 317–331.
14. Documentation of java.util.concurrent package, http://download.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html (дата обращения: 15.12.2011).
15. *Elmas T., Qadeer S., Tasiran S.* Goldilocks: A Race and Transaction-Aware Java Runtime. In Proceedings of The 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07), 2007. P. 245–255.
16. *Elmas T., Qadeer S., Tasiran S.* Precise Race Detection and Efficient Model Checking Using Locksets. Microsoft Tech Report, 2006.
17. *Engler D., Ashcraft K.* RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In Proceedings of The Nineteenth ACM Symposium on Operating Systems Principles, 2003. P. 237–252.
18. Extended Static Checker for Java, http://secure.ucd.ie/products/opensource/ESCJava2/ (дата обращения: 15.12.2011).
19. *Flanagan C., Freund S.* Detecting race conditions in large programs. In Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE'01), June 2001. P. 90–96.
20. *Flanagan C., Freund S.* FastTrack: Efficient and Precise Dynamic Race Detection. In ACM Conference on Programming Language Design and Implementation, 2009. P. 121–133.
21. *Flanagan C., Freund S.* Type-Based Race Detection for Java. In Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, 2000. P. 219–232.
22. *Flanagan C., Joshi R., Rustan K., Leino M.* Annotation Inference for Modular Checkers. Information Processing Letters, Vol. 77, Issue 2–4, 2001. P. 97–108.

23. *Flanagan C., Qadeer S.* Types for atomicity. In Proceedings of the 2003 ACM SIGPLAN international Workshop on Types in Languages Design and Implementation, 2003. P. 1–12.

24. Java Language Specification, Third Edition. Threads and Locks. http://java.sun.com/docs/books/jls/third_edition/html/memory.html (дата обращения: 15.12.2011).

25. Java PathFinder, http://javapathfinder.sourceforge.net/ (дата обращения: 15.12.2011).

26. jChord project home page, http://code.google.com/p/jchord/ (дата обращения: 15.12.2011).

27. *Ha O. et al.* Empirical Comparison of Race Detection Tools for OpenMP Programs. In Communications in Computer and Information Science, Vol. 63, 2009. P. 108–116.

28. *Helmbold D., McDowell C.* A taxonomy of race detection algorithms. Technical Report UCSC-CRL-94-35, 1994.

29. Intel Threading Building Blocks, http://www.threadingbuildingblocks.org/ (дата обращения: 15.12.2011).

30. *Leino K., Nelson G., Saxe J.* ESC/Java user's manual. SRC Technical Note 2000–002, 2001.

31. *Leveson N., Turner C. S.* An Investigation of The Therac-25 Accidents. In IEEE Computer, Vol. 26, № 7, 1993. P. 18–41.

32. *Naik M., Aiken A., Whaley J.* Effective Static Race Detection for Java. In Proceedings of The 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2006. P. 308–319.

33. *Netzer R.* Race Condition Detection for Debugging Shared-Memory Parallel Programs. PhD Thesis, Madison, 1991. 109 p.

34. *Netzer R., Miller B.* What Are Race Conditions? Some Issues and Formalizations. In ACM Letters On Programming Languages and Systems, 1(1), 1992. P. 74–88.

35. *Pozniansky E., Schuster A.* Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In Proceedings of The Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2003. P. 179–190.

36. *Praun C., Gross T.* Static conflict analysis for multi-threaded object-oriented programs. In Proceedings of the Conference on Programming Language Design and Implementation (PLDI'03), 2003. P. 115–129.

37. *Raza A.* A Review of Race Detection Mechanisms. Lecture Notes in Computer Science, Vol. 3967, 2006. P. 534–543.

38. *Ronse M., De Bosschere K.* RecPlay : A Fully Integrated Practical Record/Replay System. In ACM Transactions on Computer Systems, Vol.17, No.2, 1999. P. 133–132.

39. *Savage S., Burrows M., Nelson G., Sobalvarro P., Anderson T.* Eraser: A Dynamic Data Race Detector for Multithreaded Programs. In ACM Transactions on Computer Systems, Vol. 15, Issue 4. 1997. P. 391–411.

Abstract

One of the most hazardous and hardly reproducible errors that occur in multithreaded programs are data races – unsynchronized accesses to same shared memory fragment from several threads, where one access is write. Generally data races are weakly localized and damage global data structures. Manual detection of data races is very complicated. There was a lot of research in this area, but automatic data race detection remains an actual issue. In this review static and post-portem approaches to data race detection are covered with an emphasis on race detection in Java programs.

Keywords: concurrency, data race, automatic bugs detection.

Трифанов Виталий Юрьевич,
аспирант кафедры системного программирования математико-механического факультета СПбГУ,
инженер-программист компании «Эксперт-Система»,
vitaly.trifanov@gmail.com,

Цителов Дмитрий Игоревич,
руководитель группы внутренних разработок компании «Эксперт-Система»,
tsitelov@acm.org.